PostgreSQL 8.2.6 Documentation

The PostgreSQL Global Development Group

PostgreSQL 8.2.6 Documentation

by The PostgreSQL Global Development Group

Copyright © 1996-2006 The PostgreSQL Global Development Group

Legal Notice

PostgreSQL is Copyright © 1996-2006 by the PostgreSQL Global Development Group and is distributed under the terms of the license of the University of California below.

Postgres95 is Copyright © 1994-5 by the Regents of the University of California.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN “AS-IS” BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Table of Contents

Preface ..... **xli**

1. What is PostgreSQL? xli
2. A Brief History of PostgreSQL xlii
    2.1. The Berkeley POSTGRES Project xlii
    2.2. Postgres95 xliii
    2.3. PostgreSQL xliii
3. Conventions xliii
4. Further Information xliv
5. Bug Reporting Guidelines xlv
    5.1. Identifying Bugs xlv
    5.2. What to report xlv
    5.3. Where to report bugs xlvii

I. Tutorial **1**

1. Getting Started 1
    1.1. Installation 1
    1.2. Architectural Fundamentals 1
    1.3. Creating a Database 2
    1.4. Accessing a Database 3
2. The SQL Language 6
    2.1. Introduction 6
    2.2. Concepts 6
    2.3. Creating a New Table 6
    2.4. Populating a Table With Rows 7
    2.5. Querying a Table 8
    2.6. Joins Between Tables 10
    2.7. Aggregate Functions 12
    2.8. Updates 14
    2.9. Deletions 14
3. Advanced Features 16
    3.1. Introduction 16
    3.2. Views 16
    3.3. Foreign Keys 16
    3.4. Transactions 17
    3.5. Inheritance 19
    3.6. Conclusion 21

II. The SQL Language **22**

4. SQL Syntax 24
    4.1. Lexical Structure 24
        4.1.1. Identifiers and Key Words 24
        4.1.2. Constants 25
            4.1.2.1. String Constants 25
            4.1.2.2. Dollar-Quoted String Constants 26
            4.1.2.3. Bit-String Constants 27
            4.1.2.4. Numeric Constants 28
            4.1.2.5. Constants of Other Types 28

4.1.3. Operators 29
4.1.4. Special Characters 30
4.1.5. Comments 30
4.1.6. Lexical Precedence 31
4.2. Value Expressions 32
4.2.1. Column References 33
4.2.2. Positional Parameters 33
4.2.3. Subscripts 33
4.2.4. Field Selection 34
4.2.5. Operator Invocations 34
4.2.6. Function Calls 35
4.2.7. Aggregate Expressions 35
4.2.8. Type Casts 36
4.2.9. Scalar Subqueries 37
4.2.10. Array Constructors 37
4.2.11. Row Constructors 38
4.2.12. Expression Evaluation Rules 40
5. Data Definition 41
5.1. Table Basics 41
5.2. Default Values 42
5.3. Constraints 43
5.3.1. Check Constraints 43
5.3.2. Not-Null Constraints 45
5.3.3. Unique Constraints 46
5.3.4. Primary Keys 47
5.3.5. Foreign Keys 48
5.4. System Columns 50
5.5. Modifying Tables 52
5.5.1. Adding a Column 52
5.5.2. Removing a Column 53
5.5.3. Adding a Constraint 53
5.5.4. Removing a Constraint 53
5.5.5. Changing a Column's Default Value 54
5.5.6. Changing a Column's Data Type 54
5.5.7. Renaming a Column 54
5.5.8. Renaming a Table 54
5.6. Privileges 55
5.7. Schemas 55
5.7.1. Creating a Schema 56
5.7.2. The Public Schema 57
5.7.3. The Schema Search Path 57
5.7.4. Schemas and Privileges 59
5.7.5. The System Catalog Schema 59
5.7.6. Usage Patterns 59
5.7.7. Portability 60
5.8. Inheritance 60
5.8.1. Caveats 63
5.9. Partitioning 63

5.9.1. Overview 63
5.9.2. Implementing Partitioning 64
5.9.3. Managing Partitions 67
5.9.4. Partitioning and Constraint Exclusion 68
5.9.5. Caveats 69
5.10. Other Database Objects 70
5.11. Dependency Tracking 70
6. Data Manipulation 72
6.1. Inserting Data 72
6.2. Updating Data 73
6.3. Deleting Data 74
7. Queries 75
7.1. Overview 75
7.2. Table Expressions 75
7.2.1. The FROM Clause 76
7.2.1.1. Joined Tables 76
7.2.1.2. Table and Column Aliases 79
7.2.1.3. Subqueries 81
7.2.1.4. Table Functions 81
7.2.2. The WHERE Clause 82
7.2.3. The GROUP BY and HAVING Clauses 83
7.3. Select Lists 85
7.3.1. Select-List Items 85
7.3.2. Column Labels 86
7.3.3. DISTINCT 86
7.4. Combining Queries 87
7.5. Sorting Rows 87
7.6. LIMIT and OFFSET 88
7.7. VALUES Lists 89
8. Data Types 91
8.1. Numeric Types 92
8.1.1. Integer Types 93
8.1.2. Arbitrary Precision Numbers 93
8.1.3. Floating-Point Types 94
8.1.4. Serial Types 95
8.2. Monetary Types 96
8.3. Character Types 96
8.4. Binary Data Types 98
8.5. Date/Time Types 100
8.5.1. Date/Time Input 101
8.5.1.1. Dates 102
8.5.1.2. Times 102
8.5.1.3. Time Stamps 103
8.5.1.4. Intervals 104
8.5.1.5. Special Values 105
8.5.2. Date/Time Output 105
8.5.3. Time Zones 106
8.5.4. Internals 108

8.6. Boolean Type 108
8.7. Geometric Types 109
8.7.1. Points 110
8.7.2. Line Segments 110
8.7.3. Boxes 110
8.7.4. Paths 110
8.7.5. Polygons 111
8.7.6. Circles 111
8.8. Network Address Types 111
8.8.1. `inet` 112
8.8.2. `cidr` 112
8.8.3. `inet` vs. `cidr` 113
8.8.4. `macaddr` 113
8.9. Bit String Types 113
8.10. Arrays 114
8.10.1. Declaration of Array Types 114
8.10.2. Array Value Input 115
8.10.3. Accessing Arrays 116
8.10.4. Modifying Arrays 118
8.10.5. Searching in Arrays 121
8.10.6. Array Input and Output Syntax 121
8.11. Composite Types 123
8.11.1. Declaration of Composite Types 123
8.11.2. Composite Value Input 124
8.11.3. Accessing Composite Types 125
8.11.4. Modifying Composite Types 125
8.11.5. Composite Type Input and Output Syntax 126
8.12. Object Identifier Types 127
8.13. Pseudo-Types 128
8.14. XML Document Support 129
9. Functions and Operators 131
9.1. Logical Operators 131
9.2. Comparison Operators 131
9.3. Mathematical Functions and Operators 133
9.4. String Functions and Operators 136
9.5. Binary String Functions and Operators 146
9.6. Bit String Functions and Operators 148
9.7. Pattern Matching 149
9.7.1. `LIKE` 149
9.7.2. `SIMILAR TO` Regular Expressions 150
9.7.3. POSIX Regular Expressions 151
9.7.3.1. Regular Expression Details 152
9.7.3.2. Bracket Expressions 155
9.7.3.3. Regular Expression Escapes 156
9.7.3.4. Regular Expression Metasyntax 158
9.7.3.5. Regular Expression Matching Rules 160
9.7.3.6. Limits and Compatibility 161
9.7.3.7. Basic Regular Expressions 162

9.8. Data Type Formatting Functions 162
9.9. Date/Time Functions and Operators 168
9.9.1. EXTRACT, date_part 172
9.9.2. date_trunc 176
9.9.3. AT TIME ZONE 176
9.9.4. Current Date/Time 177
9.9.5. Delaying Execution 179
9.10. Geometric Functions and Operators 179
9.11. Network Address Functions and Operators 183
9.12. Sequence Manipulation Functions 186
9.13. Conditional Expressions 188
9.13.1. CASE 188
9.13.2. COALESCE 189
9.13.3. NULLIF 190
9.13.4. GREATEST and LEAST 190
9.14. Array Functions and Operators 190
9.15. Aggregate Functions 192
9.16. Subquery Expressions 195
9.16.1. EXISTS 196
9.16.2. IN 196
9.16.3. NOT IN 197
9.16.4. ANY/SOME 197
9.16.5. ALL 198
9.16.6. Row-wise Comparison 198
9.17. Row and Array Comparisons 199
9.17.1. IN 199
9.17.2. NOT IN 199
9.17.3. ANY/SOME (array) 200
9.17.4. ALL (array) 200
9.17.5. Row-wise Comparison 200
9.18. Set Returning Functions 201
9.19. System Information Functions 202
9.20. System Administration Functions 209
10. Type Conversion 215
10.1. Overview 215
10.2. Operators 216
10.3. Functions 219
10.4. Value Storage 222
10.5. UNION, CASE, and Related Constructs 222
11. Indexes 225
11.1. Introduction 225
11.2. Index Types 226
11.3. Multicolumn Indexes 227
11.4. Combining Multiple Indexes 228
11.5. Unique Indexes 229
11.6. Indexes on Expressions 230
11.7. Partial Indexes 230
11.8. Operator Classes 233

11.9. Examining Index Usage 234
12. Concurrency Control 236
12.1. Introduction 236
12.2. Transaction Isolation 236
12.2.1. Read Committed Isolation Level 237
12.2.2. Serializable Isolation Level 238
12.2.2.1. Serializable Isolation versus True Serializability 239
12.3. Explicit Locking 240
12.3.1. Table-Level Locks 240
12.3.2. Row-Level Locks 242
12.3.3. Deadlocks 242
12.3.4. Advisory Locks 243
12.4. Data Consistency Checks at the Application Level 244
12.5. Locking and Indexes 245
13. Performance Tips 246
13.1. Using EXPLAIN 246
13.2. Statistics Used by the Planner 251
13.3. Controlling the Planner with Explicit JOIN Clauses 252
13.4. Populating a Database 254
13.4.1. Disable Autocommit 254
13.4.2. Use COPY 254
13.4.3. Remove Indexes 254
13.4.4. Remove Foreign Key Constraints 255
13.4.5. Increase maintenance_work_mem 255
13.4.6. Increase checkpoint_segments 255
13.4.7. Run ANALYZE Afterwards 255
13.4.8. Some Notes About pg_dump 255
III. Server Administration 257
14. Installation Instructions 259
14.1. Short Version 259
14.2. Requirements 259
14.3. Getting The Source 261
14.4. If You Are Upgrading 261
14.5. Installation Procedure 262
14.6. Post-Installation Setup 270
14.6.1. Shared Libraries 270
14.6.2. Environment Variables 271
14.7. Supported Platforms 271
15. Client-Only Installation on Windows 278
16. Operating System Environment 280
16.1. The PostgreSQL User Account 280
16.2. Creating a Database Cluster 280
16.3. Starting the Database Server 281
16.3.1. Server Start-up Failures 282
16.3.2. Client Connection Problems 283
16.4. Managing Kernel Resources 284
16.4.1. Shared Memory and Semaphores 284

16.4.2. Resource Limits 290
16.4.3. Linux Memory Overcommit 291
16.5. Shutting Down the Server 291
16.6. Encryption Options 292
16.7. Secure TCP/IP Connections with SSL 293
16.8. Secure TCP/IP Connections with SSH Tunnels 294
17. Server Configuration 296
17.1. Setting Parameters 296
17.2. File Locations 297
17.3. Connections and Authentication 298
17.3.1. Connection Settings 298
17.3.2. Security and Authentication 300
17.4. Resource Consumption 301
17.4.1. Memory 301
17.4.2. Free Space Map 303
17.4.3. Kernel Resource Usage 303
17.4.4. Cost-Based Vacuum Delay 304
17.4.5. Background Writer 305
17.5. Write Ahead Log 306
17.5.1. Settings 306
17.5.2. Checkpoints 308
17.5.3. Archiving 308
17.6. Query Planning 309
17.6.1. Planner Method Configuration 309
17.6.2. Planner Cost Constants 310
17.6.3. Genetic Query Optimizer 311
17.6.4. Other Planner Options 312
17.7. Error Reporting and Logging 313
17.7.1. Where To Log 313
17.7.2. When To Log 314
17.7.3. What To Log 316
17.8. Run-Time Statistics 318
17.8.1. Query and Index Statistics Collector 319
17.8.2. Statistics Monitoring 319
17.9. Automatic Vacuuming 320
17.10. Client Connection Defaults 321
17.10.1. Statement Behavior 321
17.10.2. Locale and Formatting 323
17.10.3. Other Defaults 324
17.11. Lock Management 325
17.12. Version and Platform Compatibility 326
17.12.1. Previous PostgreSQL Versions 326
17.12.2. Platform and Client Compatibility 327
17.13. Preset Options 328
17.14. Customized Options 329
17.15. Developer Options 329
17.16. Short Options 331
18. Database Roles and Privileges 333

18.1. Database Roles 333
18.2. Role Attributes 334
18.3. Privileges 335
18.4. Role Membership 336
18.5. Functions and Triggers 337
19. Managing Databases 339
19.1. Overview 339
19.2. Creating a Database 339
19.3. Template Databases 340
19.4. Database Configuration 341
19.5. Destroying a Database 342
19.6. Tablespaces 342
20. Client Authentication 345
20.1. The `pg_hba.conf` file 345
20.2. Authentication methods 350
20.2.1. Trust authentication 350
20.2.2. Password authentication 350
20.2.3. Kerberos authentication 351
20.2.4. Ident-based authentication 352
20.2.4.1. Ident Authentication over TCP/IP 352
20.2.4.2. Ident Authentication over Local Sockets 352
20.2.4.3. Ident Maps 353
20.2.5. LDAP authentication 354
20.2.6. PAM authentication 354
20.3. Authentication problems 354
21. Localization 356
21.1. Locale Support 356
21.1.1. Overview 356
21.1.2. Behavior 357
21.1.3. Problems 358
21.2. Character Set Support 358
21.2.1. Supported Character Sets 358
21.2.2. Setting the Character Set 361
21.2.3. Automatic Character Set Conversion Between Server and Client 362
21.2.4. Further Reading 364
22. Routine Database Maintenance Tasks 366
22.1. Routine Vacuuming 366
22.1.1. Recovering disk space 366
22.1.2. Updating planner statistics 367
22.1.3. Preventing transaction ID wraparound failures 368
22.1.4. The auto-vacuum daemon 370
22.2. Routine Reindexing 371
22.3. Log File Maintenance 372
23. Backup and Restore 373
23.1. SQL Dump 373
23.1.1. Restoring the dump 373
23.1.2. Using pg_dumpall 374
23.1.3. Handling large databases 375

23.2. File System Level Backup 376
23.3. Continuous Archiving and Point-In-Time Recovery (PITR) 377
23.3.1. Setting up WAL archiving 378
23.3.2. Making a Base Backup 379
23.3.3. Recovering using a Continuous Archive Backup 381
23.3.3.1. Recovery Settings 383
23.3.4. Timelines 384
23.3.5. Caveats 385
23.4. Warm Standby Servers for High Availability 385
23.4.1. Planning 386
23.4.2. Implementation 387
23.4.3. Failover 388
23.4.4. Record-based Log Shipping 388
23.4.5. Incrementally Updated Backups 389
23.5. Migration Between Releases 389
24. High Availability and Load Balancing 391
25. Monitoring Database Activity 394
25.1. Standard Unix Tools 394
25.2. The Statistics Collector 395
25.2.1. Statistics Collection Configuration 395
25.2.2. Viewing Collected Statistics 395
25.3. Viewing Locks 401
25.4. Dynamic Tracing 402
25.4.1. Compiling for Dynamic Tracing 402
25.4.2. Built-in Trace Points 402
25.4.3. Using Trace Points 403
25.4.4. Defining Trace Points 404
26. Monitoring Disk Usage 406
26.1. Determining Disk Usage 406
26.2. Disk Full Failure 407
27. Reliability and the Write-Ahead Log 408
27.1. Reliability 408
27.2. Write-Ahead Logging (WAL) 409
27.3. WAL Configuration 409
27.4. WAL Internals 411
28. Regression Tests 412
28.1. Running the Tests 412
28.2. Test Evaluation 413
28.2.1. Error message differences 414
28.2.2. Locale differences 414
28.2.3. Date and time differences 414
28.2.4. Floating-point differences 414
28.2.5. Row ordering differences 414
28.2.6. Insufficient stack depth 415
28.2.7. The “random” test 415
28.3. Variant Comparison Files 415

IV. Client Interfaces 417
29. libpq - C Library 419
29.1. Database Connection Control Functions 419
29.2. Connection Status Functions 425
29.3. Command Execution Functions 428
29.3.1. Main Functions 428
29.3.2. Retrieving Query Result Information 435
29.3.3. Retrieving Result Information for Other Commands 439
29.3.4. Escaping Strings for Inclusion in SQL Commands 439
29.3.5. Escaping Binary Strings for Inclusion in SQL Commands 440
29.4. Asynchronous Command Processing 442
29.5. Cancelling Queries in Progress 446
29.6. The Fast-Path Interface 447
29.7. Asynchronous Notification 448
29.8. Functions Associated with the COPY Command 449
29.8.1. Functions for Sending COPY Data 450
29.8.2. Functions for Receiving COPY Data 450
29.8.3. Obsolete Functions for COPY 451
29.9. Control Functions 453
29.10. Miscellaneous Functions 454
29.11. Notice Processing 454
29.12. Environment Variables 455
29.13. The Password File 457
29.14. The Connection Service File 457
29.15. LDAP Lookup of Connection Parameters 458
29.16. SSL Support 458
29.17. Behavior in Threaded Programs 459
29.18. Building libpq Programs 460
29.19. Example Programs 461
30. Large Objects 471
30.1. Introduction 471
30.2. Implementation Features 471
30.3. Client Interfaces 471
30.3.1. Creating a Large Object 471
30.3.2. Importing a Large Object 472
30.3.3. Exporting a Large Object 472
30.3.4. Opening an Existing Large Object 472
30.3.5. Writing Data to a Large Object 473
30.3.6. Reading Data from a Large Object 473
30.3.7. Seeking in a Large Object 473
30.3.8. Obtaining the Seek Position of a Large Object 474
30.3.9. Closing a Large Object Descriptor 474
30.3.10. Removing a Large Object 474
30.4. Server-Side Functions 474
30.5. Example Program 475
31. ECPG - Embedded SQL in C 481
31.1. The Concept 481
31.2. Connecting to the Database Server 481

31.3. Closing a Connection 482
31.4. Running SQL Commands 483
31.5. Choosing a Connection 484
31.6. Using Host Variables 484
31.6.1. Overview 485
31.6.2. Declare Sections 485
31.6.3. Different types of host variables 486
31.6.4. `SELECT INTO` and `FETCH INTO` 487
31.6.5. Indicators 487
31.7. Dynamic SQL 488
31.8. pgtypes library 489
31.8.1. The numeric type 489
31.8.2. The date type 492
31.8.3. The timestamp type 496
31.8.4. The interval type 500
31.8.5. The decimal type 500
31.8.6. errno values of pgtypeslib 501
31.8.7. Special constants of pgtypeslib 502
31.9. Informix compatibility mode 502
31.9.1. Additional embedded SQL statements 502
31.9.2. Additional functions 503
31.9.3. Additional constants 511
31.10. Using SQL Descriptor Areas 513
31.11. Error Handling 514
31.11.1. Setting Callbacks 515
31.11.2. sqlca 516
31.11.3. `SQLSTATE` vs `SQLCODE` 517
31.12. Preprocessor directives 520
31.12.1. Including files 520
31.12.2. The #define and #undef directives 520
31.12.3. ifdef, ifndef, else, elif and endif directives 521
31.13. Processing Embedded SQL Programs 522
31.14. Library Functions 523
31.15. Internals 523
32. The Information Schema 526
32.1. The Schema 526
32.2. Data Types 526
32.3. `information_schema_catalog_name` 527
32.4. `administrable_role_authorizations` 527
32.5. `applicable_roles` 527
32.6. `attributes` 528
32.7. `check_constraint_routine_usage` 531
32.8. `check_constraints` 531
32.9. `column_domain_usage` 532
32.10. `column_privileges` 532
32.11. `column_udt_usage` 533
32.12. `columns` 534
32.13. `constraint_column_usage` 538

32.14. constraint_table_usage 539
32.15. data_type_privileges 540
32.16. domain_constraints 541
32.17. domain_udt_usage 541
32.18. domains 542
32.19. element_types 545
32.20. enabled_roles 547
32.21. key_column_usage 548
32.22. parameters 548
32.23. referential_constraints 551
32.24. role_column_grants 552
32.25. role_routine_grants 553
32.26. role_table_grants 553
32.27. role_usage_grants 554
32.28. routine_privileges 555
32.29. routines 556
32.30. schemata 561
32.31. sequences 562
32.32. sql_features 563
32.33. sql_implementation_info 564
32.34. sql_languages 565
32.35. sql_packages 565
32.36. sql_parts 566
32.37. sql_sizing 566
32.38. sql_sizing_profiles 567
32.39. table_constraints 567
32.40. table_privileges 568
32.41. tables 569
32.42. triggers 570
32.43. usage_privileges 571
32.44. view_column_usage 572
32.45. view_routine_usage 573
32.46. view_table_usage 573
32.47. views 574

V. Server Programming 575

33. Extending SQL 577
33.1. How Extensibility Works 577
33.2. The PostgreSQL Type System 577
33.2.1. Base Types 577
33.2.2. Composite Types 578
33.2.3. Domains 578
33.2.4. Pseudo-Types 578
33.2.5. Polymorphic Types 578
33.3. User-Defined Functions 579
33.4. Query Language (SQL) Functions 579
33.4.1. SQL Functions on Base Types 580
33.4.2. SQL Functions on Composite Types 581

33.4.3. Functions with Output Parameters 585
33.4.4. SQL Functions as Table Sources 586
33.4.5. SQL Functions Returning Sets 586
33.4.6. Polymorphic SQL Functions 587
33.5. Function Overloading 589
33.6. Function Volatility Categories 589
33.7. Procedural Language Functions 591
33.8. Internal Functions 591
33.9. C-Language Functions 591
33.9.1. Dynamic Loading 592
33.9.2. Base Types in C-Language Functions 593
33.9.3. Version 0 Calling Conventions 596
33.9.4. Version 1 Calling Conventions 598
33.9.5. Writing Code 601
33.9.6. Compiling and Linking Dynamically-Loaded Functions 601
33.9.7. Extension Building Infrastructure 604
33.9.8. Composite-Type Arguments 605
33.9.9. Returning Rows (Composite Types) 607
33.9.10. Returning Sets 609
33.9.11. Polymorphic Arguments and Return Types 614
33.9.12. Shared Memory and LWLocks 615
33.10. User-Defined Aggregates 616
33.11. User-Defined Types 618
33.12. User-Defined Operators 621
33.13. Operator Optimization Information 622
33.13.1. COMMUTATOR 622
33.13.2. NEGATOR 623
33.13.3. RESTRICT 624
33.13.4. JOIN 625
33.13.5. HASHES 625
33.13.6. MERGES (SORT1, SORT2, LTCMP, GTCMP) 626
33.14. Interfacing Extensions To Indexes 627
33.14.1. Index Methods and Operator Classes 627
33.14.2. Index Method Strategies 628
33.14.3. Index Method Support Routines 629
33.14.4. An Example 631
33.14.5. Cross-Data-Type Operator Classes 633
33.14.6. System Dependencies on Operator Classes 634
33.14.7. Special Features of Operator Classes 635
34. Triggers 637
34.1. Overview of Trigger Behavior 637
34.2. Visibility of Data Changes 638
34.3. Writing Trigger Functions in C 639
34.4. A Complete Example 641
35. The Rule System 646
35.1. The Query Tree 646
35.2. Views and the Rule System 648
35.2.1. How SELECT Rules Work 648

35.2.2. View Rules in Non-SELECT Statements 653
35.2.3. The Power of Views in PostgreSQL 654
35.2.4. Updating a View 654
35.3. Rules on INSERT, UPDATE, and DELETE 655
35.3.1. How Update Rules Work 655
35.3.1.1. A First Rule Step by Step 656
35.3.2. Cooperation with Views 659
35.4. Rules and Privileges 666
35.5. Rules and Command Status 667
35.6. Rules versus Triggers 667
36. Procedural Languages 671
36.1. Installing Procedural Languages 671
37. PL/pgSQL - SQL Procedural Language 673
37.1. Overview 673
37.1.1. Advantages of Using PL/pgSQL 674
37.1.2. Supported Argument and Result Data Types 674
37.2. Tips for Developing in PL/pgSQL 675
37.2.1. Handling of Quotation Marks 675
37.3. Structure of PL/pgSQL 677
37.4. Declarations 678
37.4.1. Aliases for Function Parameters 679
37.4.2. Copying Types 681
37.4.3. Row Types 682
37.4.4. Record Types 682
37.4.5. RENAME 683
37.5. Expressions 683
37.6. Basic Statements 684
37.6.1. Assignment 684
37.6.2. Executing a Query With No Result 685
37.6.3. Executing a Query with a Single-Row Result 686
37.6.4. Doing Nothing At All 687
37.6.5. Executing Dynamic Commands 688
37.6.6. Obtaining the Result Status 689
37.7. Control Structures 690
37.7.1. Returning From a Function 690
37.7.1.1. RETURN 690
37.7.1.2. RETURN NEXT 690
37.7.2. Conditionals 691
37.7.2.1. IF-THEN 691
37.7.2.2. IF-THEN-ELSE 692
37.7.2.3. IF-THEN-ELSE IF 692
37.7.2.4. IF-THEN-ELSIF-ELSE 693
37.7.2.5. IF-THEN-ELSEIF-ELSE 693
37.7.3. Simple Loops 693
37.7.3.1. LOOP 694
37.7.3.2. EXIT 694
37.7.3.3. CONTINUE 695
37.7.3.4. WHILE 695

37.7.3.5. FOR (integer variant) 695
37.7.4. Looping Through Query Results 696
37.7.5. Trapping Errors 697
37.8. Cursors 699
37.8.1. Declaring Cursor Variables 699
37.8.2. Opening Cursors 700
37.8.2.1. OPEN FOR *query* 700
37.8.2.2. OPEN FOR EXECUTE 700
37.8.2.3. Opening a Bound Cursor 701
37.8.3. Using Cursors 701
37.8.3.1. FETCH 701
37.8.3.2. CLOSE 702
37.8.3.3. Returning Cursors 702
37.9. Errors and Messages 703
37.10. Trigger Procedures 704
37.11. Porting from Oracle PL/SQL 710
37.11.1. Porting Examples 710
37.11.2. Other Things to Watch For 716
37.11.2.1. Implicit Rollback after Exceptions 716
37.11.2.2. EXECUTE 717
37.11.2.3. Optimizing PL/pgSQL Functions 717
37.11.3. Appendix 717
38. PL/Tcl - Tcl Procedural Language 720
38.1. Overview 720
38.2. PL/Tcl Functions and Arguments 720
38.3. Data Values in PL/Tcl 722
38.4. Global Data in PL/Tcl 722
38.5. Database Access from PL/Tcl 722
38.6. Trigger Procedures in PL/Tcl 724
38.7. Modules and the unknown command 726
38.8. Tcl Procedure Names 726
39. PL/Perl - Perl Procedural Language 728
39.1. PL/Perl Functions and Arguments 728
39.2. Database Access from PL/Perl 731
39.3. Data Values in PL/Perl 734
39.4. Global Values in PL/Perl 734
39.5. Trusted and Untrusted PL/Perl 735
39.6. PL/Perl Triggers 736
39.7. Limitations and Missing Features 737
40. PL/Python - Python Procedural Language 739
40.1. PL/Python Functions 739
40.2. Trigger Functions 743
40.3. Database Access 743
41. Server Programming Interface 745
41.1. Interface Functions 745
SPI_connect 745
SPI_finish 747
SPI_push 748

SPI_pop 749
SPI_execute 750
SPI_exec 753
SPI_prepare 754
SPI_getargcount 756
SPI_getargtypeid 757
SPI_is_cursor_plan 758
SPI_execute_plan 759
SPI_execp 761
SPI_cursor_open 762
SPI_cursor_find 764
SPI_cursor_fetch 765
SPI_cursor_move 766
SPI_cursor_close 767
SPI_saveplan 768
41.2. Interface Support Functions 769
SPI_fname 769
SPI_fnumber 770
SPI_getvalue 771
SPI_getbinval 772
SPI_gettype 773
SPI_gettypeid 774
SPI_getrelname 775
SPI_getnspname 776
41.3. Memory Management 777
SPI_palloc 777
SPI_repalloc 779
SPI_pfree 780
SPI_copytuple 781
SPI_returntuple 782
SPI_modifytuple 783
SPI_freetuple 785
SPI_freetuptable 786
SPI_freeplan 787
41.4. Visibility of Data Changes 788
41.5. Examples 788
VI. Reference 792
I. SQL Commands 794
ABORT 795
ALTER AGGREGATE 797
ALTER CONVERSION 799
ALTER DATABASE 801
ALTER DOMAIN 803
ALTER FUNCTION 806
ALTER GROUP 809
ALTER INDEX 811
ALTER LANGUAGE 814

ALTER OPERATOR 815
ALTER OPERATOR CLASS 817
ALTER ROLE 818
ALTER SCHEMA 821
ALTER SEQUENCE 822
ALTER TABLE 825
ALTER TABLESPACE 834
ALTER TRIGGER 836
ALTER TYPE 838
ALTER USER 840
ANALYZE 841
BEGIN 843
CHECKPOINT 845
CLOSE 846
CLUSTER 848
COMMENT 851
COMMIT 854
COMMIT PREPARED 856
COPY 857
CREATE AGGREGATE 866
CREATE CAST 870
CREATE CONSTRAINT TRIGGER 874
CREATE CONVERSION 876
CREATE DATABASE 878
CREATE DOMAIN 881
CREATE FUNCTION 884
CREATE GROUP 890
CREATE INDEX 891
CREATE LANGUAGE 896
CREATE OPERATOR 899
CREATE OPERATOR CLASS 903
CREATE ROLE 906
CREATE RULE 911
CREATE SCHEMA 914
CREATE SEQUENCE 917
CREATE TABLE 921
CREATE TABLE AS 933
CREATE TABLESPACE 936
CREATE TRIGGER 938
CREATE TYPE 941
CREATE USER 947
CREATE VIEW 948
DEALLOCATE 951
DECLARE 952
DELETE 955
DROP AGGREGATE 958
DROP CAST 960
DROP CONVERSION 962

DROP DATABASE 964
DROP DOMAIN 965
DROP FUNCTION 967
DROP GROUP 969
DROP INDEX 970
DROP LANGUAGE 972
DROP OPERATOR 974
DROP OPERATOR CLASS 976
DROP OWNED 978
DROP ROLE 980
DROP RULE 982
DROP SCHEMA 984
DROP SEQUENCE 986
DROP TABLE 988
DROP TABLESPACE 990
DROP TRIGGER 992
DROP TYPE 994
DROP USER 996
DROP VIEW 997
END 999
EXECUTE 1001
EXPLAIN 1003
FETCH 1006
GRANT 1010
INSERT 1016
LISTEN 1020
LOAD 1022
LOCK 1023
MOVE 1026
NOTIFY 1028
PREPARE 1030
PREPARE TRANSACTION 1033
REASSIGN OWNED 1035
REINDEX 1037
RELEASE SAVEPOINT 1040
RESET 1042
REVOKE 1044
ROLLBACK 1048
ROLLBACK PREPARED 1050
ROLLBACK TO SAVEPOINT 1051
SAVEPOINT 1053
SELECT 1055
SELECT INTO 1068
SET 1070
SET CONSTRAINTS 1073
SET ROLE 1074
SET SESSION AUTHORIZATION 1076
SET TRANSACTION 1078

SHOW 1080
START TRANSACTION 1083
TRUNCATE 1084
UNLISTEN 1086
UPDATE 1088
VACUUM 1092
VALUES 1095
II. PostgreSQL Client Applications 1098
clusterdb 1099
createdb 1102
createlang 1105
createuser 1108
dropdb 1112
droplang 1115
dropuser 1118
ecpg 1121
pg_config 1123
pg_dump 1126
pg_dumpall 1135
pg_restore 1139
psql 1146
reindexdb 1172
vacuumdb 1175
III. PostgreSQL Server Applications 1178
initdb 1179
ipcclean 1182
pg_controldata 1183
pg_ctl 1184
pg_resetxlog 1189
postgres 1191
postmaster 1199
VII. Internals 1200
42. Overview of PostgreSQL Internals 1202
42.1. The Path of a Query 1202
42.2. How Connections are Established 1202
42.3. The Parser Stage 1203
42.3.1. Parser 1203
42.3.2. Transformation Process 1204
42.4. The PostgreSQL Rule System 1204
42.5. Planner/Optimizer 1205
42.5.1. Generating Possible Plans 1205
42.6. Executor 1206
43. System Catalogs 1208
43.1. Overview 1208
43.2. `pg_aggregate` 1209
43.3. `pg_am` 1210
43.4. `pg_amop` 1211

43.5. pg_amproc 1212
43.6. pg_attrdef 1212
43.7. pg_attribute 1213
43.8. pg_authid 1216
43.9. pg_auth_members 1217
43.10. pg_autovacuum 1218
43.11. pg_cast 1219
43.12. pg_class 1220
43.13. pg_constraint 1224
43.14. pg_conversion 1225
43.15. pg_database 1225
43.16. pg_depend 1227
43.17. pg_description 1229
43.18. pg_index 1229
43.19. pg_inherits 1231
43.20. pg_language 1232
43.21. pg_largeobject 1233
43.22. pg_listener 1234
43.23. pg_namespace 1234
43.24. pg_opclass 1235
43.25. pg_operator 1236
43.26. pg_pltemplate 1237
43.27. pg_proc 1238
43.28. pg_rewrite 1241
43.29. pg_shdepend 1242
43.30. pg_shdescription 1243
43.31. pg_statistic 1244
43.32. pg_tablespace 1246
43.33. pg_trigger 1246
43.34. pg_type 1247
43.35. System Views 1253
43.36. pg_cursors 1254
43.37. pg_group 1255
43.38. pg_indexes 1256
43.39. pg_locks 1256
43.40. pg_prepared_statements 1259
43.41. pg_prepared_xacts 1260
43.42. pg_roles 1261
43.43. pg_rules 1262
43.44. pg_settings 1262
43.45. pg_shadow 1263
43.46. pg_stats 1264
43.47. pg_tables 1266
43.48. pg_timezone_abbrevs 1267
43.49. pg_timezone_names 1267
43.50. pg_user 1268
43.51. pg_views 1268
44. Frontend/Backend Protocol 1270

44.1. Overview 1270
44.1.1. Messaging Overview 1270
44.1.2. Extended Query Overview 1271
44.1.3. Formats and Format Codes 1271
44.2. Message Flow 1272
44.2.1. Start-Up 1272
44.2.2. Simple Query 1274
44.2.3. Extended Query 1275
44.2.4. Function Call 1278
44.2.5. COPY Operations 1279
44.2.6. Asynchronous Operations 1280
44.2.7. Cancelling Requests in Progress 1280
44.2.8. Termination 1281
44.2.9. SSL Session Encryption 1282
44.3. Message Data Types 1282
44.4. Message Formats 1283
44.5. Error and Notice Message Fields 1298
44.6. Summary of Changes since Protocol 2.0 1299
45. PostgreSQL Coding Conventions 1301
45.1. Formatting 1301
45.2. Reporting Errors Within the Server 1301
45.3. Error Message Style Guide 1303
45.3.1. What goes where 1303
45.3.2. Formatting 1304
45.3.3. Quotation marks 1304
45.3.4. Use of quotes 1304
45.3.5. Grammar and punctuation 1304
45.3.6. Upper case vs. lower case 1305
45.3.7. Avoid passive voice 1305
45.3.8. Present vs past tense 1305
45.3.9. Type of the object 1305
45.3.10. Brackets 1306
45.3.11. Assembling error messages 1306
45.3.12. Reasons for errors 1306
45.3.13. Function names 1306
45.3.14. Tricky words to avoid 1307
45.3.15. Proper spelling 1307
45.3.16. Localization 1307
46. Native Language Support 1308
46.1. For the Translator 1308
46.1.1. Requirements 1308
46.1.2. Concepts 1308
46.1.3. Creating and maintaining message catalogs 1309
46.1.4. Editing the PO files 1310
46.2. For the Programmer 1311
46.2.1. Mechanics 1311
46.2.2. Message-writing guidelines 1312
47. Writing A Procedural Language Handler 1314

48. Genetic Query Optimizer 1316
48.1. Query Handling as a Complex Optimization Problem 1316
48.2. Genetic Algorithms 1316
48.3. Genetic Query Optimization (GEQO) in PostgreSQL 1317
48.3.1. Future Implementation Tasks for PostgreSQL GEQO 1318
48.4. Further Reading 1318
49. Index Access Method Interface Definition 1319
49.1. Catalog Entries for Indexes 1319
49.2. Index Access Method Functions 1320
49.3. Index Scanning 1323
49.4. Index Locking Considerations 1324
49.5. Index Uniqueness Checks 1325
49.6. Index Cost Estimation Functions 1326
50. GiST Indexes 1329
50.1. Introduction 1329
50.2. Extensibility 1329
50.3. Implementation 1329
50.4. Examples 1330
50.5. Crash Recovery 1331
51. GIN Indexes 1332
51.1. Introduction 1332
51.2. Extensibility 1332
51.3. Implementation 1333
51.4. GIN tips and tricks 1333
51.5. Limitations 1333
51.6. Examples 1334
52. Database Physical Storage 1335
52.1. Database File Layout 1335
52.2. TOAST 1336
52.3. Database Page Layout 1338
53. BKI Backend Interface 1341
53.1. BKI File Format 1341
53.2. BKI Commands 1341
53.3. Structure of the Bootstrap BKI File 1342
53.4. Example 1343
54. How the Planner Uses Statistics 1344
54.1. Row Estimation Examples 1344
VIII. Appendixes 1349
A. PostgreSQL Error Codes 1350
B. Date/Time Support 1359
B.1. Date/Time Input Interpretation 1359
B.2. Date/Time Key Words 1360
B.3. Date/Time Configuration Files 1361
B.4. History of Units 1362
C. SQL Key Words 1364
D. SQL Conformance 1385
D.1. Supported Features 1386

D.2. Unsupported Features 1397
E. Release Notes 1406
E.1. Release 8.2.6 1406
E.1.1. Migration to Version 8.2.6 1406
E.1.2. Changes 1406
E.2. Release 8.2.5 1408
E.2.1. Migration to Version 8.2.5 1408
E.2.2. Changes 1408
E.3. Release 8.2.4 1409
E.3.1. Migration to Version 8.2.4 1409
E.3.2. Changes 1409
E.4. Release 8.2.3 1410
E.4.1. Migration to Version 8.2.3 1410
E.4.2. Changes 1410
E.5. Release 8.2.2 1410
E.5.1. Migration to Version 8.2.2 1410
E.5.2. Changes 1411
E.6. Release 8.2.1 1411
E.6.1. Migration to Version 8.2.1 1412
E.6.2. Changes 1412
E.7. Release 8.2 1412
E.7.1. Overview 1412
E.7.2. Migration to Version 8.2 1413
E.7.3. Changes 1415
E.7.3.1. Performance Improvements 1415
E.7.3.2. Server Changes 1416
E.7.3.3. Query Changes 1418
E.7.3.4. Object Manipulation Changes 1419
E.7.3.5. Utility Command Changes 1420
E.7.3.6. Date/Time Changes 1421
E.7.3.7. Other Data Type and Function Changes 1421
E.7.3.8. PL/PgSQL Server-Side Language Changes 1422
E.7.3.9. PL/Perl Server-Side Language Changes 1422
E.7.3.10. PL/Python Server-Side Language Changes 1423
E.7.3.11. psql Changes 1423
E.7.3.12. pg_dump Changes 1423
E.7.3.13. libpq Changes 1424
E.7.3.14. ecpg Changes 1424
E.7.3.15. Windows Port 1424
E.7.3.16. Source Code Changes 1425
E.7.3.17. Contrib Changes 1426
E.8. Release 8.1.11 1427
E.8.1. Migration to Version 8.1.11 1427
E.8.2. Changes 1427
E.9. Release 8.1.10 1429
E.9.1. Migration to Version 8.1.10 1429
E.9.2. Changes 1429
E.10. Release 8.1.9 1430

E.10.1. Migration to Version 8.1.9 1430
E.10.2. Changes 1430
E.11. Release 8.1.8 1430
E.11.1. Migration to Version 8.1.8 1431
E.11.2. Changes 1431
E.12. Release 8.1.7 1431
E.12.1. Migration to Version 8.1.7 1431
E.12.2. Changes 1431
E.13. Release 8.1.6 1432
E.13.1. Migration to Version 8.1.6 1432
E.13.2. Changes 1432
E.14. Release 8.1.5 1433
E.14.1. Migration to Version 8.1.5 1433
E.14.2. Changes 1433
E.15. Release 8.1.4 1434
E.15.1. Migration to Version 8.1.4 1434
E.15.2. Changes 1434
E.16. Release 8.1.3 1436
E.16.1. Migration to Version 8.1.3 1436
E.16.2. Changes 1436
E.17. Release 8.1.2 1437
E.17.1. Migration to Version 8.1.2 1437
E.17.2. Changes 1437
E.18. Release 8.1.1 1438
E.18.1. Migration to Version 8.1.1 1439
E.18.2. Changes 1439
E.19. Release 8.1 1440
E.19.1. Overview 1440
E.19.2. Migration to Version 8.1 1441
E.19.3. Additional Changes 1444
E.19.3.1. Performance Improvements 1444
E.19.3.2. Server Changes 1445
E.19.3.3. Query Changes 1446
E.19.3.4. Object Manipulation Changes 1446
E.19.3.5. Utility Command Changes 1447
E.19.3.6. Data Type and Function Changes 1447
E.19.3.7. Encoding and Locale Changes 1449
E.19.3.8. General Server-Side Language Changes 1450
E.19.3.9. PL/PgSQL Server-Side Language Changes 1450
E.19.3.10. PL/Perl Server-Side Language Changes 1451
E.19.3.11. psql Changes 1451
E.19.3.12. pg_dump Changes 1452
E.19.3.13. libpq Changes 1452
E.19.3.14. Source Code Changes 1453
E.19.3.15. Contrib Changes 1453
E.20. Release 8.0.15 1454
E.20.1. Migration to Version 8.0.15 1454
E.20.2. Changes 1455

E.21. Release 8.0.14 1456
E.21.1. Migration to Version 8.0.14 1456
E.21.2. Changes 1456
E.22. Release 8.0.13 1457
E.22.1. Migration to Version 8.0.13 1457
E.22.2. Changes 1457
E.23. Release 8.0.12 1457
E.23.1. Migration to Version 8.0.12 1458
E.23.2. Changes 1458
E.24. Release 8.0.11 1458
E.24.1. Migration to Version 8.0.11 1458
E.24.2. Changes 1458
E.25. Release 8.0.10 1458
E.25.1. Migration to Version 8.0.10 1459
E.25.2. Changes 1459
E.26. Release 8.0.9 1459
E.26.1. Migration to Version 8.0.9 1459
E.26.2. Changes 1460
E.27. Release 8.0.8 1460
E.27.1. Migration to Version 8.0.8 1460
E.27.2. Changes 1460
E.28. Release 8.0.7 1461
E.28.1. Migration to Version 8.0.7 1462
E.28.2. Changes 1462
E.29. Release 8.0.6 1463
E.29.1. Migration to Version 8.0.6 1463
E.29.2. Changes 1463
E.30. Release 8.0.5 1464
E.30.1. Migration to Version 8.0.5 1464
E.30.2. Changes 1464
E.31. Release 8.0.4 1465
E.31.1. Migration to Version 8.0.4 1465
E.31.2. Changes 1465
E.32. Release 8.0.3 1466
E.32.1. Migration to Version 8.0.3 1466
E.32.2. Changes 1467
E.33. Release 8.0.2 1468
E.33.1. Migration to Version 8.0.2 1468
E.33.2. Changes 1468
E.34. Release 8.0.1 1470
E.34.1. Migration to Version 8.0.1 1470
E.34.2. Changes 1470
E.35. Release 8.0 1471
E.35.1. Overview 1471
E.35.2. Migration to Version 8.0 1472
E.35.3. Deprecated Features 1474
E.35.4. Changes 1474
E.35.4.1. Performance Improvements 1474

E.35.4.2. Server Changes 1476
E.35.4.3. Query Changes 1477
E.35.4.4. Object Manipulation Changes 1478
E.35.4.5. Utility Command Changes 1479
E.35.4.6. Data Type and Function Changes 1480
E.35.4.7. Server-Side Language Changes 1482
E.35.4.8. psql Changes 1483
E.35.4.9. pg_dump Changes 1483
E.35.4.10. libpq Changes 1484
E.35.4.11. Source Code Changes 1484
E.35.4.12. Contrib Changes 1485
E.36. Release 7.4.19 1486
E.36.1. Migration to Version 7.4.19 1486
E.36.2. Changes 1486
E.37. Release 7.4.18 1487
E.37.1. Migration to Version 7.4.18 1487
E.37.2. Changes 1487
E.38. Release 7.4.17 1488
E.38.1. Migration to Version 7.4.17 1488
E.38.2. Changes 1488
E.39. Release 7.4.16 1488
E.39.1. Migration to Version 7.4.16 1489
E.39.2. Changes 1489
E.40. Release 7.4.15 1489
E.40.1. Migration to Version 7.4.15 1489
E.40.2. Changes 1489
E.41. Release 7.4.14 1490
E.41.1. Migration to Version 7.4.14 1490
E.41.2. Changes 1490
E.42. Release 7.4.13 1490
E.42.1. Migration to Version 7.4.13 1491
E.42.2. Changes 1491
E.43. Release 7.4.12 1492
E.43.1. Migration to Version 7.4.12 1492
E.43.2. Changes 1492
E.44. Release 7.4.11 1492
E.44.1. Migration to Version 7.4.11 1493
E.44.2. Changes 1493
E.45. Release 7.4.10 1493
E.45.1. Migration to Version 7.4.10 1494
E.45.2. Changes 1494
E.46. Release 7.4.9 1494
E.46.1. Migration to Version 7.4.9 1494
E.46.2. Changes 1494
E.47. Release 7.4.8 1495
E.47.1. Migration to Version 7.4.8 1495
E.47.2. Changes 1496
E.48. Release 7.4.7 1498

E.48.1. Migration to Version 7.4.7 1498
E.48.2. Changes 1498
E.49. Release 7.4.6 1498
E.49.1. Migration to Version 7.4.6 1499
E.49.2. Changes 1499
E.50. Release 7.4.5 1499
E.50.1. Migration to Version 7.4.5 1500
E.50.2. Changes 1500
E.51. Release 7.4.4 1500
E.51.1. Migration to Version 7.4.4 1500
E.51.2. Changes 1500
E.52. Release 7.4.3 1501
E.52.1. Migration to Version 7.4.3 1501
E.52.2. Changes 1501
E.53. Release 7.4.2 1502
E.53.1. Migration to Version 7.4.2 1502
E.53.2. Changes 1503
E.54. Release 7.4.1 1504
E.54.1. Migration to Version 7.4.1 1504
E.54.2. Changes 1504
E.55. Release 7.4 1505
E.55.1. Overview 1506
E.55.2. Migration to Version 7.4 1507
E.55.3. Changes 1508
E.55.3.1. Server Operation Changes 1508
E.55.3.2. Performance Improvements 1509
E.55.3.3. Server Configuration Changes 1511
E.55.3.4. Query Changes 1512
E.55.3.5. Object Manipulation Changes 1513
E.55.3.6. Utility Command Changes 1513
E.55.3.7. Data Type and Function Changes 1515
E.55.3.8. Server-Side Language Changes 1516
E.55.3.9. psql Changes 1517
E.55.3.10. pg_dump Changes 1518
E.55.3.11. libpq Changes 1518
E.55.3.12. JDBC Changes 1519
E.55.3.13. Miscellaneous Interface Changes 1519
E.55.3.14. Source Code Changes 1519
E.55.3.15. Contrib Changes 1520
E.56. Release 7.3.21 1521
E.56.1. Migration to Version 7.3.21 1521
E.56.2. Changes 1521
E.57. Release 7.3.20 1522
E.57.1. Migration to Version 7.3.20 1522
E.57.2. Changes 1522
E.58. Release 7.3.19 1522
E.58.1. Migration to Version 7.3.19 1523
E.58.2. Changes 1523

E.59. Release 7.3.18 1523
E.59.1. Migration to Version 7.3.18 1523
E.59.2. Changes 1523
E.60. Release 7.3.17 1524
E.60.1. Migration to Version 7.3.17 1524
E.60.2. Changes 1524
E.61. Release 7.3.16 1524
E.61.1. Migration to Version 7.3.16 1524
E.61.2. Changes 1525
E.62. Release 7.3.15 1525
E.62.1. Migration to Version 7.3.15 1525
E.62.2. Changes 1525
E.63. Release 7.3.14 1526
E.63.1. Migration to Version 7.3.14 1526
E.63.2. Changes 1526
E.64. Release 7.3.13 1527
E.64.1. Migration to Version 7.3.13 1527
E.64.2. Changes 1527
E.65. Release 7.3.12 1528
E.65.1. Migration to Version 7.3.12 1528
E.65.2. Changes 1528
E.66. Release 7.3.11 1528
E.66.1. Migration to Version 7.3.11 1528
E.66.2. Changes 1529
E.67. Release 7.3.10 1529
E.67.1. Migration to Version 7.3.10 1529
E.67.2. Changes 1530
E.68. Release 7.3.9 1531
E.68.1. Migration to Version 7.3.9 1531
E.68.2. Changes 1531
E.69. Release 7.3.8 1532
E.69.1. Migration to Version 7.3.8 1532
E.69.2. Changes 1532
E.70. Release 7.3.7 1532
E.70.1. Migration to Version 7.3.7 1533
E.70.2. Changes 1533
E.71. Release 7.3.6 1533
E.71.1. Migration to Version 7.3.6 1533
E.71.2. Changes 1533
E.72. Release 7.3.5 1534
E.72.1. Migration to Version 7.3.5 1534
E.72.2. Changes 1534
E.73. Release 7.3.4 1535
E.73.1. Migration to Version 7.3.4 1535
E.73.2. Changes 1535
E.74. Release 7.3.3 1535
E.74.1. Migration to Version 7.3.3 1536
E.74.2. Changes 1536

E.75. Release 7.3.2 1538
E.75.1. Migration to Version 7.3.2 1538
E.75.2. Changes 1538
E.76. Release 7.3.1 1539
E.76.1. Migration to Version 7.3.1 1539
E.76.2. Changes 1539
E.77. Release 7.3 1540
E.77.1. Overview 1540
E.77.2. Migration to Version 7.3 1541
E.77.3. Changes 1542
E.77.3.1. Server Operation 1542
E.77.3.2. Performance 1542
E.77.3.3. Privileges 1543
E.77.3.4. Server Configuration 1543
E.77.3.5. Queries 1544
E.77.3.6. Object Manipulation 1544
E.77.3.7. Utility Commands 1545
E.77.3.8. Data Types and Functions 1546
E.77.3.9. Internationalization 1548
E.77.3.10. Server-side Languages 1548
E.77.3.11. psql 1548
E.77.3.12. libpq 1549
E.77.3.13. JDBC 1549
E.77.3.14. Miscellaneous Interfaces 1549
E.77.3.15. Source Code 1550
E.77.3.16. Contrib 1551
E.78. Release 7.2.8 1552
E.78.1. Migration to Version 7.2.8 1552
E.78.2. Changes 1552
E.79. Release 7.2.7 1553
E.79.1. Migration to Version 7.2.7 1553
E.79.2. Changes 1553
E.80. Release 7.2.6 1553
E.80.1. Migration to Version 7.2.6 1554
E.80.2. Changes 1554
E.81. Release 7.2.5 1554
E.81.1. Migration to Version 7.2.5 1554
E.81.2. Changes 1554
E.82. Release 7.2.4 1555
E.82.1. Migration to Version 7.2.4 1555
E.82.2. Changes 1555
E.83. Release 7.2.3 1555
E.83.1. Migration to Version 7.2.3 1556
E.83.2. Changes 1556
E.84. Release 7.2.2 1556
E.84.1. Migration to Version 7.2.2 1556
E.84.2. Changes 1556
E.85. Release 7.2.1 1557

E.85.1. Migration to Version 7.2.1 1557
E.85.2. Changes 1557
E.86. Release 7.2 1558
E.86.1. Overview 1558
E.86.2. Migration to Version 7.2 1559
E.86.3. Changes 1559
E.86.3.1. Server Operation 1559
E.86.3.2. Performance 1560
E.86.3.3. Privileges 1560
E.86.3.4. Client Authentication 1560
E.86.3.5. Server Configuration 1561
E.86.3.6. Queries 1561
E.86.3.7. Schema Manipulation 1561
E.86.3.8. Utility Commands 1562
E.86.3.9. Data Types and Functions 1562
E.86.3.10. Internationalization 1564
E.86.3.11. PL/pgSQL 1564
E.86.3.12. PL/Perl 1564
E.86.3.13. PL/Tcl 1564
E.86.3.14. PL/Python 1564
E.86.3.15. psql 1565
E.86.3.16. libpq 1565
E.86.3.17. JDBC 1565
E.86.3.18. ODBC 1566
E.86.3.19. ECPG 1566
E.86.3.20. Misc. Interfaces 1567
E.86.3.21. Build and Install 1567
E.86.3.22. Source Code 1567
E.86.3.23. Contrib 1568
E.87. Release 7.1.3 1568
E.87.1. Migration to Version 7.1.3 1568
E.87.2. Changes 1568
E.88. Release 7.1.2 1569
E.88.1. Migration to Version 7.1.2 1569
E.88.2. Changes 1569
E.89. Release 7.1.1 1569
E.89.1. Migration to Version 7.1.1 1570
E.89.2. Changes 1570
E.90. Release 7.1 1570
E.90.1. Migration to Version 7.1 1571
E.90.2. Changes 1571
E.91. Release 7.0.3 1575
E.91.1. Migration to Version 7.0.3 1575
E.91.2. Changes 1575
E.92. Release 7.0.2 1576
E.92.1. Migration to Version 7.0.2 1576
E.92.2. Changes 1576
E.93. Release 7.0.1 1576

E.93.1. Migration to Version 7.0.1 1577
E.93.2. Changes 1577
E.94. Release 7.0 1577
E.94.1. Migration to Version 7.0 1578
E.94.2. Changes 1578
E.95. Release 6.5.3 1584
E.95.1. Migration to Version 6.5.3 1585
E.95.2. Changes 1585
E.96. Release 6.5.2 1585
E.96.1. Migration to Version 6.5.2 1585
E.96.2. Changes 1585
E.97. Release 6.5.1 1586
E.97.1. Migration to Version 6.5.1 1586
E.97.2. Changes 1586
E.98. Release 6.5 1587
E.98.1. Migration to Version 6.5 1588
E.98.1.1. Multiversion Concurrency Control 1588
E.98.2. Changes 1589
E.99. Release 6.4.2 1592
E.99.1. Migration to Version 6.4.2 1592
E.99.2. Changes 1592
E.100. Release 6.4.1 1592
E.100.1. Migration to Version 6.4.1 1593
E.100.2. Changes 1593
E.101. Release 6.4 1593
E.101.1. Migration to Version 6.4 1594
E.101.2. Changes 1594
E.102. Release 6.3.2 1598
E.102.1. Changes 1599
E.103. Release 6.3.1 1599
E.103.1. Changes 1600
E.104. Release 6.3 1600
E.104.1. Migration to Version 6.3 1602
E.104.2. Changes 1602
E.105. Release 6.2.1 1605
E.105.1. Migration from version 6.2 to version 6.2.1 1605
E.105.2. Changes 1606
E.106. Release 6.2 1606
E.106.1. Migration from version 6.1 to version 6.2 1606
E.106.2. Migration from version 1.*x* to version 6.2 1606
E.106.3. Changes 1606
E.107. Release 6.1.1 1609
E.107.1. Migration from version 6.1 to version 6.1.1 1609
E.107.2. Changes 1609
E.108. Release 6.1 1609
E.108.1. Migration to Version 6.1 1610
E.108.2. Changes 1610
E.109. Release 6.0 1612

E.109.1. Migration from version 1.09 to version 6.0 1612
E.109.2. Migration from pre-1.09 to version 6.0 1612
E.109.3. Changes 1613
E.110. Release 1.09 1615
E.111. Release 1.02 1615
E.111.1. Migration from version 1.02 to version 1.02.1 1615
E.111.2. Dump/Reload Procedure 1616
E.111.3. Changes 1616
E.112. Release 1.01 1617
E.112.1. Migration from version 1.0 to version 1.01 1617
E.112.2. Changes 1618
E.113. Release 1.0 1619
E.113.1. Changes 1619
E.114. Postgres95 Release 0.03 1620
E.114.1. Changes 1620
E.115. Postgres95 Release 0.02 1623
E.115.1. Changes 1623
E.116. Postgres95 Release 0.01 1624
F. The CVS Repository 1625
F.1. Getting The Source Via Anonymous CVS 1625
F.2. CVS Tree Organization 1626
F.3. Getting The Source Via CVSup 1627
F.3.1. Preparing A CVSup Client System 1627
F.3.2. Running a CVSup Client 1628
G. Documentation 1631
G.1. DocBook 1631
G.2. Tool Sets 1631
G.2.1. Linux RPM Installation 1632
G.2.2. FreeBSD Installation 1632
G.2.3. Debian Packages 1633
G.2.4. Manual Installation from Source 1633
G.2.4.1. Installing OpenJade 1633
G.2.4.2. Installing the DocBook DTD Kit 1634
G.2.4.3. Installing the DocBook DSSSL Style Sheets 1635
G.2.4.4. Installing JadeTeX 1635
G.2.5. Detection by `configure` 1635
G.3. Building The Documentation 1636
G.3.1. HTML 1636
G.3.2. Manpages 1636
G.3.3. Print Output via JadeTex 1637
G.3.4. Print Output via RTF 1637
G.3.5. Plain Text Files 1639
G.3.6. Syntax Check 1639
G.4. Documentation Authoring 1639
G.4.1. Emacs/PSGML 1639
G.4.2. Other Emacs modes 1640
G.5. Style Guide 1641
G.5.1. Reference Pages 1641

H. External Projects 1643
H.1. Client Interfaces 1643
H.2. Procedural Languages 1644
H.3. Extensions 1645
Bibliography 1646
Index 1648

List of Tables

4-1. Operator Precedence (decreasing) 31
8-1. Data Types 91
8-2. Numeric Types 92
8-3. Monetary Types 96
8-4. Character Types 97
8-5. Special Character Types 98
8-6. Binary Data Types 98
8-7. `bytea` Literal Escaped Octets 99
8-8. `bytea` Output Escaped Octets 100
8-9. Date/Time Types 100
8-10. Date Input 102
8-11. Time Input 103
8-12. Time Zone Input 103
8-13. Special Date/Time Inputs 105
8-14. Date/Time Output Styles 106
8-15. Date Order Conventions 106
8-16. Geometric Types 109
8-17. Network Address Types 111
8-18. `cidr` Type Input Examples 112
8-19. Object Identifier Types 127
8-20. Pseudo-Types 128
9-1. Comparison Operators 132
9-2. Mathematical Operators 134
9-3. Mathematical Functions 134
9-4. Trigonometric Functions 136
9-5. SQL String Functions and Operators 137
9-6. Other String Functions 138
9-7. Built-in Conversions 142
9-8. SQL Binary String Functions and Operators 146
9-9. Other Binary String Functions 147
9-10. Bit String Operators 148
9-11. Regular Expression Match Operators 151
9-12. Regular Expression Atoms 153
9-13. Regular Expression Quantifiers 154
9-14. Regular Expression Constraints 154
9-15. Regular Expression Character-Entry Escapes 156
9-16. Regular Expression Class-Shorthand Escapes 157
9-17. Regular Expression Constraint Escapes 158
9-18. Regular Expression Back References 158
9-19. ARE Embedded-Option Letters 159
9-20. Formatting Functions 162
9-21. Template Patterns for Date/Time Formatting 163
9-22. Template Pattern Modifiers for Date/Time Formatting 165
9-23. Template Patterns for Numeric Formatting 166
9-24. `to_char` Examples 167
9-25. Date/Time Operators 168

9-26. Date/Time Functions 169
9-27. AT TIME ZONE Variants 176
9-28. Geometric Operators 180
9-29. Geometric Functions 181
9-30. Geometric Type Conversion Functions 182
9-31. cidr and inet Operators 184
9-32. cidr and inet Functions 184
9-33. macaddr Functions 185
9-34. Sequence Functions 186
9-35. array Operators 190
9-36. array Functions 191
9-37. General-Purpose Aggregate Functions 192
9-38. Aggregate Functions for Statistics 194
9-39. Series Generating Functions 201
9-40. Session Information Functions 202
9-41. Access Privilege Inquiry Functions 204
9-42. Schema Visibility Inquiry Functions 206
9-43. System Catalog Information Functions 206
9-44. Comment Information Functions 208
9-45. Configuration Settings Functions 209
9-46. Server Signalling Functions 209
9-47. Backup Control Functions 210
9-48. Database Object Size Functions 211
9-49. Generic File Access Functions 212
9-50. Advisory Lock Functions 213
12-1. SQL Transaction Isolation Levels 236
16-1. System V IPC parameters 285
16-2. Configuration parameters affecting PostgreSQL's shared memory usage 289
17-1. Short option key 331
21-1. PostgreSQL Character Sets 359
21-2. Client/Server Character Set Conversions 362
25-1. Standard Statistics Views 396
25-2. Statistics Access Functions 398
25-3. Built-in Trace Points 402
31-1. Valid input formats for PGTYPESdate_from_asc 493
31-2. Valid input formats for PGTYPESdate_fmt_asc 495
31-3. Valid input formats for rdefmtdate 495
31-4. Valid input formats for PGTYPEStimestamp_from_asc 496
32-1. information_schema_catalog_name Columns 527
32-2. administrable_role_authorizations Columns 527
32-3. applicable_roles Columns 527
32-4. attributes Columns 528
32-5. check_constraint_routine_usage Columns 531
32-6. check_constraints Columns 532
32-7. column_domain_usage Columns 532
32-8. column_privileges Columns 533
32-9. column_udt_usage Columns 533
32-10. columns Columns 534

32-11. constraint_column_usage Columns 539
32-12. constraint_table_usage Columns 539
32-13. data_type_privileges Columns 540
32-14. domain_constraints Columns 541
32-15. domain_udt_usage Columns 541
32-16. domains Columns 542
32-17. element_types Columns 545
32-18. enabled_roles Columns 548
32-19. key_column_usage Columns 548
32-20. parameters Columns 549
32-21. referential_constraints Columns 551
32-22. role_column_grants Columns 552
32-23. role_routine_grants Columns 553
32-24. role_table_grants Columns 554
32-25. role_usage_grants Columns 554
32-26. routine_privileges Columns 555
32-27. routines Columns 556
32-28. schemata Columns 562
32-29. sequences Columns 562
32-30. sql_features Columns 563
32-31. sql_implementation_info Columns 564
32-32. sql_languages Columns 565
32-33. sql_packages Columns 565
32-34. sql_parts Columns 566
32-35. sql_sizing Columns 566
32-36. sql_sizing_profiles Columns 567
32-37. table_constraints Columns 567
32-38. table_privileges Columns 568
32-39. tables Columns 569
32-40. triggers Columns 570
32-41. usage_privileges Columns 571
32-42. view_column_usage Columns 572
32-43. view_routine_usage Columns 573
32-44. view_table_usage Columns 573
32-45. views Columns 574
33-1. Equivalent C Types for Built-In SQL Types 595
33-2. B-tree Strategies 628
33-3. Hash Strategies 628
33-4. GiST Two-Dimensional “R-tree” Strategies 629
33-5. GIN Array Strategies 629
33-6. B-tree Support Functions 630
33-7. Hash Support Functions 630
33-8. GiST Support Functions 630
33-9. GIN Support Functions 631
43-1. System Catalogs 1208
43-2. pg_aggregate Columns 1209
43-3. pg_am Columns 1210
43-4. pg_amop Columns 1211

43-5. pg_amproc Columns 1212
43-6. pg_attrdef Columns 1212
43-7. pg_attribute Columns 1213
43-8. pg_authid Columns 1216
43-9. pg_auth_members Columns 1217
43-10. pg_autovacuum Columns 1218
43-11. pg_cast Columns 1219
43-12. pg_class Columns 1220
43-13. pg_constraint Columns 1224
43-14. pg_conversion Columns 1225
43-15. pg_database Columns 1226
43-16. pg_depend Columns 1227
43-17. pg_description Columns 1229
43-18. pg_index Columns 1230
43-19. pg_inherits Columns 1232
43-20. pg_language Columns 1232
43-21. pg_largeobject Columns 1233
43-22. pg_listener Columns 1234
43-23. pg_namespace Columns 1235
43-24. pg_opclass Columns 1235
43-25. pg_operator Columns 1236
43-26. pg_pltemplate Columns 1237
43-27. pg_proc Columns 1238
43-28. pg_rewrite Columns 1241
43-29. pg_shdepend Columns 1242
43-30. pg_shdescription Columns 1243
43-31. pg_statistic Columns 1244
43-32. pg_tablespace Columns 1246
43-33. pg_trigger Columns 1246
43-34. pg_type Columns 1247
43-35. System Views 1254
43-36. pg_cursors Columns 1255
43-37. pg_group Columns 1256
43-38. pg_indexes Columns 1256
43-39. pg_locks Columns 1257
43-40. pg_prepared_statements Columns 1259
43-41. pg_prepared_xacts Columns 1260
43-42. pg_roles Columns 1261
43-43. pg_rules Columns 1262
43-44. pg_settings Columns 1263
43-45. pg_shadow Columns 1264
43-46. pg_stats Columns 1264
43-47. pg_tables Columns 1266
43-48. pg_timezone_abbrevs Columns 1267
43-49. pg_timezone_names Columns 1267
43-50. pg_user Columns 1268
43-51. pg_views Columns 1268
52-1. Contents of PGDATA 1335

52-2. Overall Page Layout 1338
52-3. PageHeaderData Layout 1339
52-4. HeapTupleHeaderData Layout 1340
A-1. PostgreSQL Error Codes 1350
B-1. Month Names 1360
B-2. Day of the Week Names 1360
B-3. Date/Time Field Modifiers 1361
C-1. SQL Key Words 1364
H-1. Externally Maintained Client Interfaces 1643
H-2. Externally Maintained Procedural Languages 1644

Preface

This book is the official documentation of PostgreSQL. It is being written by the PostgreSQL developers and other volunteers in parallel to the development of the PostgreSQL software. It describes all the functionality that the current version of PostgreSQL officially supports.

To make the large amount of information about PostgreSQL manageable, this book has been organized in several parts. Each part is targeted at a different class of users, or at users in different stages of their PostgreSQL experience:

- Part I is an informal introduction for new users.
- Part II documents the SQL query language environment, including data types and functions, as well as user-level performance tuning. Every PostgreSQL user should read this.
- Part III describes the installation and administration of the server. Everyone who runs a PostgreSQL server, be it for private use or for others, should read this part.
- Part IV describes the programming interfaces for PostgreSQL client programs.
- Part V contains information for advanced users about the extensibility capabilities of the server. Topics are, for instance, user-defined data types and functions.
- Part VI contains reference information about SQL commands, client and server programs. This part supports the other parts with structured information sorted by command or program.
- Part VII contains assorted information that may be of use to PostgreSQL developers.

1. What is PostgreSQL?

PostgreSQL is an object-relational database management system (ORDBMS) based on POSTGRES, Version 4.2[1], developed at the University of California at Berkeley Computer Science Department. POSTGRES pioneered many concepts that only became available in some commercial database systems much later.

PostgreSQL is an open-source descendant of this original Berkeley code. It supports a large part of the SQL standard and offers many modern features:

- complex queries
- foreign keys
- triggers
- views
- transactional integrity
- multiversion concurrency control

Also, PostgreSQL can be extended by the user in many ways, for example by adding new

- data types

1. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/postgres.html

- functions
- operators
- aggregate functions
- index methods
- procedural languages

And because of the liberal license, PostgreSQL can be used, modified, and distributed by everyone free of charge for any purpose, be it private, commercial, or academic.

2. A Brief History of PostgreSQL

The object-relational database management system now known as PostgreSQL is derived from the POSTGRES package written at the University of California at Berkeley. With over a decade of development behind it, PostgreSQL is now the most advanced open-source database available anywhere.

2.1. The Berkeley POSTGRES Project

The POSTGRES project, led by Professor Michael Stonebraker, was sponsored by the Defense Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), the National Science Foundation (NSF), and ESL, Inc. The implementation of POSTGRES began in 1986. The initial concepts for the system were presented in *The design of POSTGRES* , and the definition of the initial data model appeared in *The POSTGRES data model* . The design of the rule system at that time was described in *The design of the POSTGRES rules system*. The rationale and architecture of the storage manager were detailed in *The design of the POSTGRES storage system* .

POSTGRES has undergone several major releases since then. The first "demoware" system became operational in 1987 and was shown at the 1988 ACM-SIGMOD Conference. Version 1, described in *The implementation of POSTGRES* , was released to a few external users in June 1989. In response to a critique of the first rule system (*A commentary on the POSTGRES rules system*), the rule system was redesigned (*On Rules, Procedures, Caching and Views in Database Systems*), and Version 2 was released in June 1990 with the new rule system. Version 3 appeared in 1991 and added support for multiple storage managers, an improved query executor, and a rewritten rule system. For the most part, subsequent releases until Postgres95 (see below) focused on portability and reliability.

POSTGRES has been used to implement many different research and production applications. These include: a financial data analysis system, a jet engine performance monitoring package, an asteroid tracking database, a medical information database, and several geographic information systems. POSTGRES has also been used as an educational tool at several universities. Finally, Illustra Information Technologies (later merged into Informix[2], which is now owned by IBM[3]) picked up the code and commercialized it. In late 1992, POSTGRES became the primary data manager for the Sequoia 2000 scientific computing project[4].

The size of the external user community nearly doubled during 1993. It became increasingly obvious that maintenance of the prototype code and support was taking up large amounts of time that should have been

2. http://www.informix.com/
3. http://www.ibm.com/
4. http://meteora.ucsd.edu/s2k/s2k_home.html

devoted to database research. In an effort to reduce this support burden, the Berkeley POSTGRES project officially ended with Version 4.2.

2.2. Postgres95

In 1994, Andrew Yu and Jolly Chen added a SQL language interpreter to POSTGRES. Under a new name, Postgres95 was subsequently released to the web to find its own way in the world as an open-source descendant of the original POSTGRES Berkeley code.

Postgres95 code was completely ANSI C and trimmed in size by 25%. Many internal changes improved performance and maintainability. Postgres95 release 1.0.x ran about 30-50% faster on the Wisconsin Benchmark compared to POSTGRES, Version 4.2. Apart from bug fixes, the following were the major enhancements:

- The query language PostQUEL was replaced with SQL (implemented in the server). Subqueries were not supported until PostgreSQL (see below), but they could be imitated in Postgres95 with user-defined SQL functions. Aggregate functions were re-implemented. Support for the `GROUP BY` query clause was also added.
- A new program (psql) was provided for interactive SQL queries, which used GNU Readline. This largely superseded the old monitor program.
- A new front-end library, `libpgtcl`, supported Tcl-based clients. A sample shell, `pgtclsh`, provided new Tcl commands to interface Tcl programs with the Postgres95 server.
- The large-object interface was overhauled. The inversion large objects were the only mechanism for storing large objects. (The inversion file system was removed.)
- The instance-level rule system was removed. Rules were still available as rewrite rules.
- A short tutorial introducing regular SQL features as well as those of Postgres95 was distributed with the source code
- GNU make (instead of BSD make) was used for the build. Also, Postgres95 could be compiled with an unpatched GCC (data alignment of doubles was fixed).

2.3. PostgreSQL

By 1996, it became clear that the name “Postgres95” would not stand the test of time. We chose a new name, PostgreSQL, to reflect the relationship between the original POSTGRES and the more recent versions with SQL capability. At the same time, we set the version numbering to start at 6.0, putting the numbers back into the sequence originally begun by the Berkeley POSTGRES project.

The emphasis during development of Postgres95 was on identifying and understanding existing problems in the server code. With PostgreSQL, the emphasis has shifted to augmenting features and capabilities, although work continues in all areas.

Details about what has happened in PostgreSQL since then can be found in Appendix E.

3. Conventions

This book uses the following typographical conventions to mark certain portions of text: new terms, foreign phrases, and other important passages are emphasized in *italics*. Everything that represents input or output of the computer, in particular commands, program code, and screen output, is shown in a monospaced font (`example`). Within such passages, italics (*`example`*) indicate placeholders; you must insert an actual value instead of the placeholder. On occasion, parts of program code are emphasized in bold face (**`example`**), if they have been added or changed since the preceding example.

The following conventions are used in the synopsis of a command: brackets (`[` and `]`) indicate optional parts. (In the synopsis of a Tcl command, question marks (`?`) are used instead, as is usual in Tcl.) Braces (`{` and `}`) and vertical lines (`|`) indicate that you must choose one alternative. Dots (`...`) mean that the preceding element can be repeated.

Where it enhances the clarity, SQL commands are preceded by the prompt `=>`, and shell commands are preceded by the prompt `$`. Normally, prompts are not shown, though.

An *administrator* is generally a person who is in charge of installing and running the server. A *user* could be anyone who is using, or wants to use, any part of the PostgreSQL system. These terms should not be interpreted too narrowly; this book does not have fixed presumptions about system administration procedures.

4. Further Information

Besides the documentation, that is, this book, there are other resources about PostgreSQL:

FAQs

The FAQ list contains continuously updated answers to frequently asked questions.

READMEs

`README` files are available for most contributed packages.

Web Site

The PostgreSQL web site[5] carries details on the latest release and other information to make your work or play with PostgreSQL more productive.

Mailing Lists

The mailing lists are a good place to have your questions answered, to share experiences with other users, and to contact the developers. Consult the PostgreSQL web site for details.

Yourself!

PostgreSQL is an open-source project. As such, it depends on the user community for ongoing support. As you begin to use PostgreSQL, you will rely on others for help, either through the documentation or through the mailing lists. Consider contributing your knowledge back. Read the mailing lists and answer questions. If you learn something which is not in the documentation, write it up and contribute it. If you add features to the code, contribute them.

5. http://www.postgresql.org

5. Bug Reporting Guidelines

When you find a bug in PostgreSQL we want to hear about it. Your bug reports play an important part in making PostgreSQL more reliable because even the utmost care cannot guarantee that every part of PostgreSQL will work on every platform under every circumstance.

The following suggestions are intended to assist you in forming bug reports that can be handled in an effective fashion. No one is required to follow them but doing so tends to be to everyone's advantage.

We cannot promise to fix every bug right away. If the bug is obvious, critical, or affects a lot of users, chances are good that someone will look into it. It could also happen that we tell you to update to a newer version to see if the bug happens there. Or we might decide that the bug cannot be fixed before some major rewrite we might be planning is done. Or perhaps it is simply too hard and there are more important things on the agenda. If you need help immediately, consider obtaining a commercial support contract.

5.1. Identifying Bugs

Before you report a bug, please read and re-read the documentation to verify that you can really do whatever it is you are trying. If it is not clear from the documentation whether you can do something or not, please report that too; it is a bug in the documentation. If it turns out that a program does something different from what the documentation says, that is a bug. That might include, but is not limited to, the following circumstances:

- A program terminates with a fatal signal or an operating system error message that would point to a problem in the program. (A counterexample might be a "disk full" message, since you have to fix that yourself.)
- A program produces the wrong output for any given input.
- A program refuses to accept valid input (as defined in the documentation).
- A program accepts invalid input without a notice or error message. But keep in mind that your idea of invalid input might be our idea of an extension or compatibility with traditional practice.
- PostgreSQL fails to compile, build, or install according to the instructions on supported platforms.

Here "program" refers to any executable, not only the backend server.

Being slow or resource-hogging is not necessarily a bug. Read the documentation or ask on one of the mailing lists for help in tuning your applications. Failing to comply to the SQL standard is not necessarily a bug either, unless compliance for the specific feature is explicitly claimed.

Before you continue, check on the TODO list and in the FAQ to see if your bug is already known. If you cannot decode the information on the TODO list, report your problem. The least we can do is make the TODO list clearer.

5.2. What to report

The most important thing to remember about bug reporting is to state all the facts and only facts. Do not speculate what you think went wrong, what "it seemed to do", or which part of the program has a fault. If you are not familiar with the implementation you would probably guess wrong and not help us a bit.

And even if you are, educated explanations are a great supplement to but no substitute for facts. If we are going to fix the bug we still have to see it happen for ourselves first. Reporting the bare facts is relatively straightforward (you can probably copy and paste them from the screen) but all too often important details are left out because someone thought it does not matter or the report would be understood anyway.

The following items should be contained in every bug report:

- The exact sequence of steps *from program start-up* necessary to reproduce the problem. This should be self-contained; it is not enough to send in a bare `SELECT` statement without the preceding `CREATE TABLE` and `INSERT` statements, if the output should depend on the data in the tables. We do not have the time to reverse-engineer your database schema, and if we are supposed to make up our own data we would probably miss the problem.

  The best format for a test case for SQL-related problems is a file that can be run through the psql frontend that shows the problem. (Be sure to not have anything in your `~/.psqlrc` start-up file.) An easy start at this file is to use pg_dump to dump out the table declarations and data needed to set the scene, then add the problem query. You are encouraged to minimize the size of your example, but this is not absolutely necessary. If the bug is reproducible, we will find it either way.

  If your application uses some other client interface, such as PHP, then please try to isolate the offending queries. We will probably not set up a web server to reproduce your problem. In any case remember to provide the exact input files; do not guess that the problem happens for "large files" or "midsize databases", etc. since this information is too inexact to be of use.

- The output you got. Please do not say that it "didn't work" or "crashed". If there is an error message, show it, even if you do not understand it. If the program terminates with an operating system error, say which. If nothing at all happens, say so. Even if the result of your test case is a program crash or otherwise obvious it might not happen on our platform. The easiest thing is to copy the output from the terminal, if possible.

  > **Note:** If you are reporting an error message, please obtain the most verbose form of the message. In psql, say `\set VERBOSITY verbose` beforehand. If you are extracting the message from the server log, set the run-time parameter log_error_verbosity to `verbose` so that all details are logged.

  > **Note:** In case of fatal errors, the error message reported by the client might not contain all the information available. Please also look at the log output of the database server. If you do not keep your server's log output, this would be a good time to start doing so.

- The output you expected is very important to state. If you just write "This command gives me that output." or "This is not what I expected.", we might run it ourselves, scan the output, and think it looks OK and is exactly what we expected. We should not have to spend the time to decode the exact semantics behind your commands. Especially refrain from merely saying that "This is not what SQL says/Oracle does." Digging out the correct behavior from SQL is not a fun undertaking, nor do we all know how all the other relational databases out there behave. (If your problem is a program crash, you can obviously omit this item.)

- Any command line options and other start-up options, including any relevant environment variables or configuration files that you changed from the default. Again, please provide exact information. If you are using a prepackaged distribution that starts the database server at boot time, you should try to find out how that is done.
- Anything you did at all differently from the installation instructions.
- The PostgreSQL version. You can run the command `SELECT version();` to find out the version of the server you are connected to. Most executable programs also support a `--version` option; at least `postgres --version` and `psql --version` should work. If the function or the options do not exist then your version is more than old enough to warrant an upgrade. If you run a prepackaged version, such as RPMs, say so, including any subversion the package may have. If you are talking about a CVS snapshot, mention that, including its date and time.

  If your version is older than 8.2.6 we will almost certainly tell you to upgrade. There are many bug fixes and improvements in each new release, so it is quite possible that a bug you have encountered in an older release of PostgreSQL has already been fixed. We can only provide limited support for sites using older releases of PostgreSQL; if you require more than we can provide, consider acquiring a commercial support contract.

- Platform information. This includes the kernel name and version, C library, processor, memory information, and so on. In most cases it is sufficient to report the vendor and version, but do not assume everyone knows what exactly "Debian" contains or that everyone runs on Pentiums. If you have installation problems then information about the toolchain on your machine (compiler, make, and so on) is also necessary.

Do not be afraid if your bug report becomes rather lengthy. That is a fact of life. It is better to report everything the first time than us having to squeeze the facts out of you. On the other hand, if your input files are huge, it is fair to ask first whether somebody is interested in looking into it. Here is an article[6] that outlines some more tips on reporting bugs.

Do not spend all your time to figure out which changes in the input make the problem go away. This will probably not help solving it. If it turns out that the bug cannot be fixed right away, you will still have time to find and share your work-around. Also, once again, do not waste your time guessing why the bug exists. We will find that out soon enough.

When writing a bug report, please avoid confusing terminology. The software package in total is called "PostgreSQL", sometimes "Postgres" for short. If you are specifically talking about the backend server, mention that, do not just say "PostgreSQL crashes". A crash of a single backend server process is quite different from crash of the parent "postgres" process; please don't say "the server crashed" when you mean a single backend process went down, nor vice versa. Also, client programs such as the interactive frontend "psql" are completely separate from the backend. Please try to be specific about whether the problem is on the client or server side.

5.3. Where to report bugs

In general, send bug reports to the bug report mailing list at <pgsql-bugs@postgresql.org>. You are requested to use a descriptive subject for your email message, perhaps parts of the error message.

6. http://www.chiark.greenend.org.uk/~sgtatham/bugs.html

Another method is to fill in the bug report web-form available at the project's web site[7]. Entering a bug report this way causes it to be mailed to the <pgsql-bugs@postgresql.org> mailing list.

If your bug report has security implications and you'd prefer that it not become immediately visible in public archives, don't send it to pgsql-bugs. Security issues can be reported privately to <security@postgresql.org>.

Do not send bug reports to any of the user mailing lists, such as <pgsql-sql@postgresql.org> or <pgsql-general@postgresql.org>. These mailing lists are for answering user questions, and their subscribers normally do not wish to receive bug reports. More importantly, they are unlikely to fix them.

Also, please do *not* send reports to the developers' mailing list <pgsql-hackers@postgresql.org>. This list is for discussing the development of PostgreSQL, and it would be nice if we could keep the bug reports separate. We might choose to take up a discussion about your bug report on pgsql-hackers, if the problem needs more review.

If you have a problem with the documentation, the best place to report it is the documentation mailing list <pgsql-docs@postgresql.org>. Please be specific about what part of the documentation you are unhappy with.

If your bug is a portability problem on a non-supported platform, send mail to <pgsql-ports@postgresql.org>, so we (and you) can work on porting PostgreSQL to your platform.

> **Note:** Due to the unfortunate amount of spam going around, all of the above email addresses are closed mailing lists. That is, you need to be subscribed to a list to be allowed to post on it. (You need not be subscribed to use the bug-report web form, however.) If you would like to send mail but do not want to receive list traffic, you can subscribe and set your subscription option to nomail. For more information send mail to <majordomo@postgresql.org> with the single word help in the body of the message.

7. http://www.postgresql.org/

I. Tutorial

Welcome to the PostgreSQL Tutorial. The following few chapters are intended to give a simple introduction to PostgreSQL, relational database concepts, and the SQL language to those who are new to any one of these aspects. We only assume some general knowledge about how to use computers. No particular Unix or programming experience is required. This part is mainly intended to give you some hands-on experience with important aspects of the PostgreSQL system. It makes no attempt to be a complete or thorough treatment of the topics it covers.

After you have worked through this tutorial you might want to move on to reading Part II to gain a more formal knowledge of the SQL language, or Part IV for information about developing applications for PostgreSQL. Those who set up and manage their own server should also read Part III.

Chapter 1. Getting Started

1.1. Installation

Before you can use PostgreSQL you need to install it, of course. It is possible that PostgreSQL is already installed at your site, either because it was included in your operating system distribution or because the system administrator already installed it. If that is the case, you should obtain information from the operating system documentation or your system administrator about how to access PostgreSQL.

If you are not sure whether PostgreSQL is already available or whether you can use it for your experimentation then you can install it yourself. Doing so is not hard and it can be a good exercise. PostgreSQL can be installed by any unprivileged user; no superuser (root) access is required.

If you are installing PostgreSQL yourself, then refer to Chapter 14 for instructions on installation, and return to this guide when the installation is complete. Be sure to follow closely the section about setting up the appropriate environment variables.

If your site administrator has not set things up in the default way, you may have some more work to do. For example, if the database server machine is a remote machine, you will need to set the `PGHOST` environment variable to the name of the database server machine. The environment variable `PGPORT` may also have to be set. The bottom line is this: if you try to start an application program and it complains that it cannot connect to the database, you should consult your site administrator or, if that is you, the documentation to make sure that your environment is properly set up. If you did not understand the preceding paragraph then read the next section.

1.2. Architectural Fundamentals

Before we proceed, you should understand the basic PostgreSQL system architecture. Understanding how the parts of PostgreSQL interact will make this chapter somewhat clearer.

In database jargon, PostgreSQL uses a client/server model. A PostgreSQL session consists of the following cooperating processes (programs):

- A server process, which manages the database files, accepts connections to the database from client applications, and performs actions on the database on behalf of the clients. The database server program is called `postgres`.
- The user's client (frontend) application that wants to perform database operations. Client applications can be very diverse in nature: a client could be a text-oriented tool, a graphical application, a web server that accesses the database to display web pages, or a specialized database maintenance tool. Some client applications are supplied with the PostgreSQL distribution; most are developed by users.

As is typical of client/server applications, the client and the server can be on different hosts. In that case they communicate over a TCP/IP network connection. You should keep this in mind, because the files that can be accessed on a client machine might not be accessible (or might only be accessible using a different file name) on the database server machine.

The PostgreSQL server can handle multiple concurrent connections from clients. For that purpose it starts ("forks") a new process for each connection. From that point on, the client and the new server process communicate without intervention by the original `postgres` process. Thus, the master server process is always running, waiting for client connections, whereas client and associated server processes come and go. (All of this is of course invisible to the user. We only mention it here for completeness.)

1.3. Creating a Database

The first test to see whether you can access the database server is to try to create a database. A running PostgreSQL server can manage many databases. Typically, a separate database is used for each project or for each user.

Possibly, your site administrator has already created a database for your use. He should have told you what the name of your database is. In that case you can omit this step and skip ahead to the next section.

To create a new database, in this example named `mydb`, you use the following command:

```
$ createdb mydb
```

This should produce as response:

```
CREATE DATABASE
```

If so, this step was successful and you can skip over the remainder of this section.

If you see a message similar to

```
createdb: command not found
```

then PostgreSQL was not installed properly. Either it was not installed at all or the search path was not set correctly. Try calling the command with an absolute path instead:

```
$ /usr/local/pgsql/bin/createdb mydb
```

The path at your site might be different. Contact your site administrator or check back in the installation instructions to correct the situation.

Another response could be this:

```
createdb: could not connect to database postgres: could not connect to server: No such file
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

This means that the server was not started, or it was not started where `createdb` expected it. Again, check the installation instructions or consult the administrator.

Another response could be this:

```
createdb: could not connect to database postgres: FATAL:  role "joe" does not exist
```

where your own login name is mentioned. This will happen if the administrator has not created a PostgreSQL user account for you. (PostgreSQL user accounts are distinct from operating system user accounts.) If you are the administrator, see Chapter 18 for help creating accounts. You will need to become

the operating system user under which PostgreSQL was installed (usually `postgres`) to create the first user account. It could also be that you were assigned a PostgreSQL user name that is different from your operating system user name; in that case you need to use the `-U` switch or set the `PGUSER` environment variable to specify your PostgreSQL user name.

If you have a user account but it does not have the privileges required to create a database, you will see the following:

```
createdb: database creation failed: ERROR:  permission denied to create database
```

Not every user has authorization to create new databases. If PostgreSQL refuses to create databases for you then the site administrator needs to grant you permission to create databases. Consult your site administrator if this occurs. If you installed PostgreSQL yourself then you should log in for the purposes of this tutorial under the user account that you started the server as. [1]

You can also create databases with other names. PostgreSQL allows you to create any number of databases at a given site. Database names must have an alphabetic first character and are limited to 63 characters in length. A convenient choice is to create a database with the same name as your current user name. Many tools assume that database name as the default, so it can save you some typing. To create that database, simply type

```
$ createdb
```

If you do not want to use your database anymore you can remove it. For example, if you are the owner (creator) of the database `mydb`, you can destroy it using the following command:

```
$ dropdb mydb
```

(For this command, the database name does not default to the user account name. You always need to specify it.) This action physically removes all files associated with the database and cannot be undone, so this should only be done with a great deal of forethought.

More about `createdb` and `dropdb` may be found in createdb and dropdb respectively.

1.4. Accessing a Database

Once you have created a database, you can access it by:

- Running the PostgreSQL interactive terminal program, called *psql*, which allows you to interactively enter, edit, and execute SQL commands.
- Using an existing graphical frontend tool like PgAccess or an office suite with ODBC support to create and manipulate a database. These possibilities are not covered in this tutorial.

1. As an explanation for why this works: PostgreSQL user names are separate from operating system user accounts. When you connect to a database, you can choose what PostgreSQL user name to connect as; if you don't, it will default to the same name as your current operating system account. As it happens, there will always be a PostgreSQL user account that has the same name as the operating system user that started the server, and it also happens that that user always has permission to create databases. Instead of logging in as that user you can also specify the `-U` option everywhere to select a PostgreSQL user name to connect as.

- Writing a custom application, using one of the several available language bindings. These possibilities are discussed further in Part IV.

You probably want to start up `psql`, to try out the examples in this tutorial. It can be activated for the `mydb` database by typing the command:

```
$ psql mydb
```

If you leave off the database name then it will default to your user account name. You already discovered this scheme in the previous section.

In `psql`, you will be greeted with the following message:

```
Welcome to psql 8.2.6, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

mydb=>
```

The last line could also be

```
mydb=#
```

That would mean you are a database superuser, which is most likely the case if you installed PostgreSQL yourself. Being a superuser means that you are not subject to access controls. For the purposes of this tutorial that is not of importance.

If you encounter problems starting `psql` then go back to the previous section. The diagnostics of `createdb` and `psql` are similar, and if the former worked the latter should work as well.

The last line printed out by `psql` is the prompt, and it indicates that `psql` is listening to you and that you can type SQL queries into a work space maintained by `psql`. Try out these commands:

```
mydb=> SELECT version();
                              version
-----------------------------------------------------------------
 PostgreSQL 8.2.6 on i586-pc-linux-gnu, compiled by GCC 2.96
(1 row)

mydb=> SELECT current_date;
    date
------------
 2002-08-31
(1 row)

mydb=> SELECT 2 + 2;
 ?column?
----------
        4
(1 row)
```

The `psql` program has a number of internal commands that are not SQL commands. They begin with the backslash character, "`\`". Some of these commands were listed in the welcome message. For example, you can get help on the syntax of various PostgreSQL SQL commands by typing:

```
mydb=> \h
```

To get out of `psql`, type

```
mydb=> \q
```

and `psql` will quit and return you to your command shell. (For more internal commands, type `\?` at the `psql` prompt.) The full capabilities of `psql` are documented in psql. If PostgreSQL is installed correctly you can also type `man psql` at the operating system shell prompt to see the documentation. In this tutorial we will not use these features explicitly, but you can use them yourself when you see fit.

Chapter 2. The SQL Language

2.1. Introduction

This chapter provides an overview of how to use SQL to perform simple operations. This tutorial is only intended to give you an introduction and is in no way a complete tutorial on SQL. Numerous books have been written on SQL, including *Understanding the New SQL* and *A Guide to the SQL Standard*. You should be aware that some PostgreSQL language features are extensions to the standard.

In the examples that follow, we assume that you have created a database named `mydb`, as described in the previous chapter, and have been able to start psql.

Examples in this manual can also be found in the PostgreSQL source distribution in the directory `src/tutorial/`. To use those files, first change to that directory and run make:

```
$ cd ..../src/tutorial
$ make
```

This creates the scripts and compiles the C files containing user-defined functions and types. (If you installed a pre-packaged version of PostgreSQL rather than building from source, look for a directory named `tutorial` within the PostgreSQL documentation. The "make" part should already have been done for you.) Then, to start the tutorial, do the following:

```
$ cd ..../tutorial
$ psql -s mydb
...

mydb=> \i basics.sql
```

The `\i` command reads in commands from the specified file. The `-s` option puts you in single step mode which pauses before sending each statement to the server. The commands used in this section are in the file `basics.sql`.

2.2. Concepts

PostgreSQL is a *relational database management system* (RDBMS). That means it is a system for managing data stored in *relations*. Relation is essentially a mathematical term for *table*. The notion of storing data in tables is so commonplace today that it might seem inherently obvious, but there are a number of other ways of organizing databases. Files and directories on Unix-like operating systems form an example of a hierarchical database. A more modern development is the object-oriented database.

Each table is a named collection of *rows*. Each row of a given table has the same set of named *columns*, and each column is of a specific data type. Whereas columns have a fixed order in each row, it is important to remember that SQL does not guarantee the order of the rows within the table in any way (although they can be explicitly sorted for display).

Tables are grouped into databases, and a collection of databases managed by a single PostgreSQL server instance constitutes a database *cluster*.

2.3. Creating a New Table

You can create a new table by specifying the table name, along with all column names and their types:

```
CREATE TABLE weather (
    city            varchar(80),
    temp_lo         int,           -- low temperature
    temp_hi         int,           -- high temperature
    prcp            real,          -- precipitation
    date            date
);
```

You can enter this into `psql` with the line breaks. `psql` will recognize that the command is not terminated until the semicolon.

White space (i.e., spaces, tabs, and newlines) may be used freely in SQL commands. That means you can type the command aligned differently than above, or even all on one line. Two dashes ("`--`") introduce comments. Whatever follows them is ignored up to the end of the line. SQL is case insensitive about key words and identifiers, except when identifiers are double-quoted to preserve the case (not done above).

`varchar(80)` specifies a data type that can store arbitrary character strings up to 80 characters in length. `int` is the normal integer type. `real` is a type for storing single precision floating-point numbers. `date` should be self-explanatory. (Yes, the column of type `date` is also named `date`. This may be convenient or confusing — you choose.)

PostgreSQL supports the standard SQL types `int`, `smallint`, `real`, `double precision`, `char(`*N*`)`, `varchar(`*N*`)`, `date`, `time`, `timestamp`, and `interval`, as well as other types of general utility and a rich set of geometric types. PostgreSQL can be customized with an arbitrary number of user-defined data types. Consequently, type names are not syntactical key words, except where required to support special cases in the SQL standard.

The second example will store cities and their associated geographical location:

```
CREATE TABLE cities (
    name            varchar(80),
    location        point
);
```

The `point` type is an example of a PostgreSQL-specific data type.

Finally, it should be mentioned that if you don't need a table any longer or want to recreate it differently you can remove it using the following command:

```
DROP TABLE tablename;
```

2.4. Populating a Table With Rows

The `INSERT` statement is used to populate a table with rows:

```
INSERT INTO weather VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
```

Note that all data types use rather obvious input formats. Constants that are not simple numeric values usually must be surrounded by single quotes (’), as in the example. The date type is actually quite flexible in what it accepts, but for this tutorial we will stick to the unambiguous format shown here.

The point type requires a coordinate pair as input, as shown here:

```
INSERT INTO cities VALUES (’San Francisco’, ’(-194.0, 53.0)’);
```

The syntax used so far requires you to remember the order of the columns. An alternative syntax allows you to list the columns explicitly:

```
INSERT INTO weather (city, temp_lo, temp_hi, prcp, date)
    VALUES (’San Francisco’, 43, 57, 0.0, ’1994-11-29’);
```

You can list the columns in a different order if you wish or even omit some columns, e.g., if the precipitation is unknown:

```
INSERT INTO weather (date, city, temp_hi, temp_lo)
    VALUES (’1994-11-29’, ’Hayward’, 54, 37);
```

Many developers consider explicitly listing the columns better style than relying on the order implicitly.

Please enter all the commands shown above so you have some data to work with in the following sections.

You could also have used COPY to load large amounts of data from flat-text files. This is usually faster because the COPY command is optimized for this application while allowing less flexibility than INSERT. An example would be:

```
COPY weather FROM ’/home/user/weather.txt’;
```

where the file name for the source file must be available to the backend server machine, not the client, since the backend server reads the file directly. You can read more about the COPY command in *COPY*.

2.5. Querying a Table

To retrieve data from a table, the table is *queried*. An SQL SELECT statement is used to do this. The statement is divided into a select list (the part that lists the columns to be returned), a table list (the part that lists the tables from which to retrieve the data), and an optional qualification (the part that specifies any restrictions). For example, to retrieve all the rows of table weather, type:

```
SELECT * FROM weather;
```

Here * is a shorthand for “all columns”. [1] So the same result would be had with:

```
SELECT city, temp_lo, temp_hi, prcp, date FROM weather;
```

The output should be:

1. While SELECT * is useful for off-the-cuff queries, it is widely considered bad style in production code, since adding a column to the table would change the results.

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      43 |      57 |    0 | 1994-11-29
 Hayward       |      37 |      54 |      | 1994-11-29
(3 rows)
```

You can write expressions, not just simple column references, in the select list. For example, you can do:

```
SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date FROM weather;
```

This should give:

```
     city      | temp_avg |    date
---------------+----------+------------
 San Francisco |       48 | 1994-11-27
 San Francisco |       50 | 1994-11-29
 Hayward       |       45 | 1994-11-29
(3 rows)
```

Notice how the `AS` clause is used to relabel the output column. (The `AS` clause is optional.)

A query can be "qualified" by adding a `WHERE` clause that specifies which rows are wanted. The `WHERE` clause contains a Boolean (truth value) expression, and only rows for which the Boolean expression is true are returned. The usual Boolean operators (`AND`, `OR`, and `NOT`) are allowed in the qualification. For example, the following retrieves the weather of San Francisco on rainy days:

```
SELECT * FROM weather
    WHERE city = 'San Francisco' AND prcp > 0.0;
```

Result:

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
(1 row)
```

You can request that the results of a query be returned in sorted order:

```
SELECT * FROM weather
    ORDER BY city;

     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 Hayward       |      37 |      54 |      | 1994-11-29
 San Francisco |      43 |      57 |    0 | 1994-11-29
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
```

In this example, the sort order isn't fully specified, and so you might get the San Francisco rows in either order. But you'd always get the results shown above if you do

```
SELECT * FROM weather
    ORDER BY city, temp_lo;
```

You can request that duplicate rows be removed from the result of a query:

```
SELECT DISTINCT city
    FROM weather;

     city
---------------
 Hayward
 San Francisco
(2 rows)
```

Here again, the result row ordering might vary. You can ensure consistent results by using `DISTINCT` and `ORDER BY` together: [2]

```
SELECT DISTINCT city
    FROM weather
    ORDER BY city;
```

2.6. Joins Between Tables

Thus far, our queries have only accessed one table at a time. Queries can access multiple tables at once, or access the same table in such a way that multiple rows of the table are being processed at the same time. A query that accesses multiple rows of the same or different tables at one time is called a *join* query. As an example, say you wish to list all the weather records together with the location of the associated city. To do that, we need to compare the city column of each row of the weather table with the name column of all rows in the cities table, and select the pairs of rows where these values match.

> **Note:** This is only a conceptual model. The join is usually performed in a more efficient manner than actually comparing each possible pair of rows, but this is invisible to the user.

This would be accomplished by the following query:

```
SELECT *
    FROM weather, cities
    WHERE city = name;

     city      | temp_lo | temp_hi | prcp |    date    |     name      | location
---------------+---------+---------+------+------------+---------------+-----------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco | (-194,53)
(2 rows)
```

2. In some database systems, including older versions of PostgreSQL, the implementation of `DISTINCT` automatically orders the rows and so `ORDER BY` is unnecessary. But this is not required by the SQL standard, and current PostgreSQL doesn't guarantee that `DISTINCT` causes the rows to be ordered.

Observe two things about the result set:

- There is no result row for the city of Hayward. This is because there is no matching entry in the `cities` table for Hayward, so the join ignores the unmatched rows in the weather table. We will see shortly how this can be fixed.
- There are two columns containing the city name. This is correct because the lists of columns of the `weather` and the `cities` table are concatenated. In practice this is undesirable, though, so you will probably want to list the output columns explicitly rather than using `*`:

```
SELECT city, temp_lo, temp_hi, prcp, date, location
    FROM weather, cities
    WHERE city = name;
```

Exercise: Attempt to find out the semantics of this query when the `WHERE` clause is omitted.

Since the columns all had different names, the parser automatically found out which table they belong to. If there were duplicate column names in the two tables you'd need to *qualify* the column names to show which one you meant, as in:

```
SELECT weather.city, weather.temp_lo, weather.temp_hi,
       weather.prcp, weather.date, cities.location
    FROM weather, cities
    WHERE cities.name = weather.city;
```

It is widely considered good style to qualify all column names in a join query, so that the query won't fail if a duplicate column name is later added to one of the tables.

Join queries of the kind seen thus far can also be written in this alternative form:

```
SELECT *
    FROM weather INNER JOIN cities ON (weather.city = cities.name);
```

This syntax is not as commonly used as the one above, but we show it here to help you understand the following topics.

Now we will figure out how we can get the Hayward records back in. What we want the query to do is to scan the `weather` table and for each row to find the matching `cities` row(s). If no matching row is found we want some "empty values" to be substituted for the `cities` table's columns. This kind of query is called an *outer join*. (The joins we have seen so far are inner joins.) The command looks like this:

```
SELECT *
    FROM weather LEFT OUTER JOIN cities ON (weather.city = cities.name);

     city      | temp_lo | temp_hi | prcp |    date    |     name      | location
---------------+---------+---------+------+------------+---------------+-----------
 Hayward       |      37 |      54 |      | 1994-11-29 |               |
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco | (-194,53)
(3 rows)
```

This query is called a *left outer join* because the table mentioned on the left of the join operator will have each of its rows in the output at least once, whereas the table on the right will only have those rows output that match some row of the left table. When outputting a left-table row for which there is no right-table match, empty (null) values are substituted for the right-table columns.

Exercise: There are also right outer joins and full outer joins. Try to find out what those do.

We can also join a table against itself. This is called a *self join*. As an example, suppose we wish to find all the weather records that are in the temperature range of other weather records. So we need to compare the `temp_lo` and `temp_hi` columns of each `weather` row to the `temp_lo` and `temp_hi` columns of all other `weather` rows. We can do this with the following query:

```
SELECT W1.city, W1.temp_lo AS low, W1.temp_hi AS high,
    W2.city, W2.temp_lo AS low, W2.temp_hi AS high
    FROM weather W1, weather W2
    WHERE W1.temp_lo < W2.temp_lo
    AND W1.temp_hi > W2.temp_hi;

     city      | low | high |     city      | low | high
---------------+-----+------+---------------+-----+------
 San Francisco |  43 |   57 | San Francisco |  46 |   50
 Hayward       |  37 |   54 | San Francisco |  46 |   50
(2 rows)
```

Here we have relabeled the weather table as `W1` and `W2` to be able to distinguish the left and right side of the join. You can also use these kinds of aliases in other queries to save some typing, e.g.:

```
SELECT *
    FROM weather w, cities c
    WHERE w.city = c.name;
```

You will encounter this style of abbreviating quite frequently.

2.7. Aggregate Functions

Like most other relational database products, PostgreSQL supports aggregate functions. An aggregate function computes a single result from multiple input rows. For example, there are aggregates to compute the `count`, `sum`, `avg` (average), `max` (maximum) and `min` (minimum) over a set of rows.

As an example, we can find the highest low-temperature reading anywhere with

```
SELECT max(temp_lo) FROM weather;

 max
-----
  46
(1 row)
```

If we wanted to know what city (or cities) that reading occurred in, we might try

```
SELECT city FROM weather WHERE temp_lo = max(temp_lo);     WRONG
```

but this will not work since the aggregate `max` cannot be used in the `WHERE` clause. (This restriction exists because the `WHERE` clause determines which rows will be included in the aggregate calculation; so obviously it has to be evaluated before aggregate functions are computed.) However, as is often the case the query can be restated to accomplish the desired result, here by using a *subquery*:

```
SELECT city FROM weather
    WHERE temp_lo = (SELECT max(temp_lo) FROM weather);

     city
---------------
 San Francisco
(1 row)
```

This is OK because the subquery is an independent computation that computes its own aggregate separately from what is happening in the outer query.

Aggregates are also very useful in combination with `GROUP BY` clauses. For example, we can get the maximum low temperature observed in each city with

```
SELECT city, max(temp_lo)
    FROM weather
    GROUP BY city;

     city      | max
---------------+-----
 Hayward       |  37
 San Francisco |  46
(2 rows)
```

which gives us one output row per city. Each aggregate result is computed over the table rows matching that city. We can filter these grouped rows using `HAVING`:

```
SELECT city, max(temp_lo)
    FROM weather
    GROUP BY city
    HAVING max(temp_lo) < 40;

  city   | max
---------+-----
 Hayward |  37
(1 row)
```

which gives us the same results for only the cities that have all `temp_lo` values below 40. Finally, if we only care about cities whose names begin with "`S`", we might do

```
SELECT city, max(temp_lo)
    FROM weather
    WHERE city LIKE 'S%'❶
    GROUP BY city
    HAVING max(temp_lo) < 40;
```

❶ The LIKE operator does pattern matching and is explained in Section 9.7.

It is important to understand the interaction between aggregates and SQL's WHERE and HAVING clauses. The fundamental difference between WHERE and HAVING is this: WHERE selects input rows before groups and aggregates are computed (thus, it controls which rows go into the aggregate computation), whereas HAVING selects group rows after groups and aggregates are computed. Thus, the WHERE clause must not contain aggregate functions; it makes no sense to try to use an aggregate to determine which rows will be inputs to the aggregates. On the other hand, the HAVING clause always contains aggregate functions. (Strictly speaking, you are allowed to write a HAVING clause that doesn't use aggregates, but it's seldom useful. The same condition could be used more efficiently at the WHERE stage.)

In the previous example, we can apply the city name restriction in WHERE, since it needs no aggregate. This is more efficient than adding the restriction to HAVING, because we avoid doing the grouping and aggregate calculations for all rows that fail the WHERE check.

2.8. Updates

You can update existing rows using the UPDATE command. Suppose you discover the temperature readings are all off by 2 degrees after November 28. You may correct the data as follows:

```
UPDATE weather
    SET temp_hi = temp_hi - 2,  temp_lo = temp_lo - 2
    WHERE date > '1994-11-28';
```

Look at the new state of the data:

```
SELECT * FROM weather;

     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
 Hayward       |      35 |      52 |      | 1994-11-29
(3 rows)
```

2.9. Deletions

Rows can be removed from a table using the DELETE command. Suppose you are no longer interested in the weather of Hayward. Then you can do the following to delete those rows from the table:

```
DELETE FROM weather WHERE city = 'Hayward';
```

All weather records belonging to Hayward are removed.

```
SELECT * FROM weather;

     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
(2 rows)
```

One should be wary of statements of the form

```
DELETE FROM tablename;
```

Without a qualification, `DELETE` will remove *all* rows from the given table, leaving it empty. The system will not request confirmation before doing this!

Chapter 3. Advanced Features

3.1. Introduction

In the previous chapter we have covered the basics of using SQL to store and access your data in PostgreSQL. We will now discuss some more advanced features of SQL that simplify management and prevent loss or corruption of your data. Finally, we will look at some PostgreSQL extensions.

This chapter will on occasion refer to examples found in Chapter 2 to change or improve them, so it will be of advantage if you have read that chapter. Some examples from this chapter can also be found in `advanced.sql` in the tutorial directory. This file also contains some example data to load, which is not repeated here. (Refer to Section 2.1 for how to use the file.)

3.2. Views

Refer back to the queries in Section 2.6. Suppose the combined listing of weather records and city location is of particular interest to your application, but you do not want to type the query each time you need it. You can create a *view* over the query, which gives a name to the query that you can refer to like an ordinary table.

```
CREATE VIEW myview AS
    SELECT city, temp_lo, temp_hi, prcp, date, location
        FROM weather, cities
        WHERE city = name;

SELECT * FROM myview;
```

Making liberal use of views is a key aspect of good SQL database design. Views allow you to encapsulate the details of the structure of your tables, which may change as your application evolves, behind consistent interfaces.

Views can be used in almost any place a real table can be used. Building views upon other views is not uncommon.

3.3. Foreign Keys

Recall the `weather` and `cities` tables from Chapter 2. Consider the following problem: You want to make sure that no one can insert rows in the `weather` table that do not have a matching entry in the `cities` table. This is called maintaining the *referential integrity* of your data. In simplistic database systems this would be implemented (if at all) by first looking at the `cities` table to check if a matching record exists, and then inserting or rejecting the new `weather` records. This approach has a number of problems and is very inconvenient, so PostgreSQL can do this for you.

The new declaration of the tables would look like this:

```
CREATE TABLE cities (
        city     varchar(80) primary key,
        location point
);

CREATE TABLE weather (
        city      varchar(80) references cities(city),
        temp_lo   int,
        temp_hi   int,
        prcp      real,
        date      date
);
```

Now try inserting an invalid record:

```
INSERT INTO weather VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');

ERROR:  insert or update on table "weather" violates foreign key constraint "weather_city_f
DETAIL:  Key (city)=(Berkeley) is not present in table "cities".
```

The behavior of foreign keys can be finely tuned to your application. We will not go beyond this simple example in this tutorial, but just refer you to Chapter 5 for more information. Making correct use of foreign keys will definitely improve the quality of your database applications, so you are strongly encouraged to learn about them.

3.4. Transactions

Transactions are a fundamental concept of all database systems. The essential point of a transaction is that it bundles multiple steps into a single, all-or-nothing operation. The intermediate states between the steps are not visible to other concurrent transactions, and if some failure occurs that prevents the transaction from completing, then none of the steps affect the database at all.

For example, consider a bank database that contains balances for various customer accounts, as well as total deposit balances for branches. Suppose that we want to record a payment of $100.00 from Alice's account to Bob's account. Simplifying outrageously, the SQL commands for this might look like

```
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
UPDATE branches SET balance = balance - 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Alice');
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
UPDATE branches SET balance = balance + 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Bob');
```

The details of these commands are not important here; the important point is that there are several separate updates involved to accomplish this rather simple operation. Our bank's officers will want to be assured that either all these updates happen, or none of them happen. It would certainly not do for a system failure

to result in Bob receiving $100.00 that was not debited from Alice. Nor would Alice long remain a happy customer if she was debited without Bob being credited. We need a guarantee that if something goes wrong partway through the operation, none of the steps executed so far will take effect. Grouping the updates into a *transaction* gives us this guarantee. A transaction is said to be *atomic*: from the point of view of other transactions, it either happens completely or not at all.

We also want a guarantee that once a transaction is completed and acknowledged by the database system, it has indeed been permanently recorded and won't be lost even if a crash ensues shortly thereafter. For example, if we are recording a cash withdrawal by Bob, we do not want any chance that the debit to his account will disappear in a crash just after he walks out the bank door. A transactional database guarantees that all the updates made by a transaction are logged in permanent storage (i.e., on disk) before the transaction is reported complete.

Another important property of transactional databases is closely related to the notion of atomic updates: when multiple transactions are running concurrently, each one should not be able to see the incomplete changes made by others. For example, if one transaction is busy totalling all the branch balances, it would not do for it to include the debit from Alice's branch but not the credit to Bob's branch, nor vice versa. So transactions must be all-or-nothing not only in terms of their permanent effect on the database, but also in terms of their visibility as they happen. The updates made so far by an open transaction are invisible to other transactions until the transaction completes, whereupon all the updates become visible simultaneously.

In PostgreSQL, a transaction is set up by surrounding the SQL commands of the transaction with `BEGIN` and `COMMIT` commands. So our banking transaction would actually look like

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
-- etc etc
COMMIT;
```

If, partway through the transaction, we decide we do not want to commit (perhaps we just noticed that Alice's balance went negative), we can issue the command `ROLLBACK` instead of `COMMIT`, and all our updates so far will be canceled.

PostgreSQL actually treats every SQL statement as being executed within a transaction. If you do not issue a `BEGIN` command, then each individual statement has an implicit `BEGIN` and (if successful) `COMMIT` wrapped around it. A group of statements surrounded by `BEGIN` and `COMMIT` is sometimes called a *transaction block*.

> **Note:** Some client libraries issue `BEGIN` and `COMMIT` commands automatically, so that you may get the effect of transaction blocks without asking. Check the documentation for the interface you are using.

It's possible to control the statements in a transaction in a more granular fashion through the use of *savepoints*. Savepoints allow you to selectively discard parts of the transaction, while committing the rest. After defining a savepoint with `SAVEPOINT`, you can if needed roll back to the savepoint with `ROLLBACK TO`. All the transaction's database changes between defining the savepoint and rolling back to it are discarded, but changes earlier than the savepoint are kept.

After rolling back to a savepoint, it continues to be defined, so you can roll back to it several times. Conversely, if you are sure you won't need to roll back to a particular savepoint again, it can be released, so the system can free some resources. Keep in mind that either releasing or rolling back to a savepoint will automatically release all savepoints that were defined after it.

All this is happening within the transaction block, so none of it is visible to other database sessions. When and if you commit the transaction block, the committed actions become visible as a unit to other sessions, while the rolled-back actions never become visible at all.

Remembering the bank database, suppose we debit $100.00 from Alice's account, and credit Bob's account, only to find later that we should have credited Wally's account. We could do it using savepoints like this:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
-- oops ... forget that and use Wally's account
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Wally';
COMMIT;
```

This example is, of course, oversimplified, but there's a lot of control to be had over a transaction block through the use of savepoints. Moreover, `ROLLBACK TO` is the only way to regain control of a transaction block that was put in aborted state by the system due to an error, short of rolling it back completely and starting again.

3.5. Inheritance

Inheritance is a concept from object-oriented databases. It opens up interesting new possibilities of database design.

Let's create two tables: A table `cities` and a table `capitals`. Naturally, capitals are also cities, so you want some way to show the capitals implicitly when you list all cities. If you're really clever you might invent some scheme like this:

```
CREATE TABLE capitals (
  name       text,
  population real,
  altitude   int,    -- (in ft)
  state      char(2)
);

CREATE TABLE non_capitals (
  name       text,
  population real,
  altitude   int     -- (in ft)
```

```
);

CREATE VIEW cities AS
  SELECT name, population, altitude FROM capitals
    UNION
  SELECT name, population, altitude FROM non_capitals;
```

This works OK as far as querying goes, but it gets ugly when you need to update several rows, for one thing.

A better solution is this:

```
CREATE TABLE cities (
  name       text,
  population real,
  altitude   int     -- (in ft)
);

CREATE TABLE capitals (
  state      char(2)
) INHERITS (cities);
```

In this case, a row of `capitals` *inherits* all columns (`name`, `population`, and `altitude`) from its *parent*, `cities`. The type of the column `name` is `text`, a native PostgreSQL type for variable length character strings. State capitals have an extra column, `state`, that shows their state. In PostgreSQL, a table can inherit from zero or more other tables.

For example, the following query finds the names of all cities, including state capitals, that are located at an altitude over 500 ft.:

```
SELECT name, altitude
  FROM cities
  WHERE altitude > 500;
```

which returns:

```
   name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
(3 rows)
```

On the other hand, the following query finds all the cities that are not state capitals and are situated at an altitude of 500 ft. or higher:

```
SELECT name, altitude
    FROM ONLY cities
    WHERE altitude > 500;

   name    | altitude
```

```
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
(2 rows)
```

Here the `ONLY` before `cities` indicates that the query should be run over only the `cities` table, and not tables below `cities` in the inheritance hierarchy. Many of the commands that we have already discussed — `SELECT`, `UPDATE`, and `DELETE` — support this `ONLY` notation.

> **Note:** Although inheritance is frequently useful, it has not been integrated with unique constraints or foreign keys, which limits its usefulness. See Section 5.8 for more detail.

3.6. Conclusion

PostgreSQL has many features not touched upon in this tutorial introduction, which has been oriented toward newer users of SQL. These features are discussed in more detail in the remainder of this book.

If you feel you need more introductory material, please visit the PostgreSQL web site[1] for links to more resources.

1. http://www.postgresql.org

II. The SQL Language

This part describes the use of the SQL language in PostgreSQL. We start with describing the general syntax of SQL, then explain how to create the structures to hold data, how to populate the database, and how to query it. The middle part lists the available data types and functions for use in SQL commands. The rest treats several aspects that are important for tuning a database for optimal performance.

The information in this part is arranged so that a novice user can follow it start to end to gain a full understanding of the topics without having to refer forward too many times. The chapters are intended to be self-contained, so that advanced users can read the chapters individually as they choose. The information in this part is presented in a narrative fashion in topical units. Readers looking for a complete description of a particular command should look into Part VI.

Readers of this part should know how to connect to a PostgreSQL database and issue SQL commands. Readers that are unfamiliar with these issues are encouraged to read Part I first. SQL commands are typically entered using the PostgreSQL interactive terminal psql, but other programs that have similar functionality can be used as well.

Chapter 4. SQL Syntax

This chapter describes the syntax of SQL. It forms the foundation for understanding the following chapters which will go into detail about how the SQL commands are applied to define and modify data.

We also advise users who are already familiar with SQL to read this chapter carefully because there are several rules and concepts that are implemented inconsistently among SQL databases or that are specific to PostgreSQL.

4.1. Lexical Structure

SQL input consists of a sequence of *commands*. A command is composed of a sequence of *tokens*, terminated by a semicolon (";"). The end of the input stream also terminates a command. Which tokens are valid depends on the syntax of the particular command.

A token can be a *key word*, an *identifier*, a *quoted identifier*, a *literal* (or constant), or a special character symbol. Tokens are normally separated by whitespace (space, tab, newline), but need not be if there is no ambiguity (which is generally only the case if a special character is adjacent to some other token type).

Additionally, *comments* can occur in SQL input. They are not tokens, they are effectively equivalent to whitespace.

For example, the following is (syntactically) valid SQL input:

```
SELECT * FROM MY_TABLE;
UPDATE MY_TABLE SET A = 5;
INSERT INTO MY_TABLE VALUES (3, 'hi there');
```

This is a sequence of three commands, one per line (although this is not required; more than one command can be on a line, and commands can usefully be split across lines).

The SQL syntax is not very consistent regarding what tokens identify commands and which are operands or parameters. The first few tokens are generally the command name, so in the above example we would usually speak of a "SELECT", an "UPDATE", and an "INSERT" command. But for instance the `UPDATE` command always requires a `SET` token to appear in a certain position, and this particular variation of `INSERT` also requires a `VALUES` in order to be complete. The precise syntax rules for each command are described in Part VI.

4.1.1. Identifiers and Key Words

Tokens such as `SELECT`, `UPDATE`, or `VALUES` in the example above are examples of *key words*, that is, words that have a fixed meaning in the SQL language. The tokens `MY_TABLE` and `A` are examples of *identifiers*. They identify names of tables, columns, or other database objects, depending on the command they are used in. Therefore they are sometimes simply called "names". Key words and identifiers have the same lexical structure, meaning that one cannot know whether a token is an identifier or a key word without knowing the language. A complete list of key words can be found in Appendix C.

SQL identifiers and key words must begin with a letter (`a-z`, but also letters with diacritical marks and non-Latin letters) or an underscore (_). Subsequent characters in an identifier or key word can be letters, underscores, digits (`0-9`), or dollar signs (`$`). Note that dollar signs are not allowed in identifiers according

to the letter of the SQL standard, so their use may render applications less portable. The SQL standard will not define a key word that contains digits or starts or ends with an underscore, so identifiers of this form are safe against possible conflict with future extensions of the standard.

The system uses no more than NAMEDATALEN-1 characters of an identifier; longer names can be written in commands, but they will be truncated. By default, NAMEDATALEN is 64 so the maximum identifier length is 63. If this limit is problematic, it can be raised by changing the NAMEDATALEN constant in src/include/postgres_ext.h.

Identifier and key word names are case insensitive. Therefore

```
UPDATE MY_TABLE SET A = 5;
```

can equivalently be written as

```
uPDaTE my_TabLE SeT a = 5;
```

A convention often used is to write key words in upper case and names in lower case, e.g.,

```
UPDATE my_table SET a = 5;
```

There is a second kind of identifier: the *delimited identifier* or *quoted identifier*. It is formed by enclosing an arbitrary sequence of characters in double-quotes ("). A delimited identifier is always an identifier, never a key word. So "select" could be used to refer to a column or table named "select", whereas an unquoted select would be taken as a key word and would therefore provoke a parse error when used where a table or column name is expected. The example can be written with quoted identifiers like this:

```
UPDATE "my_table" SET "a" = 5;
```

Quoted identifiers can contain any character, except the character with code zero. (To include a double quote, write two double quotes.) This allows constructing table or column names that would otherwise not be possible, such as ones containing spaces or ampersands. The length limitation still applies.

Quoting an identifier also makes it case-sensitive, whereas unquoted names are always folded to lower case. For example, the identifiers FOO, foo, and "foo" are considered the same by PostgreSQL, but "Foo" and "FOO" are different from these three and each other. (The folding of unquoted names to lower case in PostgreSQL is incompatible with the SQL standard, which says that unquoted names should be folded to upper case. Thus, foo should be equivalent to "FOO" not "foo" according to the standard. If you want to write portable applications you are advised to always quote a particular name or never quote it.)

4.1.2. Constants

There are three kinds of *implicitly-typed constants* in PostgreSQL: strings, bit strings, and numbers. Constants can also be specified with explicit types, which can enable more accurate representation and more efficient handling by the system. These alternatives are discussed in the following subsections.

4.1.2.1. String Constants

A string constant in SQL is an arbitrary sequence of characters bounded by single quotes (`'`), for example `'This is a string'`. To include a single-quote character within a string constant, write two adjacent single quotes, e.g. `'Dianne''s horse'`. Note that this is *not* the same as a double-quote character (`"`).

Two string constants that are only separated by whitespace *with at least one newline* are concatenated and effectively treated as if the string had been written as one constant. For example:

```
SELECT 'foo'
'bar';
```

is equivalent to

```
SELECT 'foobar';
```

but

```
SELECT 'foo'      'bar';
```

is not valid syntax. (This slightly bizarre behavior is specified by SQL; PostgreSQL is following the standard.)

PostgreSQL also accepts "escape" string constants, which are an extension to the SQL standard. An escape string constant is specified by writing the letter `E` (upper or lower case) just before the opening single quote, e.g. `E'foo'`. (When continuing an escape string constant across lines, write `E` only before the first opening quote.) Within an escape string, a backslash character (`\`) begins a C-like *backslash escape* sequence, in which the combination of backslash and following character(s) represents a special byte value. `\b` is a backspace, `\f` is a form feed, `\n` is a newline, `\r` is a carriage return, `\t` is a tab. Also supported are `\`*`digits`*, where *`digits`* represents an octal byte value, and `\x`*`hexdigits`*, where *`hexdigits`* represents a hexadecimal byte value. (It is your responsibility that the byte sequences you create are valid characters in the server character set encoding.) Any other character following a backslash is taken literally. Thus, to include a backslash character, write two backslashes (`\\`). Also, a single quote can be included in an escape string by writing `\'`, in addition to the normal way of `''`.

> **Caution**
>
> If the configuration parameter standard_conforming_strings is `off`, then PostgreSQL recognizes backslash escapes in both regular and escape string constants. This is for backward compatibility with the historical behavior, in which backslash escapes were always recognized. Although `standard_conforming_strings` currently defaults to `off`, the default will change to `on` in a future release for improved standards compliance. Applications are therefore encouraged to migrate away from using backslash escapes. If you need to use a backslash escape to represent a special character, write the constant with an `E` to be sure it will be handled the same way in future releases.
>
> In addition to `standard_conforming_strings`, the configuration parameters escape_string_warning and backslash_quote govern treatment of backslashes in string constants.

The character with the code zero cannot be in a string constant.

4.1.2.2. Dollar-Quoted String Constants

While the standard syntax for specifying string constants is usually convenient, it can be difficult to understand when the desired string contains many single quotes or backslashes, since each of those must be doubled. To allow more readable queries in such situations, PostgreSQL provides another way, called "dollar quoting", to write string constants. A dollar-quoted string constant consists of a dollar sign (`$`), an optional "tag" of zero or more characters, another dollar sign, an arbitrary sequence of characters that makes up the string content, a dollar sign, the same tag that began this dollar quote, and a dollar sign. For example, here are two different ways to specify the string "Dianne's horse" using dollar quoting:

```
$$Dianne's horse$$
$SomeTag$Dianne's horse$SomeTag$
```

Notice that inside the dollar-quoted string, single quotes can be used without needing to be escaped. Indeed, no characters inside a dollar-quoted string are ever escaped: the string content is always written literally. Backslashes are not special, and neither are dollar signs, unless they are part of a sequence matching the opening tag.

It is possible to nest dollar-quoted string constants by choosing different tags at each nesting level. This is most commonly used in writing function definitions. For example:

```
$function$
BEGIN
    RETURN ($1 ~ $q$[\t\r\n\v\\]$q$);
END;
$function$
```

Here, the sequence `$q$[\t\r\n\v\\]$q$` represents a dollar-quoted literal string `[\t\r\n\v\\]`, which will be recognized when the function body is executed by PostgreSQL. But since the sequence does not match the outer dollar quoting delimiter `$function$`, it is just some more characters within the constant so far as the outer string is concerned.

The tag, if any, of a dollar-quoted string follows the same rules as an unquoted identifier, except that it cannot contain a dollar sign. Tags are case sensitive, so `$tag$String content$tag$` is correct, but `$TAG$String content$tag$` is not.

A dollar-quoted string that follows a keyword or identifier must be separated from it by whitespace; otherwise the dollar quoting delimiter would be taken as part of the preceding identifier.

Dollar quoting is not part of the SQL standard, but it is often a more convenient way to write complicated string literals than the standard-compliant single quote syntax. It is particularly useful when representing string constants inside other constants, as is often needed in procedural function definitions. With single-quote syntax, each backslash in the above example would have to be written as four backslashes, which would be reduced to two backslashes in parsing the original string constant, and then to one when the inner string constant is re-parsed during function execution.

4.1.2.3. Bit-String Constants

Bit-string constants look like regular string constants with a `B` (upper or lower case) immediately before the opening quote (no intervening whitespace), e.g., `B'1001'`. The only characters allowed within bit-string constants are `0` and `1`.

Alternatively, bit-string constants can be specified in hexadecimal notation, using a leading `X` (upper or lower case), e.g., `X'1FF'`. This notation is equivalent to a bit-string constant with four binary digits for each hexadecimal digit.

Both forms of bit-string constant can be continued across lines in the same way as regular string constants. Dollar quoting cannot be used in a bit-string constant.

4.1.2.4. Numeric Constants

Numeric constants are accepted in these general forms:

```
digits
digits.[digits][e[+-]digits]
[digits].digits[e[+-]digits]
digitse[+-]digits
```

where *digits* is one or more decimal digits (0 through 9). At least one digit must be before or after the decimal point, if one is used. At least one digit must follow the exponent marker (`e`), if one is present. There may not be any spaces or other characters embedded in the constant. Note that any leading plus or minus sign is not actually considered part of the constant; it is an operator applied to the constant.

These are some examples of valid numeric constants:

42
3.5
4.
.001
5e2
1.925e-3

A numeric constant that contains neither a decimal point nor an exponent is initially presumed to be type `integer` if its value fits in type `integer` (32 bits); otherwise it is presumed to be type `bigint` if its value fits in type `bigint` (64 bits); otherwise it is taken to be type `numeric`. Constants that contain decimal points and/or exponents are always initially presumed to be type `numeric`.

The initially assigned data type of a numeric constant is just a starting point for the type resolution algorithms. In most cases the constant will be automatically coerced to the most appropriate type depending on context. When necessary, you can force a numeric value to be interpreted as a specific data type by casting it. For example, you can force a numeric value to be treated as type `real` (`float4`) by writing

```
REAL '1.23'  -- string style
1.23::REAL   -- PostgreSQL (historical) style
```

These are actually just special cases of the general casting notations discussed next.

4.1.2.5. Constants of Other Types

A constant of an *arbitrary* type can be entered using any one of the following notations:

```
type 'string'
'string'::type
CAST ( 'string' AS type )
```

The string constant's text is passed to the input conversion routine for the type called `type`. The result is a constant of the indicated type. The explicit type cast may be omitted if there is no ambiguity as to the type the constant must be (for example, when it is assigned directly to a table column), in which case it is automatically coerced.

The string constant can be written using either regular SQL notation or dollar-quoting.

It is also possible to specify a type coercion using a function-like syntax:

```
typename ( 'string' )
```

but not all type names may be used in this way; see Section 4.2.8 for details.

The `::`, `CAST()`, and function-call syntaxes can also be used to specify run-time type conversions of arbitrary expressions, as discussed in Section 4.2.8. But the form `type 'string'` can only be used to specify the type of a literal constant. Another restriction on `type 'string'` is that it does not work for array types; use `::` or `CAST()` to specify the type of an array constant.

The `CAST()` syntax conforms to SQL. The `type 'string'` syntax is a generalization of the standard: SQL specifies this syntax only for a few data types, but PostgreSQL allows it for all types. The syntax with `::` is historical PostgreSQL usage, as is the function-call syntax.

4.1.3. Operators

An operator name is a sequence of up to `NAMEDATALEN`-1 (63 by default) characters from the following list:

+ - * / < > = ~ ! @ # % ^ & | ` ?

There are a few restrictions on operator names, however:

- `--` and `/*` cannot appear anywhere in an operator name, since they will be taken as the start of a comment.
- A multiple-character operator name cannot end in `+` or `-`, unless the name also contains at least one of these characters:

  ~ ! @ # % ^ & | ` ?

  For example, `@-` is an allowed operator name, but `*-` is not. This restriction allows PostgreSQL to parse SQL-compliant queries without requiring spaces between tokens.

When working with non-SQL-standard operator names, you will usually need to separate adjacent operators with spaces to avoid ambiguity. For example, if you have defined a left unary operator named `@`, you cannot write `X*@Y`; you must write `X* @Y` to ensure that PostgreSQL reads it as two operator names not one.

4.1.4. Special Characters

Some characters that are not alphanumeric have a special meaning that is different from being an operator. Details on the usage can be found at the location where the respective syntax element is described. This section only exists to advise the existence and summarize the purposes of these characters.

- A dollar sign (`$`) followed by digits is used to represent a positional parameter in the body of a function definition or a prepared statement. In other contexts the dollar sign may be part of an identifier or a dollar-quoted string constant.
- Parentheses (`()`) have their usual meaning to group expressions and enforce precedence. In some cases parentheses are required as part of the fixed syntax of a particular SQL command.
- Brackets (`[]`) are used to select the elements of an array. See Section 8.10 for more information on arrays.
- Commas (`,`) are used in some syntactical constructs to separate the elements of a list.
- The semicolon (`;`) terminates an SQL command. It cannot appear anywhere within a command, except within a string constant or quoted identifier.
- The colon (`:`) is used to select "slices" from arrays. (See Section 8.10.) In certain SQL dialects (such as Embedded SQL), the colon is used to prefix variable names.
- The asterisk (`*`) is used in some contexts to denote all the fields of a table row or composite value. It also has a special meaning when used as the argument of an aggregate function, namely that the aggregate does not require any explicit parameter.
- The period (`.`) is used in numeric constants, and to separate schema, table, and column names.

4.1.5. Comments

A comment is an arbitrary sequence of characters beginning with double dashes and extending to the end of the line, e.g.:

```
-- This is a standard SQL comment
```

Alternatively, C-style block comments can be used:

```
/* multiline comment
 * with nesting: /* nested block comment */
 */
```

where the comment begins with /* and extends to the matching occurrence of */. These block comments nest, as specified in the SQL standard but unlike C, so that one can comment out larger blocks of code that may contain existing block comments.

A comment is removed from the input stream before further syntax analysis and is effectively replaced by whitespace.

4.1.6. Lexical Precedence

Table 4-1 shows the precedence and associativity of the operators in PostgreSQL. Most operators have the same precedence and are left-associative. The precedence and associativity of the operators is hard-wired into the parser. This may lead to non-intuitive behavior; for example the Boolean operators < and > have a different precedence than the Boolean operators <= and >=. Also, you will sometimes need to add parentheses when using combinations of binary and unary operators. For instance

```
SELECT 5 ! - 6;
```

will be parsed as

```
SELECT 5 ! (- 6);
```

because the parser has no idea — until it is too late — that ! is defined as a postfix operator, not an infix one. To get the desired behavior in this case, you must write

```
SELECT (5 !) - 6;
```

This is the price one pays for extensibility.

Table 4-1. Operator Precedence (decreasing)

Operator/Element	Associativity	Description
.	left	table/column name separator
::	left	PostgreSQL-style typecast
[ ]	left	array element selection
-	right	unary minus
^	left	exponentiation
* / %	left	multiplication, division, modulo
+ -	left	addition, subtraction
IS		IS TRUE, IS FALSE, IS UNKNOWN, IS NULL
ISNULL		test for null
NOTNULL		test for not null
(any other)	left	all other native and user-defined operators
IN		set membership
BETWEEN		range containment

Operator/Element	Associativity	Description
OVERLAPS		time interval overlap
LIKE ILIKE SIMILAR		string pattern matching
< >		less than, greater than
=	right	equality, assignment
NOT	right	logical negation
AND	left	logical conjunction
OR	left	logical disjunction

Note that the operator precedence rules also apply to user-defined operators that have the same names as the built-in operators mentioned above. For example, if you define a "+" operator for some custom data type it will have the same precedence as the built-in "+" operator, no matter what yours does.

When a schema-qualified operator name is used in the OPERATOR syntax, as for example in

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

the OPERATOR construct is taken to have the default precedence shown in Table 4-1 for "any other" operator. This is true no matter which specific operator name appears inside OPERATOR().

4.2. Value Expressions

Value expressions are used in a variety of contexts, such as in the target list of the SELECT command, as new column values in INSERT or UPDATE, or in search conditions in a number of commands. The result of a value expression is sometimes called a *scalar*, to distinguish it from the result of a table expression (which is a table). Value expressions are therefore also called *scalar expressions* (or even simply *expressions*). The expression syntax allows the calculation of values from primitive parts using arithmetic, logical, set, and other operations.

A value expression is one of the following:

- A constant or literal value.
- A column reference.
- A positional parameter reference, in the body of a function definition or prepared statement.
- A subscripted expression.
- A field selection expression.
- An operator invocation.
- A function call.
- An aggregate expression.
- A type cast.
- A scalar subquery.
- An array constructor.

- A row constructor.
- Another value expression in parentheses, useful to group subexpressions and override precedence.

In addition to this list, there are a number of constructs that can be classified as an expression but do not follow any general syntax rules. These generally have the semantics of a function or operator and are explained in the appropriate location in Chapter 9. An example is the `IS NULL` clause.

We have already discussed constants in Section 4.1.2. The following sections discuss the remaining options.

4.2.1. Column References

A column can be referenced in the form

```
correlation.columnname
```

correlation is the name of a table (possibly qualified with a schema name), or an alias for a table defined by means of a `FROM` clause, or one of the key words `NEW` or `OLD`. (`NEW` and `OLD` can only appear in rewrite rules, while other correlation names can be used in any SQL statement.) The correlation name and separating dot may be omitted if the column name is unique across all the tables being used in the current query. (See also Chapter 7.)

4.2.2. Positional Parameters

A positional parameter reference is used to indicate a value that is supplied externally to an SQL statement. Parameters are used in SQL function definitions and in prepared queries. Some client libraries also support specifying data values separately from the SQL command string, in which case parameters are used to refer to the out-of-line data values. The form of a parameter reference is:

```
$number
```

For example, consider the definition of a function, `dept`, as

```
CREATE FUNCTION dept(text) RETURNS dept
    AS $$ SELECT * FROM dept WHERE name = $1 $$
    LANGUAGE SQL;
```

Here the `$1` references the value of the first function argument whenever the function is invoked.

4.2.3. Subscripts

If an expression yields a value of an array type, then a specific element of the array value can be extracted by writing

```
expression[subscript]
```

or multiple adjacent elements (an "array slice") can be extracted by writing

```
expression[lower_subscript:upper_subscript]
```

(Here, the brackets `[ ]` are meant to appear literally.) Each *subscript* is itself an expression, which must yield an integer value.

In general the array *expression* must be parenthesized, but the parentheses may be omitted when the expression to be subscripted is just a column reference or positional parameter. Also, multiple subscripts can be concatenated when the original array is multidimensional. For example,

```
mytable.arraycolumn[4]
mytable.two_d_column[17][34]
$1[10:42]
(arrayfunction(a,b))[42]
```

The parentheses in the last example are required. See Section 8.10 for more about arrays.

4.2.4. Field Selection

If an expression yields a value of a composite type (row type), then a specific field of the row can be extracted by writing

```
expression.fieldname
```

In general the row *expression* must be parenthesized, but the parentheses may be omitted when the expression to be selected from is just a table reference or positional parameter. For example,

```
mytable.mycolumn
$1.somecolumn
(rowfunction(a,b)).col3
```

(Thus, a qualified column reference is actually just a special case of the field selection syntax.)

4.2.5. Operator Invocations

There are three possible syntaxes for an operator invocation:

expression operator expression (binary infix operator)
operator expression (unary prefix operator)
expression operator (unary postfix operator)

where the *operator* token follows the syntax rules of Section 4.1.3, or is one of the key words AND, OR, and NOT, or is a qualified operator name in the form

```
OPERATOR(schema.operatorname)
```

Which particular operators exist and whether they are unary or binary depends on what operators have been defined by the system or the user. Chapter 9 describes the built-in operators.

4.2.6. Function Calls

The syntax for a function call is the name of a function (possibly qualified with a schema name), followed by its argument list enclosed in parentheses:

```
function ([expression [, expression ... ]] )
```

For example, the following computes the square root of 2:

```
sqrt(2)
```

The list of built-in functions is in Chapter 9. Other functions may be added by the user.

4.2.7. Aggregate Expressions

An *aggregate expression* represents the application of an aggregate function across the rows selected by a query. An aggregate function reduces multiple inputs to a single output value, such as the sum or average of the inputs. The syntax of an aggregate expression is one of the following:

```
aggregate_name (expression [ , ... ] )
aggregate_name (ALL expression [ , ... ] )
aggregate_name (DISTINCT expression [ , ... ] )
aggregate_name ( * )
```

where *aggregate_name* is a previously defined aggregate (possibly qualified with a schema name), and *expression* is any value expression that does not itself contain an aggregate expression.

The first form of aggregate expression invokes the aggregate across all input rows for which the given expression(s) yield non-null values. (Actually, it is up to the aggregate function whether to ignore null values or not — but all the standard ones do.) The second form is the same as the first, since ALL is the default. The third form invokes the aggregate for all distinct non-null values of the expressions found in the input rows. The last form invokes the aggregate once for each input row regardless of null or non-null values; since no particular input value is specified, it is generally only useful for the count(*) aggregate function.

For example, count(*) yields the total number of input rows; count(f1) yields the number of input rows in which f1 is non-null; count(distinct f1) yields the number of distinct non-null values of f1.

The predefined aggregate functions are described in Section 9.15. Other aggregate functions may be added by the user.

An aggregate expression may only appear in the result list or HAVING clause of a SELECT command. It is forbidden in other clauses, such as WHERE, because those clauses are logically evaluated before the results of aggregates are formed.

When an aggregate expression appears in a subquery (see Section 4.2.9 and Section 9.16), the aggregate is normally evaluated over the rows of the subquery. But an exception occurs if the aggregate's arguments contain only outer-level variables: the aggregate then belongs to the nearest such outer level, and is evaluated over the rows of that query. The aggregate expression as a whole is then an outer reference for the subquery it appears in, and acts as a constant over any one evaluation of that subquery. The restriction about appearing only in the result list or HAVING clause applies with respect to the query level that the aggregate belongs to.

Note: PostgreSQL currently does not support DISTINCT with more than one input expression.

4.2.8. Type Casts

A type cast specifies a conversion from one data type to another. PostgreSQL accepts two equivalent syntaxes for type casts:

```
CAST ( expression AS type )
expression::type
```

The CAST syntax conforms to SQL; the syntax with :: is historical PostgreSQL usage.

When a cast is applied to a value expression of a known type, it represents a run-time type conversion. The cast will succeed only if a suitable type conversion operation has been defined. Notice that this is subtly different from the use of casts with constants, as shown in Section 4.1.2.5. A cast applied to an unadorned string literal represents the initial assignment of a type to a literal constant value, and so it will succeed for any type (if the contents of the string literal are acceptable input syntax for the data type).

An explicit type cast may usually be omitted if there is no ambiguity as to the type that a value expression must produce (for example, when it is assigned to a table column); the system will automatically apply a type cast in such cases. However, automatic casting is only done for casts that are marked "OK to apply implicitly" in the system catalogs. Other casts must be invoked with explicit casting syntax. This restriction is intended to prevent surprising conversions from being applied silently.

It is also possible to specify a type cast using a function-like syntax:

```
typename ( expression )
```

However, this only works for types whose names are also valid as function names. For example, double precision can't be used this way, but the equivalent float8 can. Also, the names interval, time, and timestamp can only be used in this fashion if they are double-quoted, because of syntactic conflicts. Therefore, the use of the function-like cast syntax leads to inconsistencies and should probably be avoided in new applications. (The function-like syntax is in fact just a function call. When one of the two standard cast syntaxes is used to do a run-time conversion, it will internally invoke a registered function to perform

the conversion. By convention, these conversion functions have the same name as their output type, and thus the "function-like syntax" is nothing more than a direct invocation of the underlying conversion function. Obviously, this is not something that a portable application should rely on.)

4.2.9. Scalar Subqueries

A scalar subquery is an ordinary `SELECT` query in parentheses that returns exactly one row with one column. (See Chapter 7 for information about writing queries.) The `SELECT` query is executed and the single returned value is used in the surrounding value expression. It is an error to use a query that returns more than one row or more than one column as a scalar subquery. (But if, during a particular execution, the subquery returns no rows, there is no error; the scalar result is taken to be null.) The subquery can refer to variables from the surrounding query, which will act as constants during any one evaluation of the subquery. See also Section 9.16 for other expressions involving subqueries.

For example, the following finds the largest city population in each state:

```
SELECT name, (SELECT max(pop) FROM cities WHERE cities.state = states.name)
    FROM states;
```

4.2.10. Array Constructors

An array constructor is an expression that builds an array value from values for its member elements. A simple array constructor consists of the key word `ARRAY`, a left square bracket `[`, one or more expressions (separated by commas) for the array element values, and finally a right square bracket `]`. For example,

```
SELECT ARRAY[1,2,3+4];
  array
---------
 {1,2,7}
(1 row)
```

The array element type is the common type of the member expressions, determined using the same rules as for `UNION` or `CASE` constructs (see Section 10.5).

Multidimensional array values can be built by nesting array constructors. In the inner constructors, the key word `ARRAY` may be omitted. For example, these produce the same result:

```
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)

SELECT ARRAY[[1,2],[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)
```

Since multidimensional arrays must be rectangular, inner constructors at the same level must produce sub-arrays of identical dimensions.

Multidimensional array constructor elements can be anything yielding an array of the proper kind, not only a sub-ARRAY construct. For example:

```
CREATE TABLE arr(f1 int[], f2 int[]);

INSERT INTO arr VALUES (ARRAY[[1,2],[3,4]], ARRAY[[5,6],[7,8]]);

SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;
                     array
------------------------------------------------
 {{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}}
(1 row)
```

It is also possible to construct an array from the results of a subquery. In this form, the array constructor is written with the key word ARRAY followed by a parenthesized (not bracketed) subquery. For example:

```
SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE 'bytea%');
                           ?column?
------------------------------------------------------------
 {2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31}
(1 row)
```

The subquery must return a single column. The resulting one-dimensional array will have an element for each row in the subquery result, with an element type matching that of the subquery's output column.

The subscripts of an array value built with ARRAY always begin with one. For more information about arrays, see Section 8.10.

4.2.11. Row Constructors

A row constructor is an expression that builds a row value (also called a composite value) from values for its member fields. A row constructor consists of the key word ROW, a left parenthesis, zero or more expressions (separated by commas) for the row field values, and finally a right parenthesis. For example,

```
SELECT ROW(1,2.5,'this is a test');
```

The key word ROW is optional when there is more than one expression in the list.

A row constructor can include the syntax *rowvalue*.*, which will be expanded to a list of the elements of the row value, just as occurs when the .* syntax is used at the top level of a SELECT list. For example, if table t has columns f1 and f2, these are the same:

```
SELECT ROW(t.*, 42) FROM t;
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

> **Note:** Before PostgreSQL 8.2, the `.*` syntax was not expanded, so that writing `ROW(t.*, 42)` created a two-field row whose first field was another row value. The new behavior is usually more useful. If you need the old behavior of nested row values, write the inner row value without `.*`, for instance `ROW(t, 42)`.

By default, the value created by a `ROW` expression is of an anonymous record type. If necessary, it can be cast to a named composite type — either the row type of a table, or a composite type created with `CREATE TYPE AS`. An explicit cast may be needed to avoid ambiguity. For example:

```
CREATE TABLE mytable(f1 int, f2 float, f3 text);

CREATE FUNCTION getf1(mytable) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- No cast needed since only one getf1() exists
SELECT getf1(ROW(1,2.5,'this is a test'));
 getf1
-------
     1
(1 row)

CREATE TYPE myrowtype AS (f1 int, f2 text, f3 numeric);

CREATE FUNCTION getf1(myrowtype) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- Now we need a cast to indicate which function to call:
SELECT getf1(ROW(1,2.5,'this is a test'));
ERROR:  function getf1(record) is not unique

SELECT getf1(ROW(1,2.5,'this is a test')::mytable);
 getf1
-------
     1
(1 row)

SELECT getf1(CAST(ROW(11,'this is a test',2.5) AS myrowtype));
 getf1
-------
    11
(1 row)
```

Row constructors can be used to build composite values to be stored in a composite-type table column, or to be passed to a function that accepts a composite parameter. Also, it is possible to compare two row values or test a row with `IS NULL` or `IS NOT NULL`, for example

```
SELECT ROW(1,2.5,'this is a test') = ROW(1, 3, 'not the same');

SELECT ROW(table.*) IS NULL FROM table;  -- detect all-null rows
```

For more detail see Section 9.17. Row constructors can also be used in connection with subqueries, as discussed in Section 9.16.

4.2.12. Expression Evaluation Rules

The order of evaluation of subexpressions is not defined. In particular, the inputs of an operator or function are not necessarily evaluated left-to-right or in any other fixed order.

Furthermore, if the result of an expression can be determined by evaluating only some parts of it, then other subexpressions might not be evaluated at all. For instance, if one wrote

```
SELECT true OR somefunc();
```

then `somefunc()` would (probably) not be called at all. The same would be the case if one wrote

```
SELECT somefunc() OR true;
```

Note that this is not the same as the left-to-right "short-circuiting" of Boolean operators that is found in some programming languages.

As a consequence, it is unwise to use functions with side effects as part of complex expressions. It is particularly dangerous to rely on side effects or evaluation order in `WHERE` and `HAVING` clauses, since those clauses are extensively reprocessed as part of developing an execution plan. Boolean expressions (`AND`/`OR`/`NOT` combinations) in those clauses may be reorganized in any manner allowed by the laws of Boolean algebra.

When it is essential to force evaluation order, a `CASE` construct (see Section 9.13) may be used. For example, this is an untrustworthy way of trying to avoid division by zero in a `WHERE` clause:

```
SELECT ... WHERE x <> 0 AND y/x > 1.5;
```

But this is safe:

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

A `CASE` construct used in this fashion will defeat optimization attempts, so it should only be done when necessary. (In this particular example, it would doubtless be best to sidestep the problem by writing `y > 1.5*x` instead.)

Chapter 5. Data Definition

This chapter covers how one creates the database structures that will hold one's data. In a relational database, the raw data is stored in tables, so the majority of this chapter is devoted to explaining how tables are created and modified and what features are available to control what data is stored in the tables. Subsequently, we discuss how tables can be organized into schemas, and how privileges can be assigned to tables. Finally, we will briefly look at other features that affect the data storage, such as inheritance, views, functions, and triggers.

5.1. Table Basics

A table in a relational database is much like a table on paper: It consists of rows and columns. The number and order of the columns is fixed, and each column has a name. The number of rows is variable — it reflects how much data is stored at a given moment. SQL does not make any guarantees about the order of the rows in a table. When a table is read, the rows will appear in random order, unless sorting is explicitly requested. This is covered in Chapter 7. Furthermore, SQL does not assign unique identifiers to rows, so it is possible to have several completely identical rows in a table. This is a consequence of the mathematical model that underlies SQL but is usually not desirable. Later in this chapter we will see how to deal with this issue.

Each column has a data type. The data type constrains the set of possible values that can be assigned to a column and assigns semantics to the data stored in the column so that it can be used for computations. For instance, a column declared to be of a numerical type will not accept arbitrary text strings, and the data stored in such a column can be used for mathematical computations. By contrast, a column declared to be of a character string type will accept almost any kind of data but it does not lend itself to mathematical calculations, although other operations such as string concatenation are available.

PostgreSQL includes a sizable set of built-in data types that fit many applications. Users can also define their own data types. Most built-in data types have obvious names and semantics, so we defer a detailed explanation to Chapter 8. Some of the frequently used data types are `integer` for whole numbers, `numeric` for possibly fractional numbers, `text` for character strings, `date` for dates, `time` for time-of-day values, and `timestamp` for values containing both date and time.

To create a table, you use the aptly named *CREATE TABLE* command. In this command you specify at least a name for the new table, the names of the columns and the data type of each column. For example:

```
CREATE TABLE my_first_table (
    first_column text,
    second_column integer
);
```

This creates a table named `my_first_table` with two columns. The first column is named `first_column` and has a data type of `text`; the second column has the name `second_column` and the type `integer`. The table and column names follow the identifier syntax explained in Section 4.1.1. The type names are usually also identifiers, but there are some exceptions. Note that the column list is comma-separated and surrounded by parentheses.

Of course, the previous example was heavily contrived. Normally, you would give names to your tables and columns that convey what kind of data they store. So let's look at a more realistic example:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);
```

(The `numeric` type can store fractional components, as would be typical of monetary amounts.)

> **Tip:** When you create many interrelated tables it is wise to choose a consistent naming pattern for the tables and columns. For instance, there is a choice of using singular or plural nouns for table names, both of which are favored by some theorist or other.

There is a limit on how many columns a table can contain. Depending on the column types, it is between 250 and 1600. However, defining a table with anywhere near this many columns is highly unusual and often a questionable design.

If you no longer need a table, you can remove it using the *DROP TABLE* command. For example:

```
DROP TABLE my_first_table;
DROP TABLE products;
```

Attempting to drop a table that does not exist is an error. Nevertheless, it is common in SQL script files to unconditionally try to drop each table before creating it, ignoring any error messages, so that the script works whether or not the table exists. (If you like, you can use the `DROP TABLE IF EXISTS` variant to avoid the error messages, but this is not standard SQL.)

If you need to modify a table that already exists look into Section 5.5 later in this chapter.

With the tools discussed so far you can create fully functional tables. The remainder of this chapter is concerned with adding features to the table definition to ensure data integrity, security, or convenience. If you are eager to fill your tables with data now you can skip ahead to Chapter 6 and read the rest of this chapter later.

5.2. Default Values

A column can be assigned a default value. When a new row is created and no values are specified for some of the columns, those columns will be filled with their respective default values. A data manipulation command can also request explicitly that a column be set to its default value, without having to know what that value is. (Details about data manipulation commands are in Chapter 6.)

If no default value is declared explicitly, the default value is the null value. This usually makes sense because a null value can be considered to represent unknown data.

In a table definition, default values are listed after the column data type. For example:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric DEFAULT 9.99
);
```

The default value may be an expression, which will be evaluated whenever the default value is inserted (*not* when the table is created). A common example is that a `timestamp` column may have a default of `now()`, so that it gets set to the time of row insertion. Another common example is generating a "serial number" for each row. In PostgreSQL this is typically done by something like

```
CREATE TABLE products (
    product_no integer DEFAULT nextval('products_product_no_seq'),
    ...
);
```

where the `nextval()` function supplies successive values from a *sequence object* (see Section 9.12). This arrangement is sufficiently common that there's a special shorthand for it:

```
CREATE TABLE products (
    product_no SERIAL,
    ...
);
```

The `SERIAL` shorthand is discussed further in Section 8.1.4.

5.3. Constraints

Data types are a way to limit the kind of data that can be stored in a table. For many applications, however, the constraint they provide is too coarse. For example, a column containing a product price should probably only accept positive values. But there is no standard data type that accepts only positive numbers. Another issue is that you might want to constrain column data with respect to other columns or rows. For example, in a table containing product information, there should be only one row for each product number.

To that end, SQL allows you to define constraints on columns and tables. Constraints give you as much control over the data in your tables as you wish. If a user attempts to store data in a column that would violate a constraint, an error is raised. This applies even if the value came from the default value definition.

5.3.1. Check Constraints

A check constraint is the most generic constraint type. It allows you to specify that the value in a certain column must satisfy a Boolean (truth-value) expression. For instance, to require positive product prices, you could use:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0)
);
```

As you see, the constraint definition comes after the data type, just like default value definitions. Default values and constraints can be listed in any order. A check constraint consists of the key word CHECK followed by an expression in parentheses. The check constraint expression should involve the column thus constrained, otherwise the constraint would not make too much sense.

You can also give the constraint a separate name. This clarifies error messages and allows you to refer to the constraint when you need to change it. The syntax is:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CONSTRAINT positive_price CHECK (price > 0)
);
```

So, to specify a named constraint, use the key word CONSTRAINT followed by an identifier followed by the constraint definition. (If you don't specify a constraint name in this way, the system chooses a name for you.)

A check constraint can also refer to several columns. Say you store a regular price and a discounted price, and you want to ensure that the discounted price is lower than the regular price.

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

The first two constraints should look familiar. The third one uses a new syntax. It is not attached to a particular column, instead it appears as a separate item in the comma-separated column list. Column definitions and these constraint definitions can be listed in mixed order.

We say that the first two constraints are column constraints, whereas the third one is a table constraint because it is written separately from any one column definition. Column constraints can also be written as table constraints, while the reverse is not necessarily possible, since a column constraint is supposed to refer to only the column it is attached to. (PostgreSQL doesn't enforce that rule, but you should follow it if you want your table definitions to work with other database systems.) The above example could also be written as

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

or even

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0 AND price > discounted_price)
);
```

It's a matter of taste.

Names can be assigned to table constraints in just the same way as for column constraints:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CONSTRAINT valid_discount CHECK (price > discounted_price)
);
```

It should be noted that a check constraint is satisfied if the check expression evaluates to true or the null value. Since most expressions will evaluate to the null value if any operand is null, they will not prevent null values in the constrained columns. To ensure that a column does not contain null values, the not-null constraint described in the next section can be used.

5.3.2. Not-Null Constraints

A not-null constraint simply specifies that a column must not assume the null value. A syntax example:

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric
);
```

A not-null constraint is always written as a column constraint. A not-null constraint is functionally equivalent to creating a check constraint `CHECK (column_name IS NOT NULL)`, but in PostgreSQL creating an explicit not-null constraint is more efficient. The drawback is that you cannot give explicit names to not-null constraints created this way.

Of course, a column can have more than one constraint. Just write the constraints one after another:

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric NOT NULL CHECK (price > 0)
);
```

The order doesn't matter. It does not necessarily determine in which order the constraints are checked.

The `NOT NULL` constraint has an inverse: the `NULL` constraint. This does not mean that the column must be null, which would surely be useless. Instead, this simply selects the default behavior that the column may be null. The `NULL` constraint is not present in the SQL standard and should not be used in portable applications. (It was only added to PostgreSQL to be compatible with some other database systems.) Some users, however, like it because it makes it easy to toggle the constraint in a script file. For example, you could start with

```
CREATE TABLE products (
    product_no integer NULL,
    name text NULL,
    price numeric NULL
);
```

and then insert the `NOT` key word where desired.

> **Tip:** In most database designs the majority of columns should be marked not null.

5.3.3. Unique Constraints

Unique constraints ensure that the data contained in a column or a group of columns is unique with respect to all the rows in the table. The syntax is

```
CREATE TABLE products (
    product_no integer UNIQUE,
    name text,
    price numeric
);
```

when written as a column constraint, and

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    UNIQUE (product_no)
);
```

when written as a table constraint.

If a unique constraint refers to a group of columns, the columns are listed separated by commas:

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    UNIQUE (a, c)
);
```

This specifies that the combination of values in the indicated columns is unique across the whole table, though any one of the columns need not be (and ordinarily isn't) unique.

You can assign your own name for a unique constraint, in the usual way:

```
CREATE TABLE products (
    product_no integer CONSTRAINT must_be_different UNIQUE,
    name text,
    price numeric
);
```

In general, a unique constraint is violated when there are two or more rows in the table where the values of all of the columns included in the constraint are equal. However, two null values are not considered equal in this comparison. That means even in the presence of a unique constraint it is possible to store duplicate rows that contain a null value in at least one of the constrained columns. This behavior conforms to the SQL standard, but we have heard that other SQL databases may not follow this rule. So be careful when developing applications that are intended to be portable.

5.3.4. Primary Keys

Technically, a primary key constraint is simply a combination of a unique constraint and a not-null constraint. So, the following two table definitions accept the same data:

```
CREATE TABLE products (
    product_no integer UNIQUE NOT NULL,
    name text,
    price numeric
);

CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

Primary keys can also constrain more than one column; the syntax is similar to unique constraints:

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    PRIMARY KEY (a, c)
);
```

A primary key indicates that a column or group of columns can be used as a unique identifier for rows in the table. (This is a direct consequence of the definition of a primary key. Note that a unique constraint does not, by itself, provide a unique identifier because it does not exclude null values.) This is useful

both for documentation purposes and for client applications. For example, a GUI application that allows modifying row values probably needs to know the primary key of a table to be able to identify rows uniquely.

A table can have at most one primary key. (There can be any number of unique and not-null constraints, which are functionally the same thing, but only one can be identified as the primary key.) Relational database theory dictates that every table must have a primary key. This rule is not enforced by PostgreSQL, but it is usually best to follow it.

5.3.5. Foreign Keys

A foreign key constraint specifies that the values in a column (or a group of columns) must match the values appearing in some row of another table. We say this maintains the *referential integrity* between two related tables.

Say you have the product table that we have used several times already:

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

Let's also assume you have a table storing orders of those products. We want to ensure that the orders table only contains orders of products that actually exist. So we define a foreign key constraint in the orders table that references the products table:

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products (product_no),
    quantity integer
);
```

Now it is impossible to create orders with `product_no` entries that do not appear in the products table.

We say that in this situation the orders table is the *referencing* table and the products table is the *referenced* table. Similarly, there are referencing and referenced columns.

You can also shorten the above command to

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products,
    quantity integer
);
```

because in absence of a column list the primary key of the referenced table is used as the referenced column(s).

A foreign key can also constrain and reference a group of columns. As usual, it then needs to be written in table constraint form. Here is a contrived syntax example:

```
CREATE TABLE t1 (
```

```
  a integer PRIMARY KEY,
  b integer,
  c integer,
  FOREIGN KEY (b, c) REFERENCES other_table (c1, c2)
);
```

Of course, the number and type of the constrained columns need to match the number and type of the referenced columns.

You can assign your own name for a foreign key constraint, in the usual way.

A table can contain more than one foreign key constraint. This is used to implement many-to-many relationships between tables. Say you have tables about products and orders, but now you want to allow one order to contain possibly many products (which the structure above did not allow). You could use this table structure:

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
    ...
);

CREATE TABLE order_items (
    product_no integer REFERENCES products,
    order_id integer REFERENCES orders,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

Notice that the primary key overlaps with the foreign keys in the last table.

We know that the foreign keys disallow creation of orders that do not relate to any products. But what if a product is removed after an order is created that references it? SQL allows you to handle that as well. Intuitively, we have a few options:

- Disallow deleting a referenced product
- Delete the orders as well
- Something else?

To illustrate this, let's implement the following policy on the many-to-many relationship example above: when someone wants to remove a product that is still referenced by an order (via `order_items`), we disallow it. If someone removes an order, the order items are removed as well.

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
```

```
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
    ...
);

CREATE TABLE order_items (
    product_no integer REFERENCES products ON DELETE RESTRICT,
    order_id integer REFERENCES orders ON DELETE CASCADE,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

Restricting and cascading deletes are the two most common options. `RESTRICT` prevents deletion of a referenced row. `NO ACTION` means that if any referencing rows still exist when the constraint is checked, an error is raised; this is the default behavior if you do not specify anything. (The essential difference between these two choices is that `NO ACTION` allows the check to be deferred until later in the transaction, whereas `RESTRICT` does not.) `CASCADE` specifies that when a referenced row is deleted, row(s) referencing it should be automatically deleted as well. There are two other options: `SET NULL` and `SET DEFAULT`. These cause the referencing columns to be set to nulls or default values, respectively, when the referenced row is deleted. Note that these do not excuse you from observing any constraints. For example, if an action specifies `SET DEFAULT` but the default value would not satisfy the foreign key, the operation will fail.

Analogous to `ON DELETE` there is also `ON UPDATE` which is invoked when a referenced column is changed (updated). The possible actions are the same.

More information about updating and deleting data is in Chapter 6.

Finally, we should mention that a foreign key must reference columns that either are a primary key or form a unique constraint. If the foreign key references a unique constraint, there are some additional possibilities regarding how null values are matched. These are explained in the reference documentation for *CREATE TABLE*.

5.4. System Columns

Every table has several *system columns* that are implicitly defined by the system. Therefore, these names cannot be used as names of user-defined columns. (Note that these restrictions are separate from whether the name is a key word or not; quoting a name will not allow you to escape these restrictions.) You do not really need to be concerned about these columns, just know they exist.

`oid`

The object identifier (object ID) of a row. This column is only present if the table was created using `WITH OIDS`, or if the default_with_oids configuration variable was set at the time. This column is of type `oid` (same name as the column); see Section 8.12 for more information about the type.

`tableoid`

The OID of the table containing this row. This column is particularly handy for queries that select from inheritance hierarchies (see Section 5.8), since without it, it's difficult to tell which individual table a row came from. The `tableoid` can be joined against the `oid` column of `pg_class` to obtain the table name.

`xmin`

The identity (transaction ID) of the inserting transaction for this row version. (A row version is an individual state of a row; each update of a row creates a new row version for the same logical row.)

`cmin`

The command identifier (starting at zero) within the inserting transaction.

`xmax`

The identity (transaction ID) of the deleting transaction, or zero for an undeleted row version. It is possible for this column to be nonzero in a visible row version. That usually indicates that the deleting transaction hasn't committed yet, or that an attempted deletion was rolled back.

`cmax`

The command identifier within the deleting transaction, or zero.

`ctid`

The physical location of the row version within its table. Note that although the `ctid` can be used to locate the row version very quickly, a row's `ctid` will change each time it is updated or moved by `VACUUM FULL`. Therefore `ctid` is useless as a long-term row identifier. The OID, or even better a user-defined serial number, should be used to identify logical rows.

OIDs are 32-bit quantities and are assigned from a single cluster-wide counter. In a large or long-lived database, it is possible for the counter to wrap around. Hence, it is bad practice to assume that OIDs are unique, unless you take steps to ensure that this is the case. If you need to identify the rows in a table, using a sequence generator is strongly recommended. However, OIDs can be used as well, provided that a few additional precautions are taken:

- A unique constraint should be created on the OID column of each table for which the OID will be used to identify rows. When such a unique constraint (or unique index) exists, the system takes care not to generate an OID matching an already-existing row. (Of course, this is only possible if the table contains fewer than 2^{32} (4 billion) rows, and in practice the table size had better be much less than that, or performance may suffer.)
- OIDs should never be assumed to be unique across tables; use the combination of `tableoid` and row OID if you need a database-wide identifier.
- Of course, the tables in question must be created `WITH OIDS`. As of PostgreSQL 8.1, `WITHOUT OIDS` is the default.

Transaction identifiers are also 32-bit quantities. In a long-lived database it is possible for transaction IDs to wrap around. This is not a fatal problem given appropriate maintenance procedures; see Chapter 22 for details. It is unwise, however, to depend on the uniqueness of transaction IDs over the long term (more than one billion transactions).

Command identifiers are also 32-bit quantities. This creates a hard limit of 2^{32} (4 billion) SQL commands within a single transaction. In practice this limit is not a problem — note that the limit is on number of SQL commands, not number of rows processed.

5.5. Modifying Tables

When you create a table and you realize that you made a mistake, or the requirements of the application change, then you can drop the table and create it again. But this is not a convenient option if the table is already filled with data, or if the table is referenced by other database objects (for instance a foreign key constraint). Therefore PostgreSQL provides a family of commands to make modifications to existing tables. Note that this is conceptually distinct from altering the data contained in the table: here we are interested in altering the definition, or structure, of the table.

You can

- Add columns,
- Remove columns,
- Add constraints,
- Remove constraints,
- Change default values,
- Change column data types,
- Rename columns,
- Rename tables.

All these actions are performed using the *ALTER TABLE* command, whose reference page contains details beyond those given here.

5.5.1. Adding a Column

To add a column, use a command like this:

```
ALTER TABLE products ADD COLUMN description text;
```

The new column is initially filled with whatever default value is given (null if you don't specify a `DEFAULT` clause).

You can also define constraints on the column at the same time, using the usual syntax:

```
ALTER TABLE products ADD COLUMN description text CHECK (description <> '');
```

In fact all the options that can be applied to a column description in `CREATE TABLE` can be used here. Keep in mind however that the default value must satisfy the given constraints, or the `ADD` will fail. Alternatively, you can add constraints later (see below) after you've filled in the new column correctly.

> **Tip:** Adding a column with a default requires updating each row of the table (to store the new column value). However, if no default is specified, PostgreSQL is able to avoid the physical update. So if you intend to fill the column with mostly nondefault values, it's best to add the column with no default, insert the correct values using `UPDATE`, and then add any desired default as described below.

5.5.2. Removing a Column

To remove a column, use a command like this:

```
ALTER TABLE products DROP COLUMN description;
```

Whatever data was in the column disappears. Table constraints involving the column are dropped, too. However, if the column is referenced by a foreign key constraint of another table, PostgreSQL will not silently drop that constraint. You can authorize dropping everything that depends on the column by adding `CASCADE`:

```
ALTER TABLE products DROP COLUMN description CASCADE;
```

See Section 5.11 for a description of the general mechanism behind this.

5.5.3. Adding a Constraint

To add a constraint, the table constraint syntax is used. For example:

```
ALTER TABLE products ADD CHECK (name <> '');
ALTER TABLE products ADD CONSTRAINT some_name UNIQUE (product_no);
ALTER TABLE products ADD FOREIGN KEY (product_group_id) REFERENCES product_groups;
```

To add a not-null constraint, which cannot be written as a table constraint, use this syntax:

```
ALTER TABLE products ALTER COLUMN product_no SET NOT NULL;
```

The constraint will be checked immediately, so the table data must satisfy the constraint before it can be added.

5.5.4. Removing a Constraint

To remove a constraint you need to know its name. If you gave it a name then that's easy. Otherwise the system assigned a generated name, which you need to find out. The psql command `\d` *`tablename`* can be helpful here; other interfaces might also provide a way to inspect table details. Then the command is:

```
ALTER TABLE products DROP CONSTRAINT some_name;
```

(If you are dealing with a generated constraint name like `$2`, don't forget that you'll need to double-quote it to make it a valid identifier.)

As with dropping a column, you need to add `CASCADE` if you want to drop a constraint that something else depends on. An example is that a foreign key constraint depends on a unique or primary key constraint on the referenced column(s).

This works the same for all constraint types except not-null constraints. To drop a not null constraint use

```
ALTER TABLE products ALTER COLUMN product_no DROP NOT NULL;
```

(Recall that not-null constraints do not have names.)

5.5.5. Changing a Column's Default Value

To set a new default for a column, use a command like this:

```
ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;
```

Note that this doesn't affect any existing rows in the table, it just changes the default for future `INSERT` commands.

To remove any default value, use

```
ALTER TABLE products ALTER COLUMN price DROP DEFAULT;
```

This is effectively the same as setting the default to null. As a consequence, it is not an error to drop a default where one hadn't been defined, because the default is implicitly the null value.

5.5.6. Changing a Column's Data Type

To convert a column to a different data type, use a command like this:

```
ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);
```

This will succeed only if each existing entry in the column can be converted to the new type by an implicit cast. If a more complex conversion is needed, you can add a `USING` clause that specifies how to compute the new values from the old.

PostgreSQL will attempt to convert the column's default value (if any) to the new type, as well as any constraints that involve the column. But these conversions may fail, or may produce surprising results. It's often best to drop any constraints on the column before altering its type, and then add back suitably modified constraints afterwards.

5.5.7. Renaming a Column

To rename a column:

```
ALTER TABLE products RENAME COLUMN product_no TO product_number;
```

5.5.8. Renaming a Table

To rename a table:

```
ALTER TABLE products RENAME TO items;
```

5.6. Privileges

When you create a database object, you become its owner. By default, only the owner of an object can do anything with the object. In order to allow other users to use it, *privileges* must be granted. (However, users that have the superuser attribute can always access any object.)

There are several different privileges: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `REFERENCES`, `TRIGGER`, `CREATE`, `CONNECT`, `TEMPORARY`, `EXECUTE`, and `USAGE`. The privileges applicable to a particular object vary depending on the object's type (table, function, etc). For complete information on the different types of privileges supported by PostgreSQL, refer to the *GRANT* reference page. The following sections and chapters will also show you how those privileges are used.

The right to modify or destroy an object is always the privilege of the owner only.

> **Note:** To change the owner of a table, index, sequence, or view, use the *ALTER TABLE* command. There are corresponding `ALTER` commands for other object types.

To assign privileges, the `GRANT` command is used. For example, if `joe` is an existing user, and `accounts` is an existing table, the privilege to update the table can be granted with

```
GRANT UPDATE ON accounts TO joe;
```

Writing `ALL` in place of a specific privilege grants all privileges that are relevant for the object type.

The special "user" name `PUBLIC` can be used to grant a privilege to every user on the system. Also, "group" roles can be set up to help manage privileges when there are many users of a database — for details see Chapter 18.

To revoke a privilege, use the fittingly named `REVOKE` command:

```
REVOKE ALL ON accounts FROM PUBLIC;
```

The special privileges of the object owner (i.e., the right to do `DROP`, `GRANT`, `REVOKE`, etc.) are always implicit in being the owner, and cannot be granted or revoked. But the object owner can choose to revoke his own ordinary privileges, for example to make a table read-only for himself as well as others.

Ordinarily, only the object's owner (or a superuser) can grant or revoke privileges on an object. However, it is possible to grant a privilege "with grant option", which gives the recipient the right to grant it in turn to others. If the grant option is subsequently revoked then all who received the privilege from that recipient (directly or through a chain of grants) will lose the privilege. For details see the *GRANT* and *REVOKE* reference pages.

5.7. Schemas

A PostgreSQL database cluster contains one or more named databases. Users and groups of users are shared across the entire cluster, but no other data is shared across databases. Any given client connection to the server can access only the data in a single database, the one specified in the connection request.

> **Note:** Users of a cluster do not necessarily have the privilege to access every database in the cluster. Sharing of user names means that there cannot be different users named, say, `joe` in two databases in the same cluster; but the system can be configured to allow `joe` access to only some of the databases.

A database contains one or more named *schemas*, which in turn contain tables. Schemas also contain other kinds of named objects, including data types, functions, and operators. The same object name can be used in different schemas without conflict; for example, both `schema1` and `myschema` may contain tables named `mytable`. Unlike databases, schemas are not rigidly separated: a user may access objects in any of the schemas in the database he is connected to, if he has privileges to do so.

There are several reasons why one might want to use schemas:

- To allow many users to use one database without interfering with each other.
- To organize database objects into logical groups to make them more manageable.
- Third-party applications can be put into separate schemas so they cannot collide with the names of other objects.

Schemas are analogous to directories at the operating system level, except that schemas cannot be nested.

5.7.1. Creating a Schema

To create a schema, use the *CREATE SCHEMA* command. Give the schema a name of your choice. For example:

```
CREATE SCHEMA myschema;
```

To create or access objects in a schema, write a *qualified name* consisting of the schema name and table name separated by a dot:

```
schema.table
```

This works anywhere a table name is expected, including the table modification commands and the data access commands discussed in the following chapters. (For brevity we will speak of tables only, but the same ideas apply to other kinds of named objects, such as types and functions.)

Actually, the even more general syntax

```
database.schema.table
```

can be used too, but at present this is just for *pro forma* compliance with the SQL standard. If you write a database name, it must be the same as the database you are connected to.

So to create a table in the new schema, use

```
CREATE TABLE myschema.mytable (
 ...
);
```

To drop a schema if it's empty (all objects in it have been dropped), use

```
DROP SCHEMA myschema;
```

To drop a schema including all contained objects, use

```
DROP SCHEMA myschema CASCADE;
```

See Section 5.11 for a description of the general mechanism behind this.

Often you will want to create a schema owned by someone else (since this is one of the ways to restrict the activities of your users to well-defined namespaces). The syntax for that is:

```
CREATE SCHEMA schemaname AUTHORIZATION username;
```

You can even omit the schema name, in which case the schema name will be the same as the user name. See Section 5.7.6 for how this can be useful.

Schema names beginning with `pg_` are reserved for system purposes and may not be created by users.

5.7.2. The Public Schema

In the previous sections we created tables without specifying any schema names. By default, such tables (and other objects) are automatically put into a schema named "public". Every new database contains such a schema. Thus, the following are equivalent:

```
CREATE TABLE products ( ... );
```

and

```
CREATE TABLE public.products ( ... );
```

5.7.3. The Schema Search Path

Qualified names are tedious to write, and it's often best not to wire a particular schema name into applications anyway. Therefore tables are often referred to by *unqualified names*, which consist of just the table name. The system determines which table is meant by following a *search path*, which is a list of schemas to look in. The first matching table in the search path is taken to be the one wanted. If there is no match in the search path, an error is reported, even if matching table names exist in other schemas in the database.

The first schema named in the search path is called the current schema. Aside from being the first schema searched, it is also the schema in which new tables will be created if the `CREATE TABLE` command does not specify a schema name.

To show the current search path, use the following command:

```
SHOW search_path;
```

In the default setup this returns:

```
 search_path
--------------
 "$user",public
```

The first element specifies that a schema with the same name as the current user is to be searched. If no such schema exists, the entry is ignored. The second element refers to the public schema that we have seen already.

The first schema in the search path that exists is the default location for creating new objects. That is the reason that by default objects are created in the public schema. When objects are referenced in any other context without schema qualification (table modification, data modification, or query commands) the search path is traversed until a matching object is found. Therefore, in the default configuration, any unqualified access again can only refer to the public schema.

To put our new schema in the path, we use

```
SET search_path TO myschema,public;
```

(We omit the `$user` here because we have no immediate need for it.) And then we can access the table without schema qualification:

```
DROP TABLE mytable;
```

Also, since `myschema` is the first element in the path, new objects would by default be created in it.

We could also have written

```
SET search_path TO myschema;
```

Then we no longer have access to the public schema without explicit qualification. There is nothing special about the public schema except that it exists by default. It can be dropped, too.

See also Section 9.19 for other ways to manipulate the schema search path.

The search path works in the same way for data type names, function names, and operator names as it does for table names. Data type and function names can be qualified in exactly the same way as table names. If you need to write a qualified operator name in an expression, there is a special provision: you must write

```
OPERATOR(schema.operator)
```

This is needed to avoid syntactic ambiguity. An example is

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

In practice one usually relies on the search path for operators, so as not to have to write anything so ugly as that.

5.7.4. Schemas and Privileges

By default, users cannot access any objects in schemas they do not own. To allow that, the owner of the schema needs to grant the `USAGE` privilege on the schema. To allow users to make use of the objects in the schema, additional privileges may need to be granted, as appropriate for the object.

A user can also be allowed to create objects in someone else's schema. To allow that, the `CREATE` privilege on the schema needs to be granted. Note that by default, everyone has `CREATE` and `USAGE` privileges on the schema `public`. This allows all users that are able to connect to a given database to create objects in its `public` schema. If you do not want to allow that, you can revoke that privilege:

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

(The first "public" is the schema, the second "public" means "every user". In the first sense it is an identifier, in the second sense it is a key word, hence the different capitalization; recall the guidelines from Section 4.1.1.)

5.7.5. The System Catalog Schema

In addition to `public` and user-created schemas, each database contains a `pg_catalog` schema, which contains the system tables and all the built-in data types, functions, and operators. `pg_catalog` is always effectively part of the search path. If it is not named explicitly in the path then it is implicitly searched *before* searching the path's schemas. This ensures that built-in names will always be findable. However, you may explicitly place `pg_catalog` at the end of your search path if you prefer to have user-defined names override built-in names.

In PostgreSQL versions before 7.3, table names beginning with `pg_` were reserved. This is no longer true: you may create such a table name if you wish, in any non-system schema. However, it's best to continue to avoid such names, to ensure that you won't suffer a conflict if some future version defines a system table named the same as your table. (With the default search path, an unqualified reference to your table name would be resolved as the system table instead.) System tables will continue to follow the convention of having names beginning with `pg_`, so that they will not conflict with unqualified user-table names so long as users avoid the `pg_` prefix.

5.7.6. Usage Patterns

Schemas can be used to organize your data in many ways. There are a few usage patterns that are recommended and are easily supported by the default configuration:

- If you do not create any schemas then all users access the public schema implicitly. This simulates the situation where schemas are not available at all. This setup is mainly recommended when there is only a single user or a few cooperating users in a database. This setup also allows smooth transition from the non-schema-aware world.

- You can create a schema for each user with the same name as that user. Recall that the default search path starts with `$user`, which resolves to the user name. Therefore, if each user has a separate schema, they access their own schemas by default.

  If you use this setup then you might also want to revoke access to the public schema (or drop it altogether), so users are truly constrained to their own schemas.

- To install shared applications (tables to be used by everyone, additional functions provided by third parties, etc.), put them into separate schemas. Remember to grant appropriate privileges to allow the other users to access them. Users can then refer to these additional objects by qualifying the names with a schema name, or they can put the additional schemas into their search path, as they choose.

5.7.7. Portability

In the SQL standard, the notion of objects in the same schema being owned by different users does not exist. Moreover, some implementations do not allow you to create schemas that have a different name than their owner. In fact, the concepts of schema and user are nearly equivalent in a database system that implements only the basic schema support specified in the standard. Therefore, many users consider qualified names to really consist of *`username`*`.`*`tablename`*. This is how PostgreSQL will effectively behave if you create a per-user schema for every user.

Also, there is no concept of a `public` schema in the SQL standard. For maximum conformance to the standard, you should not use (perhaps even remove) the `public` schema.

Of course, some SQL database systems might not implement schemas at all, or provide namespace support by allowing (possibly limited) cross-database access. If you need to work with those systems, then maximum portability would be achieved by not using schemas at all.

5.8. Inheritance

PostgreSQL implements table inheritance, which can be a useful tool for database designers. (SQL:1999 and later define a type inheritance feature, which differs in many respects from the features described here.)

Let's start with an example: suppose we are trying to build a data model for cities. Each state has many cities, but only one capital. We want to be able to quickly retrieve the capital city for any particular state. This can be done by creating two tables, one for state capitals and one for cities that are not capitals. However, what happens when we want to ask for data about a city, regardless of whether it is a capital or not? The inheritance feature can help to resolve this problem. We define the `capitals` table so that it inherits from `cities`:

```
CREATE TABLE cities (
    name            text,
    population      float,
    altitude        int     -- in feet
);
```

```
CREATE TABLE capitals (
    state           char(2)
) INHERITS (cities);
```

In this case, the `capitals` table *inherits* all the columns of its parent table, `cities`. State capitals also have an extra column, `state`, that shows their state.

In PostgreSQL, a table can inherit from zero or more other tables, and a query can reference either all rows of a table or all rows of a table plus all of its descendant tables. The latter behavior is the default. For example, the following query finds the names of all cities, including state capitals, that are located at an altitude over 500ft:

```
SELECT name, altitude
    FROM cities
    WHERE altitude > 500;
```

Given the sample data from the PostgreSQL tutorial (see Section 2.1), this returns:

```
   name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
```

On the other hand, the following query finds all the cities that are not state capitals and are situated at an altitude over 500ft:

```
SELECT name, altitude
    FROM ONLY cities
    WHERE altitude > 500;

   name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
```

Here the `ONLY` keyword indicates that the query should apply only to `cities`, and not any tables below `cities` in the inheritance hierarchy. Many of the commands that we have already discussed — `SELECT`, `UPDATE` and `DELETE` — support the `ONLY` keyword.

In some cases you may wish to know which table a particular row originated from. There is a system column called `tableoid` in each table which can tell you the originating table:

```
SELECT c.tableoid, c.name, c.altitude
FROM cities c
WHERE c.altitude > 500;
```

which returns:

```
 tableoid |   name    | altitude
----------+-----------+----------
```

```
   139793 | Las Vegas |     2174
   139793 | Mariposa  |     1953
   139798 | Madison   |      845
```

(If you try to reproduce this example, you will probably get different numeric OIDs.) By doing a join with `pg_class` you can see the actual table names:

```
SELECT p.relname, c.name, c.altitude
FROM cities c, pg_class p
WHERE c.altitude > 500 and c.tableoid = p.oid;
```

which returns:

```
 relname  |   name    | altitude
----------+-----------+----------
 cities   | Las Vegas |     2174
 cities   | Mariposa  |     1953
 capitals | Madison   |      845
```

Inheritance does not automatically propagate data from `INSERT` or `COPY` commands to other tables in the inheritance hierarchy. In our example, the following `INSERT` statement will fail:

```
INSERT INTO cities (name, population, altitude, state)
VALUES ('New York', NULL, NULL, 'NY');
```

We might hope that the data would somehow be routed to the `capitals` table, but this does not happen: `INSERT` always inserts into exactly the table specified. In some cases it is possible to redirect the insertion using a rule (see Chapter 35). However that does not help for the above case because the `cities` table does not contain the column `state`, and so the command will be rejected before the rule can be applied.

All check constraints and not-null constraints on a parent table are automatically inherited by its children. Other types of constraints (unique, primary key, and foreign key constraints) are not inherited.

A table can inherit from more than one parent table, in which case it has the union of the columns defined by the parent tables. Any columns declared in the child table's definition are added to these. If the same column name appears in multiple parent tables, or in both a parent table and the child's definition, then these columns are "merged" so that there is only one such column in the child table. To be merged, columns must have the same data types, else an error is raised. The merged column will have copies of all the check constraints coming from any one of the column definitions it came from, and will be marked not-null if any of them are.

Table inheritance is typically established when the child table is created, using the `INHERITS` clause of the *CREATE TABLE* statement. Alternatively, a table which is already defined in a compatible way can have a new parent relationship added, using the `INHERIT` variant of *ALTER TABLE*. To do this the new child table must already include columns with the same names and types as the columns of the parent. It must also include check constraints with the same names and check expressions as those of the parent. Similarly an inheritance link can be removed from a child using the `NO INHERIT` variant of `ALTER TABLE`. Dynamically adding and removing inheritance links like this can be useful when the inheritance relationship is being used for table partitioning (see Section 5.9).

One convenient way to create a compatible table that will later be made a new child is to use the `LIKE` clause in `CREATE TABLE`. This creates a new table with the same columns as the source table. If there are

any CHECK constraints defined on the source table, the INCLUDING CONSTRAINTS option to LIKE should be specified, as the new child must have constraints matching the parent to be considered compatible.

A parent table cannot be dropped while any of its children remain. Neither can columns of child tables be dropped or altered if they are inherited from any parent tables. If you wish to remove a table and all of its descendants, one easy way is to drop the parent table with the CASCADE option.

ALTER TABLE will propagate any changes in column data definitions and check constraints down the inheritance hierarchy. Again, dropping columns or constraints on parent tables is only possible when using the CASCADE option. ALTER TABLE follows the same rules for duplicate column merging and rejection that apply during CREATE TABLE.

5.8.1. Caveats

Table access permissions are not automatically inherited. Therefore, a user attempting to access a parent table must either have permissions to do the operation on all its child tables as well, or must use the ONLY notation. When adding a new child table to an existing inheritance hierarchy, be careful to grant all the needed permissions on it.

A serious limitation of the inheritance feature is that indexes (including unique constraints) and foreign key constraints only apply to single tables, not to their inheritance children. This is true on both the referencing and referenced sides of a foreign key constraint. Thus, in the terms of the above example:

- If we declared cities.name to be UNIQUE or a PRIMARY KEY, this would not stop the capitals table from having rows with names duplicating rows in cities. And those duplicate rows would by default show up in queries from cities. In fact, by default capitals would have no unique constraint at all, and so could contain multiple rows with the same name. You could add a unique constraint to capitals, but this would not prevent duplication compared to cities.
- Similarly, if we were to specify that cities.name REFERENCES some other table, this constraint would not automatically propagate to capitals. In this case you could work around it by manually adding the same REFERENCES constraint to capitals.
- Specifying that another table's column REFERENCES cities(name) would allow the other table to contain city names, but not capital names. There is no good workaround for this case.

These deficiencies will probably be fixed in some future release, but in the meantime considerable care is needed in deciding whether inheritance is useful for your problem.

> **Deprecated:** In releases of PostgreSQL prior to 7.1, the default behavior was not to include child tables in queries. This was found to be error prone and also in violation of the SQL standard. You can get the pre-7.1 behavior by turning off the sql_inheritance configuration option.

5.9. Partitioning

PostgreSQL supports basic table partitioning. This section describes why and how to implement partitioning as part of your database design.

5.9.1. Overview

Partitioning refers to splitting what is logically one large table into smaller physical pieces. Partitioning can provide several benefits:

- Query performance can be improved dramatically in certain situations, particularly when most of the heavily accessed rows of the table are in a single partition or a small number of partitions. The partitioning substitutes for leading columns of indexes, reducing index size and making it more likely that the heavily-used parts of the indexes fit in memory.
- When queries or updates access a large percentage of a single partition, performance can be improved by taking advantage of sequential scan of that partition instead of using an index and random access reads scattered across the whole table.
- Bulk loads and deletes may be accomplished by adding or removing partitions, if that requirement is planned into the partitioning design. `ALTER TABLE` is far faster than a bulk operation. It also entirely avoids the `VACUUM` overhead caused by a bulk `DELETE`.
- Seldom-used data can be migrated to cheaper and slower storage media.

The benefits will normally be worthwhile only when a table would otherwise be very large. The exact point at which a table will benefit from partitioning depends on the application, although a rule of thumb is that the size of the table should exceed the physical memory of the database server.

Currently, PostgreSQL supports partitioning via table inheritance. Each partition must be created as a child table of a single parent table. The parent table itself is normally empty; it exists just to represent the entire data set. You should be familiar with inheritance (see Section 5.8) before attempting to set up partitioning.

The following forms of partitioning can be implemented in PostgreSQL:

Range Partitioning

: The table is partitioned into "ranges" defined by a key column or set of columns, with no overlap between the ranges of values assigned to different partitions. For example one might partition by date ranges, or by ranges of identifiers for particular business objects.

List Partitioning

: The table is partitioned by explicitly listing which key values appear in each partition.

5.9.2. Implementing Partitioning

To set up a partitioned table, do the following:

1. Create the "master" table, from which all of the partitions will inherit.

   This table will contain no data. Do not define any check constraints on this table, unless you intend them to be applied equally to all partitions. There is no point in defining any indexes or unique constraints on it, either.

2. Create several "child" tables that each inherit from the master table. Normally, these tables will not add any columns to the set inherited from the master.

   We will refer to the child tables as partitions, though they are in every way normal PostgreSQL tables.
3. Add table constraints to the partition tables to define the allowed key values in each partition.

   Typical examples would be:

   ```
   CHECK ( x = 1 )
   CHECK ( county IN ( 'Oxfordshire', 'Buckinghamshire', 'Warwickshire' ))
   CHECK ( outletID >= 100 AND outletID < 200 )
   ```

   Ensure that the constraints guarantee that there is no overlap between the key values permitted in different partitions. A common mistake is to set up range constraints like this:

   ```
   CHECK ( outletID BETWEEN 100 AND 200 )
   CHECK ( outletID BETWEEN 200 AND 300 )
   ```

   This is wrong since it is not clear which partition the key value 200 belongs in.

   Note that there is no difference in syntax between range and list partitioning; those terms are descriptive only.
4. For each partition, create an index on the key column(s), as well as any other indexes you might want. (The key index is not strictly necessary, but in most scenarios it is helpful. If you intend the key values to be unique then you should always create a unique or primary-key constraint for each partition.)
5. Optionally, define a rule or trigger to redirect modifications of the master table to the appropriate partition.
6. Ensure that the constraint_exclusion configuration parameter is enabled in `postgresql.conf`. Without this, queries will not be optimized as desired.

For example, suppose we are constructing a database for a large ice cream company. The company measures peak temperatures every day as well as ice cream sales in each region. Conceptually, we want a table like this:

```
CREATE TABLE measurement (
    city_id         int not null,
    logdate         date not null,
    peaktemp        int,
    unitsales       int
);
```

We know that most queries will access just the last week's, month's or quarter's data, since the main use of this table will be to prepare online reports for management. To reduce the amount of old data that needs to be stored, we decide to only keep the most recent 3 years worth of data. At the beginning of each month we will remove the oldest month's data.

In this situation we can use partitioning to help us meet all of our different requirements for the measurements table. Following the steps outlined above, partitioning can be set up as follows:

1. The master table is the `measurement` table, declared exactly as above.
2. Next we create one partition for each active month:

   ```
   CREATE TABLE measurement_y2004m02 ( ) INHERITS (measurement);
   CREATE TABLE measurement_y2004m03 ( ) INHERITS (measurement);
   ...
   ```

```
CREATE TABLE measurement_y2005m11 ( ) INHERITS (measurement);
CREATE TABLE measurement_y2005m12 ( ) INHERITS (measurement);
CREATE TABLE measurement_y2006m01 ( ) INHERITS (measurement);
```

Each of the partitions are complete tables in their own right, but they inherit their definition from the `measurement` table.

This solves one of our problems: deleting old data. Each month, all we will need to do is perform a `DROP TABLE` on the oldest child table and create a new child table for the new month's data.

3. We must add non-overlapping table constraints, so that our table creation script becomes:

```
CREATE TABLE measurement_y2004m02 (
    CHECK ( logdate >= DATE '2004-02-01' AND logdate < DATE '2004-03-01' )
) INHERITS (measurement);
CREATE TABLE measurement_y2004m03 (
    CHECK ( logdate >= DATE '2004-03-01' AND logdate < DATE '2004-04-01' )
) INHERITS (measurement);
...
CREATE TABLE measurement_y2005m11 (
    CHECK ( logdate >= DATE '2005-11-01' AND logdate < DATE '2005-12-01' )
) INHERITS (measurement);
CREATE TABLE measurement_y2005m12 (
    CHECK ( logdate >= DATE '2005-12-01' AND logdate < DATE '2006-01-01' )
) INHERITS (measurement);
CREATE TABLE measurement_y2006m01 (
    CHECK ( logdate >= DATE '2006-01-01' AND logdate < DATE '2006-02-01' )
) INHERITS (measurement);
```

4. We probably need indexes on the key columns too:

```
CREATE INDEX measurement_y2004m02_logdate ON measurement_y2004m02 (logdate);
CREATE INDEX measurement_y2004m03_logdate ON measurement_y2004m03 (logdate);
...
CREATE INDEX measurement_y2005m11_logdate ON measurement_y2005m11 (logdate);
CREATE INDEX measurement_y2005m12_logdate ON measurement_y2005m12 (logdate);
CREATE INDEX measurement_y2006m01_logdate ON measurement_y2006m01 (logdate);
```

We choose not to add further indexes at this time.

5. If data will be added only to the latest partition, we can set up a very simple rule to insert data. We must redefine this each month so that it always points to the current partition.

```
CREATE OR REPLACE RULE measurement_current_partition AS
ON INSERT TO measurement
DO INSTEAD
    INSERT INTO measurement_y2006m01 VALUES ( NEW.city_id,
                                              NEW.logdate,
                                              NEW.peaktemp,
                                              NEW.unitsales );
```

We might want to insert data and have the server automatically locate the partition into which the row should be added. We could do this with a more complex set of rules as shown below.

```
CREATE RULE measurement_insert_y2004m02 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2004-02-01' AND logdate < DATE '2004-03-01' )
DO INSTEAD
    INSERT INTO measurement_y2004m02 VALUES ( NEW.city_id,
                                              NEW.logdate,
                                              NEW.peaktemp,
```

```
                                                NEW.unitsales );
...
CREATE RULE measurement_insert_y2005m12 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2005-12-01' AND logdate < DATE '2006-01-01' )
DO INSTEAD
    INSERT INTO measurement_y2005m12 VALUES ( NEW.city_id,
                                              NEW.logdate,
                                              NEW.peaktemp,
                                              NEW.unitsales );
CREATE RULE measurement_insert_y2006m01 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2006-01-01' AND logdate < DATE '2006-02-01' )
DO INSTEAD
    INSERT INTO measurement_y2006m01 VALUES ( NEW.city_id,
                                              NEW.logdate,
                                              NEW.peaktemp,
                                              NEW.unitsales );
```

Note that the `WHERE` clause in each rule exactly matches the `CHECK` constraint for its partition.

As we can see, a complex partitioning scheme could require a substantial amount of DDL. In the above example we would be creating a new partition each month, so it may be wise to write a script that generates the required DDL automatically.

Partitioning can also be arranged using a `UNION ALL` view:

```
CREATE VIEW measurement AS
          SELECT * FROM measurement_y2004m02
UNION ALL SELECT * FROM measurement_y2004m03
...
UNION ALL SELECT * FROM measurement_y2005m11
UNION ALL SELECT * FROM measurement_y2005m12
UNION ALL SELECT * FROM measurement_y2006m01;
```

However, the need to recreate the view adds an extra step to adding and dropping individual partitions of the data set.

5.9.3. Managing Partitions

Normally the set of partitions established when initially defining the table are not intended to remain static. It is common to want to remove old partitions of data and periodically add new partitions for new data. One of the most important advantages of partitioning is precisely that it allows this otherwise painful task to be executed nearly instantaneously by manipulating the partition structure, rather than physically moving large amounts of data around.

The simplest option for removing old data is simply to drop the partition that is no longer necessary:

```
DROP TABLE measurement_y2003m02;
```

This can very quickly delete millions of records because it doesn't have to individually delete every record.

Another option that is often preferable is to remove the partition from the partitioned table but retain access to it as a table in its own right:

```
ALTER TABLE measurement_y2003m02 NO INHERIT measurement;
```

This allows further operations to be performed on the data before it is dropped. For example, this is often a useful time to back up the data using COPY, pg_dump, or similar tools. It can also be a useful time to aggregate data into smaller formats, perform other data manipulations, or run reports.

Similarly we can add a new partition to handle new data. We can create an empty partition in the partitioned table just as the original partitions were created above.

```
CREATE TABLE measurement_y2006m02 (
    CHECK ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )
) INHERITS (measurement);
```

As an alternative, it is sometimes more convenient to create the new table outside the partition structure, and make it a proper partition later. This allows the data to be loaded, checked, and transformed prior to it appearing in the partitioned table.

```
CREATE TABLE measurement_y2006m02
  (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS);
ALTER TABLE measurement_y2006m02 ADD CONSTRAINT y2006m02
   CHECK ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' );
\copy measurement_y2006m02 from 'measurement_y2006m02'
-- possibly some other data preparation work
ALTER TABLE measurement_y2006m02 INHERIT measurement;
```

5.9.4. Partitioning and Constraint Exclusion

Constraint exclusion is a query optimization technique that improves performance for partitioned tables defined in the fashion described above. As an example:

```
SET constraint_exclusion = on;
SELECT count(*) FROM measurement WHERE logdate >= DATE '2006-01-01';
```

Without constraint exclusion, the above query would scan each of the partitions of the measurement table. With constraint exclusion enabled, the planner will examine the constraints of each partition and try to prove that the partition need not be scanned because it could not contain any rows meeting the query's WHERE clause. When the planner can prove this, it excludes the partition from the query plan.

You can use the EXPLAIN command to show the difference between a plan with constraint_exclusion on and a plan with it off. A typical default plan for this type of table setup is:

```
SET constraint_exclusion = off;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2006-01-01';

                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Aggregate  (cost=158.66..158.68 rows=1 width=0)
```

```
   ->  Append  (cost=0.00..151.88 rows=2715 width=0)
         ->  Seq Scan on measurement  (cost=0.00..30.38 rows=543 width=0)
               Filter: (logdate >= '2006-01-01'::date)
         ->  Seq Scan on measurement_y2004m02 measurement  (cost=0.00..30.38 rows=543 width
               Filter: (logdate >= '2006-01-01'::date)
         ->  Seq Scan on measurement_y2004m03 measurement  (cost=0.00..30.38 rows=543 width
               Filter: (logdate >= '2006-01-01'::date)
...
         ->  Seq Scan on measurement_y2005m12 measurement  (cost=0.00..30.38 rows=543 width
               Filter: (logdate >= '2006-01-01'::date)
         ->  Seq Scan on measurement_y2006m01 measurement  (cost=0.00..30.38 rows=543 width
               Filter: (logdate >= '2006-01-01'::date)
```

Some or all of the partitions might use index scans instead of full-table sequential scans, but the point here is that there is no need to scan the older partitions at all to answer this query. When we enable constraint exclusion, we get a significantly reduced plan that will deliver the same answer:

```
SET constraint_exclusion = on;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2006-01-01';
                                       QUERY PLAN
-----------------------------------------------------------------------------------------
 Aggregate  (cost=63.47..63.48 rows=1 width=0)
   ->  Append  (cost=0.00..60.75 rows=1086 width=0)
         ->  Seq Scan on measurement  (cost=0.00..30.38 rows=543 width=0)
               Filter: (logdate >= '2006-01-01'::date)
         ->  Seq Scan on measurement_y2006m01 measurement  (cost=0.00..30.38 rows=543 width
               Filter: (logdate >= '2006-01-01'::date)
```

Note that constraint exclusion is driven only by `CHECK` constraints, not by the presence of indexes. Therefore it isn't necessary to define indexes on the key columns. Whether an index needs to be created for a given partition depends on whether you expect that queries that scan the partition will generally scan a large part of the partition or just a small part. An index will be helpful in the latter case but not the former.

5.9.5. Caveats

The following caveats apply to partitioned tables:

- There is currently no way to verify that all of the `CHECK` constraints are mutually exclusive. Care is required by the database designer.
- There is currently no simple way to specify that rows must not be inserted into the master table. A `CHECK (false)` constraint on the master table would be inherited by all child tables, so that cannot be used for this purpose. One possibility is to set up an `ON INSERT` trigger on the master table that always raises an error. (Alternatively, such a trigger could be used to redirect the data into the proper child table, instead of using a set of rules as suggested above.)

The following caveats apply to constraint exclusion:

- Constraint exclusion only works when the query's `WHERE` clause contains constants. A parameterized query will not be optimized, since the planner cannot know what partitions the parameter value might select at run time. For the same reason, "stable" functions such as `CURRENT_DATE` must be avoided.
- Avoid cross-data type comparisons in the `CHECK` constraints, as the planner will currently fail to prove such conditions false. For example, the following constraint will work if `x` is an `integer` column, but not if `x` is a `bigint`:

  ```
  CHECK ( x = 1 )
  ```
  For a `bigint` column we must use a constraint like:

  ```
  CHECK ( x = 1::bigint )
  ```
  The problem is not limited to the `bigint` data type — it can occur whenever the default data type of the constant does not match the data type of the column to which it is being compared. Cross-data type comparisons in the supplied queries are usually OK, just not in the `CHECK` conditions.
- All constraints on all partitions of the master table are considered for constraint exclusion, so large numbers of partitions are likely to increase query planning time considerably.
- Don't forget that you still need to run `ANALYZE` on each partition individually. A command like

  ```
  ANALYZE measurement;
  ```
  will only process the master table.

5.10. Other Database Objects

Tables are the central objects in a relational database structure, because they hold your data. But they are not the only objects that exist in a database. Many other kinds of objects can be created to make the use and management of the data more efficient or convenient. They are not discussed in this chapter, but we give you a list here so that you are aware of what is possible.

- Views
- Functions and operators
- Data types and domains
- Triggers and rewrite rules

Detailed information on these topics appears in Part V.

5.11. Dependency Tracking

When you create complex database structures involving many tables with foreign key constraints, views, triggers, functions, etc. you will implicitly create a net of dependencies between the objects. For instance, a table with a foreign key constraint depends on the table it references.

To ensure the integrity of the entire database structure, PostgreSQL makes sure that you cannot drop objects that other objects still depend on. For example, attempting to drop the products table we had

considered in Section 5.3.5, with the orders table depending on it, would result in an error message such as this:

```
DROP TABLE products;

NOTICE:  constraint orders_product_no_fkey on table orders depends on table products
ERROR:  cannot drop table products because other objects depend on it
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

The error message contains a useful hint: if you do not want to bother deleting all the dependent objects individually, you can run

```
DROP TABLE products CASCADE;
```

and all the dependent objects will be removed. In this case, it doesn't remove the orders table, it only removes the foreign key constraint. (If you want to check what `DROP ... CASCADE` will do, run `DROP` without `CASCADE` and read the `NOTICE` messages.)

All drop commands in PostgreSQL support specifying `CASCADE`. Of course, the nature of the possible dependencies varies with the type of the object. You can also write `RESTRICT` instead of `CASCADE` to get the default behavior, which is to prevent drops of objects that other objects depend on.

> **Note:** According to the SQL standard, specifying either `RESTRICT` or `CASCADE` is required. No database system actually enforces that rule, but whether the default behavior is `RESTRICT` or `CASCADE` varies across systems.

> **Note:** Foreign key constraint dependencies and serial column dependencies from PostgreSQL versions prior to 7.3 are *not* maintained or created during the upgrade process. All other dependency types will be properly created during an upgrade from a pre-7.3 database.

Chapter 6. Data Manipulation

The previous chapter discussed how to create tables and other structures to hold your data. Now it is time to fill the tables with data. This chapter covers how to insert, update, and delete table data. We also introduce ways to effect automatic data changes when certain events occur: triggers and rewrite rules. The chapter after this will finally explain how to extract your long-lost data back out of the database.

6.1. Inserting Data

When a table is created, it contains no data. The first thing to do before a database can be of much use is to insert data. Data is conceptually inserted one row at a time. Of course you can also insert more than one row, but there is no way to insert less than one row at a time. Even if you know only some column values, a complete row must be created.

To create a new row, use the *INSERT* command. The command requires the table name and a value for each of the columns of the table. For example, consider the products table from Chapter 5:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);
```

An example command to insert a row would be:

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

The data values are listed in the order in which the columns appear in the table, separated by commas. Usually, the data values will be literals (constants), but scalar expressions are also allowed.

The above syntax has the drawback that you need to know the order of the columns in the table. To avoid that you can also list the columns explicitly. For example, both of the following commands have the same effect as the one above:

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', 9.99);
INSERT INTO products (name, price, product_no) VALUES ('Cheese', 9.99, 1);
```

Many users consider it good practice to always list the column names.

If you don't have values for all the columns, you can omit some of them. In that case, the columns will be filled with their default values. For example,

```
INSERT INTO products (product_no, name) VALUES (1, 'Cheese');
INSERT INTO products VALUES (1, 'Cheese');
```

The second form is a PostgreSQL extension. It fills the columns from the left with as many values as are given, and the rest will be defaulted.

For clarity, you can also request default values explicitly, for individual columns or for the entire row:

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', DEFAULT);
INSERT INTO products DEFAULT VALUES;
```

You can insert multiple rows in a single command:

```
INSERT INTO products (product_no, name, price) VALUES
    (1, 'Cheese', 9.99),
    (2, 'Bread', 1.99),
    (3, 'Milk', 2.99);
```

Tip: When inserting a lot of data at the same time, considering using the *COPY* command. It is not as flexible as the *INSERT* command, but is more efficient. Refer to Section 13.4 for more information on improving bulk loading performance.

6.2. Updating Data

The modification of data that is already in the database is referred to as updating. You can update individual rows, all the rows in a table, or a subset of all rows. Each column can be updated separately; the other columns are not affected.

To perform an update, you need three pieces of information:

1. The name of the table and column to update,
2. The new value of the column,
3. Which row(s) to update.

Recall from Chapter 5 that SQL does not, in general, provide a unique identifier for rows. Therefore it is not necessarily possible to directly specify which row to update. Instead, you specify which conditions a row must meet in order to be updated. Only if you have a primary key in the table (no matter whether you declared it or not) can you reliably address individual rows, by choosing a condition that matches the primary key. Graphical database access tools rely on this fact to allow you to update rows individually.

For example, this command updates all products that have a price of 5 to have a price of 10:

```
UPDATE products SET price = 10 WHERE price = 5;
```

This may cause zero, one, or many rows to be updated. It is not an error to attempt an update that does not match any rows.

Let's look at that command in detail. First is the key word `UPDATE` followed by the table name. As usual, the table name may be schema-qualified, otherwise it is looked up in the path. Next is the key word `SET` followed by the column name, an equals sign and the new column value. The new column value can be any scalar expression, not just a constant. For example, if you want to raise the price of all products by 10% you could use:

```
UPDATE products SET price = price * 1.10;
```

As you see, the expression for the new value can refer to the existing value(s) in the row. We also left out the `WHERE` clause. If it is omitted, it means that all rows in the table are updated. If it is present, only those rows that match the `WHERE` condition are updated. Note that the equals sign in the `SET` clause is an assignment while the one in the `WHERE` clause is a comparison, but this does not create any ambiguity. Of course, the `WHERE` condition does not have to be an equality test. Many other operators are available (see Chapter 9). But the expression needs to evaluate to a Boolean result.

You can update more than one column in an `UPDATE` command by listing more than one assignment in the `SET` clause. For example:

```
UPDATE mytable SET a = 5, b = 3, c = 1 WHERE a > 0;
```

6.3. Deleting Data

So far we have explained how to add data to tables and how to change data. What remains is to discuss how to remove data that is no longer needed. Just as adding data is only possible in whole rows, you can only remove entire rows from a table. In the previous section we explained that SQL does not provide a way to directly address individual rows. Therefore, removing rows can only be done by specifying conditions that the rows to be removed have to match. If you have a primary key in the table then you can specify the exact row. But you can also remove groups of rows matching a condition, or you can remove all rows in the table at once.

You use the *DELETE* command to remove rows; the syntax is very similar to the `UPDATE` command. For instance, to remove all rows from the products table that have a price of 10, use

```
DELETE FROM products WHERE price = 10;
```

If you simply write

```
DELETE FROM products;
```

then all rows in the table will be deleted! Caveat programmer.

Chapter 7. Queries

The previous chapters explained how to create tables, how to fill them with data, and how to manipulate that data. Now we finally discuss how to retrieve the data out of the database.

7.1. Overview

The process of retrieving or the command to retrieve data from a database is called a *query*. In SQL the *SELECT* command is used to specify queries. The general syntax of the `SELECT` command is

```
SELECT select_list FROM table_expression [sort_specification]
```

The following sections describe the details of the select list, the table expression, and the sort specification.

A simple kind of query has the form

```
SELECT * FROM table1;
```

Assuming that there is a table called `table1`, this command would retrieve all rows and all columns from `table1`. (The method of retrieval depends on the client application. For example, the psql program will display an ASCII-art table on the screen, while client libraries will offer functions to extract individual values from the query result.) The select list specification `*` means all columns that the table expression happens to provide. A select list can also select a subset of the available columns or make calculations using the columns. For example, if `table1` has columns named `a`, `b`, and `c` (and perhaps others) you can make the following query:

```
SELECT a, b + c FROM table1;
```

(assuming that `b` and `c` are of a numerical data type). See Section 7.3 for more details.

`FROM table1` is a particularly simple kind of table expression: it reads just one table. In general, table expressions can be complex constructs of base tables, joins, and subqueries. But you can also omit the table expression entirely and use the `SELECT` command as a calculator:

```
SELECT 3 * 4;
```

This is more useful if the expressions in the select list return varying results. For example, you could call a function this way:

```
SELECT random();
```

7.2. Table Expressions

A *table expression* computes a table. The table expression contains a `FROM` clause that is optionally followed by `WHERE`, `GROUP BY`, and `HAVING` clauses. Trivial table expressions simply refer to a table on

disk, a so-called base table, but more complex expressions can be used to modify or combine base tables in various ways.

The optional WHERE, GROUP BY, and HAVING clauses in the table expression specify a pipeline of successive transformations performed on the table derived in the FROM clause. All these transformations produce a virtual table that provides the rows that are passed to the select list to compute the output rows of the query.

7.2.1. The FROM Clause

The *FROM Clause* derives a table from one or more other tables given in a comma-separated table reference list.

```
FROM table_reference [, table_reference [, ...]]
```

A table reference may be a table name (possibly schema-qualified), or a derived table such as a subquery, a table join, or complex combinations of these. If more than one table reference is listed in the FROM clause they are cross-joined (see below) to form the intermediate virtual table that may then be subject to transformations by the WHERE, GROUP BY, and HAVING clauses and is finally the result of the overall table expression.

When a table reference names a table that is the parent of a table inheritance hierarchy, the table reference produces rows of not only that table but all of its descendant tables, unless the key word ONLY precedes the table name. However, the reference produces only the columns that appear in the named table — any columns added in subtables are ignored.

7.2.1.1. Joined Tables

A joined table is a table derived from two other (real or derived) tables according to the rules of the particular join type. Inner, outer, and cross-joins are available.

Join Types

Cross join

```
T1 CROSS JOIN T2
```

For each combination of rows from *T1* and *T2*, the derived table will contain a row consisting of all columns in *T1* followed by all columns in *T2*. If the tables have N and M rows respectively, the joined table will have N * M rows.

FROM *T1* CROSS JOIN *T2* is equivalent to FROM *T1*, *T2*. It is also equivalent to FROM *T1* INNER JOIN *T2* ON TRUE (see below).

Qualified joins

```
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 ON boolean_expression
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 USING ( join column list )
T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
```

The words INNER and OUTER are optional in all forms. INNER is the default; LEFT, RIGHT, and FULL imply an outer join.

The *join condition* is specified in the `ON` or `USING` clause, or implicitly by the word `NATURAL`. The join condition determines which rows from the two source tables are considered to "match", as explained in detail below.

The `ON` clause is the most general kind of join condition: it takes a Boolean value expression of the same kind as is used in a `WHERE` clause. A pair of rows from *`T1`* and *`T2`* match if the `ON` expression evaluates to true for them.

`USING` is a shorthand notation: it takes a comma-separated list of column names, which the joined tables must have in common, and forms a join condition specifying equality of each of these pairs of columns. Furthermore, the output of a `JOIN USING` has one column for each of the equated pairs of input columns, followed by all of the other columns from each table. Thus, `USING (a, b, c)` is equivalent to `ON (t1.a = t2.a AND t1.b = t2.b AND t1.c = t2.c)` with the exception that if `ON` is used there will be two columns `a`, `b`, and `c` in the result, whereas with `USING` there will be only one of each.

Finally, `NATURAL` is a shorthand form of `USING`: it forms a `USING` list consisting of exactly those column names that appear in both input tables. As with `USING`, these columns appear only once in the output table.

The possible types of qualified join are:

`INNER JOIN`

For each row R1 of T1, the joined table has a row for each row in T2 that satisfies the join condition with R1.

`LEFT OUTER JOIN`

First, an inner join is performed. Then, for each row in T1 that does not satisfy the join condition with any row in T2, a joined row is added with null values in columns of T2. Thus, the joined table unconditionally has at least one row for each row in T1.

`RIGHT OUTER JOIN`

First, an inner join is performed. Then, for each row in T2 that does not satisfy the join condition with any row in T1, a joined row is added with null values in columns of T1. This is the converse of a left join: the result table will unconditionally have a row for each row in T2.

`FULL OUTER JOIN`

First, an inner join is performed. Then, for each row in T1 that does not satisfy the join condition with any row in T2, a joined row is added with null values in columns of T2. Also, for each row of T2 that does not satisfy the join condition with any row in T1, a joined row with null values in the columns of T1 is added.

Joins of all types can be chained together or nested: either or both of *`T1`* and *`T2`* may be joined tables. Parentheses may be used around `JOIN` clauses to control the join order. In the absence of parentheses, `JOIN` clauses nest left-to-right.

To put this together, assume we have tables `t1`

```
 num | name
-----+------
   1 | a
```

```
   2 | b
   3 | c
```

and `t2`

```
 num | value
-----+-------
   1 | xxx
   3 | yyy
   5 | zzz
```

then we get the following results for the various joins:

```
=> SELECT * FROM t1 CROSS JOIN t2;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   1 | a    |   3 | yyy
   1 | a    |   5 | zzz
   2 | b    |   1 | xxx
   2 | b    |   3 | yyy
   2 | b    |   5 | zzz
   3 | c    |   1 | xxx
   3 | c    |   3 | yyy
   3 | c    |   5 | zzz
(9 rows)

=> SELECT * FROM t1 INNER JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
(2 rows)

=> SELECT * FROM t1 INNER JOIN t2 USING (num);
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 NATURAL INNER JOIN t2;
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
```

```
   3 | c    |   3 | yyy
(3 rows)

=> SELECT * FROM t1 LEFT JOIN t2 USING (num);
 num | name | value
-----+------+-------
   1 | a    | xxx
   2 | b    |
   3 | c    | yyy
(3 rows)

=> SELECT * FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
     |      |   5 | zzz
(3 rows)

=> SELECT * FROM t1 FULL JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
     |      |   5 | zzz
(4 rows)
```

The join condition specified with ON can also contain conditions that do not relate directly to the join. This can prove useful for some queries but needs to be thought out carefully. For example:

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num AND t2.value = 'xxx';
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |     |
(3 rows)
```

7.2.1.2. Table and Column Aliases

A temporary name can be given to tables and complex table references to be used for references to the derived table in the rest of the query. This is called a *table alias*.

To create a table alias, write

```
FROM table_reference AS alias
```

or

```
FROM table_reference alias
```

The `AS` key word is noise. *`alias`* can be any identifier.

A typical application of table aliases is to assign short identifiers to long table names to keep the join clauses readable. For example:

```
SELECT * FROM some_very_long_table_name s JOIN another_fairly_long_name a ON s.id = a.num;
```

The alias becomes the new name of the table reference for the current query — it is no longer possible to refer to the table by the original name. Thus

```
SELECT * FROM my_table AS m WHERE my_table.a > 5;
```

is not valid according to the SQL standard. In PostgreSQL this will draw an error if the add_missing_from configuration variable is `off` (as it is by default). If it is `on`, an implicit table reference will be added to the `FROM` clause, so the query is processed as if it were written as

```
SELECT * FROM my_table AS m, my_table AS my_table WHERE my_table.a > 5;
```

That will result in a cross join, which is usually not what you want.

Table aliases are mainly for notational convenience, but it is necessary to use them when joining a table to itself, e.g.,

```
SELECT * FROM people AS mother JOIN people AS child ON mother.id = child.mother_id;
```

Additionally, an alias is required if the table reference is a subquery (see Section 7.2.1.3).

Parentheses are used to resolve ambiguities. In the following example, the first statement assigns the alias `b` to the second instance of `my_table`, but the second statement assigns the alias to the result of the join:

```
SELECT * FROM my_table AS a CROSS JOIN my_table AS b ...
SELECT * FROM (my_table AS a CROSS JOIN my_table) AS b ...
```

Another form of table aliasing gives temporary names to the columns of the table, as well as the table itself:

```
FROM table_reference [AS] alias ( column1 [, column2 [, ...]] )
```

If fewer column aliases are specified than the actual table has columns, the remaining columns are not renamed. This syntax is especially useful for self-joins or subqueries.

When an alias is applied to the output of a `JOIN` clause, using any of these forms, the alias hides the original names within the `JOIN`. For example,

```
SELECT a.* FROM my_table AS a JOIN your_table AS b ON ...
```

is valid SQL, but

```
SELECT a.* FROM (my_table AS a JOIN your_table AS b ON ...) AS c
```

is not valid: the table alias `a` is not visible outside the alias `c`.

7.2.1.3. Subqueries

Subqueries specifying a derived table must be enclosed in parentheses and *must* be assigned a table alias name. (See Section 7.2.1.2.) For example:

```
FROM (SELECT * FROM table1) AS alias_name
```

This example is equivalent to `FROM table1 AS alias_name`. More interesting cases, which can't be reduced to a plain join, arise when the subquery involves grouping or aggregation.

A subquery can also be a `VALUES` list:

```
FROM (VALUES ('anne', 'smith'), ('bob', 'jones'), ('joe', 'blow'))
     AS names(first, last)
```

Again, a table alias is required. Assigning alias names to the columns of the `VALUES` list is optional, but is good practice. For more information see Section 7.7.

7.2.1.4. Table Functions

Table functions are functions that produce a set of rows, made up of either base data types (scalar types) or composite data types (table rows). They are used like a table, view, or subquery in the `FROM` clause of a query. Columns returned by table functions may be included in `SELECT`, `JOIN`, or `WHERE` clauses in the same manner as a table, view, or subquery column.

If a table function returns a base data type, the single result column is named like the function. If the function returns a composite type, the result columns get the same names as the individual attributes of the type.

A table function may be aliased in the `FROM` clause, but it also may be left unaliased. If a function is used in the `FROM` clause with no alias, the function name is used as the resulting table name.

Some examples:

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);

CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;

SELECT * FROM foo
    WHERE foosubid IN (select foosubid from getfoo(foo.fooid) z
                           where z.fooid = foo.fooid);

CREATE VIEW vw_getfoo AS SELECT * FROM getfoo(1);

SELECT * FROM vw_getfoo;
```

In some cases it is useful to define table functions that can return different column sets depending on how they are invoked. To support this, the table function can be declared as returning the pseudotype `record`. When such a function is used in a query, the expected row structure must be specified in the query itself, so that the system can know how to parse and plan the query. Consider this example:

```
SELECT *
    FROM dblink('dbname=mydb', 'select proname, prosrc from pg_proc')
      AS t1(proname name, prosrc text)
    WHERE proname LIKE 'bytea%';
```

The `dblink` function executes a remote query (see `contrib/dblink`). It is declared to return `record` since it might be used for any kind of query. The actual column set must be specified in the calling query so that the parser knows, for example, what `*` should expand to.

7.2.2. The `WHERE` Clause

The syntax of the *WHERE Clause* is

```
WHERE search_condition
```

where *search_condition* is any value expression (see Section 4.2) that returns a value of type `boolean`.

After the processing of the `FROM` clause is done, each row of the derived virtual table is checked against the search condition. If the result of the condition is true, the row is kept in the output table, otherwise (that is, if the result is false or null) it is discarded. The search condition typically references at least some column of the table generated in the `FROM` clause; this is not required, but otherwise the `WHERE` clause will be fairly useless.

> **Note:** The join condition of an inner join can be written either in the `WHERE` clause or in the `JOIN` clause. For example, these table expressions are equivalent:
>
> ```
> FROM a, b WHERE a.id = b.id AND b.val > 5
> ```
>
> and
>
> ```
> FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
> ```
>
> or perhaps even
>
> ```
> FROM a NATURAL JOIN b WHERE b.val > 5
> ```
>
> Which one of these you use is mainly a matter of style. The `JOIN` syntax in the `FROM` clause is probably not as portable to other SQL database management systems. For outer joins there is no choice in any case: they must be done in the `FROM` clause. An `ON`/`USING` clause of an outer join is *not* equivalent to a `WHERE` condition, because it determines the addition of rows (for unmatched input rows) as well as the removal of rows from the final result.

Here are some examples of WHERE clauses:

```
SELECT ... FROM fdt WHERE c1 > 5

SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)

SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)

SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)

SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10) AND 100

SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)
```

fdt is the table derived in the FROM clause. Rows that do not meet the search condition of the WHERE clause are eliminated from fdt. Notice the use of scalar subqueries as value expressions. Just like any other query, the subqueries can employ complex table expressions. Notice also how fdt is referenced in the subqueries. Qualifying c1 as fdt.c1 is only necessary if c1 is also the name of a column in the derived input table of the subquery. But qualifying the column name adds clarity even when it is not needed. This example shows how the column naming scope of an outer query extends into its inner queries.

7.2.3. The GROUP BY and HAVING Clauses

After passing the WHERE filter, the derived input table may be subject to grouping, using the GROUP BY clause, and elimination of group rows using the HAVING clause.

```
SELECT select_list
    FROM ...
    [WHERE ...]
    GROUP BY grouping_column_reference [, grouping_column_reference]...
```

The *GROUP BY Clause* is used to group together those rows in a table that share the same values in all the columns listed. The order in which the columns are listed does not matter. The effect is to combine each set of rows sharing common values into one group row that is representative of all rows in the group. This is done to eliminate redundancy in the output and/or compute aggregates that apply to these groups. For instance:

```
=> SELECT * FROM test1;
 x | y
---+---
 a | 3
 c | 2
 b | 5
 a | 1
(4 rows)

=> SELECT x FROM test1 GROUP BY x;
 x
---
 a
 b
```

```
 c
(3 rows)
```

In the second query, we could not have written `SELECT * FROM test1 GROUP BY x`, because there is no single value for the column `y` that could be associated with each group. The grouped-by columns can be referenced in the select list since they have a single value in each group.

In general, if a table is grouped, columns that are not used in the grouping cannot be referenced except in aggregate expressions. An example with aggregate expressions is:

```
=> SELECT x, sum(y) FROM test1 GROUP BY x;
 x | sum
---+-----
 a |   4
 b |   5
 c |   2
(3 rows)
```

Here `sum` is an aggregate function that computes a single value over the entire group. More information about the available aggregate functions can be found in Section 9.15.

> **Tip:** Grouping without aggregate expressions effectively calculates the set of distinct values in a column. This can also be achieved using the `DISTINCT` clause (see Section 7.3.3).

Here is another example: it calculates the total sales for each product (rather than the total sales on all products).

```
SELECT product_id, p.name, (sum(s.units) * p.price) AS sales
    FROM products p LEFT JOIN sales s USING (product_id)
    GROUP BY product_id, p.name, p.price;
```

In this example, the columns `product_id`, `p.name`, and `p.price` must be in the `GROUP BY` clause since they are referenced in the query select list. (Depending on how exactly the products table is set up, name and price may be fully dependent on the product ID, so the additional groupings could theoretically be unnecessary, but this is not implemented yet.) The column `s.units` does not have to be in the `GROUP BY` list since it is only used in an aggregate expression (`sum(...)`), which represents the sales of a product. For each product, the query returns a summary row about all sales of the product.

In strict SQL, `GROUP BY` can only group by columns of the source table but PostgreSQL extends this to also allow `GROUP BY` to group by columns in the select list. Grouping by value expressions instead of simple column names is also allowed.

If a table has been grouped using a `GROUP BY` clause, but then only certain groups are of interest, the `HAVING` clause can be used, much like a `WHERE` clause, to eliminate groups from a grouped table. The syntax is:

```
SELECT select_list FROM ... [WHERE ...] GROUP BY ... HAVING boolean_expression
```

Expressions in the `HAVING` clause can refer both to grouped expressions and to ungrouped expressions (which necessarily involve an aggregate function).

Example:

```
=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING sum(y) > 3;
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)

=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING x < 'c';
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)
```

Again, a more realistic example:

```
SELECT product_id, p.name, (sum(s.units) * (p.price - p.cost)) AS profit
    FROM products p LEFT JOIN sales s USING (product_id)
    WHERE s.date > CURRENT_DATE - INTERVAL '4 weeks'
    GROUP BY product_id, p.name, p.price, p.cost
    HAVING sum(p.price * s.units) > 5000;
```

In the example above, the `WHERE` clause is selecting rows by a column that is not grouped (the expression is only true for sales during the last four weeks), while the `HAVING` clause restricts the output to groups with total gross sales over 5000. Note that the aggregate expressions do not necessarily need to be the same in all parts of the query.

7.3. Select Lists

As shown in the previous section, the table expression in the `SELECT` command constructs an intermediate virtual table by possibly combining tables, views, eliminating rows, grouping, etc. This table is finally passed on to processing by the *select list*. The select list determines which *columns* of the intermediate table are actually output.

7.3.1. Select-List Items

The simplest kind of select list is `*` which emits all columns that the table expression produces. Otherwise, a select list is a comma-separated list of value expressions (as defined in Section 4.2). For instance, it could be a list of column names:

```
SELECT a, b, c FROM ...
```

The columns names `a`, `b`, and `c` are either the actual names of the columns of tables referenced in the `FROM` clause, or the aliases given to them as explained in Section 7.2.1.2. The name space available in the

select list is the same as in the WHERE clause, unless grouping is used, in which case it is the same as in the HAVING clause.

If more than one table has a column of the same name, the table name must also be given, as in

```
SELECT tbl1.a, tbl2.a, tbl1.b FROM ...
```

When working with multiple tables, it can also be useful to ask for all the columns of a particular table:

```
SELECT tbl1.*, tbl2.a FROM ...
```

(See also Section 7.2.2.)

If an arbitrary value expression is used in the select list, it conceptually adds a new virtual column to the returned table. The value expression is evaluated once for each result row, with the row's values substituted for any column references. But the expressions in the select list do not have to reference any columns in the table expression of the FROM clause; they could be constant arithmetic expressions as well, for instance.

7.3.2. Column Labels

The entries in the select list can be assigned names for further processing. The "further processing" in this case is an optional sort specification and the client application (e.g., column headers for display). For example:

```
SELECT a AS value, b + c AS sum FROM ...
```

If no output column name is specified using AS, the system assigns a default name. For simple column references, this is the name of the referenced column. For function calls, this is the name of the function. For complex expressions, the system will generate a generic name.

> **Note:** The naming of output columns here is different from that done in the FROM clause (see Section 7.2.1.2). This pipeline will in fact allow you to rename the same column twice, but the name chosen in the select list is the one that will be passed on.

7.3.3. DISTINCT

After the select list has been processed, the result table may optionally be subject to the elimination of duplicate rows. The DISTINCT key word is written directly after SELECT to specify this:

```
SELECT DISTINCT select_list ...
```

(Instead of DISTINCT the key word ALL can be used to specify the default behavior of retaining all rows.)

Obviously, two rows are considered distinct if they differ in at least one column value. Null values are considered equal in this comparison.

Alternatively, an arbitrary expression can determine what rows are to be considered distinct:

```
SELECT DISTINCT ON (expression [, expression ...]) select_list ...
```

Here *expression* is an arbitrary value expression that is evaluated for all rows. A set of rows for which all the expressions are equal are considered duplicates, and only the first row of the set is kept in the output. Note that the "first row" of a set is unpredictable unless the query is sorted on enough columns to guarantee a unique ordering of the rows arriving at the DISTINCT filter. (DISTINCT ON processing occurs after ORDER BY sorting.)

The DISTINCT ON clause is not part of the SQL standard and is sometimes considered bad style because of the potentially indeterminate nature of its results. With judicious use of GROUP BY and subqueries in FROM the construct can be avoided, but it is often the most convenient alternative.

7.4. Combining Queries

The results of two queries can be combined using the set operations union, intersection, and difference. The syntax is

```
query1 UNION [ALL] query2
query1 INTERSECT [ALL] query2
query1 EXCEPT [ALL] query2
```

query1 and *query2* are queries that can use any of the features discussed up to this point. Set operations can also be nested and chained, for example

```
query1 UNION query2 UNION query3
```

which really says

```
(query1 UNION query2) UNION query3
```

UNION effectively appends the result of *query2* to the result of *query1* (although there is no guarantee that this is the order in which the rows are actually returned). Furthermore, it eliminates duplicate rows from its result, in the same way as DISTINCT, unless UNION ALL is used.

INTERSECT returns all rows that are both in the result of *query1* and in the result of *query2*. Duplicate rows are eliminated unless INTERSECT ALL is used.

EXCEPT returns all rows that are in the result of *query1* but not in the result of *query2*. (This is sometimes called the *difference* between two queries.) Again, duplicates are eliminated unless EXCEPT ALL is used.

In order to calculate the union, intersection, or difference of two queries, the two queries must be "union compatible", which means that they return the same number of columns and the corresponding columns have compatible data types, as described in Section 10.5.

7.5. Sorting Rows

After a query has produced an output table (after the select list has been processed) it can optionally be sorted. If sorting is not chosen, the rows will be returned in an unspecified order. The actual order in that case will depend on the scan and join plan types and the order on disk, but it must not be relied on. A particular output ordering can only be guaranteed if the sort step is explicitly chosen.

The `ORDER BY` clause specifies the sort order:

```
SELECT select_list
    FROM table_expression
    ORDER BY sort_expression1 [ASC | DESC] [, sort_expression2 [ASC | DESC] ...]
```

The sort expression(s) can be any expression that would be valid in the query's select list. An example is

```
SELECT a, b FROM table1 ORDER BY a + b, c;
```

When more than one expression is specified, the later values are used to sort rows that are equal according to the earlier values. Each expression may be followed by an optional `ASC` or `DESC` keyword to set the sort direction to ascending or descending. `ASC` order is the default. Ascending order puts smaller values first, where "smaller" is defined in terms of the < operator. Similarly, descending order is determined with the > operator. [1]

For backwards compatibility with the SQL92 version of the standard, a *sort_expression* can instead be the name or number of an output column, as in

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum;
SELECT a, max(b) FROM table1 GROUP BY a ORDER BY 1;
```

both of which sort by the first output column. Note that an output column name has to stand alone, it's not allowed as part of an expression — for example, this is *not* correct:

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum + c;          -- wrong
```

This restriction is made to reduce ambiguity. There is still ambiguity if an `ORDER BY` item is a simple name that could match either an output column name or a column from the table expression. The output column is used in such cases. This would only cause confusion if you use `AS` to rename an output column to match some other table column's name.

`ORDER BY` can be applied to the result of a `UNION`, `INTERSECT`, or `EXCEPT` combination, but in this case it is only permitted to sort by output column names or numbers, not by expressions.

7.6. `LIMIT` and `OFFSET`

`LIMIT` and `OFFSET` allow you to retrieve just a portion of the rows that are generated by the rest of the query:

```
SELECT select_list
```

1. Actually, PostgreSQL uses the *default B-tree operator class* for the expression's data type to determine the sort ordering for `ASC` and `DESC`. Conventionally, data types will be set up so that the < and > operators correspond to this sort ordering, but a user-defined data type's designer could choose to do something different.

```
FROM table_expression
[ ORDER BY sort_expression1 [ASC | DESC] [, sort_expression2 [ASC | DESC] ...] ]
[ LIMIT { number | ALL } ] [ OFFSET number ]
```

If a limit count is given, no more than that many rows will be returned (but possibly less, if the query itself yields less rows). `LIMIT ALL` is the same as omitting the `LIMIT` clause.

`OFFSET` says to skip that many rows before beginning to return rows. `OFFSET 0` is the same as omitting the `OFFSET` clause. If both `OFFSET` and `LIMIT` appear, then `OFFSET` rows are skipped before starting to count the `LIMIT` rows that are returned.

When using `LIMIT`, it is important to use an `ORDER BY` clause that constrains the result rows into a unique order. Otherwise you will get an unpredictable subset of the query's rows. You may be asking for the tenth through twentieth rows, but tenth through twentieth in what ordering? The ordering is unknown, unless you specified `ORDER BY`.

The query optimizer takes `LIMIT` into account when generating a query plan, so you are very likely to get different plans (yielding different row orders) depending on what you give for `LIMIT` and `OFFSET`. Thus, using different `LIMIT`/`OFFSET` values to select different subsets of a query result *will give inconsistent results* unless you enforce a predictable result ordering with `ORDER BY`. This is not a bug; it is an inherent consequence of the fact that SQL does not promise to deliver the results of a query in any particular order unless `ORDER BY` is used to constrain the order.

The rows skipped by an `OFFSET` clause still have to be computed inside the server; therefore a large `OFFSET` can be inefficient.

7.7. `VALUES` Lists

`VALUES` provides a way to generate a "constant table" that can be used in a query without having to actually create and populate a table on-disk. The syntax is

```
VALUES ( expression [, ...] ) [, ...]
```

Each parenthesized list of expressions generates a row in the table. The lists must all have the same number of elements (i.e., the number of columns in the table), and corresponding entries in each list must have compatible data types. The actual data type assigned to each column of the result is determined using the same rules as for `UNION` (see Section 10.5).

As an example,

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

will return a table of two columns and three rows. It's effectively equivalent to

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

By default, PostgreSQL assigns the names `column1`, `column2`, etc. to the columns of a `VALUES` table. The column names are not specified by the SQL standard and different database systems do it differently, so it's usually better to override the default names with a table alias list.

Syntactically, `VALUES` followed by expression lists is treated as equivalent to

```
SELECT select_list FROM table_expression
```

and can appear anywhere a `SELECT` can. For example, you can use it as an arm of a `UNION`, or attach a *`sort_specification`* (`ORDER BY`, `LIMIT`, and/or `OFFSET`) to it. `VALUES` is most commonly used as the data source in an `INSERT` command, and next most commonly as a subquery.

For more information see *VALUES*.

Chapter 8. Data Types

PostgreSQL has a rich set of native data types available to users. Users may add new types to PostgreSQL using the *CREATE TYPE* command.

Table 8-1 shows all the built-in general-purpose data types. Most of the alternative names listed in the "Aliases" column are the names used internally by PostgreSQL for historical reasons. In addition, some internally used or deprecated types are available, but they are not listed here.

Table 8-1. Data Types

Name	Aliases	Description
bigint	int8	signed eight-byte integer
bigserial	serial8	autoincrementing eight-byte integer
bit [(*n*)]		fixed-length bit string
bit varying [(*n*)]	varbit	variable-length bit string
boolean	bool	logical Boolean (true/false)
box		rectangular box in the plane
bytea		binary data ("byte array")
character varying [(*n*)]	varchar [(*n*)]	variable-length character string
character [(*n*)]	char [(*n*)]	fixed-length character string
cidr		IPv4 or IPv6 network address
circle		circle in the plane
date		calendar date (year, month, day)
double precision	float8	double precision floating-point number
inet		IPv4 or IPv6 host address
integer	int, int4	signed four-byte integer
interval [(*p*)]		time span
line		infinite line in the plane
lseg		line segment in the plane
macaddr		MAC address
money		currency amount
numeric [(*p*, *s*)]	decimal [(*p*, *s*)]	exact numeric of selectable precision
path		geometric path in the plane
point		geometric point in the plane
polygon		closed geometric path in the plane

Name	Aliases	Description
real	float4	single precision floating-point number
smallint	int2	signed two-byte integer
serial	serial4	autoincrementing four-byte integer
text		variable-length character string
time [(*p*)] [without time zone]		time of day
time [(*p*)] with time zone	timetz	time of day, including time zone
timestamp [(*p*)] [without time zone]		date and time
timestamp [(*p*)] with time zone	timestamptz	date and time, including time zone

Compatibility: The following types (or spellings thereof) are specified by SQL: bit, bit varying, boolean, char, character varying, character, varchar, date, double precision, integer, interval, numeric, decimal, real, smallint, time (with or without time zone), timestamp (with or without time zone).

Each data type has an external representation determined by its input and output functions. Many of the built-in types have obvious external formats. However, several types are either unique to PostgreSQL, such as geometric paths, or have several possibilities for formats, such as the date and time types. Some of the input and output functions are not invertible. That is, the result of an output function may lose accuracy when compared to the original input.

8.1. Numeric Types

Numeric types consist of two-, four-, and eight-byte integers, four- and eight-byte floating-point numbers, and selectable-precision decimals. Table 8-2 lists the available types.

Table 8-2. Numeric Types

Name	Storage Size	Description	Range
smallint	2 bytes	small-range integer	-32768 to +32767
integer	4 bytes	usual choice for integer	-2147483648 to +2147483647
bigint	8 bytes	large-range integer	-9223372036854775808 to 9223372036854775807

Name	Storage Size	Description	Range
`decimal`	variable	user-specified precision, exact	no limit
`numeric`	variable	user-specified precision, exact	no limit
`real`	4 bytes	variable-precision, inexact	6 decimal digits precision
`double precision`	8 bytes	variable-precision, inexact	15 decimal digits precision
`serial`	4 bytes	autoincrementing integer	1 to 2147483647
`bigserial`	8 bytes	large autoincrementing integer	1 to 9223372036854775807

The syntax of constants for the numeric types is described in Section 4.1.2. The numeric types have a full set of corresponding arithmetic operators and functions. Refer to Chapter 9 for more information. The following sections describe the types in detail.

8.1.1. Integer Types

The types `smallint`, `integer`, and `bigint` store whole numbers, that is, numbers without fractional components, of various ranges. Attempts to store values outside of the allowed range will result in an error.

The type `integer` is the usual choice, as it offers the best balance between range, storage size, and performance. The `smallint` type is generally only used if disk space is at a premium. The `bigint` type should only be used if the `integer` range is not sufficient, because the latter is definitely faster.

The `bigint` type may not function correctly on all platforms, since it relies on compiler support for eight-byte integers. On a machine without such support, `bigint` acts the same as `integer` (but still takes up eight bytes of storage). However, we are not aware of any reasonable platform where this is actually the case.

SQL only specifies the integer types `integer` (or `int`) and `smallint`. The type `bigint`, and the type names `int2`, `int4`, and `int8` are extensions, which are shared with various other SQL database systems.

8.1.2. Arbitrary Precision Numbers

The type `numeric` can store numbers with up to 1000 digits of precision and perform calculations exactly. It is especially recommended for storing monetary amounts and other quantities where exactness is required. However, arithmetic on `numeric` values is very slow compared to the integer types, or to the floating-point types described in the next section.

In what follows we use these terms: The *scale* of a `numeric` is the count of decimal digits in the fractional part, to the right of the decimal point. The *precision* of a `numeric` is the total count of significant digits in the whole number, that is, the number of digits to both sides of the decimal point. So the number 23.5141 has a precision of 6 and a scale of 4. Integers can be considered to have a scale of zero.

Both the maximum precision and the maximum scale of a `numeric` column can be configured. To declare a column of type `numeric` use the syntax

```
NUMERIC(precision, scale)
```

The precision must be positive, the scale zero or positive. Alternatively,

```
NUMERIC(precision)
```

selects a scale of 0. Specifying

```
NUMERIC
```

without any precision or scale creates a column in which numeric values of any precision and scale can be stored, up to the implementation limit on precision. A column of this kind will not coerce input values to any particular scale, whereas `numeric` columns with a declared scale will coerce input values to that scale. (The SQL standard requires a default scale of 0, i.e., coercion to integer precision. We find this a bit useless. If you're concerned about portability, always specify the precision and scale explicitly.)

If the scale of a value to be stored is greater than the declared scale of the column, the system will round the value to the specified number of fractional digits. Then, if the number of digits to the left of the decimal point exceeds the declared precision minus the declared scale, an error is raised.

Numeric values are physically stored without any extra leading or trailing zeroes. Thus, the declared precision and scale of a column are maximums, not fixed allocations. (In this sense the `numeric` type is more akin to `varchar(n)` than to `char(n)`.) The actual storage requirement is two bytes for each group of four decimal digits, plus eight bytes overhead.

In addition to ordinary numeric values, the `numeric` type allows the special value `NaN`, meaning "not-a-number". Any operation on `NaN` yields another `NaN`. When writing this value as a constant in a SQL command, you must put quotes around it, for example `UPDATE table SET x = 'NaN'`. On input, the string `NaN` is recognized in a case-insensitive manner.

The types `decimal` and `numeric` are equivalent. Both types are part of the SQL standard.

8.1.3. Floating-Point Types

The data types `real` and `double precision` are inexact, variable-precision numeric types. In practice, these types are usually implementations of IEEE Standard 754 for Binary Floating-Point Arithmetic (single and double precision, respectively), to the extent that the underlying processor, operating system, and compiler support it.

Inexact means that some values cannot be converted exactly to the internal format and are stored as approximations, so that storing and printing back out a value may show slight discrepancies. Managing these errors and how they propagate through calculations is the subject of an entire branch of mathematics and computer science and will not be discussed further here, except for the following points:

- If you require exact storage and calculations (such as for monetary amounts), use the `numeric` type instead.

- If you want to do complicated calculations with these types for anything important, especially if you rely on certain behavior in boundary cases (infinity, underflow), you should evaluate the implementation carefully.
- Comparing two floating-point values for equality may or may not work as expected.

On most platforms, the `real` type has a range of at least 1E-37 to 1E+37 with a precision of at least 6 decimal digits. The `double precision` type typically has a range of around 1E-307 to 1E+308 with a precision of at least 15 digits. Values that are too large or too small will cause an error. Rounding may take place if the precision of an input number is too high. Numbers too close to zero that are not representable as distinct from zero will cause an underflow error.

In addition to ordinary numeric values, the floating-point types have several special values:

```
Infinity
-Infinity
NaN
```

These represent the IEEE 754 special values "infinity", "negative infinity", and "not-a-number", respectively. (On a machine whose floating-point arithmetic does not follow IEEE 754, these values will probably not work as expected.) When writing these values as constants in a SQL command, you must put quotes around them, for example `UPDATE table SET x = 'Infinity'`. On input, these strings are recognized in a case-insensitive manner.

PostgreSQL also supports the SQL-standard notations `float` and `float(`*p*`)` for specifying inexact numeric types. Here, *p* specifies the minimum acceptable precision in binary digits. PostgreSQL accepts `float(1)` to `float(24)` as selecting the `real` type, while `float(25)` to `float(53)` select `double precision`. Values of *p* outside the allowed range draw an error. `float` with no precision specified is taken to mean `double precision`.

> **Note:** Prior to PostgreSQL 7.4, the precision in `float(`*p*`)` was taken to mean so many decimal digits. This has been corrected to match the SQL standard, which specifies that the precision is measured in binary digits. The assumption that `real` and `double precision` have exactly 24 and 53 bits in the mantissa respectively is correct for IEEE-standard floating point implementations. On non-IEEE platforms it may be off a little, but for simplicity the same ranges of *p* are used on all platforms.

8.1.4. Serial Types

The data types `serial` and `bigserial` are not true types, but merely a notational convenience for setting up unique identifier columns (similar to the `AUTO_INCREMENT` property supported by some other databases). In the current implementation, specifying

```
CREATE TABLE tablename (
    colname SERIAL
);
```

is equivalent to specifying:

```
CREATE SEQUENCE tablename_colname_seq;
CREATE TABLE tablename (
    colname integer NOT NULL DEFAULT nextval('tablename_colname_seq')
);
ALTER SEQUENCE tablename_colname_seq OWNED BY tablename.colname;
```

Thus, we have created an integer column and arranged for its default values to be assigned from a sequence generator. A `NOT NULL` constraint is applied to ensure that a null value cannot be explicitly inserted, either. (In most cases you would also want to attach a `UNIQUE` or `PRIMARY KEY` constraint to prevent duplicate values from being inserted by accident, but this is not automatic.) Lastly, the sequence is marked as "owned by" the column, so that it will be dropped if the column or table is dropped.

> **Note:** Prior to PostgreSQL 7.3, `serial` implied `UNIQUE`. This is no longer automatic. If you wish a serial column to be in a unique constraint or a primary key, it must now be specified, same as with any other data type.

To insert the next value of the sequence into the `serial` column, specify that the `serial` column should be assigned its default value. This can be done either by excluding the column from the list of columns in the `INSERT` statement, or through the use of the `DEFAULT` key word.

The type names `serial` and `serial4` are equivalent: both create `integer` columns. The type names `bigserial` and `serial8` work just the same way, except that they create a `bigint` column. `bigserial` should be used if you anticipate the use of more than 2^{31} identifiers over the lifetime of the table.

The sequence created for a `serial` column is automatically dropped when the owning column is dropped. You can drop the sequence without dropping the column, but this will force removal of the column default expression.

8.2. Monetary Types

> **Note:** The `money` type is deprecated. Use `numeric` or `decimal` instead, in combination with the `to_char` function.

The `money` type stores a currency amount with a fixed fractional precision; see Table 8-3. Input is accepted in a variety of formats, including integer and floating-point literals, as well as "typical" currency formatting, such as `'$1,000.00'`. Output is generally in the latter form but depends on the locale.

Table 8-3. Monetary Types

Name	Storage Size	Description	Range
money	4 bytes	currency amount	-21474836.48 to +21474836.47

8.3. Character Types

Table 8-4. Character Types

Name	Description
`character varying(n)`, `varchar(n)`	variable-length with limit
`character(n)`, `char(n)`	fixed-length, blank padded
`text`	variable unlimited length

Table 8-4 shows the general-purpose character types available in PostgreSQL.

SQL defines two primary character types: `character varying(n)` and `character(n)`, where *n* is a positive integer. Both of these types can store strings up to *n* characters in length. An attempt to store a longer string into a column of these types will result in an error, unless the excess characters are all spaces, in which case the string will be truncated to the maximum length. (This somewhat bizarre exception is required by the SQL standard.) If the string to be stored is shorter than the declared length, values of type `character` will be space-padded; values of type `character varying` will simply store the shorter string.

If one explicitly casts a value to `character varying(n)` or `character(n)`, then an over-length value will be truncated to *n* characters without raising an error. (This too is required by the SQL standard.)

The notations `varchar(n)` and `char(n)` are aliases for `character varying(n)` and `character(n)`, respectively. `character` without length specifier is equivalent to `character(1)`. If `character varying` is used without length specifier, the type accepts strings of any size. The latter is a PostgreSQL extension.

In addition, PostgreSQL provides the `text` type, which stores strings of any length. Although the type `text` is not in the SQL standard, several other SQL database management systems have it as well.

Values of type `character` are physically padded with spaces to the specified width *n*, and are stored and displayed that way. However, the padding spaces are treated as semantically insignificant. Trailing spaces are disregarded when comparing two values of type `character`, and they will be removed when converting a `character` value to one of the other string types. Note that trailing spaces *are* semantically significant in `character varying` and `text` values.

The storage requirement for data of these types is 4 bytes plus the actual string, and in case of `character` plus the padding. Long strings are compressed by the system automatically, so the physical requirement on disk may be less. Long values are also stored in background tables so they do not interfere with rapid access to the shorter column values. In any case, the longest possible character string that can be stored is about 1 GB. (The maximum value that will be allowed for *n* in the data type declaration is less than that. It wouldn't be very useful to change this because with multibyte character encodings the number of characters and bytes can be quite different anyway. If you desire to store long strings with no specific upper limit, use `text` or `character varying` without a length specifier, rather than making up an arbitrary length limit.)

> **Tip:** There are no performance differences between these three types, apart from the increased storage size when using the blank-padded type. While `character(n)` has performance advantages in some other database systems, it has no such advantages in PostgreSQL. In most situations `text` or `character varying` should be used instead.

Refer to Section 4.1.2.1 for information about the syntax of string literals, and to Chapter 9 for information about available operators and functions. The database character set determines the character set used to store textual values; for more information on character set support, refer to Section 21.2.

Example 8-1. Using the character types

```
CREATE TABLE test1 (a character(4));
INSERT INTO test1 VALUES ('ok');
SELECT a, char_length(a) FROM test1; -- ❶
  a   | char_length
------+-------------
 ok   |           2

CREATE TABLE test2 (b varchar(5));
INSERT INTO test2 VALUES ('ok');
INSERT INTO test2 VALUES ('good      ');
INSERT INTO test2 VALUES ('too long');
ERROR:  value too long for type character varying(5)
INSERT INTO test2 VALUES ('too long'::varchar(5)); -- explicit truncation
SELECT b, char_length(b) FROM test2;
   b   | char_length
-------+-------------
 ok    |           2
 good  |           5
 too l |           5
```

❶ The `char_length` function is discussed in Section 9.4.

There are two other fixed-length character types in PostgreSQL, shown in Table 8-5. The `name` type exists *only* for storage of identifiers in the internal system catalogs and is not intended for use by the general user. Its length is currently defined as 64 bytes (63 usable characters plus terminator) but should be referenced using the constant `NAMEDATALEN`. The length is set at compile time (and is therefore adjustable for special uses); the default maximum length may change in a future release. The type `"char"` (note the quotes) is different from `char(1)` in that it only uses one byte of storage. It is internally used in the system catalogs as a poor-man's enumeration type.

Table 8-5. Special Character Types

Name	Storage Size	Description
`"char"`	1 byte	single-character internal type
`name`	64 bytes	internal type for object names

8.4. Binary Data Types

The `bytea` data type allows storage of binary strings; see Table 8-6.

Table 8-6. Binary Data Types

Name	Storage Size	Description
`bytea`	4 bytes plus the actual binary string	variable-length binary string

A binary string is a sequence of octets (or bytes). Binary strings are distinguished from character strings by two characteristics: First, binary strings specifically allow storing octets of value zero and other "non-printable" octets (usually, octets outside the range 32 to 126). Character strings disallow zero octets, and also disallow any other octet values and sequences of octet values that are invalid according to the database's selected character set encoding. Second, operations on binary strings process the actual bytes, whereas the processing of character strings depends on locale settings. In short, binary strings are appropriate for storing data that the programmer thinks of as "raw bytes", whereas character strings are appropriate for storing text.

When entering `bytea` values, octets of certain values *must* be escaped (but all octet values *can* be escaped) when used as part of a string literal in an SQL statement. In general, to escape an octet, it is converted into the three-digit octal number equivalent of its decimal octet value, and preceded by two backslashes. Table 8-7 shows the characters that must be escaped, and gives the alternate escape sequences where applicable.

Table 8-7. `bytea` Literal Escaped Octets

Decimal Octet Value	Description	Escaped Input Representation	Example	Output Representation
0	zero octet	`E'\\000'`	`SELECT E'\\000'::bytea;`	`\000`
39	single quote	`''` or `E'\\047'`	`SELECT E'\''::bytea;`	`'`
92	backslash	`E'\\\\'` or `E'\\134'`	`SELECT E'\\\\'::bytea;`	`\\`
0 to 31 and 127 to 255	"non-printable" octets	`E'\\`*xxx*`'` (octal value)	`SELECT E'\\001'::bytea;`	`\001`

The requirement to escape "non-printable" octets actually varies depending on locale settings. In some instances you can get away with leaving them unescaped. Note that the result in each of the examples in Table 8-7 was exactly one octet in length, even though the output representation of the zero octet and backslash are more than one character.

The reason that you have to write so many backslashes, as shown in Table 8-7, is that an input string written as a string literal must pass through two parse phases in the PostgreSQL server. The first backslash of each pair is interpreted as an escape character by the string-literal parser (assuming escape string syntax is used) and is therefore consumed, leaving the second backslash of the pair. (Dollar-quoted strings can be used to avoid this level of escaping.) The remaining backslash is then recognized by the `bytea` input function as starting either a three digit octal value or escaping another backslash. For example, a string literal passed to the server as `E'\\001'` becomes `\001` after passing through the escape string parser. The `\001` is then sent to the `bytea` input function, where it is converted to a single octet with a decimal value

of 1. Note that the single-quote character is not treated specially by `bytea`, so it follows the normal rules for string literals. (See also Section 4.1.2.1.)

`Bytea` octets are also escaped in the output. In general, each "non-printable" octet is converted into its equivalent three-digit octal value and preceded by one backslash. Most "printable" octets are represented by their standard representation in the client character set. The octet with decimal value 92 (backslash) has a special alternative output representation. Details are in Table 8-8.

Table 8-8. `bytea` Output Escaped Octets

Decimal Octet Value	Description	Escaped Output Representation	Example	Output Result
92	backslash	`\\`	`SELECT E'\\134'::bytea;`	`\\`
0 to 31 and 127 to 255	"non-printable" octets	`\`*xxx* (octal value)	`SELECT E'\\001'::bytea;`	`\001`
32 to 126	"printable" octets	client character set representation	`SELECT E'\\176'::bytea;`	~

Depending on the front end to PostgreSQL you use, you may have additional work to do in terms of escaping and unescaping `bytea` strings. For example, you may also have to escape line feeds and carriage returns if your interface automatically translates these.

The SQL standard defines a different binary string type, called `BLOB` or `BINARY LARGE OBJECT`. The input format is different from `bytea`, but the provided functions and operators are mostly the same.

8.5. Date/Time Types

PostgreSQL supports the full set of SQL date and time types, shown in Table 8-9. The operations available on these data types are described in Section 9.9.

Table 8-9. Date/Time Types

Name	Storage Size	Description	Low Value	High Value	Resolution
`timestamp [ (`*p*`) ] [ without time zone ]`	8 bytes	both date and time	4713 BC	5874897 AD	1 microsecond / 14 digits
`timestamp [ (`*p*`) ] with time zone`	8 bytes	both date and time, with time zone	4713 BC	5874897 AD	1 microsecond / 14 digits
`interval [ (`*p*`) ]`	12 bytes	time intervals	-178000000 years	178000000 years	1 microsecond / 14 digits

Name	Storage Size	Description	Low Value	High Value	Resolution
date	4 bytes	dates only	4713 BC	5874897 AD	1 day
time [(p)] [without time zone]	8 bytes	times of day only	00:00:00	24:00:00	1 microsecond / 14 digits
time [(p)] with time zone	12 bytes	times of day only, with time zone	00:00:00+1459	24:00:00-1459	1 microsecond / 14 digits

Note: Prior to PostgreSQL 7.3, writing just `timestamp` was equivalent to `timestamp with time zone`. This was changed for SQL compliance.

`time`, `timestamp`, and `interval` accept an optional precision value *p* which specifies the number of fractional digits retained in the seconds field. By default, there is no explicit bound on precision. The allowed range of *p* is from 0 to 6 for the `timestamp` and `interval` types.

Note: When `timestamp` values are stored as double precision floating-point numbers (currently the default), the effective limit of precision may be less than 6. `timestamp` values are stored as seconds before or after midnight 2000-01-01. Microsecond precision is achieved for dates within a few years of 2000-01-01, but the precision degrades for dates further away. When `timestamp` values are stored as eight-byte integers (a compile-time option), microsecond precision is available over the full range of values. However eight-byte integer timestamps have a more limited range of dates than shown above: from 4713 BC up to 294276 AD. The same compile-time option also determines whether `time` and `interval` values are stored as floating-point or eight-byte integers. In the floating-point case, large `interval` values degrade in precision as the size of the interval increases.

For the `time` types, the allowed range of *p* is from 0 to 6 when eight-byte integer storage is used, or from 0 to 10 when floating-point storage is used.

The type `time with time zone` is defined by the SQL standard, but the definition exhibits properties which lead to questionable usefulness. In most cases, a combination of `date`, `time`, `timestamp without time zone`, and `timestamp with time zone` should provide a complete range of date/time functionality required by any application.

The types `abstime` and `reltime` are lower precision types which are used internally. You are discouraged from using these types in new applications and are encouraged to move any old ones over when appropriate. Any or all of these internal types might disappear in a future release.

8.5.1. Date/Time Input

Date and time input is accepted in almost any reasonable format, including ISO 8601, SQL-compatible, traditional POSTGRES, and others. For some formats, ordering of month, day, and year in date input is ambiguous and there is support for specifying the expected ordering of these fields. Set the DateStyle parameter to `MDY` to select month-day-year interpretation, `DMY` to select day-month-year interpretation, or `YMD` to select year-month-day interpretation.

PostgreSQL is more flexible in handling date/time input than the SQL standard requires. See Appendix B for the exact parsing rules of date/time input and for the recognized text fields including months, days of the week, and time zones.

Remember that any date or time literal input needs to be enclosed in single quotes, like text strings. Refer to Section 4.1.2.5 for more information. SQL requires the following syntax

```
type [ (p) ] 'value'
```

where *p* in the optional precision specification is an integer corresponding to the number of fractional digits in the seconds field. Precision can be specified for `time`, `timestamp`, and `interval` types. The allowed values are mentioned above. If no precision is specified in a constant specification, it defaults to the precision of the literal value.

8.5.1.1. Dates

Table 8-10 shows some possible inputs for the `date` type.

Table 8-10. Date Input

Example	Description
January 8, 1999	unambiguous in any `datestyle` input mode
1999-01-08	ISO 8601; January 8 in any mode (recommended format)
1/8/1999	January 8 in `MDY` mode; August 1 in `DMY` mode
1/18/1999	January 18 in `MDY` mode; rejected in other modes
01/02/03	January 2, 2003 in `MDY` mode; February 1, 2003 in `DMY` mode; February 3, 2001 in `YMD` mode
1999-Jan-08	January 8 in any mode
Jan-08-1999	January 8 in any mode
08-Jan-1999	January 8 in any mode
99-Jan-08	January 8 in `YMD` mode, else error
08-Jan-99	January 8, except error in `YMD` mode
Jan-08-99	January 8, except error in `YMD` mode
19990108	ISO 8601; January 8, 1999 in any mode
990108	ISO 8601; January 8, 1999 in any mode
1999.008	year and day of year
J2451187	Julian day
January 8, 99 BC	year 99 before the Common Era

8.5.1.2. Times

The time-of-day types are `time [ (p) ] without time zone` and `time [ (p) ] with time zone`. Writing just `time` is equivalent to `time without time zone`.

Valid input for these types consists of a time of day followed by an optional time zone. (See Table 8-11 and Table 8-12.) If a time zone is specified in the input for `time without time zone`, it is silently ignored. You can also specify a date but it will be ignored, except when you use a time zone name that involves a daylight-savings rule, such as `America/New_York`. In this case specifying the date is required in order to determine whether standard or daylight-savings time applies. The appropriate time zone offset is recorded in the `time with time zone` value.

Table 8-11. Time Input

Example	Description
`04:05:06.789`	ISO 8601
`04:05:06`	ISO 8601
`04:05`	ISO 8601
`040506`	ISO 8601
`04:05 AM`	same as 04:05; AM does not affect value
`04:05 PM`	same as 16:05; input hour must be <= 12
`04:05:06.789-8`	ISO 8601
`04:05:06-08:00`	ISO 8601
`04:05-08:00`	ISO 8601
`040506-08`	ISO 8601
`04:05:06 PST`	time zone specified by abbreviation
`2003-04-12 04:05:06 America/New_York`	time zone specified by full name

Table 8-12. Time Zone Input

Example	Description
`PST`	Abbreviation (for Pacific Standard Time)
`America/New_York`	Full time zone name
`PST8PDT`	POSIX-style time zone specification
`-8:00`	ISO-8601 offset for PST
`-800`	ISO-8601 offset for PST
`-8`	ISO-8601 offset for PST
`zulu`	Military abbreviation for UTC
`z`	Short form of `zulu`

Refer to Section 8.5.3 for more information on how to specify time zones.

8.5.1.3. Time Stamps

Valid input for the time stamp types consists of a concatenation of a date and a time, followed by an optional time zone, followed by an optional `AD` or `BC`. (Alternatively, `AD`/`BC` can appear before the time

zone, but this is not the preferred ordering.) Thus

```
1999-01-08 04:05:06
```

and

```
1999-01-08 04:05:06 -8:00
```

are valid values, which follow the ISO 8601 standard. In addition, the wide-spread format

```
January 8 04:05:06 1999 PST
```

is supported.

The SQL standard differentiates `timestamp without time zone` and `timestamp with time zone` literals by the presence of a "+" or "-". Hence, according to the standard,

```
TIMESTAMP '2004-10-19 10:23:54'
```

is a `timestamp without time zone`, while

```
TIMESTAMP '2004-10-19 10:23:54+02'
```

is a `timestamp with time zone`. PostgreSQL never examines the content of a literal string before determining its type, and therefore will treat both of the above as `timestamp without time zone`. To ensure that a literal is treated as `timestamp with time zone`, give it the correct explicit type:

```
TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'
```

In a literal that has been decided to be `timestamp without time zone`, PostgreSQL will silently ignore any time zone indication. That is, the resulting value is derived from the date/time fields in the input value, and is not adjusted for time zone.

For `timestamp with time zone`, the internally stored value is always in UTC (Universal Coordinated Time, traditionally known as Greenwich Mean Time, GMT). An input value that has an explicit time zone specified is converted to UTC using the appropriate offset for that time zone. If no time zone is stated in the input string, then it is assumed to be in the time zone indicated by the system's timezone parameter, and is converted to UTC using the offset for the `timezone` zone.

When a `timestamp with time zone` value is output, it is always converted from UTC to the current `timezone` zone, and displayed as local time in that zone. To see the time in another time zone, either change `timezone` or use the `AT TIME ZONE` construct (see Section 9.9.3).

Conversions between `timestamp without time zone` and `timestamp with time zone` normally assume that the `timestamp without time zone` value should be taken or given as `timezone` local time. A different zone reference can be specified for the conversion using `AT TIME ZONE`.

8.5.1.4. Intervals

`interval` values can be written with the following syntax:

```
[@] quantity unit [quantity unit...] [direction]
```

Where: *quantity* is a number (possibly signed); *unit* is microsecond, millisecond, second, minute, hour, day, week, month, year, decade, century, millennium, or abbreviations or plurals of these units; *direction* can be ago or empty. The at sign (@) is optional noise. The amounts of different units are implicitly added up with appropriate sign accounting.

Quantities of days, hours, minutes, and seconds can be specified without explicit unit markings. For example, '1 12:59:10' is read the same as '1 day 12 hours 59 min 10 sec'.

The optional subsecond precision *p* should be between 0 and 6, and defaults to the precision of the input literal.

Internally interval values are stored as months, days, and seconds. This is done because the number of days in a month varies, and a day can have 23 or 25 hours if a daylight savings time adjustment is involved. Because intervals are usually created from constant strings or timestamp subtraction, this storage method works well in most cases. Functions justify_days and justify_hours are available for adjusting days and hours that overflow their normal periods.

8.5.1.5. Special Values

PostgreSQL supports several special date/time input values for convenience, as shown in Table 8-13. The values infinity and -infinity are specially represented inside the system and will be displayed the same way; but the others are simply notational shorthands that will be converted to ordinary date/time values when read. (In particular, now and related strings are converted to a specific time value as soon as they are read.) All of these values need to be written in single quotes when used as constants in SQL commands.

Table 8-13. Special Date/Time Inputs

Input String	Valid Types	Description
epoch	date, timestamp	1970-01-01 00:0 system time zero
infinity	timestamp	later than all othe
-infinity	timestamp	earlier than all ot
now	date, time, timestamp	current transactio
today	date, timestamp	midnight today
tomorrow	date, timestamp	midnight tomorro
yesterday	date, timestamp	midnight yesterd
allballs	time	00:00:00.00 UTC

The following SQL-compatible functions can also be used to obtain the current time value for the corresponding data type: CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, LOCALTIME, LOCALTIMESTAMP. The latter four accept an optional subsecond precision specification. (See Section 9.9.4.) Note however that these are SQL functions and are *not* recognized as data input strings.

8.5.2. Date/Time Output

The output format of the date/time types can be set to one of the four styles ISO 8601, SQL (Ingres), traditional POSTGRES, and German, using the command `SET datestyle`. The default is the ISO format. (The SQL standard requires the use of the ISO 8601 format. The name of the "SQL" output format is a historical accident.) Table 8-14 shows examples of each output style. The output of the `date` and `time` types is of course only the date or time part in accordance with the given examples.

Table 8-14. Date/Time Output Styles

Style Specification	Description	Example
ISO	ISO 8601/SQL standard	1997-12-17 07:37:16-08
SQL	traditional style	12/17/1997 07:37:16.00 PST
POSTGRES	original style	Wed Dec 17 07:37:16 1997 PST
German	regional style	17.12.1997 07:37:16.00 PST

In the SQL and POSTGRES styles, day appears before month if DMY field ordering has been specified, otherwise month appears before day. (See Section 8.5.1 for how this setting also affects interpretation of input values.) Table 8-15 shows an example.

Table 8-15. Date Order Conventions

`datestyle` Setting	Input Ordering	Example Output
`SQL, DMY`	*day*/*month*/*year*	17/12/1997 15:37:16.00 CET
`SQL, MDY`	*month*/*day*/*year*	12/17/1997 07:37:16.00 PST
`Postgres, DMY`	*day*/*month*/*year*	Wed 17 Dec 07:37:16 1997 PST

`interval` output looks like the input format, except that units like `century` or `week` are converted to years and days and `ago` is converted to an appropriate sign. In ISO mode the output looks like

```
[ quantity unit [ ... ] ] [ days ] [ hours:minutes:seconds ]
```

The date/time styles can be selected by the user using the `SET datestyle` command, the DateStyle parameter in the `postgresql.conf` configuration file, or the `PGDATESTYLE` environment variable on the server or client. The formatting function `to_char` (see Section 9.8) is also available as a more flexible way to format the date/time output.

8.5.3. Time Zones

Time zones, and time-zone conventions, are influenced by political decisions, not just earth geometry. Time zones around the world became somewhat standardized during the 1900's, but continue to be prone to arbitrary changes, particularly with respect to daylight-savings rules. PostgreSQL currently supports daylight-savings rules over the time period 1902 through 2038 (corresponding to the full range of conventional Unix system time). Times outside that range are taken to be in "standard time" for the selected time zone, no matter what part of the year they fall in.

PostgreSQL endeavors to be compatible with the SQL standard definitions for typical usage. However, the SQL standard has an odd mix of date and time types and capabilities. Two obvious problems are:

- Although the `date` type does not have an associated time zone, the `time` type can. Time zones in the real world have little meaning unless associated with a date as well as a time, since the offset may vary through the year with daylight-saving time boundaries.
- The default time zone is specified as a constant numeric offset from UTC. It is therefore not possible to adapt to daylight-saving time when doing date/time arithmetic across DST boundaries.

To address these difficulties, we recommend using date/time types that contain both date and time when using time zones. We recommend *not* using the type `time with time zone` (though it is supported by PostgreSQL for legacy applications and for compliance with the SQL standard). PostgreSQL assumes your local time zone for any type containing only date or time.

All timezone-aware dates and times are stored internally in UTC. They are converted to local time in the zone specified by the timezone configuration parameter before being displayed to the client.

PostgreSQL allows you to specify time zones in three different forms:

- A full time zone name, for example `America/New_York`. The recognized time zone names are listed in the `pg_timezone_names` view (see Section 43.49). PostgreSQL uses the widely-used `zic` time zone data for this purpose, so the same names are also recognized by much other software.
- A time zone abbreviation, for example `PST`. Such a specification merely defines a particular offset from UTC, in contrast to full time zone names which may imply a set of daylight savings transition-date rules as well. The recognized abbreviations are listed in the `pg_timezone_abbrevs` view (see Section 43.48). You cannot set the configuration parameter timezone using a time zone abbreviation, but you can use abbreviations in date/time input values and with the `AT TIME ZONE` operator.
- In addition to the timezone names and abbreviations, PostgreSQL will accept POSIX-style time zone specifications of the form *`STDoffset`* or *`STDoffsetDST`*, where *`STD`* is a zone abbreviation, *`offset`* is a numeric offset in hours west from UTC, and *`DST`* is an optional daylight-savings zone abbreviation, assumed to stand for one hour ahead of the given offset. For example, if `EST5EDT` were not already a recognized zone name, it would be accepted and would be functionally equivalent to USA East Coast time. When a daylight-savings zone name is present, it is assumed to be used according to the same daylight-savings transition rules used in the `zic` time zone database's `posixrules` entry. In a standard PostgreSQL installation, `posixrules` is the same as `US/Eastern`, so that POSIX-style time zone specifications follow USA daylight-savings rules. If needed, you can adjust this behavior by replacing the `posixrules` file.

There is a conceptual and practical difference between the abbreviations and the full names: abbreviations always represent a fixed offset from UTC, whereas most of the full names imply a local daylight-savings time rule and so have two possible UTC offsets.

One should be wary that the POSIX-style time zone feature can lead to silently accepting bogus input, since there is no check on the reasonableness of the zone abbreviations. For example, `SET TIMEZONE TO FOOBAR0` will work, leaving the system effectively using a rather peculiar abbreviation for UTC.

In all cases, timezone names are recognized case-insensitively. (This is a change from PostgreSQL versions prior to 8.2, which were case-sensitive in some contexts and not others.)

Neither full names nor abbreviations are hard-wired into the server; they are obtained from configuration files stored under .../share/timezone/ and .../share/timezonesets/ of the installation directory (see Section B.3).

The timezone configuration parameter can be set in the file postgresql.conf, or in any of the other standard ways described in Chapter 17. There are also several special ways to set it:

- If timezone is not specified in postgresql.conf nor as a server command-line option, the server attempts to use the value of the TZ environment variable as the default time zone. If TZ is not defined or is not any of the time zone names known to PostgreSQL, the server attempts to determine the operating system's default time zone by checking the behavior of the C library function localtime(). The default time zone is selected as the closest match among PostgreSQL's known time zones.
- The SQL command SET TIME ZONE sets the time zone for the session. This is an alternative spelling of SET TIMEZONE TO with a more SQL-spec-compatible syntax.
- The PGTZ environment variable, if set at the client, is used by libpq applications to send a SET TIME ZONE command to the server upon connection.

8.5.4. Internals

PostgreSQL uses Julian dates for all date/time calculations. They have the nice property of correctly predicting/calculating any date more recent than 4713 BC to far into the future, using the assumption that the length of the year is 365.2425 days.

Date conventions before the 19th century make for interesting reading, but are not consistent enough to warrant coding into a date/time handler.

8.6. Boolean Type

PostgreSQL provides the standard SQL type boolean. boolean can have one of only two states: "true" or "false". A third state, "unknown", is represented by the SQL null value.

Valid literal values for the "true" state are:

```
TRUE
't'
'true'
'y'
'yes'
'1'
```

For the "false" state, the following values can be used:

```
FALSE
'f'
'false'
```

```
  'n'
  'no'
  '0'
```

Using the key words `TRUE` and `FALSE` is preferred (and SQL-compliant).

Example 8-2. Using the `boolean` type

```
CREATE TABLE test1 (a boolean, b text);
INSERT INTO test1 VALUES (TRUE, 'sic est');
INSERT INTO test1 VALUES (FALSE, 'non est');
SELECT * FROM test1;
 a |    b
---+---------
 t | sic est
 f | non est

SELECT * FROM test1 WHERE a;
 a |    b
---+---------
 t | sic est
```

Example 8-2 shows that `boolean` values are output using the letters `t` and `f`.

`boolean` uses 1 byte of storage.

8.7. Geometric Types

Geometric data types represent two-dimensional spatial objects. Table 8-16 shows the geometric types available in PostgreSQL. The most fundamental type, the point, forms the basis for all of the other types.

Table 8-16. Geometric Types

Name	Storage Size	Representation	Description
`point`	16 bytes	Point on the plane	(x,y)
`line`	32 bytes	Infinite line (not fully implemented)	((x1,y1),(x2,y2))
`lseg`	32 bytes	Finite line segment	((x1,y1),(x2,y2))
`box`	32 bytes	Rectangular box	((x1,y1),(x2,y2))
`path`	16+16n bytes	Closed path (similar to polygon)	((x1,y1),...)
`path`	16+16n bytes	Open path	[(x1,y1),...]
`polygon`	40+16n bytes	Polygon (similar to closed path)	((x1,y1),...)
`circle`	24 bytes	Circle	<(x,y),r> (center and radius)

A rich set of functions and operators is available to perform various geometric operations such as scaling,

translation, rotation, and determining intersections. They are explained in Section 9.10.

8.7.1. Points

Points are the fundamental two-dimensional building block for geometric types. Values of type `point` are specified using the following syntax:

```
( x , y )
  x , y
```

where *x* and *y* are the respective coordinates as floating-point numbers.

8.7.2. Line Segments

Line segments (`lseg`) are represented by pairs of points. Values of type `lseg` are specified using the following syntax:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

where `(x1,y1)` and `(x2,y2)` are the end points of the line segment.

8.7.3. Boxes

Boxes are represented by pairs of points that are opposite corners of the box. Values of type `box` are specified using the following syntax:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

where `(x1,y1)` and `(x2,y2)` are any two opposite corners of the box.

Boxes are output using the first syntax. The corners are reordered on input to store the upper right corner, then the lower left corner. Other corners of the box can be entered, but the lower left and upper right corners are determined from the input and stored.

8.7.4. Paths

Paths are represented by lists of connected points. Paths can be *open*, where the first and last points in the list are not considered connected, or *closed*, where the first and last points are considered connected.

Values of type `path` are specified using the following syntax:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
  ( x1 , y1 ) , ... , ( xn , yn )
```

```
( x1 , y1   , ... ,   xn , yn )
  x1 , y1   , ... ,   xn , yn
```

where the points are the end points of the line segments comprising the path. Square brackets ([]) indicate an open path, while parentheses (()) indicate a closed path.

Paths are output using the first syntax.

8.7.5. Polygons

Polygons are represented by lists of points (the vertexes of the polygon). Polygons should probably be considered equivalent to closed paths, but are stored differently and have their own set of support routines.

Values of type `polygon` are specified using the following syntax:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

where the points are the end points of the line segments comprising the boundary of the polygon.

Polygons are output using the first syntax.

8.7.6. Circles

Circles are represented by a center point and a radius. Values of type `circle` are specified using the following syntax:

```
< ( x , y ) , r >
( ( x , y ) , r )
  ( x , y ) , r
    x , y   , r
```

where `(x,y)` is the center and `r` is the radius of the circle.

Circles are output using the first syntax.

8.8. Network Address Types

PostgreSQL offers data types to store IPv4, IPv6, and MAC addresses, as shown in Table 8-17. It is preferable to use these types instead of plain text types to store network addresses, because these types offer input error checking and several specialized operators and functions (see Section 9.11).

Name	Storage Size	Description

Table 8-17. Network Address Types

Name	Storage Size	Description
`cidr`	12 or 24 bytes	IPv4 and IPv6 networks
`inet`	12 or 24 bytes	IPv4 and IPv6 hosts and networks
`macaddr`	6 bytes	MAC addresses

When sorting `inet` or `cidr` data types, IPv4 addresses will always sort before IPv6 addresses, including IPv4 addresses encapsulated or mapped into IPv6 addresses, such as ::10.2.3.4 or ::ffff:10.4.3.2.

8.8.1. `inet`

The `inet` type holds an IPv4 or IPv6 host address, and optionally the identity of the subnet it is in, all in one field. The subnet identity is represented by stating how many bits of the host address represent the network address (the "netmask"). If the netmask is 32 and the address is IPv4, then the value does not indicate a subnet, only a single host. In IPv6, the address length is 128 bits, so 128 bits specify a unique host address. Note that if you want to accept networks only, you should use the `cidr` type rather than `inet`.

The input format for this type is *`address/y`* where *`address`* is an IPv4 or IPv6 address and *`y`* is the number of bits in the netmask. If the *`/y`* part is left off, then the netmask is 32 for IPv4 and 128 for IPv6, so the value represents just a single host. On display, the *`/y`* portion is suppressed if the netmask specifies a single host.

8.8.2. `cidr`

The `cidr` type holds an IPv4 or IPv6 network specification. Input and output formats follow Classless Internet Domain Routing conventions. The format for specifying networks is *`address/y`* where *`address`* is the network represented as an IPv4 or IPv6 address, and *`y`* is the number of bits in the netmask. If *`y`* is omitted, it is calculated using assumptions from the older classful network numbering system, except that it will be at least large enough to include all of the octets written in the input. It is an error to specify a network address that has bits set to the right of the specified netmask.

Table 8-18 shows some examples.

Table 8-18. `cidr` Type Input Examples

`cidr` Input	`cidr` Output	`abbrev(cidr)`
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24

cidr Input	cidr Output	abbrev(cidr)
128.1	128.1.0.0/16	128.1/16
128	128.0.0.0/16	128.0/16
128.1.2	128.1.2.0/24	128.1.2/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/32
2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8:3:ba::/64
2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1
::ffff:1.2.3.0/120	::ffff:1.2.3.0/120	::ffff:1.2.3/120
::ffff:1.2.3.0/128	::ffff:1.2.3.0/128	::ffff:1.2.3.0/128

8.8.3. inet vs. cidr

The essential difference between `inet` and `cidr` data types is that `inet` accepts values with nonzero bits to the right of the netmask, whereas `cidr` does not.

> **Tip:** If you do not like the output format for `inet` or `cidr` values, try the functions `host`, `text`, and `abbrev`.

8.8.4. macaddr

The `macaddr` type stores MAC addresses, i.e., Ethernet card hardware addresses (although MAC addresses are used for other purposes as well). Input is accepted in various customary formats, including

```
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'08-00-2b-01-02-03'
'08:00:2b:01:02:03'
```

which would all specify the same address. Upper and lower case is accepted for the digits `a` through `f`. Output is always in the last of the forms shown.

8.9. Bit String Types

Bit strings are strings of 1's and 0's. They can be used to store or visualize bit masks. There are two SQL bit types: `bit(n)` and `bit varying(n)`, where *n* is a positive integer.

`bit` type data must match the length *n* exactly; it is an error to attempt to store shorter or longer bit strings. `bit varying` data is of variable length up to the maximum length *n*; longer strings will be rejected. Writing `bit` without a length is equivalent to `bit(1)`, while `bit varying` without a length specification means unlimited length.

> **Note:** If one explicitly casts a bit-string value to `bit(n)`, it will be truncated or zero-padded on the right to be exactly *n* bits, without raising an error. Similarly, if one explicitly casts a bit-string value to `bit varying(n)`, it will be truncated on the right if it is more than *n* bits.

Refer to Section 4.1.2.3 for information about the syntax of bit string constants. Bit-logical operators and string manipulation functions are available; see Section 9.6.

Example 8-3. Using the bit string types

```
CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B'101', B'00');
INSERT INTO test VALUES (B'10', B'101');
ERROR:  bit string length 2 does not match type bit(3)
INSERT INTO test VALUES (B'10'::bit(3), B'101');
SELECT * FROM test;
  a  |  b
-----+-----
 101 | 00
 100 | 101
```

8.10. Arrays

PostgreSQL allows columns of a table to be defined as variable-length multidimensional arrays. Arrays of any built-in or user-defined base type can be created. (Arrays of composite types or domains are not yet supported, however.)

8.10.1. Declaration of Array Types

To illustrate the use of array types, we create this table:

```
CREATE TABLE sal_emp (
    name            text,
    pay_by_quarter  integer[],
    schedule        text[][]
);
```

As shown, an array data type is named by appending square brackets ([]) to the data type name of the array elements. The above command will create a table named sal_emp with a column of type text (name), a one-dimensional array of type integer (pay_by_quarter), which represents the employee's salary by quarter, and a two-dimensional array of text (schedule), which represents the employee's weekly schedule.

The syntax for CREATE TABLE allows the exact size of arrays to be specified, for example:

```
CREATE TABLE tictactoe (
    squares   integer[3][3]
);
```

However, the current implementation does not enforce the array size limits — the behavior is the same as for arrays of unspecified length.

Actually, the current implementation does not enforce the declared number of dimensions either. Arrays of a particular element type are all considered to be of the same type, regardless of size or number of dimensions. So, declaring number of dimensions or sizes in CREATE TABLE is simply documentation, it does not affect run-time behavior.

An alternative syntax, which conforms to the SQL standard, may be used for one-dimensional arrays. pay_by_quarter could have been defined as:

```
    pay_by_quarter  integer ARRAY[4],
```

This syntax requires an integer constant to denote the array size. As before, however, PostgreSQL does not enforce the size restriction.

8.10.2. Array Value Input

To write an array value as a literal constant, enclose the element values within curly braces and separate them by commas. (If you know C, this is not unlike the C syntax for initializing structures.) You may put double quotes around any element value, and must do so if it contains commas or curly braces. (More details appear below.) Thus, the general format of an array constant is the following:

```
'{ val1 delim val2 delim ... }'
```

where *delim* is the delimiter character for the type, as recorded in its pg_type entry. Among the standard data types provided in the PostgreSQL distribution, type box uses a semicolon (;) but all the others use comma (,). Each *val* is either a constant of the array element type, or a subarray. An example of an array constant is

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

This constant is a two-dimensional, 3-by-3 array consisting of three subarrays of integers.

To set an element of an array constant to NULL, write NULL for the element value. (Any upper- or lower-case variant of NULL will do.) If you want an actual string value "NULL", you must put double quotes around it.

(These kinds of array constants are actually only a special case of the generic type constants discussed in Section 4.1.2.5. The constant is initially treated as a string and passed to the array input conversion routine. An explicit type specification might be necessary.)

Now we can show some `INSERT` statements.

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"training", "presentation"}}');

INSERT INTO sal_emp
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"breakfast", "consulting"}, {"meeting", "lunch"}}');
```

The result of the previous two inserts looks like this:

```
SELECT * FROM sal_emp;
 name  |      pay_by_quarter       |                  schedule
-------+---------------------------+-------------------------------------------
 Bill  | {10000,10000,10000,10000} | {{meeting,lunch},{training,presentation}}
 Carol | {20000,25000,25000,25000} | {{breakfast,consulting},{meeting,lunch}}
(2 rows)
```

The `ARRAY` constructor syntax may also be used:

```
INSERT INTO sal_emp
    VALUES ('Bill',
    ARRAY[10000, 10000, 10000, 10000],
    ARRAY[['meeting', 'lunch'], ['training', 'presentation']]);

INSERT INTO sal_emp
    VALUES ('Carol',
    ARRAY[20000, 25000, 25000, 25000],
    ARRAY[['breakfast', 'consulting'], ['meeting', 'lunch']]);
```

Notice that the array elements are ordinary SQL constants or expressions; for instance, string literals are single quoted, instead of double quoted as they would be in an array literal. The `ARRAY` constructor syntax is discussed in more detail in Section 4.2.10.

Multidimensional arrays must have matching extents for each dimension. A mismatch causes an error report, for example:

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"meeting"}}');
ERROR:  multidimensional arrays must have array expressions with matching dimensions
```

8.10.3. Accessing Arrays

Now, we can run some queries on the table. First, we show how to access a single element of an array at a time. This query retrieves the names of the employees whose pay changed in the second quarter:

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <> pay_by_quarter[2];

 name
-------
 Carol
(1 row)
```

The array subscript numbers are written within square brackets. By default PostgreSQL uses the one-based numbering convention for arrays, that is, an array of *n* elements starts with `array[1]` and ends with `array[`*n*`]`.

This query retrieves the third quarter pay of all employees:

```
SELECT pay_by_quarter[3] FROM sal_emp;

 pay_by_quarter
----------------
          10000
          25000
(2 rows)
```

We can also access arbitrary rectangular slices of an array, or subarrays. An array slice is denoted by writing *lower-bound*`:`*upper-bound* for one or more array dimensions. For example, this query retrieves the first item on Bill's schedule for the first two days of the week:

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';

        schedule
------------------------
 {{meeting},{training}}
(1 row)
```

If any dimension is written as a slice, i.e contains a colon, then all dimensions are treated as slices. If a dimension is missing, it is assumed to be `[1:1]`. If a dimension has only a single number (no colon), that dimension is treated as being from `1` to the number specified. For example, `[2]` is treated as `[1:2]`, as in this example:

```
SELECT schedule[1:2][2] FROM sal_emp WHERE name = 'Bill';

                schedule
-----------------------------------------
 {{meeting,lunch},{training,presentation}}
(1 row)
```

An array subscript expression will return null if either the array itself or any of the subscript expressions are null. Also, null is returned if a subscript is outside the array bounds (this case does not raise an error). For example, if `schedule` currently has the dimensions `[1:3][1:2]` then referencing `schedule[3][3]` yields NULL. Similarly, an array reference with the wrong number of subscripts yields a null rather than an error.

An array slice expression likewise yields null if the array itself or any of the subscript expressions are null. However, in other corner cases such as selecting an array slice that is completely outside the current array bounds, a slice expression yields an empty (zero-dimensional) array instead of null. If the requested slice partially overlaps the array bounds, then it is silently reduced to just the overlapping region.

The current dimensions of any array value can be retrieved with the `array_dims` function:

```
SELECT array_dims(schedule) FROM sal_emp WHERE name = 'Carol';

 array_dims
------------
 [1:2][1:2]
(1 row)
```

`array_dims` produces a `text` result, which is convenient for people to read but perhaps not so convenient for programs. Dimensions can also be retrieved with `array_upper` and `array_lower`, which return the upper and lower bound of a specified array dimension, respectively.

```
SELECT array_upper(schedule, 1) FROM sal_emp WHERE name = 'Carol';

 array_upper
-------------
           2
(1 row)
```

8.10.4. Modifying Arrays

An array value can be replaced completely:

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'
    WHERE name = 'Carol';
```

or using the `ARRAY` expression syntax:

```
UPDATE sal_emp SET pay_by_quarter = ARRAY[25000,25000,27000,27000]
    WHERE name = 'Carol';
```

An array may also be updated at a single element:

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
    WHERE name = 'Bill';
```

or updated in a slice:

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
    WHERE name = 'Carol';
```

A stored array value can be enlarged by assigning to element(s) not already present. Any positions between those previously present and the newly assigned element(s) will be filled with nulls. For example, if array `myarray` currently has 4 elements, it will have six elements after an update that assigns to `myarray[6]`, and `myarray[5]` will contain a null. Currently, enlargement in this fashion is only allowed for one-dimensional arrays, not multidimensional arrays.

Subscripted assignment allows creation of arrays that do not use one-based subscripts. For example one might assign to `myarray[-2:7]` to create an array with subscript values running from -2 to 7.

New array values can also be constructed by using the concatenation operator, `||`.

```
SELECT ARRAY[1,2] || ARRAY[3,4];
 ?column?
-----------
 {1,2,3,4}
(1 row)

SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];
      ?column?
---------------------
 {{5,6},{1,2},{3,4}}
(1 row)
```

The concatenation operator allows a single element to be pushed on to the beginning or end of a one-dimensional array. It also accepts two *N*-dimensional arrays, or an *N*-dimensional and an *N+1*-dimensional array.

When a single element is pushed on to either the beginning or end of a one-dimensional array, the result is an array with the same lower bound subscript as the array operand. For example:

```
SELECT array_dims(1 || '[0:1]={2,3}'::int[]);
 array_dims
------------
 [0:2]
(1 row)

SELECT array_dims(ARRAY[1,2] || 3);
 array_dims
------------
 [1:3]
(1 row)
```

When two arrays with an equal number of dimensions are concatenated, the result retains the lower bound subscript of the left-hand operand's outer dimension. The result is an array comprising every element of the left-hand operand followed by every element of the right-hand operand. For example:

```
SELECT array_dims(ARRAY[1,2] || ARRAY[3,4,5]);
 array_dims
------------
 [1:5]
(1 row)

SELECT array_dims(ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]]);
 array_dims
------------
 [1:5][1:2]
(1 row)
```

When an *N*-dimensional array is pushed on to the beginning or end of an *N+1*-dimensional array, the result is analogous to the element-array case above. Each *N*-dimensional sub-array is essentially an element of the *N+1*-dimensional array's outer dimension. For example:

```
SELECT array_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);
 array_dims
------------
 [1:3][1:2]
(1 row)
```

An array can also be constructed by using the functions `array_prepend`, `array_append`, or `array_cat`. The first two only support one-dimensional arrays, but `array_cat` supports multidimensional arrays. Note that the concatenation operator discussed above is preferred over direct use of these functions. In fact, the functions exist primarily for use in implementing the concatenation operator. However, they may be directly useful in the creation of user-defined aggregates. Some examples:

```
SELECT array_prepend(1, ARRAY[2,3]);
 array_prepend
---------------
 {1,2,3}
(1 row)

SELECT array_append(ARRAY[1,2], 3);
 array_append
--------------
 {1,2,3}
(1 row)

SELECT array_cat(ARRAY[1,2], ARRAY[3,4]);
 array_cat
-----------
 {1,2,3,4}
(1 row)

SELECT array_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]);
      array_cat
```

```
---------------------
 {{1,2},{3,4},{5,6}}
(1 row)

SELECT array_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]);
      array_cat
---------------------
 {{5,6},{1,2},{3,4}}
```

8.10.5. Searching in Arrays

To search for a value in an array, you must check each value of the array. This can be done by hand, if you know the size of the array. For example:

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR
                            pay_by_quarter[2] = 10000 OR
                            pay_by_quarter[3] = 10000 OR
                            pay_by_quarter[4] = 10000;
```

However, this quickly becomes tedious for large arrays, and is not helpful if the size of the array is uncertain. An alternative method is described in Section 9.17. The above query could be replaced by:

```
SELECT * FROM sal_emp WHERE 10000 = ANY (pay_by_quarter);
```

In addition, you could find rows where the array had all values equal to 10000 with:

```
SELECT * FROM sal_emp WHERE 10000 = ALL (pay_by_quarter);
```

> **Tip:** Arrays are not sets; searching for specific array elements may be a sign of database misdesign. Consider using a separate table with a row for each item that would be an array element. This will be easier to search, and is likely to scale up better to large numbers of elements.

8.10.6. Array Input and Output Syntax

The external text representation of an array value consists of items that are interpreted according to the I/O conversion rules for the array's element type, plus decoration that indicates the array structure. The decoration consists of curly braces (`{` and `}`) around the array value plus delimiter characters between adjacent items. The delimiter character is usually a comma (`,`) but can be something else: it is determined by the `typdelim` setting for the array's element type. (Among the standard data types provided in the PostgreSQL distribution, type `box` uses a semicolon (`;`) but all the others use comma.) In a multidimensional array, each dimension (row, plane, cube, etc.) gets its own level of curly braces, and delimiters must be written between adjacent curly-braced entities of the same level.

The array output routine will put double quotes around element values if they are empty strings, contain curly braces, delimiter characters, double quotes, backslashes, or white space, or match the word `NULL`. Double quotes and backslashes embedded in element values will be backslash-escaped. For numeric data types it is safe to assume that double quotes will never appear, but for textual data types one should be prepared to cope with either presence or absence of quotes.

By default, the lower bound index value of an array's dimensions is set to one. To represent arrays with other lower bounds, the array subscript ranges can be specified explicitly before writing the array contents. This decoration consists of square brackets (`[]`) around each array dimension's lower and upper bounds, with a colon (`:`) delimiter character in between. The array dimension decoration is followed by an equal sign (=). For example:

```
SELECT f1[1][-2][3] AS e1, f1[1][-1][5] AS e2
 FROM (SELECT '[1:1][-2:-1][3:5]={{{1,2,3},{4,5,6}}}'::int[] AS f1) AS ss;

 e1 | e2
----+----
  1 |  6
(1 row)
```

The array output routine will include explicit dimensions in its result only when there are one or more lower bounds different from one.

If the value written for an element is `NULL` (in any case variant), the element is taken to be NULL. The presence of any quotes or backslashes disables this and allows the literal string value "NULL" to be entered. Also, for backwards compatibility with pre-8.2 versions of PostgreSQL, the array_nulls configuration parameter may be turned `off` to suppress recognition of `NULL` as a NULL.

As shown previously, when writing an array value you can write double quotes around any individual array element. You *must* do so if the element value would otherwise confuse the array-value parser. For example, elements containing curly braces, commas (or whatever the delimiter character is), double quotes, backslashes, or leading or trailing whitespace must be double-quoted. Empty strings and strings matching the word `NULL` must be quoted, too. To put a double quote or backslash in a quoted array element value, use escape string syntax and precede it with a backslash. Alternatively, you can use backslash-escaping to protect all data characters that would otherwise be taken as array syntax.

You may write whitespace before a left brace or after a right brace. You may also write whitespace before or after any individual item string. In all of these cases the whitespace will be ignored. However, whitespace within double-quoted elements, or surrounded on both sides by non-whitespace characters of an element, is not ignored.

> **Note:** Remember that what you write in an SQL command will first be interpreted as a string literal, and then as an array. This doubles the number of backslashes you need. For example, to insert a `text` array value containing a backslash and a double quote, you'd need to write
>
> ```
> INSERT ... VALUES (E'{"\\\\","\\""}');
> ```
>
> The escape string processor removes one level of backslashes, so that what arrives at the array-value parser looks like `{"\\","\""}`. In turn, the strings fed to the `text` data type's input routine become `\` and `"` respectively. (If we were working with a data type whose input routine also treated backslashes specially, `bytea` for example, we might need as many as eight backslashes in the command to get one backslash into the stored array element.) Dollar quoting (see Section 4.1.2.2) can be used to avoid the need to double backslashes.

Tip: The ARRAY constructor syntax (see Section 4.2.10) is often easier to work with than the array-literal syntax when writing array values in SQL commands. In ARRAY, individual element values are written the same way they would be written when not members of an array.

8.11. Composite Types

A *composite type* describes the structure of a row or record; it is in essence just a list of field names and their data types. PostgreSQL allows values of composite types to be used in many of the same ways that simple types can be used. For example, a column of a table can be declared to be of a composite type.

8.11.1. Declaration of Composite Types

Here are two simple examples of defining composite types:

```
CREATE TYPE complex AS (
    r       double precision,
    i       double precision
);

CREATE TYPE inventory_item AS (
    name            text,
    supplier_id     integer,
    price           numeric
);
```

The syntax is comparable to CREATE TABLE, except that only field names and types can be specified; no constraints (such as NOT NULL) can presently be included. Note that the AS keyword is essential; without it, the system will think a quite different kind of CREATE TYPE command is meant, and you'll get odd syntax errors.

Having defined the types, we can use them to create tables:

```
CREATE TABLE on_hand (
    item      inventory_item,
    count     integer
);

INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
```

or functions:

```
CREATE FUNCTION price_extension(inventory_item, integer) RETURNS numeric
AS 'SELECT $1.price * $2' LANGUAGE SQL;

SELECT price_extension(item, 10) FROM on_hand;
```

Whenever you create a table, a composite type is also automatically created, with the same name as the table, to represent the table's row type. For example, had we said

```
CREATE TABLE inventory_item (
    name            text,
    supplier_id     integer REFERENCES suppliers,
    price           numeric CHECK (price > 0)
);
```

then the same `inventory_item` composite type shown above would come into being as a byproduct, and could be used just as above. Note however an important restriction of the current implementation: since no constraints are associated with a composite type, the constraints shown in the table definition *do not apply* to values of the composite type outside the table. (A partial workaround is to use domain types as members of composite types.)

8.11.2. Composite Value Input

To write a composite value as a literal constant, enclose the field values within parentheses and separate them by commas. You may put double quotes around any field value, and must do so if it contains commas or parentheses. (More details appear below.) Thus, the general format of a composite constant is the following:

```
'( val1 , val2 , ... )'
```

An example is

```
'("fuzzy dice",42,1.99)'
```

which would be a valid value of the `inventory_item` type defined above. To make a field be NULL, write no characters at all in its position in the list. For example, this constant specifies a NULL third field:

```
'("fuzzy dice",42,)'
```

If you want an empty string rather than NULL, write double quotes:

```
'("",42,)'
```

Here the first field is a non-NULL empty string, the third is NULL.

(These constants are actually only a special case of the generic type constants discussed in Section 4.1.2.5. The constant is initially treated as a string and passed to the composite-type input conversion routine. An explicit type specification might be necessary.)

The `ROW` expression syntax may also be used to construct composite values. In most cases this is considerably simpler to use than the string-literal syntax, since you don't have to worry about multiple layers of quoting. We already used this method above:

```
ROW('fuzzy dice', 42, 1.99)
ROW(", 42, NULL)
```

The ROW keyword is actually optional as long as you have more than one field in the expression, so these can simplify to

```
('fuzzy dice', 42, 1.99)
('', 42, NULL)
```

The ROW expression syntax is discussed in more detail in Section 4.2.11.

8.11.3. Accessing Composite Types

To access a field of a composite column, one writes a dot and the field name, much like selecting a field from a table name. In fact, it's so much like selecting from a table name that you often have to use parentheses to keep from confusing the parser. For example, you might try to select some subfields from our on_hand example table with something like:

```
SELECT item.name FROM on_hand WHERE item.price > 9.99;
```

This will not work since the name item is taken to be a table name, not a field name, per SQL syntax rules. You must write it like this:

```
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;
```

or if you need to use the table name as well (for instance in a multitable query), like this:

```
SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price > 9.99;
```

Now the parenthesized object is correctly interpreted as a reference to the item column, and then the subfield can be selected from it.

Similar syntactic issues apply whenever you select a field from a composite value. For instance, to select just one field from the result of a function that returns a composite value, you'd need to write something like

```
SELECT (my_func(...)).field FROM ...
```

Without the extra parentheses, this will provoke a syntax error.

8.11.4. Modifying Composite Types

Here are some examples of the proper syntax for inserting and updating composite columns. First, inserting or updating a whole column:

```
INSERT INTO mytab (complex_col) VALUES((1.1,2.2));

UPDATE mytab SET complex_col = ROW(1.1,2.2) WHERE ...;
```

The first example omits ROW, the second uses it; we could have done it either way.

We can update an individual subfield of a composite column:

```
UPDATE mytab SET complex_col.r = (complex_col).r + 1 WHERE ...;
```

Notice here that we don't need to (and indeed cannot) put parentheses around the column name appearing just after `SET`, but we do need parentheses when referencing the same column in the expression to the right of the equal sign.

And we can specify subfields as targets for `INSERT`, too:

```
INSERT INTO mytab (complex_col.r, complex_col.i) VALUES(1.1, 2.2);
```

Had we not supplied values for all the subfields of the column, the remaining subfields would have been filled with null values.

8.11.5. Composite Type Input and Output Syntax

The external text representation of a composite value consists of items that are interpreted according to the I/O conversion rules for the individual field types, plus decoration that indicates the composite structure. The decoration consists of parentheses (`(` and `)`) around the whole value, plus commas (`,`) between adjacent items. Whitespace outside the parentheses is ignored, but within the parentheses it is considered part of the field value, and may or may not be significant depending on the input conversion rules for the field data type. For example, in

```
'(  42)'
```

the whitespace will be ignored if the field type is integer, but not if it is text.

As shown previously, when writing a composite value you may write double quotes around any individual field value. You *must* do so if the field value would otherwise confuse the composite-value parser. In particular, fields containing parentheses, commas, double quotes, or backslashes must be double-quoted. To put a double quote or backslash in a quoted composite field value, precede it with a backslash. (Also, a pair of double quotes within a double-quoted field value is taken to represent a double quote character, analogously to the rules for single quotes in SQL literal strings.) Alternatively, you can use backslash-escaping to protect all data characters that would otherwise be taken as composite syntax.

A completely empty field value (no characters at all between the commas or parentheses) represents a NULL. To write a value that is an empty string rather than NULL, write `""`.

The composite output routine will put double quotes around field values if they are empty strings or contain parentheses, commas, double quotes, backslashes, or white space. (Doing so for white space is not essential, but aids legibility.) Double quotes and backslashes embedded in field values will be doubled.

> **Note:** Remember that what you write in an SQL command will first be interpreted as a string literal, and then as a composite. This doubles the number of backslashes you need (assuming escape string syntax is used). For example, to insert a `text` field containing a double quote and a backslash in a composite value, you'd need to write
>
> ```
> INSERT ... VALUES (E'("\\"\\\\")');
> ```
>
> The string-literal processor removes one level of backslashes, so that what arrives at the composite-value parser looks like `("\"\\")`. In turn, the string fed to the `text` data type's input routine becomes `"\`. (If we were working with a data type whose input routine also treated backslashes specially, `bytea` for example, we might need as many as eight backslashes in the command to get one backslash into the stored composite field.) Dollar quoting (see Section 4.1.2.2) may be used to avoid the need to double backslashes.

Tip: The `ROW` constructor syntax is usually easier to work with than the composite-literal syntax when writing composite values in SQL commands. In `ROW`, individual field values are written the same way they would be written when not members of a composite.

8.12. Object Identifier Types

Object identifiers (OIDs) are used internally by PostgreSQL as primary keys for various system tables. OIDs are not added to user-created tables, unless `WITH OIDS` is specified when the table is created, or the default_with_oids configuration variable is enabled. Type `oid` represents an object identifier. There are also several alias types for `oid`: `regproc`, `regprocedure`, `regoper`, `regoperator`, `regclass`, and `regtype`. Table 8-19 shows an overview.

The `oid` type is currently implemented as an unsigned four-byte integer. Therefore, it is not large enough to provide database-wide uniqueness in large databases, or even in large individual tables. So, using a user-created table's OID column as a primary key is discouraged. OIDs are best used only for references to system tables.

The `oid` type itself has few operations beyond comparison. It can be cast to integer, however, and then manipulated using the standard integer operators. (Beware of possible signed-versus-unsigned confusion if you do this.)

The OID alias types have no operations of their own except for specialized input and output routines. These routines are able to accept and display symbolic names for system objects, rather than the raw numeric value that type `oid` would use. The alias types allow simplified lookup of OID values for objects. For example, to examine the `pg_attribute` rows related to a table `mytable`, one could write

```
SELECT * FROM pg_attribute WHERE attrelid = 'mytable'::regclass;
```

rather than

```
SELECT * FROM pg_attribute
  WHERE attrelid = (SELECT oid FROM pg_class WHERE relname = 'mytable');
```

While that doesn't look all that bad by itself, it's still oversimplified. A far more complicated sub-select would be needed to select the right OID if there are multiple tables named `mytable` in different schemas. The `regclass` input converter handles the table lookup according to the schema path setting, and so it does the "right thing" automatically. Similarly, casting a table's OID to `regclass` is handy for symbolic display of a numeric OID.

Table 8-19. Object Identifier Types

Name	References	Description	Value Example
`oid`	any	numeric object identifier	`564182`

Name	References	Description	Value Example
regproc	pg_proc	function name	sum
regprocedure	pg_proc	function with argument types	sum(int4)
regoper	pg_operator	operator name	+
regoperator	pg_operator	operator with argument types	*(integer,integer) or -(NONE,integer)
regclass	pg_class	relation name	pg_type
regtype	pg_type	data type name	integer

All of the OID alias types accept schema-qualified names, and will display schema-qualified names on output if the object would not be found in the current search path without being qualified. The regproc and regoper alias types will only accept input names that are unique (not overloaded), so they are of limited use; for most uses regprocedure or regoperator is more appropriate. For regoperator, unary operators are identified by writing NONE for the unused operand.

An additional property of the OID alias types is that if a constant of one of these types appears in a stored expression (such as a column default expression or view), it creates a dependency on the referenced object. For example, if a column has a default expression nextval('my_seq'::regclass), PostgreSQL understands that the default expression depends on the sequence my_seq; the system will not let the sequence be dropped without first removing the default expression.

Another identifier type used by the system is xid, or transaction (abbreviated xact) identifier. This is the data type of the system columns xmin and xmax. Transaction identifiers are 32-bit quantities.

A third identifier type used by the system is cid, or command identifier. This is the data type of the system columns cmin and cmax. Command identifiers are also 32-bit quantities.

A final identifier type used by the system is tid, or tuple identifier (row identifier). This is the data type of the system column ctid. A tuple ID is a pair (block number, tuple index within block) that identifies the physical location of the row within its table.

(The system columns are further explained in Section 5.4.)

8.13. Pseudo-Types

The PostgreSQL type system contains a number of special-purpose entries that are collectively called *pseudo-types*. A pseudo-type cannot be used as a column data type, but it can be used to declare a function's argument or result type. Each of the available pseudo-types is useful in situations where a function's behavior does not correspond to simply taking or returning a value of a specific SQL data type. Table 8-20 lists the existing pseudo-types.

Table 8-20. Pseudo-Types

Name	Description
any	Indicates that a function accepts any input data type whatever.

Name	Description
anyarray	Indicates that a function accepts any array data type (see Section 33.2.5).
anyelement	Indicates that a function accepts any data type (see Section 33.2.5).
cstring	Indicates that a function accepts or returns a null-terminated C string.
internal	Indicates that a function accepts or returns a server-internal data type.
language_handler	A procedural language call handler is declared to return language_handler.
record	Identifies a function returning an unspecified row type.
trigger	A trigger function is declared to return trigger.
void	Indicates that a function returns no value.
opaque	An obsolete type name that formerly served all the above purposes.

Functions coded in C (whether built-in or dynamically loaded) may be declared to accept or return any of these pseudo data types. It is up to the function author to ensure that the function will behave safely when a pseudo-type is used as an argument type.

Functions coded in procedural languages may use pseudo-types only as allowed by their implementation languages. At present the procedural languages all forbid use of a pseudo-type as argument type, and allow only void and record as a result type (plus trigger when the function is used as a trigger). Some also support polymorphic functions using the types anyarray and anyelement.

The internal pseudo-type is used to declare functions that are meant only to be called internally by the database system, and not by direct invocation in a SQL query. If a function has at least one internal-type argument then it cannot be called from SQL. To preserve the type safety of this restriction it is important to follow this coding rule: do not create any function that is declared to return internal unless it has at least one internal argument.

8.14. XML Document Support

XML (Extensible Markup Language) support is not one capability, but a variety of features supported by a database system. These capabilities include storage, import/export, validation, indexing, efficiency of modification, searching, transforming, and XML to SQL mapping. PostgreSQL supports some but not all of these XML capabilities. Future releases of PostgreSQL will continue to improve XML support. For an overview of XML use in databases, see http://www.rpbourret.com/xml/XMLAndDatabases.htm.

Storage

PostgreSQL does not have a specialized XML data type. Users should store XML documents in ordinary TEXT fields. If you need the document split apart into its component parts so each element

is stored separately, you must use a middle-ware solution to do that, but once done, the data becomes relational and has to be processed accordingly.

Import/Export

There is no facility for mapping XML to relational tables. An external tool must be used for this. One simple way to export XML is to use psql in HTML mode (`\pset format html`), and convert the XHTML output to XML using an external tool.

Validation

`/contrib/xml2` has a function called `xml_is_well_formed()` that can be used in a `CHECK` constraint to enforce that a field contains well-formed XML. It does not support validation against a specific XML schema. A server-side language with XML capabilities could be used to do schema-specific XML checks.

Indexing

`/contrib/xml2` functions can be used in expression indexes to index specific XML fields. To index the full contents of XML documents, the full-text indexing tool `/contrib/tsearch2` can be used. Of course, Tsearch2 indexes have no XML awareness so additional `/contrib/xml2` checks should be added to queries.

Modification

If an `UPDATE` does not modify an XML field, the XML data is shared between the old and new rows. However, if the `UPDATE` modifies an XML field, a full modified copy of the XML field must be created internally.

Searching

XPath searches are implemented using `/contrib/xml2`. It processes XML text documents and returns results based on the requested query.

Transforming

`/contrib/xml2` supports XSLT (Extensible Stylesheet Language Transformation).

XML to SQL Mapping

This involves converting XML data to and from relational structures. PostgreSQL has no internal support for such mapping, and relies on external tools to do such conversions.

Missing Features

Missing features include XQuery, SQL/XML syntax (ISO/IEC 9075-14), and an XML data type optimized for XML storage.

Chapter 9. Functions and Operators

PostgreSQL provides a large number of functions and operators for the built-in data types. Users can also define their own functions and operators, as described in Part V. The psql commands `\df` and `\do` can be used to show the list of all actually available functions and operators, respectively.

If you are concerned about portability then take note that most of the functions and operators described in this chapter, with the exception of the most trivial arithmetic and comparison operators and some explicitly marked functions, are not specified by the SQL standard. Some of the extended functionality is present in other SQL database management systems, and in many cases this functionality is compatible and consistent between the various implementations. This chapter is also not exhaustive; additional functions appear in relevant sections of the manual.

9.1. Logical Operators

The usual logical operators are available:

```
AND
OR
NOT
```

SQL uses a three-valued Boolean logic where the null value represents "unknown". Observe the following truth tables:

a	*b*	*a* AND *b*	*a* OR *b*
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	NOT *a*
TRUE	FALSE
FALSE	TRUE
NULL	NULL

The operators `AND` and `OR` are commutative, that is, you can switch the left and right operand without affecting the result. But see Section 4.2.12 for more information about the order of evaluation of subexpressions.

9.2. Comparison Operators

The usual comparison operators are available, shown in Table 9-1.

Table 9-1. Comparison Operators

Operator	Description
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
=	equal
<> or !=	not equal

> **Note:** The != operator is converted to <> in the parser stage. It is not possible to implement != and <> operators that do different things.

Comparison operators are available for all data types where this makes sense. All comparison operators are binary operators that return values of type `boolean`; expressions like `1 < 2 < 3` are not valid (because there is no < operator to compare a Boolean value with `3`).

In addition to the comparison operators, the special `BETWEEN` construct is available.

```
a BETWEEN x AND y
```

is equivalent to

```
a >= x AND a <= y
```

Similarly,

```
a NOT BETWEEN x AND y
```

is equivalent to

```
a < x OR a > y
```

There is no difference between the two respective forms apart from the CPU cycles required to rewrite the first one into the second one internally. `BETWEEN SYMMETRIC` is the same as `BETWEEN` except there is no requirement that the argument to the left of `AND` be less than or equal to the argument on the right; the proper range is automatically determined.

To check whether a value is or is not null, use the constructs

```
expression IS NULL
expression IS NOT NULL
```

or the equivalent, but nonstandard, constructs

```
expression ISNULL
```

```
expression NOTNULL
```

Do *not* write `expression = NULL` because `NULL` is not "equal to" `NULL`. (The null value represents an unknown value, and it is not known whether two unknown values are equal.) This behavior conforms to the SQL standard.

> **Tip:** Some applications may expect that `expression = NULL` returns true if `expression` evaluates to the null value. It is highly recommended that these applications be modified to comply with the SQL standard. However, if that cannot be done the transform_null_equals configuration variable is available. If it is enabled, PostgreSQL will convert `x = NULL` clauses to `x IS NULL`. This was the default behavior in PostgreSQL releases 6.5 through 7.1.

> **Note:** If the `expression` is row-valued, then `IS NULL` is true when the row expression itself is null or when all the row's fields are null, while `IS NOT NULL` is true when the row expression itself is non-null and all the row's fields are non-null. This definition conforms to the SQL standard, and is a change from the inconsistent behavior exhibited by PostgreSQL versions prior to 8.2.

The ordinary comparison operators yield null (signifying "unknown") when either input is null. Another way to do comparisons is with the `IS [ NOT ] DISTINCT FROM` construct:

```
expression IS DISTINCT FROM expression
expression IS NOT DISTINCT FROM expression
```

For non-null inputs, `IS DISTINCT FROM` is the same as the <> operator. However, when both inputs are null it will return false, and when just one input is null it will return true. Similarly, `IS NOT DISTINCT FROM` is identical to = for non-null inputs, but it returns true when both inputs are null, and false when only one input is null. Thus, these constructs effectively act as though null were a normal data value, rather than "unknown".

Boolean values can also be tested using the constructs

```
expression IS TRUE
expression IS NOT TRUE
expression IS FALSE
expression IS NOT FALSE
expression IS UNKNOWN
expression IS NOT UNKNOWN
```

These will always return true or false, never a null value, even when the operand is null. A null input is treated as the logical value "unknown". Notice that `IS UNKNOWN` and `IS NOT UNKNOWN` are effectively the same as `IS NULL` and `IS NOT NULL`, respectively, except that the input expression must be of Boolean type.

9.3. Mathematical Functions and Operators

Mathematical operators are provided for many PostgreSQL types. For types without common mathematical conventions for all possible permutations (e.g., date/time types) we describe the actual behavior in subsequent sections.

Table 9-2 shows the available mathematical operators.

Table 9-2. Mathematical Operators

Operator	Description	Example	Result
`+`	addition	`2 + 3`	`5`
`-`	subtraction	`2 - 3`	`-1`
`*`	multiplication	`2 * 3`	`6`
`/`	division (integer division truncates results)	`4 / 2`	`2`
`%`	modulo (remainder)	`5 % 4`	`1`
`^`	exponentiation	`2.0 ^ 3.0`	`8`
`|/`	square root	`|/ 25.0`	`5`
`||/`	cube root	`||/ 27.0`	`3`
`!`	factorial	`5 !`	`120`
`!!`	factorial (prefix operator)	`!! 5`	`120`
`@`	absolute value	`@ -5.0`	`5`
`&`	bitwise AND	`91 & 15`	`11`
`|`	bitwise OR	`32 | 3`	`35`
`#`	bitwise XOR	`17 # 5`	`20`
`~`	bitwise NOT	`~1`	`-2`
`<<`	bitwise shift left	`1 << 4`	`16`
`>>`	bitwise shift right	`8 >> 2`	`2`

The bitwise operators work only on integral data types, whereas the others are available for all numeric data types. The bitwise operators are also available for the bit string types `bit` and `bit varying`, as shown in Table 9-10.

Table 9-3 shows the available mathematical functions. In the table, `dp` indicates `double precision`. Many of these functions are provided in multiple forms with different argument types. Except where noted, any given form of a function returns the same data type as its argument. The functions working with `double precision` data are mostly implemented on top of the host system's C library; accuracy and behavior in boundary cases may therefore vary depending on the host system.

Table 9-3. Mathematical Functions

Function	Return Type	Description	Example	Result

Function	Return Type	Description	Example	Result
`abs(x)`	(same as *x*)	absolute value	`abs(-17.4)`	`17.4`
`cbrt(dp)`	`dp`	cube root	`cbrt(27.0)`	`3`
`ceil(dp or numeric)`	(same as input)	smallest integer not less than argument	`ceil(-42.8)`	`-42`
`ceiling(dp or numeric)`	(same as input)	smallest integer not less than argument (alias for `ceil`)	`ceiling(-95.3)`	`-95`
`degrees(dp)`	`dp`	radians to degrees	`degrees(0.5)`	`28.6478897565412`
`exp(dp or numeric)`	(same as input)	exponential	`exp(1.0)`	`2.71828182845905`
`floor(dp or numeric)`	(same as input)	largest integer not greater than argument	`floor(-42.8)`	`-43`
`ln(dp or numeric)`	(same as input)	natural logarithm	`ln(2.0)`	`0.693147180559945`
`log(dp or numeric)`	(same as input)	base 10 logarithm	`log(100.0)`	`2`
`log(b numeric, x numeric)`	`numeric`	logarithm to base `b`	`log(2.0, 64.0)`	`6.0000000000`
`mod(y, x)`	(same as argument types)	remainder of `y`/`x`	`mod(9,4)`	`1`
`pi()`	`dp`	"π" constant	`pi()`	`3.14159265358979`
`power(a dp, b dp)`	`dp`	`a` raised to the power of `b`	`power(9.0, 3.0)`	`729`
`power(a numeric, b numeric)`	`numeric`	`a` raised to the power of `b`	`power(9.0, 3.0)`	`729`
`radians(dp)`	`dp`	degrees to radians	`radians(45.0)`	`0.785398163397448`
`random()`	`dp`	random value between 0.0 and 1.0	`random()`	
`round(dp or numeric)`	(same as input)	round to nearest integer	`round(42.4)`	`42`
`round(v numeric, s int)`	`numeric`	round to `s` decimal places	`round(42.4382, 2)`	`42.44`

Function	Return Type	Description	Example	Result
`setseed(dp)`	`int`	set seed for subsequent `random()` calls (value between 0 and 1.0)	`setseed(0.54823)`	`1177314959`
`sign(dp or numeric)`	(same as input)	sign of the argument (-1, 0, +1)	`sign(-8.4)`	`-1`
`sqrt(dp or numeric)`	(same as input)	square root	`sqrt(2.0)`	`1.4142135623731`
`trunc(dp or numeric)`	(same as input)	truncate toward zero	`trunc(42.8)`	`42`
`trunc(v numeric, s int)`	`numeric`	truncate to `s` decimal places	`trunc(42.4382, 2)`	`42.43`
`width_bucket(op numeric, b1 numeric, b2 numeric, count int)`	`int`	return the bucket to which `operand` would be assigned in an equidepth histogram with `count` buckets, in the range `b1` to `b2`	`width_bucket(5.35, 0.024, 10.06, 5)`	`3`

Finally, Table 9-4 shows the available trigonometric functions. All trigonometric functions take arguments and return values of type `double precision`.

Table 9-4. Trigonometric Functions

Function	Description
`acos(`*x*`)`	inverse cosine
`asin(`*x*`)`	inverse sine
`atan(`*x*`)`	inverse tangent
`atan2(`*x*`, `*y*`)`	inverse tangent of *x*/*y*
`cos(`*x*`)`	cosine
`cot(`*x*`)`	cotangent
`sin(`*x*`)`	sine
`tan(`*x*`)`	tangent

9.4. String Functions and Operators

This section describes functions and operators for examining and manipulating string values. Strings in this context include values of all the types `character`, `character varying`, and `text`. Unless otherwise noted, all of the functions listed below work on all of these types, but be wary of potential

effects of the automatic padding when using the `character` type. Generally, the functions described here also work on data of non-string types by converting that data to a string representation first. Some functions also exist natively for the bit-string types.

SQL defines some string functions with a special syntax where certain key words rather than commas are used to separate the arguments. Details are in Table 9-5. These functions are also implemented using the regular syntax for function invocation. (See Table 9-6.)

Table 9-5. SQL String Functions and Operators

Function	Return Type	Description	Example	Result
`string || string`	`text`	String concatenation	`'Post' || 'greSQL'`	`PostgreSQL`
`bit_length(string)`	`int`	Number of bits in string	`bit_length('jose')`	`32`
`char_length(string)` or `character_length(string)`	`int`	Number of characters in string	`char_length('jose')`	`4`
`convert(string using conversion_name)`	`text`	Change encoding using specified conversion name. Conversions can be defined by `CREATE CONVERSION`. Also there are some pre-defined conversion names. See Table 9-7 for available conversion names.	`convert('PostgreSQL' using iso_8859_1_to_utf_8)`	`'PostgreSQL'` in 8-bit (Unicode, UTF8) encoding
`lower(string)`	`text`	Convert string to lower case	`lower('TOM')`	`tom`
`octet_length(string)`	`int`	Number of bytes in string	`octet_length('jose')`	`4`
`overlay(string placing string from int [for int])`	`text`	Replace substring	`overlay('Txxxxas' placing 'hom' from 2 for 4)`	`Thomas`
`position(substring in string)`	`int`	Location of specified substring	`position('om' in 'Thomas')`	`3`
`substring(string [from int] [for int])`	`text`	Extract substring	`substring('Thomas' from 2 for 3)`	`hom`

Function	Return Type	Description	Example	Result
`substring(string from pattern)`	`text`	Extract substring matching POSIX regular expression. See Section 9.7 for more information on pattern matching.	`substring('Thomas' from '...$')`	`mas`
`substring(string from pattern for escape)`	`text`	Extract substring matching SQL regular expression. See Section 9.7 for more information on pattern matching.	`substring('Thomas' from '%#"o_a#"_' for '#')`	`oma`
`trim([leading | trailing | both] [characters] from string)`	`text`	Remove the longest string containing only the `characters` (a space by default) from the start/end/both ends of the `string`	`trim(both 'x' from 'xTomxx')`	`Tom`
`upper(string)`	`text`	Convert string to uppercase	`upper('tom')`	`TOM`

Additional string manipulation functions are available and are listed in Table 9-6. Some of them are used internally to implement the SQL-standard string functions listed in Table 9-5.

Table 9-6. Other String Functions

Function	Return Type	Description	Example	Result
`ascii(string)`	`int`	ASCII code of the first byte of the argument	`ascii('x')`	`120`
`btrim(string text [, characters text])`	`text`	Remove the longest string consisting only of characters in `characters` (a space by default) from the start and end of `string`	`btrim('xyxtrimyyx', 'xy')`	`trim`
`chr(int)`	`text`	Character with the given ASCII code	`chr(65)`	`A`

Function	Return Type	Description	Example	Result
convert(string text, [src_encoding name,] dest_encoding name)	text	Convert string to dest_encoding. The original encoding is specified by src_encoding. If src_encoding is omitted, database encoding is assumed.	convert('text_in_utf8', 'UTF8', 'LATIN1')	text_in_utf8 represented in ISO 8859-1 encoding
decode(string text, type text)	bytea	Decode binary data from string previously encoded with encode. Parameter type is same as in encode.	decode('MTIzAAE=', 'base64')	123\000\001
encode(data bytea, type text)	text	Encode binary data to different representation. Supported types are: base64, hex, escape. Escape merely outputs null bytes as \000 and doubles backslashes.	encode(E'123\\000\\001', 'base64')	MTIzAAE=
initcap(string)	text	Convert the first letter of each word to uppercase and the rest to lowercase. Words are sequences of alphanumeric characters separated by non-alphanumeric characters.	initcap('hi THOMAS')	Hi Thomas
length(string)	int	Number of characters in string	length('jose')	4

Function	Return Type	Description	Example	Result
lpad(string text, length int [, fill text])	text	Fill up the string to length length by prepending the characters fill (a space by default). If the string is already longer than length then it is truncated (on the right).	lpad('hi', 5, 'xy')	xyxhi
ltrim(string text [, characters text])	text	Remove the longest string containing only characters from characters (a space by default) from the start of string	ltrim('zzzytrim', 'xyz')	trim
md5(string)	text	Calculates the MD5 hash of string, returning the result in hexadecimal	md5('abc')	900150983cd24fb0d6963f7d28e17f72
pg_client_encoding()	name	Current client encoding name	pg_client_encoding()	SQL_ASCII
quote_ident(string text)	text	Return the given string suitably quoted to be used as an identifier in an SQL statement string. Quotes are added only if necessary (i.e., if the string contains non-identifier characters or would be case-folded). Embedded quotes are properly doubled.	quote_ident('Foo bar')	"Foo bar"

Function	Return Type	Description	Example	Result
quote_literal(string text)	text	Return the given string suitably quoted to be used as a string literal in an SQL statement string. Embedded single-quotes and backslashes are properly doubled.	quote_literal('O\'Reilly')	'O''Reilly'
regexp_replace(string text, pattern text, replacement text [,flags text])	text	Replace substring matching POSIX regular expression. See Section 9.7 for more information on pattern matching.	regexp_replace('Thomas', '.[mN]a.', 'M')	ThM
repeat(string text, number int)	text	Repeat string the specified number of times	repeat('Pg', 4)	PgPgPgPg
replace(string text, from text, to text)	text	Replace all occurrences in string of substring from with substring to	replace('abcdefabcdef', 'cd', 'XX')	abXXefabXXef
rpad(string text, length int [, fill text])	text	Fill up the string to length length by appending the characters fill (a space by default). If the string is already longer than length then it is truncated.	rpad('hi', 5, 'xy')	hixyx
rtrim(string text [, characters text])	text	Remove the longest string containing only characters from characters (a space by default) from the end of string	rtrim('trimxxxx', 'x')	trim

Function	Return Type	Description	Example	Result
split_part(string text, delimiter text, field int)	text	Split string on delimiter and return the given field (counting from one)	split_part('abc~@~def~@~ghi', '~@~', 2)	def
strpos(string, substring)	int	Location of specified substring (same as position(substring in string), but note the reversed argument order)	strpos('high', 'ig')	2
substr(string, from [, count])	text	Extract substring (same as substring(string from from for count))	substr('alphabet', 3, 2)	ph
to_ascii(string text [, encoding text])	text	Convert string to ASCII from another encoding (only supports conversion from LATIN1, LATIN2, LATIN9, and WIN1250 encodings)	to_ascii('Karel')	Karel
to_hex(number int or bigint)	text	Convert number to its equivalent hexadecimal representation	to_hex(2147483647)	7fffffff
translate(string text, from text, to text)	text	Any character in string that matches a character in the from set is replaced by the corresponding character in the to set	translate('12345', '14', 'ax')	a23x5

Table 9-7. Built-in Conversions

Conversion Name [a]	Source Encoding	Destination Encoding
ascii_to_mic	SQL_ASCII	MULE_INTERNAL

Conversion Name [a]	Source Encoding	Destination Encoding
ascii_to_utf8	SQL_ASCII	UTF8
big5_to_euc_tw	BIG5	EUC_TW
big5_to_mic	BIG5	MULE_INTERNAL
big5_to_utf8	BIG5	UTF8
euc_cn_to_mic	EUC_CN	MULE_INTERNAL
euc_cn_to_utf8	EUC_CN	UTF8
euc_jp_to_mic	EUC_JP	MULE_INTERNAL
euc_jp_to_sjis	EUC_JP	SJIS
euc_jp_to_utf8	EUC_JP	UTF8
euc_kr_to_mic	EUC_KR	MULE_INTERNAL
euc_kr_to_utf8	EUC_KR	UTF8
euc_tw_to_big5	EUC_TW	BIG5
euc_tw_to_mic	EUC_TW	MULE_INTERNAL
euc_tw_to_utf8	EUC_TW	UTF8
gb18030_to_utf8	GB18030	UTF8
gbk_to_utf8	GBK	UTF8
iso_8859_10_to_utf8	LATIN6	UTF8
iso_8859_13_to_utf8	LATIN7	UTF8
iso_8859_14_to_utf8	LATIN8	UTF8
iso_8859_15_to_utf8	LATIN9	UTF8
iso_8859_16_to_utf8	LATIN10	UTF8
iso_8859_1_to_mic	LATIN1	MULE_INTERNAL
iso_8859_1_to_utf8	LATIN1	UTF8
iso_8859_2_to_mic	LATIN2	MULE_INTERNAL
iso_8859_2_to_utf8	LATIN2	UTF8
iso_8859_2_to_windows_1250	LATIN2	WIN1250
iso_8859_3_to_mic	LATIN3	MULE_INTERNAL
iso_8859_3_to_utf8	LATIN3	UTF8
iso_8859_4_to_mic	LATIN4	MULE_INTERNAL
iso_8859_4_to_utf8	LATIN4	UTF8
iso_8859_5_to_koi8_r	ISO_8859_5	KOI8
iso_8859_5_to_mic	ISO_8859_5	MULE_INTERNAL
iso_8859_5_to_utf8	ISO_8859_5	UTF8
iso_8859_5_to_windows_1251	ISO_8859_5	WIN1251
iso_8859_5_to_windows_866	ISO_8859_5	WIN866
iso_8859_6_to_utf8	ISO_8859_6	UTF8

Conversion Name [a]	Source Encoding	Destination Encoding
iso_8859_7_to_utf8	ISO_8859_7	UTF8
iso_8859_8_to_utf8	ISO_8859_8	UTF8
iso_8859_9_to_utf8	LATIN5	UTF8
johab_to_utf8	JOHAB	UTF8
koi8_r_to_iso_8859_5	KOI8	ISO_8859_5
koi8_r_to_mic	KOI8	MULE_INTERNAL
koi8_r_to_utf8	KOI8	UTF8
koi8_r_to_windows_1251	KOI8	WIN1251
koi8_r_to_windows_866	KOI8	WIN866
mic_to_ascii	MULE_INTERNAL	SQL_ASCII
mic_to_big5	MULE_INTERNAL	BIG5
mic_to_euc_cn	MULE_INTERNAL	EUC_CN
mic_to_euc_jp	MULE_INTERNAL	EUC_JP
mic_to_euc_kr	MULE_INTERNAL	EUC_KR
mic_to_euc_tw	MULE_INTERNAL	EUC_TW
mic_to_iso_8859_1	MULE_INTERNAL	LATIN1
mic_to_iso_8859_2	MULE_INTERNAL	LATIN2
mic_to_iso_8859_3	MULE_INTERNAL	LATIN3
mic_to_iso_8859_4	MULE_INTERNAL	LATIN4
mic_to_iso_8859_5	MULE_INTERNAL	ISO_8859_5
mic_to_koi8_r	MULE_INTERNAL	KOI8
mic_to_sjis	MULE_INTERNAL	SJIS
mic_to_windows_1250	MULE_INTERNAL	WIN1250
mic_to_windows_1251	MULE_INTERNAL	WIN1251
mic_to_windows_866	MULE_INTERNAL	WIN866
sjis_to_euc_jp	SJIS	EUC_JP
sjis_to_mic	SJIS	MULE_INTERNAL
sjis_to_utf8	SJIS	UTF8
tcvn_to_utf8	WIN1258	UTF8
uhc_to_utf8	UHC	UTF8
utf8_to_ascii	UTF8	SQL_ASCII
utf8_to_big5	UTF8	BIG5
utf8_to_euc_cn	UTF8	EUC_CN
utf8_to_euc_jp	UTF8	EUC_JP
utf8_to_euc_kr	UTF8	EUC_KR
utf8_to_euc_tw	UTF8	EUC_TW
utf8_to_gb18030	UTF8	GB18030
utf8_to_gbk	UTF8	GBK

Conversion Name [a]	Source Encoding	Destination Encoding
utf8_to_iso_8859_1	UTF8	LATIN1
utf8_to_iso_8859_10	UTF8	LATIN6
utf8_to_iso_8859_13	UTF8	LATIN7
utf8_to_iso_8859_14	UTF8	LATIN8
utf8_to_iso_8859_15	UTF8	LATIN9
utf8_to_iso_8859_16	UTF8	LATIN10
utf8_to_iso_8859_2	UTF8	LATIN2
utf8_to_iso_8859_3	UTF8	LATIN3
utf8_to_iso_8859_4	UTF8	LATIN4
utf8_to_iso_8859_5	UTF8	ISO_8859_5
utf8_to_iso_8859_6	UTF8	ISO_8859_6
utf8_to_iso_8859_7	UTF8	ISO_8859_7
utf8_to_iso_8859_8	UTF8	ISO_8859_8
utf8_to_iso_8859_9	UTF8	LATIN5
utf8_to_johab	UTF8	JOHAB
utf8_to_koi8_r	UTF8	KOI8
utf8_to_sjis	UTF8	SJIS
utf8_to_tcvn	UTF8	WIN1258
utf8_to_uhc	UTF8	UHC
utf8_to_windows_1250	UTF8	WIN1250
utf8_to_windows_1251	UTF8	WIN1251
utf8_to_windows_1252	UTF8	WIN1252
utf8_to_windows_1253	UTF8	WIN1253
utf8_to_windows_1254	UTF8	WIN1254
utf8_to_windows_1255	UTF8	WIN1255
utf8_to_windows_1256	UTF8	WIN1256
utf8_to_windows_1257	UTF8	WIN1257
utf8_to_windows_866	UTF8	WIN866
utf8_to_windows_874	UTF8	WIN874
windows_1250_to_iso_8859_2	WIN1250	LATIN2
windows_1250_to_mic	WIN1250	MULE_INTERNAL
windows_1250_to_utf8	WIN1250	UTF8
windows_1251_to_iso_8859_5	WIN1251	ISO_8859_5
windows_1251_to_koi8_r	WIN1251	KOI8
windows_1251_to_mic	WIN1251	MULE_INTERNAL
windows_1251_to_utf8	WIN1251	UTF8

Conversion Name [a]	Source Encoding	Destination Encoding
windows_1251_to_windows_866	WIN1251	WIN866
windows_1252_to_utf8	WIN1252	UTF8
windows_1256_to_utf8	WIN1256	UTF8
windows_866_to_iso_8859_5	WIN866	ISO_8859_5
windows_866_to_koi8_r	WIN866	KOI8
windows_866_to_mic	WIN866	MULE_INTERNAL
windows_866_to_utf8	WIN866	UTF8
windows_866_to_windows_1251	WIN866	WIN
windows_874_to_utf8	WIN874	UTF8

Notes:
a. The conversion names follow a standard naming scheme: The official name of the source encoding with all non-alphanumeric characters replaced by underscores followed by _to_ followed by the equally processed destination encoding name. Therefore the names might deviate from the customary encoding names.

9.5. Binary String Functions and Operators

This section describes functions and operators for examining and manipulating values of type bytea.

SQL defines some string functions with a special syntax where certain key words rather than commas are used to separate the arguments. Details are in Table 9-8. Some functions are also implemented using the regular syntax for function invocation. (See Table 9-9.)

Table 9-8. SQL Binary String Functions and Operators

Function	Return Type	Description	Example	Result
string || string	bytea	String concatenation	E'\\\\Post'::bytea || E'\\047gres\\000'::bytea	\\Post'gres\000
get_bit(string, offset)	int	Extract bit from string	get_bit(E'Th\\000omas'::bytea, 45)	1
get_byte(string, offset)	int	Extract byte from string	get_byte(E'Th\\000omas'::bytea, 4)	109
octet_length(string)	int	Number of bytes in binary string	octet_length(E'jo\\000se'::bytea)	5

Function	Return Type	Description	Example	Result
position(substring in string)	int	Location of specified substring	position(E'\\000om'::bytea in E'Th\\000omas'::bytea)	3
set_bit(string, offset, newvalue)	bytea	Set bit in string	set_bit(E'Th\\000omas'::bytea, 45, 0)	Th\000Tomas
set_byte(string, offset, newvalue)	bytea	Set byte in string	set_byte(E'Th\\000omas'::bytea, 4, 64)	Th\000@mas
substring(string [from int] [for int])	bytea	Extract substring	substring(E'Th\\000omas'::bytea from 2 for 3)	h\000o
trim([both] bytes from string)	bytea	Remove the longest string containing only the bytes in bytes from the start and end of string	trim(E'\\000'::bytea from E'\\000Tom\\000'::bytea)	Tom

Additional binary string manipulation functions are available and are listed in Table 9-9. Some of them are used internally to implement the SQL-standard string functions listed in Table 9-8.

Table 9-9. Other Binary String Functions

Function	Return Type	Description	Example	Result
btrim(string bytea, bytes bytea)	bytea	Remove the longest string consisting only of bytes in bytes from the start and end of string	btrim(E'\\000trim\\000'::bytea, E'\\000'::bytea)	trim
decode(string text, type text)	bytea	Decode binary string from string previously encoded with encode. Parameter type is same as in encode.	decode(E'123\\000456', 'escape')	123\000456

Function	Return Type	Description	Example	Result
encode(string bytea, type text)	text	Encode binary string to ASCII-only representation. Supported types are: base64, hex, escape.	encode(E'123\\000456'::bytea, 'escape')	123\000456
length(string)	int	Length of binary string	length(E'jo\\000se'::bytea)	5
md5(string)	text	Calculates the MD5 hash of string, returning the result in hexadecimal	md5(E'Th\\000omas'::bytea)	8ab2d3c9689aaf18 b4958c334c82d8b1

9.6. Bit String Functions and Operators

This section describes functions and operators for examining and manipulating bit strings, that is values of the types bit and bit varying. Aside from the usual comparison operators, the operators shown in Table 9-10 can be used. Bit string operands of &, |, and # must be of equal length. When bit shifting, the original length of the string is preserved, as shown in the examples.

Table 9-10. Bit String Operators

Operator	Description	Example	Result
||	concatenation	B'10001' || B'011'	10001011
&	bitwise AND	B'10001' & B'01101'	00001
|	bitwise OR	B'10001' | B'01101'	11101
#	bitwise XOR	B'10001' # B'01101'	11100
~	bitwise NOT	~ B'10001'	01110
<<	bitwise shift left	B'10001' << 3	01000
>>	bitwise shift right	B'10001' >> 2	00100

The following SQL-standard functions work on bit strings as well as character strings: length, bit_length, octet_length, position, substring.

In addition, it is possible to cast integral values to and from type bit. Some examples:

```
44::bit(10)                    0000101100
44::bit(3)                     100
```

```
cast(-44 as bit(12))           111111010100
'1110'::bit(4)::integer        14
```

Note that casting to just "bit" means casting to `bit(1)`, and so it will deliver only the least significant bit of the integer.

> **Note:** Prior to PostgreSQL 8.0, casting an integer to `bit(n)` would copy the leftmost `n` bits of the integer, whereas now it copies the rightmost `n` bits. Also, casting an integer to a bit string width wider than the integer itself will sign-extend on the left.

9.7. Pattern Matching

There are three separate approaches to pattern matching provided by PostgreSQL: the traditional SQL `LIKE` operator, the more recent `SIMILAR TO` operator (added in SQL:1999), and POSIX-style regular expressions. Additionally, a pattern matching function, `substring`, is available, using either `SIMILAR TO`-style or POSIX-style regular expressions.

> **Tip:** If you have pattern matching needs that go beyond this, consider writing a user-defined function in Perl or Tcl.

9.7.1. `LIKE`

```
string LIKE pattern [ESCAPE escape-character]
string NOT LIKE pattern [ESCAPE escape-character]
```

Every *pattern* defines a set of strings. The `LIKE` expression returns true if the *string* is contained in the set of strings represented by *pattern*. (As expected, the `NOT LIKE` expression returns false if `LIKE` returns true, and vice versa. An equivalent expression is `NOT (`*string* `LIKE` *pattern*`)`.)

If *pattern* does not contain percent signs or underscore, then the pattern only represents the string itself; in that case `LIKE` acts like the equals operator. An underscore (_) in *pattern* stands for (matches) any single character; a percent sign (`%`) matches any string of zero or more characters.

Some examples:

```
'abc' LIKE 'abc'    true
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'      false
```

`LIKE` pattern matches always cover the entire string. To match a sequence anywhere within a string, the pattern must therefore start and end with a percent sign.

To match a literal underscore or percent sign without matching other characters, the respective character in *pattern* must be preceded by the escape character. The default escape character is the backslash but a

different one may be selected by using the ESCAPE clause. To match the escape character itself, write two escape characters.

Note that the backslash already has a special meaning in string literals, so to write a pattern constant that contains a backslash you must write two backslashes in an SQL statement (assuming escape string syntax is used). Thus, writing a pattern that actually matches a literal backslash means writing four backslashes in the statement. You can avoid this by selecting a different escape character with ESCAPE; then a backslash is not special to LIKE anymore. (But it is still special to the string literal parser, so you still need two of them.)

It's also possible to select no escape character by writing ESCAPE ''. This effectively disables the escape mechanism, which makes it impossible to turn off the special meaning of underscore and percent signs in the pattern.

The key word ILIKE can be used instead of LIKE to make the match case-insensitive according to the active locale. This is not in the SQL standard but is a PostgreSQL extension.

The operator ~~ is equivalent to LIKE, and ~~* corresponds to ILIKE. There are also !~~ and !~~* operators that represent NOT LIKE and NOT ILIKE, respectively. All of these operators are PostgreSQL-specific.

9.7.2. SIMILAR TO Regular Expressions

string SIMILAR TO *pattern* [ESCAPE *escape-character*]
string NOT SIMILAR TO *pattern* [ESCAPE *escape-character*]

The SIMILAR TO operator returns true or false depending on whether its pattern matches the given string. It is much like LIKE, except that it interprets the pattern using the SQL standard's definition of a regular expression. SQL regular expressions are a curious cross between LIKE notation and common regular expression notation.

Like LIKE, the SIMILAR TO operator succeeds only if its pattern matches the entire string; this is unlike common regular expression practice, wherein the pattern may match any part of the string. Also like LIKE, SIMILAR TO uses _ and % as wildcard characters denoting any single character and any string, respectively (these are comparable to . and .* in POSIX regular expressions).

In addition to these facilities borrowed from LIKE, SIMILAR TO supports these pattern-matching metacharacters borrowed from POSIX regular expressions:

- | denotes alternation (either of two alternatives).
- * denotes repetition of the previous item zero or more times.
- + denotes repetition of the previous item one or more times.
- Parentheses () may be used to group items into a single logical item.
- A bracket expression [...] specifies a character class, just as in POSIX regular expressions.

Notice that bounded repetition (? and {...}) are not provided, though they exist in POSIX. Also, the dot (.) is not a metacharacter.

As with LIKE, a backslash disables the special meaning of any of these metacharacters; or a different escape character can be specified with ESCAPE.

Some examples:

```
'abc' SIMILAR TO 'abc'      true
'abc' SIMILAR TO 'a'        false
'abc' SIMILAR TO '%(b|d)%'  true
'abc' SIMILAR TO '(b|c)%'   false
```

The `substring` function with three parameters, `substring(`*`string`* `from` *`pattern`* `for` *`escape-character`*`)`, provides extraction of a substring that matches an SQL regular expression pattern. As with `SIMILAR TO`, the specified pattern must match to the entire data string, else the function fails and returns null. To indicate the part of the pattern that should be returned on success, the pattern must contain two occurrences of the escape character followed by a double quote (`"`). The text matching the portion of the pattern between these markers is returned.

Some examples:

```
substring('foobar' from '%#"o_b#"%' for '#')   oob
substring('foobar' from '#"o_b#"%' for '#')    NULL
```

9.7.3. POSIX Regular Expressions

Table 9-11 lists the available operators for pattern matching using POSIX regular expressions.

Table 9-11. Regular Expression Match Operators

Operator	Description	Example
`~`	Matches regular expression, case sensitive	`'thomas' ~ '.*thomas.*'`
`~*`	Matches regular expression, case insensitive	`'thomas' ~* '.*Thomas.*'`
`!~`	Does not match regular expression, case sensitive	`'thomas' !~ '.*Thomas.*'`
`!~*`	Does not match regular expression, case insensitive	`'thomas' !~* '.*vadim.*'`

POSIX regular expressions provide a more powerful means for pattern matching than the `LIKE` and `SIMILAR TO` operators. Many Unix tools such as `egrep`, `sed`, or `awk` use a pattern matching language that is similar to the one described here.

A regular expression is a character sequence that is an abbreviated definition of a set of strings (a *regular set*). A string is said to match a regular expression if it is a member of the regular set described by the regular expression. As with `LIKE`, pattern characters match string characters exactly unless they are special characters in the regular expression language — but regular expressions use different special characters than `LIKE` does. Unlike `LIKE` patterns, a regular expression is allowed to match anywhere within a string, unless the regular expression is explicitly anchored to the beginning or end of the string.

Some examples:

```
'abc' ~ 'abc'    true
'abc' ~ '^a'     true
'abc' ~ '(b|d)'  true
'abc' ~ '^(b|c)' false
```

The `substring` function with two parameters, `substring(`*`string`* `from` *`pattern`*`)`, provides extraction of a substring that matches a POSIX regular expression pattern. It returns null if there is no match, otherwise the portion of the text that matched the pattern. But if the pattern contains any parentheses, the portion of the text that matched the first parenthesized subexpression (the one whose left parenthesis comes first) is returned. You can put parentheses around the whole expression if you want to use parentheses within it without triggering this exception. If you need parentheses in the pattern before the subexpression you want to extract, see the non-capturing parentheses described below.

Some examples:

```
substring('foobar' from 'o.b')     oob
substring('foobar' from 'o(.)b')   o
```

The `regexp_replace` function provides substitution of new text for substrings that match POSIX regular expression patterns. It has the syntax `regexp_replace(`*`source`*, *`pattern`*, *`replacement`* [, *`flags`*]). The *`source`* string is returned unchanged if there is no match to the *`pattern`*. If there is a match, the *`source`* string is returned with the *`replacement`* string substituted for the matching substring. The *`replacement`* string can contain `\`*`n`*, where *`n`* is `1` through `9`, to indicate that the source substring matching the *`n`*'th parenthesized subexpression of the pattern should be inserted, and it can contain `\&` to indicate that the substring matching the entire pattern should be inserted. Write `\\` if you need to put a literal backslash in the replacement text. (As always, remember to double backslashes written in literal constant strings, assuming escape string syntax is used.) The *`flags`* parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. Flag `i` specifies case-insensitive matching, while flag `g` specifies replacement of each matching substring rather than only the first one.

Some examples:

```
regexp_replace('foobarbaz', 'b..', 'X')
                                   fooXbaz
regexp_replace('foobarbaz', 'b..', 'X', 'g')
                                   fooXX
regexp_replace('foobarbaz', 'b(..)', E'X\\1Y', 'g')
                                   fooXarYXazY
```

PostgreSQL's regular expressions are implemented using a package written by Henry Spencer. Much of the description of regular expressions below is copied verbatim from his manual entry.

9.7.3.1. Regular Expression Details

Regular expressions (REs), as defined in POSIX 1003.2, come in two forms: *extended* REs or EREs (roughly those of `egrep`), and *basic* REs or BREs (roughly those of `ed`). PostgreSQL supports both forms, and also implements some extensions that are not in the POSIX standard, but have become widely used anyway due to their availability in programming languages such as Perl and Tcl. REs using these non-POSIX extensions are called *advanced* REs or AREs in this documentation. AREs are almost an exact superset of EREs, but BREs have several notational incompatibilities (as well as being much more limited). We first describe the ARE and ERE forms, noting features that apply only to AREs, and then describe how BREs differ.

> **Note:** The form of regular expressions accepted by PostgreSQL can be chosen by setting the regex_flavor run-time parameter. The usual setting is `advanced`, but one might choose `extended` for maximum backwards compatibility with pre-7.4 releases of PostgreSQL.

A regular expression is defined as one or more *branches*, separated by `|`. It matches anything that matches one of the branches.

A branch is zero or more *quantified atoms* or *constraints*, concatenated. It matches a match for the first, followed by a match for the second, etc; an empty branch matches the empty string.

A quantified atom is an *atom* possibly followed by a single *quantifier*. Without a quantifier, it matches a match for the atom. With a quantifier, it can match some number of matches of the atom. An *atom* can be any of the possibilities shown in Table 9-12. The possible quantifiers and their meanings are shown in Table 9-13.

A *constraint* matches an empty string, but matches only when specific conditions are met. A constraint can be used where an atom could be used, except it may not be followed by a quantifier. The simple constraints are shown in Table 9-14; some more constraints are described later.

Table 9-12. Regular Expression Atoms

Atom	Description
`(`*re*`)`	(where *re* is any regular expression) matches a match for *re*, with the match noted for possible reporting
`(?:`*re*`)`	as above, but the match is not noted for reporting (a "non-capturing" set of parentheses) (AREs only)
`.`	matches any single character
`[`*chars*`]`	a *bracket expression*, matching any one of the *chars* (see Section 9.7.3.2 for more detail)
`\`*k*	(where *k* is a non-alphanumeric character) matches that character taken as an ordinary character, e.g. `\\` matches a backslash character
`\`*c*	where *c* is alphanumeric (possibly followed by other characters) is an *escape*, see Section 9.7.3.3 (AREs only; in EREs and BREs, this matches *c*)

Atom	Description
`{`	when followed by a character other than a digit, matches the left-brace character `{`; when followed by a digit, it is the beginning of a *bound* (see below)
x	where *x* is a single character with no other significance, matches that character

An RE may not end with `\`.

> **Note:** Remember that the backslash (`\`) already has a special meaning in PostgreSQL string literals. To write a pattern constant that contains a backslash, you must write two backslashes in the statement, assuming escape string syntax is used.

Table 9-13. Regular Expression Quantifiers

Quantifier	Matches
`*`	a sequence of 0 or more matches of the atom
`+`	a sequence of 1 or more matches of the atom
`?`	a sequence of 0 or 1 matches of the atom
`{`*m*`}`	a sequence of exactly *m* matches of the atom
`{`*m*`,}`	a sequence of *m* or more matches of the atom
`{`*m*`,`*n*`}`	a sequence of *m* through *n* (inclusive) matches of the atom; *m* may not exceed *n*
`*?`	non-greedy version of `*`
`+?`	non-greedy version of `+`
`??`	non-greedy version of `?`
`{`*m*`}?`	non-greedy version of `{`*m*`}`
`{`*m*`,}?`	non-greedy version of `{`*m*`,}`
`{`*m*`,`*n*`}?`	non-greedy version of `{`*m*`,`*n*`}`

The forms using `{...}` are known as *bounds*. The numbers *m* and *n* within a bound are unsigned decimal integers with permissible values from 0 to 255 inclusive.

Non-greedy quantifiers (available in AREs only) match the same possibilities as their corresponding normal (*greedy*) counterparts, but prefer the smallest number rather than the largest number of matches. See Section 9.7.3.5 for more detail.

> **Note:** A quantifier cannot immediately follow another quantifier. A quantifier cannot begin an expression or subexpression or follow `^` or `|`.

Table 9-14. Regular Expression Constraints

Constraint	Description
`^`	matches at the beginning of the string
`$`	matches at the end of the string
`(?=re)`	*positive lookahead* matches at any point where a substring matching `re` begins (AREs only)
`(?!re)`	*negative lookahead* matches at any point where no substring matching `re` begins (AREs only)

Lookahead constraints may not contain *back references* (see Section 9.7.3.3), and all parentheses within them are considered non-capturing.

9.7.3.2. Bracket Expressions

A *bracket expression* is a list of characters enclosed in `[]`. It normally matches any single character from the list (but see below). If the list begins with `^`, it matches any single character *not* from the rest of the list. If two characters in the list are separated by `-`, this is shorthand for the full range of characters between those two (inclusive) in the collating sequence, e.g. `[0-9]` in ASCII matches any decimal digit. It is illegal for two ranges to share an endpoint, e.g. `a-c-e`. Ranges are very collating-sequence-dependent, so portable programs should avoid relying on them.

To include a literal `]` in the list, make it the first character (following a possible `^`). To include a literal `-`, make it the first or last character, or the second endpoint of a range. To use a literal `-` as the first endpoint of a range, enclose it in `[.` and `.]` to make it a collating element (see below). With the exception of these characters, some combinations using `[` (see next paragraphs), and escapes (AREs only), all other special characters lose their special significance within a bracket expression. In particular, `\` is not special when following ERE or BRE rules, though it is special (as introducing an escape) in AREs.

Within a bracket expression, a collating element (a character, a multiple-character sequence that collates as if it were a single character, or a collating-sequence name for either) enclosed in `[.` and `.]` stands for the sequence of characters of that collating element. The sequence is a single element of the bracket expression's list. A bracket expression containing a multiple-character collating element can thus match more than one character, e.g. if the collating sequence includes a `ch` collating element, then the RE `[[.ch.]]*c` matches the first five characters of `chchcc`.

> **Note:** PostgreSQL currently has no multicharacter collating elements. This information describes possible future behavior.

Within a bracket expression, a collating element enclosed in `[=` and `=]` is an equivalence class, standing for the sequences of characters of all collating elements equivalent to that one, including itself. (If there are no other equivalent collating elements, the treatment is as if the enclosing delimiters were `[.` and `.]`.) For example, if `o` and `^` are the members of an equivalence class, then `[[=o=]]`, `[[=^=]]`, and `[o^]` are all synonymous. An equivalence class may not be an endpoint of a range.

Within a bracket expression, the name of a character class enclosed in `[:` and `:]` stands for the list of all characters belonging to that class. Standard character class names are: `alnum`, `alpha`, `blank`, `cntrl`,

digit, graph, lower, print, punct, space, upper, xdigit. These stand for the character classes defined in ctype. A locale may provide others. A character class may not be used as an endpoint of a range.

There are two special cases of bracket expressions: the bracket expressions [[:<:]] and [[:>:]] are constraints, matching empty strings at the beginning and end of a word respectively. A word is defined as a sequence of word characters that is neither preceded nor followed by word characters. A word character is an alnum character (as defined by ctype) or an underscore. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. The constraint escapes described below are usually preferable (they are no more standard, but are certainly easier to type).

9.7.3.3. Regular Expression Escapes

Escapes are special sequences beginning with \ followed by an alphanumeric character. Escapes come in several varieties: character entry, class shorthands, constraint escapes, and back references. A \ followed by an alphanumeric character but not constituting a valid escape is illegal in AREs. In EREs, there are no escapes: outside a bracket expression, a \ followed by an alphanumeric character merely stands for that character as an ordinary character, and inside a bracket expression, \ is an ordinary character. (The latter is the one actual incompatibility between EREs and AREs.)

Character-entry escapes exist to make it easier to specify non-printing and otherwise inconvenient characters in REs. They are shown in Table 9-15.

Class-shorthand escapes provide shorthands for certain commonly-used character classes. They are shown in Table 9-16.

A *constraint escape* is a constraint, matching the empty string if specific conditions are met, written as an escape. They are shown in Table 9-17.

A *back reference* (*n*) matches the same string matched by the previous parenthesized subexpression specified by the number *n* (see Table 9-18). For example, ([bc])\1 matches bb or cc but not bc or cb. The subexpression must entirely precede the back reference in the RE. Subexpressions are numbered in the order of their leading parentheses. Non-capturing parentheses do not define subexpressions.

> **Note:** Keep in mind that an escape's leading \ will need to be doubled when entering the pattern as an SQL string constant. For example:
>
> ```
> '123' ~ E'^\\d{3}' true
> ```

Table 9-15. Regular Expression Character-Entry Escapes

Escape	Description
\a	alert (bell) character, as in C
\b	backspace, as in C

Escape	Description
\B	synonym for \ to help reduce the need for backslash doubling
\c*X*	(where *X* is any character) the character whose low-order 5 bits are the same as those of *X*, and whose other bits are all zero
\e	the character whose collating-sequence name is ESC, or failing that, the character with octal value 033
\f	form feed, as in C
\n	newline, as in C
\r	carriage return, as in C
\t	horizontal tab, as in C
\u*wxyz*	(where *wxyz* is exactly four hexadecimal digits) the UTF16 (Unicode, 16-bit) character U+*wxyz* in the local byte ordering
\U*stuvwxyz*	(where *stuvwxyz* is exactly eight hexadecimal digits) reserved for a somewhat-hypothetical Unicode extension to 32 bits
\v	vertical tab, as in C
\x*hhh*	(where *hhh* is any sequence of hexadecimal digits) the character whose hexadecimal value is 0x*hhh* (a single character no matter how many hexadecimal digits are used)
\0	the character whose value is 0
xy	(where *xy* is exactly two octal digits, and is not a *back reference*) the character whose octal value is 0*xy*
xyz	(where *xyz* is exactly three octal digits, and is not a *back reference*) the character whose octal value is 0*xyz*

Hexadecimal digits are 0-9, a-f, and A-F. Octal digits are 0-7.

The character-entry escapes are always taken as ordinary characters. For example, \135 is] in ASCII, but \135 does not terminate a bracket expression.

Table 9-16. Regular Expression Class-Shorthand Escapes

Escape	Description
\d	[[:digit:]]
\s	[[:space:]]
\w	[[:alnum:]_] (note underscore is included)
\D	[^[:digit:]]

Escape	Description
\S	[^[:space:]]
\W	[^[:alnum:]_] (note underscore is included)

Within bracket expressions, \d, \s, and \w lose their outer brackets, and \D, \S, and \W are illegal. (So, for example, [a-c\d] is equivalent to [a-c[:digit:]]. Also, [a-c\D], which is equivalent to [a-c^[:digit:]], is illegal.)

Table 9-17. Regular Expression Constraint Escapes

Escape	Description
\A	matches only at the beginning of the string (see Section 9.7.3.5 for how this differs from ^)
\m	matches only at the beginning of a word
\M	matches only at the end of a word
\y	matches only at the beginning or end of a word
\Y	matches only at a point that is not the beginning or end of a word
\Z	matches only at the end of the string (see Section 9.7.3.5 for how this differs from $)

A word is defined as in the specification of [[:<:]] and [[:>:]] above. Constraint escapes are illegal within bracket expressions.

Table 9-18. Regular Expression Back References

Escape	Description
m	(where *m* is a nonzero digit) a back reference to the *m*'th subexpression
mnn	(where *m* is a nonzero digit, and *nn* is some more digits, and the decimal value *mnn* is not greater than the number of closing capturing parentheses seen so far) a back reference to the *mnn*'th subexpression

Note: There is an inherent historical ambiguity between octal character-entry escapes and back references, which is resolved by heuristics, as hinted at above. A leading zero always indicates an octal escape. A single non-zero digit, not followed by another digit, is always taken as a back reference. A multidigit sequence not starting with a zero is taken as a back reference if it comes after a suitable subexpression (i.e. the number is in the legal range for a back reference), and otherwise is taken as octal.

9.7.3.4. Regular Expression Metasyntax

In addition to the main syntax described above, there are some special forms and miscellaneous syntactic facilities available.

Normally the flavor of RE being used is determined by `regex_flavor`. However, this can be overridden by a *director* prefix. If an RE begins with `***:`, the rest of the RE is taken as an ARE regardless of `regex_flavor`. If an RE begins with `***=`, the rest of the RE is taken to be a literal string, with all characters considered ordinary characters.

An ARE may begin with *embedded options*: a sequence `(?xyz)` (where *xyz* is one or more alphabetic characters) specifies options affecting the rest of the RE. These options override any previously determined options (including both the RE flavor and case sensitivity). The available option letters are shown in Table 9-19.

Table 9-19. ARE Embedded-Option Letters

Option	Description
b	rest of RE is a BRE
c	case-sensitive matching (overrides operator type)
e	rest of RE is an ERE
i	case-insensitive matching (see Section 9.7.3.5) (overrides operator type)
m	historical synonym for n
n	newline-sensitive matching (see Section 9.7.3.5)
p	partial newline-sensitive matching (see Section 9.7.3.5)
q	rest of RE is a literal ("quoted") string, all ordinary characters
s	non-newline-sensitive matching (default)
t	tight syntax (default; see below)
w	inverse partial newline-sensitive ("weird") matching (see Section 9.7.3.5)
x	expanded syntax (see below)

Embedded options take effect at the `)` terminating the sequence. They may appear only at the start of an ARE (after the `***:` director if any).

In addition to the usual (*tight*) RE syntax, in which all characters are significant, there is an *expanded* syntax, available by specifying the embedded `x` option. In the expanded syntax, white-space characters in the RE are ignored, as are all characters between a `#` and the following newline (or the end of the RE). This permits paragraphing and commenting a complex RE. There are three exceptions to that basic rule:

- a white-space character or `#` preceded by `\` is retained
- white space or `#` within a bracket expression is retained
- white space and comments cannot appear within multicharacter symbols, such as `(?:`

For this purpose, white-space characters are blank, tab, newline, and any character that belongs to the `space` character class.

Finally, in an ARE, outside bracket expressions, the sequence `(?#ttt)` (where `ttt` is any text not containing a `)`) is a comment, completely ignored. Again, this is not allowed between the characters of multi-character symbols, like `(?:`. Such comments are more a historical artifact than a useful facility, and their use is deprecated; use the expanded syntax instead.

None of these metasyntax extensions is available if an initial `***=` director has specified that the user's input be treated as a literal string rather than as an RE.

9.7.3.5. Regular Expression Matching Rules

In the event that an RE could match more than one substring of a given string, the RE matches the one starting earliest in the string. If the RE could match more than one substring starting at that point, either the longest possible match or the shortest possible match will be taken, depending on whether the RE is *greedy* or *non-greedy*.

Whether an RE is greedy or not is determined by the following rules:

- Most atoms, and all constraints, have no greediness attribute (because they cannot match variable amounts of text anyway).
- Adding parentheses around an RE does not change its greediness.
- A quantified atom with a fixed-repetition quantifier (`{m}` or `{m}?`) has the same greediness (possibly none) as the atom itself.
- A quantified atom with other normal quantifiers (including `{m,n}` with `m` equal to `n`) is greedy (prefers longest match).
- A quantified atom with a non-greedy quantifier (including `{m,n}?` with `m` equal to `n`) is non-greedy (prefers shortest match).
- A branch — that is, an RE that has no top-level `|` operator — has the same greediness as the first quantified atom in it that has a greediness attribute.
- An RE consisting of two or more branches connected by the `|` operator is always greedy.

The above rules associate greediness attributes not only with individual quantified atoms, but with branches and entire REs that contain quantified atoms. What that means is that the matching is done in such a way that the branch, or whole RE, matches the longest or shortest possible substring *as a whole*. Once the length of the entire match is determined, the part of it that matches any particular subexpression is determined on the basis of the greediness attribute of that subexpression, with subexpressions starting earlier in the RE taking priority over ones starting later.

An example of what this means:

```
SELECT SUBSTRING('XY1234Z', 'Y*([0-9]{1,3})');
Result: 123
SELECT SUBSTRING('XY1234Z', 'Y*?([0-9]{1,3})');
Result: 1
```

In the first case, the RE as a whole is greedy because `Y*` is greedy. It can match beginning at the `Y`, and it matches the longest possible string starting there, i.e., `Y123`. The output is the parenthesized part of that, or `123`. In the second case, the RE as a whole is non-greedy because `Y*?` is non-greedy. It can match beginning at the `Y`, and it matches the shortest possible string starting there, i.e., `Y1`. The subexpression `[0-9]{1,3}` is greedy but it cannot change the decision as to the overall match length; so it is forced to match just `1`.

In short, when an RE contains both greedy and non-greedy subexpressions, the total match length is either as long as possible or as short as possible, according to the attribute assigned to the whole RE. The attributes assigned to the subexpressions only affect how much of that match they are allowed to "eat" relative to each other.

The quantifiers `{1,1}` and `{1,1}?` can be used to force greediness or non-greediness, respectively, on a subexpression or a whole RE.

Match lengths are measured in characters, not collating elements. An empty string is considered longer than no match at all. For example: `bb*` matches the three middle characters of `abbbc`; `(week|wee)(night|knights)` matches all ten characters of `weeknights`; when `(.*).*` is matched against `abc` the parenthesized subexpression matches all three characters; and when `(a*)*` is matched against `bc` both the whole RE and the parenthesized subexpression match an empty string.

If case-independent matching is specified, the effect is much as if all case distinctions had vanished from the alphabet. When an alphabetic that exists in multiple cases appears as an ordinary character outside a bracket expression, it is effectively transformed into a bracket expression containing both cases, e.g. `x` becomes `[xX]`. When it appears inside a bracket expression, all case counterparts of it are added to the bracket expression, e.g. `[x]` becomes `[xX]` and `[^x]` becomes `[^xX]`.

If newline-sensitive matching is specified, `.` and bracket expressions using `^` will never match the newline character (so that matches will never cross newlines unless the RE explicitly arranges it) and `^`and `$` will match the empty string after and before a newline respectively, in addition to matching at beginning and end of string respectively. But the ARE escapes `\A` and `\Z` continue to match beginning or end of string *only*.

If partial newline-sensitive matching is specified, this affects `.` and bracket expressions as with newline-sensitive matching, but not `^` and `$`.

If inverse partial newline-sensitive matching is specified, this affects `^` and `$` as with newline-sensitive matching, but not `.` and bracket expressions. This isn't very useful but is provided for symmetry.

9.7.3.6. Limits and Compatibility

No particular limit is imposed on the length of REs in this implementation. However, programs intended to be highly portable should not employ REs longer than 256 bytes, as a POSIX-compliant implementation can refuse to accept such REs.

The only feature of AREs that is actually incompatible with POSIX EREs is that `\` does not lose its special significance inside bracket expressions. All other ARE features use syntax which is illegal or has undefined or unspecified effects in POSIX EREs; the `***` syntax of directors likewise is outside the POSIX syntax for both BREs and EREs.

Many of the ARE extensions are borrowed from Perl, but some have been changed to clean them up, and a few Perl extensions are not present. Incompatibilities of note include `\b`, `\B`, the lack of special treatment for a trailing newline, the addition of complemented bracket expressions to the things affected by newline-

sensitive matching, the restrictions on parentheses and back references in lookahead constraints, and the longest/shortest-match (rather than first-match) matching semantics.

Two significant incompatibilities exist between AREs and the ERE syntax recognized by pre-7.4 releases of PostgreSQL:

- In AREs, `\` followed by an alphanumeric character is either an escape or an error, while in previous releases, it was just another way of writing the alphanumeric. This should not be much of a problem because there was no reason to write such a sequence in earlier releases.
- In AREs, `\` remains a special character within `[]`, so a literal `\` within a bracket expression must be written `\\`.

While these differences are unlikely to create a problem for most applications, you can avoid them if necessary by setting `regex_flavor` to `extended`.

9.7.3.7. Basic Regular Expressions

BREs differ from EREs in several respects. `|`, `+`, and `?` are ordinary characters and there is no equivalent for their functionality. The delimiters for bounds are `\{` and `\}`, with `{` and `}` by themselves ordinary characters. The parentheses for nested subexpressions are `\(` and `\)`, with `(` and `)` by themselves ordinary characters. `^` is an ordinary character except at the beginning of the RE or the beginning of a parenthesized subexpression, `$` is an ordinary character except at the end of the RE or the end of a parenthesized subexpression, and `*` is an ordinary character if it appears at the beginning of the RE or the beginning of a parenthesized subexpression (after a possible leading `^`). Finally, single-digit back references are available, and `\<` and `\>` are synonyms for `[[:<:]]` and `[[:>:]]` respectively; no other escapes are available.

9.8. Data Type Formatting Functions

The PostgreSQL formatting functions provide a powerful set of tools for converting various data types (date/time, integer, floating point, numeric) to formatted strings and for converting from formatted strings to specific data types. Table 9-20 lists them. These functions all follow a common calling convention: the first argument is the value to be formatted and the second argument is a template that defines the output or input format.

The `to_timestamp` function can also take a single `double precision` argument to convert from Unix epoch to `timestamp with time zone`. (`Integer` Unix epochs are implicitly cast to `double precision`.)

Table 9-20. Formatting Functions

Function	Return Type	Description	Example
`to_char(timestamp, text)`	`text`	convert time stamp to string	`to_char(current_timestamp, 'HH12:MI:SS')`

Function	Return Type	Description	Example
to_char(interval, text)	text	convert interval to string	to_char(interval '15h 2m 12s', 'HH24:MI:SS')
to_char(int, text)	text	convert integer to string	to_char(125, '999')
to_char(double precision, text)	text	convert real/double precision to string	to_char(125.8::real, '999D9')
to_char(numeric, text)	text	convert numeric to string	to_char(-125.8, '999D99S')
to_date(text, text)	date	convert string to date	to_date('05 Dec 2000', 'DD Mon YYYY')
to_number(text, text)	numeric	convert string to numeric	to_number('12,454.8-', '99G999D9S')
to_timestamp(text, text)	timestamp with time zone	convert string to time stamp	to_timestamp('05 Dec 2000', 'DD Mon YYYY')
to_timestamp(double precision)	timestamp with time zone	convert UNIX epoch to time stamp	to_timestamp(200120400)

In an output template string (for to_char), there are certain patterns that are recognized and replaced with appropriately-formatted data from the value to be formatted. Any text that is not a template pattern is simply copied verbatim. Similarly, in an input template string (for anything but to_char), template patterns identify the parts of the input data string to be looked at and the values to be found there.

Table 9-21 shows the template patterns available for formatting date and time values.

Table 9-21. Template Patterns for Date/Time Formatting

Pattern	Description
HH	hour of day (01-12)
HH12	hour of day (01-12)
HH24	hour of day (00-23)
MI	minute (00-59)
SS	second (00-59)
MS	millisecond (000-999)
US	microsecond (000000-999999)
SSSS	seconds past midnight (0-86399)
AM or A.M. or PM or P.M.	meridian indicator (uppercase)
am or a.m. or pm or p.m.	meridian indicator (lowercase)
Y,YYY	year (4 and more digits) with comma
YYYY	year (4 and more digits)
YYY	last 3 digits of year
YY	last 2 digits of year

Pattern	Description
`Y`	last digit of year
`IYYY`	ISO year (4 and more digits)
`IYY`	last 3 digits of ISO year
`IY`	last 2 digits of ISO year
`I`	last digits of ISO year
`BC` or `B.C.` or `AD` or `A.D.`	era indicator (uppercase)
`bc` or `b.c.` or `ad` or `a.d.`	era indicator (lowercase)
`MONTH`	full uppercase month name (blank-padded to 9 chars)
`Month`	full mixed-case month name (blank-padded to 9 chars)
`month`	full lowercase month name (blank-padded to 9 chars)
`MON`	abbreviated uppercase month name (3 chars in English, localized lengths vary)
`Mon`	abbreviated mixed-case month name (3 chars in English, localized lengths vary)
`mon`	abbreviated lowercase month name (3 chars in English, localized lengths vary)
`MM`	month number (01-12)
`DAY`	full uppercase day name (blank-padded to 9 chars)
`Day`	full mixed-case day name (blank-padded to 9 chars)
`day`	full lowercase day name (blank-padded to 9 chars)
`DY`	abbreviated uppercase day name (3 chars in English, localized lengths vary)
`Dy`	abbreviated mixed-case day name (3 chars in English, localized lengths vary)
`dy`	abbreviated lowercase day name (3 chars in English, localized lengths vary)
`DDD`	day of year (001-366)
`DD`	day of month (01-31)
`D`	day of week (1-7; Sunday is 1)
`W`	week of month (1-5) (The first week starts on the first day of the month.)
`WW`	week number of year (1-53) (The first week starts on the first day of the year.)
`IW`	ISO week number of year (The first Thursday of the new year is in week 1.)
`CC`	century (2 digits) (The twenty-first century starts on 2001-01-01.)

Pattern	Description
J	Julian Day (days since January 1, 4712 BC)
Q	quarter
RM	month in Roman numerals (I-XII; I=January) (uppercase)
rm	month in Roman numerals (i-xii; i=January) (lowercase)
TZ	time-zone name (uppercase)
tz	time-zone name (lowercase)

Certain modifiers may be applied to any template pattern to alter its behavior. For example, FMMonth is the Month pattern with the FM modifier. Table 9-22 shows the modifier patterns for date/time formatting.

Table 9-22. Template Pattern Modifiers for Date/Time Formatting

Modifier	Description	Example
FM prefix	fill mode (suppress padding blanks and zeroes)	FMMonth
TH suffix	uppercase ordinal number suffix	DDTH
th suffix	lowercase ordinal number suffix	DDth
FX prefix	fixed format global option (see usage notes)	FX Month DD Day
TM prefix	translation mode (print localized day and month names based on lc_messages)	TMMonth
SP suffix	spell mode (not yet implemented)	DDSP

Usage notes for date/time formatting:

- FM suppresses leading zeroes and trailing blanks that would otherwise be added to make the output of a pattern be fixed-width.
- TM does not include trailing blanks.
- to_timestamp and to_date skip multiple blank spaces in the input string if the FX option is not used. FX must be specified as the first item in the template. For example to_timestamp('2000 JUN', 'YYYY MON') is correct, but to_timestamp('2000 JUN', 'FXYYYY MON') returns an error, because to_timestamp expects one space only.
- Ordinary text is allowed in to_char templates and will be output literally. You can put a substring in double quotes to force it to be interpreted as literal text even if it contains pattern key words. For example, in '"Hello Year "YYYY', the YYYY will be replaced by the year data, but the single Y in Year will not be.
- If you want to have a double quote in the output you must precede it with a backslash, for example E'\\"YYYY Month\\"'. (Two backslashes are necessary because the backslash already has a special

meaning when using the escape string syntax.)

- The `YYYY` conversion from string to `timestamp` or `date` has a restriction if you use a year with more than 4 digits. You must use some non-digit character or template after `YYYY`, otherwise the year is always interpreted as 4 digits. For example (with the year 20000): `to_date('200001131', 'YYYYMMDD')` will be interpreted as a 4-digit year; instead use a non-digit separator after the year, like `to_date('20000-1131', 'YYYY-MMDD')` or `to_date('20000Nov31', 'YYYYMonDD')`.
- In conversions from string to `timestamp` or `date`, the `CC` field is ignored if there is a `YYY`, `YYYY` or `Y,YYY` field. If `CC` is used with `YY` or `Y` then the year is computed as `(CC-1)*100+YY`.
- Millisecond (`MS`) and microsecond (`US`) values in a conversion from string to `timestamp` are used as part of the seconds after the decimal point. For example `to_timestamp('12:3', 'SS:MS')` is not 3 milliseconds, but 300, because the conversion counts it as 12 + 0.3 seconds. This means for the format `SS:MS`, the input values `12:3`, `12:30`, and `12:300` specify the same number of milliseconds. To get three milliseconds, one must use `12:003`, which the conversion counts as 12 + 0.003 = 12.003 seconds.

  Here is a more complex example: `to_timestamp('15:12:02.020.001230', 'HH:MI:SS.MS.US')` is 15 hours, 12 minutes, and 2 seconds + 20 milliseconds + 1230 microseconds = 2.021230 seconds.
- `to_char`'s day of the week numbering (see the 'D' formatting pattern) is different from that of the `extract` function.
- `to_char(interval)` formats `HH` and `HH12` as hours in a single day, while `HH24` can output hours exceeding a single day, e.g. >24.

Table 9-23 shows the template patterns available for formatting numeric values.

Table 9-23. Template Patterns for Numeric Formatting

Pattern	Description
`9`	value with the specified number of digits
`0`	value with leading zeros
`.` (period)	decimal point
`,` (comma)	group (thousand) separator
`PR`	negative value in angle brackets
`S`	sign anchored to number (uses locale)
`L`	currency symbol (uses locale)
`D`	decimal point (uses locale)
`G`	group separator (uses locale)
`MI`	minus sign in specified position (if number < 0)
`PL`	plus sign in specified position (if number > 0)
`SG`	plus/minus sign in specified position
`RN`	roman numeral (input between 1 and 3999)
`TH` or `th`	ordinal number suffix
`V`	shift specified number of digits (see notes)

Pattern	Description
EEEE	scientific notation (not implemented yet)

Usage notes for numeric formatting:

- A sign formatted using SG, PL, or MI is not anchored to the number; for example, to_char(-12, 'S9999') produces ' -12', but to_char(-12, 'MI9999') produces '- 12'. The Oracle implementation does not allow the use of MI ahead of 9, but rather requires that 9 precede MI.
- 9 results in a value with the same number of digits as there are 9s. If a digit is not available it outputs a space.
- TH does not convert values less than zero and does not convert fractional numbers.
- PL, SG, and TH are PostgreSQL extensions.
- V effectively multiplies the input values by 10^*n*, where *n* is the number of digits following V. to_char does not support the use of V combined with a decimal point. (E.g., 99.9V99 is not allowed.)

Table 9-24 shows some examples of the use of the to_char function.

Table 9-24. to_char Examples

Expression	Result
to_char(current_timestamp, 'Day, DD HH12:MI:SS')	'Tuesday , 06 05:39:18'
to_char(current_timestamp, 'FMDay, FMDD HH12:MI:SS')	'Tuesday, 6 05:39:18'
to_char(-0.1, '99.99')	' -.10'
to_char(-0.1, 'FM9.99')	'-.1'
to_char(0.1, '0.9')	' 0.1'
to_char(12, '9990999.9')	' 0012.0'
to_char(12, 'FM9990999.9')	'0012.'
to_char(485, '999')	' 485'
to_char(-485, '999')	'-485'
to_char(485, '9 9 9')	' 4 8 5'
to_char(1485, '9,999')	' 1,485'
to_char(1485, '9G999')	' 1 485'
to_char(148.5, '999.999')	' 148.500'
to_char(148.5, 'FM999.999')	'148.5'
to_char(148.5, 'FM999.990')	'148.500'
to_char(148.5, '999D999')	' 148,500'
to_char(3148.5, '9G999D999')	' 3 148,500'
to_char(-485, '999S')	'485-'

Expression	Result
to_char(-485, '999MI')	'485-'
to_char(485, '999MI')	'485 '
to_char(485, 'FM999MI')	'485'
to_char(485, 'PL999')	'+485'
to_char(485, 'SG999')	'+485'
to_char(-485, 'SG999')	'-485'
to_char(-485, '9SG99')	'4-85'
to_char(-485, '999PR')	'<485>'
to_char(485, 'L999')	'DM 485
to_char(485, 'RN')	' CDLXXXV'
to_char(485, 'FMRN')	'CDLXXXV'
to_char(5.2, 'FMRN')	'V'
to_char(482, '999th')	' 482nd'
to_char(485, '"Good number:"999')	'Good number: 485'
to_char(485.8, '"Pre:"999" Post:" .999')	'Pre: 485 Post: .800'
to_char(12, '99V999')	' 12000'
to_char(12.4, '99V999')	' 12400'
to_char(12.45, '99V9')	' 125'

9.9. Date/Time Functions and Operators

Table 9-26 shows the available functions for date/time value processing, with details appearing in the following subsections. Table 9-25 illustrates the behaviors of the basic arithmetic operators (+, *, etc.). For formatting functions, refer to Section 9.8. You should be familiar with the background information on date/time data types from Section 8.5.

All the functions and operators described below that take `time` or `timestamp` inputs actually come in two variants: one that takes `time with time zone` or `timestamp with time zone`, and one that takes `time without time zone` or `timestamp without time zone`. For brevity, these variants are not shown separately. Also, the + and * operators come in commutative pairs (for example both date + integer and integer + date); we show only one of each such pair.

Table 9-25. Date/Time Operators

Operator	Example	Result
+	date '2001-09-28' + integer '7'	date '2001-10-05'
+	date '2001-09-28' + interval '1 hour'	timestamp '2001-09-28 01:00:00'

Operator	Example	Result
+	date '2001-09-28' + time '03:00'	timestamp '2001-09-28 03:00:00'
+	interval '1 day' + interval '1 hour'	interval '1 day 01:00:00'
+	timestamp '2001-09-28 01:00' + interval '23 hours'	timestamp '2001-09-29 00:00:00'
+	time '01:00' + interval '3 hours'	time '04:00:00'
-	- interval '23 hours'	interval '-23:00:00'
-	date '2001-10-01' - date '2001-09-28'	integer '3'
-	date '2001-10-01' - integer '7'	date '2001-09-24'
-	date '2001-09-28' - interval '1 hour'	timestamp '2001-09-27 23:00:00'
-	time '05:00' - time '03:00'	interval '02:00:00'
-	time '05:00' - interval '2 hours'	time '03:00:00'
-	timestamp '2001-09-28 23:00' - interval '23 hours'	timestamp '2001-09-28 00:00:00'
-	interval '1 day' - interval '1 hour'	interval '1 day -01:00:00'
-	timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00'	interval '1 day 15:00:00'
*	900 * interval '1 second'	interval '00:15:00'
*	21 * interval '1 day'	interval '21 days'
*	double precision '3.5' * interval '1 hour'	interval '03:30:00'
/	interval '1 hour' / double precision '1.5'	interval '00:40:00'

Table 9-26. Date/Time Functions

Function	Return Type	Description	Example	Result

Function	Return Type	Description	Example	Result
age(timestamp, timestamp)	interval	Subtract arguments, producing a "symbolic" result that uses years and months	age(timestamp '2001-04-10', timestamp '1957-06-13')	43 years 9 mons 27 days
age(timestamp)	interval	Subtract from current_date	age(timestamp '1957-06-13')	43 years 8 mons 3 days
clock_timestamp()	timestamp with time zone	Current date and time (changes during statement execution); see Section 9.9.4		
current_date	date	Current date; see Section 9.9.4		
current_time	time with time zone	Current time of day; see Section 9.9.4		
current_timestamp	timestamp with time zone	Current date and time (start of current transaction); see Section 9.9.4		
date_part(text, timestamp)	double precision	Get subfield (equivalent to extract); see Section 9.9.1	date_part('hour', timestamp '2001-02-16 20:38:40')	20
date_part(text, interval)	double precision	Get subfield (equivalent to extract); see Section 9.9.1	date_part('month', interval '2 years 3 months')	3
date_trunc(text, timestamp)	timestamp	Truncate to specified precision; see also Section 9.9.2	date_trunc('hour', timestamp '2001-02-16 20:38:40')	2001-02-16 20:00:00
extract(field from timestamp)	double precision	Get subfield; see Section 9.9.1	extract(hour from timestamp '2001-02-16 20:38:40')	20
extract(field from interval)	double precision	Get subfield; see Section 9.9.1	extract(month from interval '2 years 3 months')	3

Function	Return Type	Description	Example	Result
isfinite(timestamp)	boolean	Test for finite time stamp (not equal to infinity)	isfinite(timestamp '2001-02-16 21:28:30')	true
isfinite(interval)	boolean	Test for finite interval	isfinite(interval '4 hours')	true
justify_days(interval)	interval	Adjust interval so 30-day time periods are represented as months	justify_days(interval '30 days')	1 mon
justify_hours(interval)	interval	Adjust interval so 24-hour time periods are represented as days	justify_hours(interval '24 hours')	1 day
justify_interval(interval)	interval	Adjust interval using justify_days and justify_hours, with additional sign adjustments	justify_interval('1 mon -1 hour')	29 days 23:00:00
localtime	time	Current time of day; see Section 9.9.4		
localtimestamp	timestamp	Current date and time (start of current transaction); see Section 9.9.4		
now()	timestamp with time zone	Current date and time (start of current transaction); see Section 9.9.4		
statement_timestamp()	timestamp with time zone	Current date and time (start of current statement); see Section 9.9.4		

Function	Return Type	Description	Example	Result
`timeofday()`	`text`	Current date and time (like `clock_timestamp`, but as a `text` string); see Section 9.9.4		
`transaction_timestamp()`	`timestamp with time zone`	Current date and time (start of current transaction); see Section 9.9.4		

In addition to these functions, the SQL `OVERLAPS` operator is supported:

```
(start1, end1) OVERLAPS (start2, end2)
(start1, length1) OVERLAPS (start2, length2)
```

This expression yields true when two time periods (defined by their endpoints) overlap, false when they do not overlap. The endpoints can be specified as pairs of dates, times, or time stamps; or as a date, time, or time stamp followed by an interval.

```
SELECT (DATE '2001-02-16', DATE '2001-12-21') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Result: true
SELECT (DATE '2001-02-16', INTERVAL '100 days') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Result: false
```

When adding an `interval` value to (or subtracting an `interval` value from) a `timestamp with time zone` value, the days component advances (or decrements) the date of the `timestamp with time zone` by the indicated number of days. Across daylight saving time changes (with the session time zone set to a time zone that recognizes DST), this means `interval '1 day'` does not necessarily equal `interval '24 hours'`. For example, with the session time zone set to `CST7CDT`, `timestamp with time zone '2005-04-02 12:00-07' + interval '1 day'` will produce `timestamp with time zone '2005-04-03 12:00-06'`, while adding `interval '24 hours'` to the same initial `timestamp with time zone` produces `timestamp with time zone '2005-04-03 13:00-06'`, as there is a change in daylight saving time at `2005-04-03 02:00` in time zone `CST7CDT`.

9.9.1. `EXTRACT`, `date_part`

```
EXTRACT(field FROM source)
```

The `extract` function retrieves subfields such as year or hour from date/time values. *`source`* must be a value expression of type `timestamp`, `time`, or `interval`. (Expressions of type `date` will be cast to `timestamp` and can therefore be used as well.) *`field`* is an identifier or string that selects what field to extract from the source value. The `extract` function returns values of type `double precision`. The following are valid field names:

`century`

The century

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
Result: 20
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 21
```

The first century starts at 0001-01-01 00:00:00 AD, although they did not know it at the time. This definition applies to all Gregorian calendar countries. There is no century number 0, you go from -1 to 1. If you disagree with this, please write your complaint to: Pope, Cathedral Saint-Peter of Roma, Vatican.

PostgreSQL releases before 8.0 did not follow the conventional numbering of centuries, but just returned the year field divided by 100.

`day`

The day (of the month) field (1 - 31)

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 16
```

`decade`

The year field divided by 10

```
SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 200
```

`dow`

The day of the week (0 - 6; Sunday is 0) (for `timestamp` values only)

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 5
```

Note that `extract`'s day of the week numbering is different from that of the `to_char` function.

`doy`

The day of the year (1 - 365/366) (for `timestamp` values only)

```
SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 47
```

`epoch`

For `date` and `timestamp` values, the number of seconds since 1970-01-01 00:00:00-00 (can be negative); for `interval` values, the total number of seconds in the interval

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-08');
Result: 982384720

SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
Result: 442800
```

Here is how you can convert an epoch value back to a time stamp:

```
SELECT TIMESTAMP WITH TIME ZONE 'epoch' + 982384720 * INTERVAL '1 second';
```

`hour`

The hour field (0 - 23)

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 20
```

`microseconds`

The seconds field, including fractional parts, multiplied by 1 000 000. Note that this includes full seconds.

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
Result: 28500000
```

`millennium`

The millennium

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 3
```

Years in the 1900s are in the second millennium. The third millennium starts January 1, 2001.

PostgreSQL releases before 8.0 did not follow the conventional numbering of millennia, but just returned the year field divided by 1000.

`milliseconds`

The seconds field, including fractional parts, multiplied by 1000. Note that this includes full seconds.

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
Result: 28500
```

`minute`

The minutes field (0 - 59)

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 38
```

`month`

For `timestamp` values, the number of the month within the year (1 - 12) ; for `interval` values the number of months, modulo 12 (0 - 11)

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');
Result: 3

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
Result: 1
```

`quarter`

The quarter of the year (1 - 4) that the day is in (for `timestamp` values only)

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 1
```

second

The seconds field, including fractional parts (0 - 59[1])

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 40

SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
Result: 28.5
```

timezone

The time zone offset from UTC, measured in seconds. Positive values correspond to time zones east of UTC, negative values to zones west of UTC.

timezone_hour

The hour component of the time zone offset

timezone_minute

The minute component of the time zone offset

week

The number of the week of the year that the day is in. By definition (ISO 8601), the first week of a year contains January 4 of that year. (The ISO-8601 week starts on Monday.) In other words, the first Thursday of a year is in week 1 of that year. (for timestamp values only)

Because of this, it is possible for early January dates to be part of the 52nd or 53rd week of the previous year. For example, 2005-01-01 is part of the 53rd week of year 2004, and 2006-01-01 is part of the 52nd week of year 2005.

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 7
```

year

The year field. Keep in mind there is no 0 AD, so subtracting BC years from AD years should be done with care.

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2001
```

The extract function is primarily intended for computational processing. For formatting date/time values for display, see Section 9.8.

The date_part function is modeled on the traditional Ingres equivalent to the SQL-standard function extract:

```
date_part('field', source)
```

Note that here the *field* parameter needs to be a string value, not a name. The valid field names for date_part are the same as for extract.

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
Result: 16
```

60 if leap seconds are implemented by the operating system

```
SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
Result: 4
```

9.9.2. date_trunc

The function date_trunc is conceptually similar to the trunc function for numbers.

```
date_trunc('field', source)
```

source is a value expression of type timestamp or interval. (Values of type date and time are cast automatically, to timestamp or interval respectively.) *field* selects to which precision to truncate the input value. The return value is of type timestamp or interval with all fields that are less significant than the selected one set to zero (or one, for day and month).

Valid values for *field* are:

```
microseconds
milliseconds
second
minute
hour
day
week
month
quarter
year
decade
century
millennium
```

Examples:

```
SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-02-16 20:00:00

SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-01-01 00:00:00
```

9.9.3. AT TIME ZONE

The AT TIME ZONE construct allows conversions of time stamps to different time zones. Table 9-27 shows its variants.

Table 9-27. `AT TIME ZONE` Variants

Expression	Return Type	Description
`timestamp without time zone AT TIME ZONE` *zone*	`timestamp with time zone`	Treat given time stamp *without time zone* as located in the specified time zone
`timestamp with time zone AT TIME ZONE` *zone*	`timestamp without time zone`	Convert given time stamp *with time zone* to the new time zone
`time with time zone AT TIME ZONE` *zone*	`time with time zone`	Convert given time *with time zone* to the new time zone

In these expressions, the desired time zone *zone* can be specified either as a text string (e.g., `'PST'`) or as an interval (e.g., `INTERVAL '-08:00'`). In the text case, a time zone name may be specified in any of the ways described in Section 8.5.3.

Examples (supposing that the local time zone is `PST8PDT`):

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'MST';
Result: 2001-02-16 19:38:40-08

SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'MST';
Result: 2001-02-16 18:38:40
```

The first example takes a time stamp without time zone and interprets it as MST time (UTC-7), which is then converted to PST (UTC-8) for display. The second example takes a time stamp specified in EST (UTC-5) and converts it to local time in MST (UTC-7).

The function `timezone(`*zone*`, `*timestamp*`)` is equivalent to the SQL-conforming construct *timestamp* `AT TIME ZONE` *zone*.

9.9.4. Current Date/Time

PostgreSQL provides a number of functions that return values related to the current date and time. These SQL-standard functions all return values based on the start time of the current transaction:

```
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME(precision)
CURRENT_TIMESTAMP(precision)
LOCALTIME
LOCALTIMESTAMP
LOCALTIME(precision)
LOCALTIMESTAMP(precision)
```

`CURRENT_TIME` and `CURRENT_TIMESTAMP` deliver values with time zone; `LOCALTIME` and `LOCALTIMESTAMP` deliver values without time zone.

CURRENT_TIME, CURRENT_TIMESTAMP, LOCALTIME, and LOCALTIMESTAMP can optionally be given a precision parameter, which causes the result to be rounded to that many fractional digits in the seconds field. Without a precision parameter, the result is given to the full available precision.

Some examples:

```
SELECT CURRENT_TIME;
Result: 14:39:53.662522-05

SELECT CURRENT_DATE;
Result: 2001-12-23

SELECT CURRENT_TIMESTAMP;
Result: 2001-12-23 14:39:53.662522-05

SELECT CURRENT_TIMESTAMP(2);
Result: 2001-12-23 14:39:53.66-05

SELECT LOCALTIMESTAMP;
Result: 2001-12-23 14:39:53.662522
```

Since these functions return the start time of the current transaction, their values do not change during the transaction. This is considered a feature: the intent is to allow a single transaction to have a consistent notion of the "current" time, so that multiple modifications within the same transaction bear the same time stamp.

> **Note:** Other database systems may advance these values more frequently.

PostgreSQL also provides functions that return the start time of the current statement, as well as the actual current time at the instant the function is called. The complete list of non-SQL-standard time functions is:

```
now()
transaction_timestamp()
statement_timestamp()
clock_timestamp()
timeofday()
```

now() is a traditional PostgreSQL equivalent to CURRENT_TIMESTAMP. transaction_timestamp() is likewise equivalent to CURRENT_TIMESTAMP, but is named to clearly reflect what it returns. statement_timestamp() returns the start time of the current statement (more specifically, the time of receipt of the latest command message from the client). statement_timestamp() and transaction_timestamp() return the same value during the first command of a transaction, but may differ during subsequent commands. clock_timestamp() returns the actual current time, and therefore its value changes even within a single SQL command. timeofday() is a historical PostgreSQL function. Like clock_timestamp(), it returns the actual current time, but as a formatted text string rather than a timestamp with time zone value.

All the date/time data types also accept the special literal value `now` to specify the current date and time (again, interpreted as the transaction start time). Thus, the following three all return the same result:

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now';  -- incorrect for use with DEFAULT
```

> **Tip:** You do not want to use the third form when specifying a `DEFAULT` clause while creating a table. The system will convert `now` to a `timestamp` as soon as the constant is parsed, so that when the default value is needed, the time of the table creation would be used! The first two forms will not be evaluated until the default value is used, because they are function calls. Thus they will give the desired behavior of defaulting to the time of row insertion.

9.9.5. Delaying Execution

The following function is available to delay execution of the server process:

`pg_sleep(`*`seconds`*`)`

`pg_sleep` makes the current session's process sleep until *`seconds`* seconds have elapsed. *`seconds`* is a value of type `double precision`, so fractional-second delays can be specified. For example:

```
SELECT pg_sleep(1.5);
```

> **Note:** The effective resolution of the sleep interval is platform-specific; 0.01 seconds is a common value. The sleep delay will be at least as long as specified. It may be longer depending on factors such as server load.

> **Warning**
>
> Make sure that your session does not hold more locks than necessary when calling `pg_sleep`. Otherwise other sessions might have to wait for your sleeping process, slowing down the entire system.

9.10. Geometric Functions and Operators

The geometric types `point`, `box`, `lseg`, `line`, `path`, `polygon`, and `circle` have a large set of native support functions and operators, shown in Table 9-28, Table 9-29, and Table 9-30.

Caution

Note that the "same as" operator, ~=, represents the usual notion of equality for the point, box, polygon, and circle types. Some of these types also have an = operator, but = compares for equal *areas* only. The other scalar comparison operators (<= and so on) likewise compare areas for these types.

Table 9-28. Geometric Operators

Operator	Description	Example
+	Translation	box '((0,0),(1,1))' + point '(2.0,0)'
-	Translation	box '((0,0),(1,1))' - point '(2.0,0)'
*	Scaling/rotation	box '((0,0),(1,1))' * point '(2.0,0)'
/	Scaling/rotation	box '((0,0),(2,2))' / point '(2.0,0)'
#	Point or box of intersection	'((1,-1),(-1,1))' # '((1,1),(-1,-1))'
#	Number of points in path or polygon	# '((1,0),(0,1),(-1,0))'
@-@	Length or circumference	@-@ path '((0,0),(1,0))'
@@	Center	@@ circle '((0,0),10)'
##	Closest point to first operand on second operand	point '(0,0)' ## lseg '((2,0),(0,2))'
<->	Distance between	circle '((0,0),1)' <-> circle '((5,0),1)'
&&	Overlaps?	box '((0,0),(1,1))' && box '((0,0),(2,2))'
<<	Is strictly left of?	circle '((0,0),1)' << circle '((5,0),1)'
>>	Is strictly right of?	circle '((5,0),1)' >> circle '((0,0),1)'
&<	Does not extend to the right of?	box '((0,0),(1,1))' &< box '((0,0),(2,2))'
&>	Does not extend to the left of?	box '((0,0),(3,3))' &> box '((0,0),(2,2))'
<<|	Is strictly below?	box '((0,0),(3,3))' <<| box '((3,4),(5,5))'
|>>	Is strictly above?	box '((3,4),(5,5))' |>> box '((0,0),(3,3))'

Operator	Description	Example
&<|	Does not extend above?	box '((0,0),(1,1))' &<| box '((0,0),(2,2))'
|&>	Does not extend below?	box '((0,0),(3,3))' |&> box '((0,0),(2,2))'
<^	Is below (allows touching)?	circle '((0,0),1)' <^ circle '((0,5),1)'
>^	Is above (allows touching)?	circle '((0,5),1)' >^ circle '((0,0),1)'
?#	Intersects?	lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'
?-	Is horizontal?	?- lseg '((-1,0),(1,0))'
?-	Are horizontally aligned?	point '(1,0)' ?- point '(0,0)'
?|	Is vertical?	?| lseg '((-1,0),(1,0))'
?|	Are vertically aligned?	point '(0,1)' ?| point '(0,0)'
?-|	Is perpendicular?	lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))'
?||	Are parallel?	lseg '((-1,0),(1,0))' ?|| lseg '((-1,2),(1,2))'
@>	Contains?	circle '((0,0),2)' @> point '(1,1)'
<@	Contained in or on?	point '(1,1)' <@ circle '((0,0),2)'
~=	Same as?	polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'

Note: Before PostgreSQL 8.2, the containment operators @> and <@ were respectively called ~ and @. These names are still available, but are deprecated and will eventually be retired.

Table 9-29. Geometric Functions

Function	Return Type	Description	Example
area(*object*)	double precision	area	area(box '((0,0),(1,1))')
center(*object*)	point	center	center(box '((0,0),(1,2))')
diameter(circle)	double precision	diameter of circle	diameter(circle '((0,0),2.0)')

Function	Return Type	Description	Example
height(box)	double precision	vertical size of box	height(box '((0,0),(1,1))')
isclosed(path)	boolean	a closed path?	isclosed(path '((0,0),(1,1),(2,0))')
isopen(path)	boolean	an open path?	isopen(path '[(0,0),(1,1),(2,0)]')
length(*object*)	double precision	length	length(path '((-1,0),(1,0))')
npoints(path)	int	number of points	npoints(path '[(0,0),(1,1),(2,0)]')
npoints(polygon)	int	number of points	npoints(polygon '((1,1),(0,0))')
pclose(path)	path	convert path to closed	pclose(path '[(0,0),(1,1),(2,0)]')
popen(path)	path	convert path to open	popen(path '((0,0),(1,1),(2,0))')
radius(circle)	double precision	radius of circle	radius(circle '((0,0),2.0)')
width(box)	double precision	horizontal size of box	width(box '((0,0),(1,1))')

Table 9-30. Geometric Type Conversion Functions

Function	Return Type	Description	Example
box(circle)	box	circle to box	box(circle '((0,0),2.0)')
box(point, point)	box	points to box	box(point '(0,0)', point '(1,1)')
box(polygon)	box	polygon to box	box(polygon '((0,0),(1,1),(2,0))')
circle(box)	circle	box to circle	circle(box '((0,0),(1,1))')
circle(point, double precision)	circle	center and radius to circle	circle(point '(0,0)', 2.0)
circle(polygon)	circle	polygon to circle	circle(polygon '((0,0),(1,1),(2,0))')

Function	Return Type	Description	Example
`lseg(box)`	`lseg`	box diagonal to line segment	`lseg(box '((-1,0),(1,0))')`
`lseg(point, point)`	`lseg`	points to line segment	`lseg(point '(-1,0)', point '(1,0)')`
`path(polygon)`	`point`	polygon to path	`path(polygon '((0,0),(1,1),(2,0))')`
`point(double precision, double precision)`	`point`	construct point	`point(23.4, -44.5)`
`point(box)`	`point`	center of box	`point(box '((-1,0),(1,0))')`
`point(circle)`	`point`	center of circle	`point(circle '((0,0),2.0)')`
`point(lseg)`	`point`	center of line segment	`point(lseg '((-1,0),(1,0))')`
`point(polygon)`	`point`	center of polygon	`point(polygon '((0,0),(1,1),(2,0))')`
`polygon(box)`	`polygon`	box to 4-point polygon	`polygon(box '((0,0),(1,1))')`
`polygon(circle)`	`polygon`	circle to 12-point polygon	`polygon(circle '((0,0),2.0)')`
`polygon(`*`npts`*`, circle)`	`polygon`	circle to *`npts`*-point polygon	`polygon(12, circle '((0,0),2.0)')`
`polygon(path)`	`polygon`	path to polygon	`polygon(path '((0,0),(1,1),(2,0))')`

It is possible to access the two component numbers of a `point` as though it were an array with indices 0 and 1. For example, if `t.p` is a `point` column then `SELECT p[0] FROM t` retrieves the X coordinate and `UPDATE t SET p[1] = ...` changes the Y coordinate. In the same way, a value of type `box` or `lseg` may be treated as an array of two `point` values.

The `area` function works for the types `box`, `circle`, and `path`. The `area` function only works on the `path` data type if the points in the `path` are non-intersecting. For example, the `path '((0,0),(0,1),(2,1),(2,2),(1,2),(1,0),(0,0))'::PATH` won't work, however, the following visually identical `path '((0,0),(0,1),(1,1),(1,2),(2,2),(2,1),(1,1),(1,0),(0,0))'::PATH` will work. If the concept of an intersecting versus non-intersecting `path` is confusing, draw both of the above `path`s side by side on a piece of graph paper.

9.11. Network Address Functions and Operators

Table 9-31 shows the operators available for the `cidr` and `inet` types. The operators <<, <<=, >>, and >>= test for subnet inclusion. They consider only the network parts of the two addresses, ignoring any host part, and determine whether one network part is identical to or a subnet of the other.

Table 9-31. `cidr` and `inet` Operators

Operator	Description	Example
<	is less than	`inet '192.168.1.5' < inet '192.168.1.6'`
<=	is less than or equal	`inet '192.168.1.5' <= inet '192.168.1.5'`
=	equals	`inet '192.168.1.5' = inet '192.168.1.5'`
>=	is greater or equal	`inet '192.168.1.5' >= inet '192.168.1.5'`
>	is greater than	`inet '192.168.1.5' > inet '192.168.1.4'`
<>	is not equal	`inet '192.168.1.5' <> inet '192.168.1.4'`
<<	is contained within	`inet '192.168.1.5' << inet '192.168.1/24'`
<<=	is contained within or equals	`inet '192.168.1/24' <<= inet '192.168.1/24'`
>>	contains	`inet '192.168.1/24' >> inet '192.168.1.5'`
>>=	contains or equals	`inet '192.168.1/24' >>= inet '192.168.1/24'`
~	bitwise NOT	`~ inet '192.168.1.6'`
&	bitwise AND	`inet '192.168.1.6' & inet '0.0.0.255'`
|	bitwise OR	`inet '192.168.1.6' | inet '0.0.0.255'`
+	addition	`inet '192.168.1.6' + 25`
-	subtraction	`inet '192.168.1.43' - 36`
-	subtraction	`inet '192.168.1.43' - inet '192.168.1.19'`

Table 9-32 shows the functions available for use with the `cidr` and `inet` types. The `host`, `text`, and `abbrev` functions are primarily intended to offer alternative display formats.

Table 9-32. `cidr` and `inet` Functions

Function	Return Type	Description	Example	Result

Function	Return Type	Description	Example	Result
`abbrev(inet)`	`text`	abbreviated display format as text	`abbrev(inet '10.1.0.0/16')`	`10.1.0.0/16`
`abbrev(cidr)`	`text`	abbreviated display format as text	`abbrev(cidr '10.1.0.0/16')`	`10.1/16`
`broadcast(inet)`	`inet`	broadcast address for network	`broadcast('192.168.1.5/24')`	`192.168.1.255/24`
`family(inet)`	`int`	extract family of address; `4` for IPv4, `6` for IPv6	`family('::1')`	`6`
`host(inet)`	`text`	extract IP address as text	`host('192.168.1.5/24')`	`192.168.1.5`
`hostmask(inet)`	`inet`	construct host mask for network	`hostmask('192.168.23.20/30')`	`0.0.0.3`
`masklen(inet)`	`int`	extract netmask length	`masklen('192.168.1.5/24')`	`24`
`netmask(inet)`	`inet`	construct netmask for network	`netmask('192.168.1.5/24')`	`255.255.255.0`
`network(inet)`	`cidr`	extract network part of address	`network('192.168.1.5/24')`	`192.168.1.0/24`
`set_masklen(inet, int)`	`inet`	set netmask length for `inet` value	`set_masklen('192.168.1.5/24', 16)`	`192.168.1.5/16`
`set_masklen(cidr, int)`	`cidr`	set netmask length for `cidr` value	`set_masklen('192.168.0.0/24', 16)`	`192.168.0.0/16`
`text(inet)`	`text`	extract IP address and netmask length as text	`text(inet '192.168.1.5')`	`192.168.1.5/32`

Any `cidr` value can be cast to `inet` implicitly or explicitly; therefore, the functions shown above as operating on `inet` also work on `cidr` values. (Where there are separate functions for `inet` and `cidr`, it is because the behavior should be different for the two cases.) Also, it is permitted to cast an `inet` value to `cidr`. When this is done, any bits to the right of the netmask are silently zeroed to create a valid `cidr` value. In addition, you can cast a text value to `inet` or `cidr` using normal casting syntax: for example, `inet(`*`expression`*`)` or *`colname`*`::cidr`.

Table 9-33 shows the functions available for use with the `macaddr` type. The function `trunc(macaddr)` returns a MAC address with the last 3 bytes set to zero. This can be used to associate the remaining prefix with a manufacturer.

Table 9-33. `macaddr` Functions

Function	Return Type	Description	Example	Result
`trunc(macaddr)`	`macaddr`	set last 3 bytes to zero	`trunc(macaddr '12:34:56:78:90:ab')`	`12:34:56:00:00:00`

The macaddr type also supports the standard relational operators (>, <=, etc.) for lexicographical ordering.

9.12. Sequence Manipulation Functions

This section describes PostgreSQL's functions for operating on *sequence objects*. Sequence objects (also called sequence generators or just sequences) are special single-row tables created with CREATE SEQUENCE. A sequence object is usually used to generate unique identifiers for rows of a table. The sequence functions, listed in Table 9-34, provide simple, multiuser-safe methods for obtaining successive sequence values from sequence objects.

Table 9-34. Sequence Functions

Function	Return Type	Description
currval(regclass)	bigint	Return value most recently obtained with nextval for specified sequence
nextval(regclass)	bigint	Advance sequence and return new value
setval(regclass, bigint)	bigint	Set sequence's current value
setval(regclass, bigint, boolean)	bigint	Set sequence's current value and is_called flag

The sequence to be operated on by a sequence-function call is specified by a regclass argument, which is just the OID of the sequence in the pg_class system catalog. You do not have to look up the OID by hand, however, since the regclass data type's input converter will do the work for you. Just write the sequence name enclosed in single quotes, so that it looks like a literal constant. To achieve some compatibility with the handling of ordinary SQL names, the string will be converted to lowercase unless it contains double quotes around the sequence name. Thus

```
nextval('foo')      operates on sequence foo
nextval('FOO')      operates on sequence foo
nextval('"Foo"')    operates on sequence Foo
```

The sequence name can be schema-qualified if necessary:

```
nextval('myschema.foo')     operates on myschema.foo
nextval('"myschema".foo')   same as above
nextval('foo')              searches search path for foo
```

See Section 8.12 for more information about regclass.

> **Note:** Before PostgreSQL 8.1, the arguments of the sequence functions were of type text, not regclass, and the above-described conversion from a text string to an OID value would happen at run time during each call. For backwards compatibility, this facility still exists, but internally it is now handled as an implicit coercion from text to regclass before the function is invoked.

When you write the argument of a sequence function as an unadorned literal string, it becomes a constant of type `regclass`. Since this is really just an OID, it will track the originally identified sequence despite later renaming, schema reassignment, etc. This "early binding" behavior is usually desirable for sequence references in column defaults and views. But sometimes you will want "late binding" where the sequence reference is resolved at run time. To get late-binding behavior, force the constant to be stored as a `text` constant instead of `regclass`:

```
nextval('foo'::text)      foo is looked up at runtime
```

Note that late binding was the only behavior supported in PostgreSQL releases before 8.1, so you may need to do this to preserve the semantics of old applications.

Of course, the argument of a sequence function can be an expression as well as a constant. If it is a text expression then the implicit coercion will result in a run-time lookup.

The available sequence functions are:

`nextval`

Advance the sequence object to its next value and return that value. This is done atomically: even if multiple sessions execute `nextval` concurrently, each will safely receive a distinct sequence value.

`currval`

Return the value most recently obtained by `nextval` for this sequence in the current session. (An error is reported if `nextval` has never been called for this sequence in this session.) Notice that because this is returning a session-local value, it gives a predictable answer whether or not other sessions have executed `nextval` since the current session did.

`lastval`

Return the value most recently returned by `nextval` in the current session. This function is identical to `currval`, except that instead of taking the sequence name as an argument it fetches the value of the last sequence that `nextval` was used on in the current session. It is an error to call `lastval` if `nextval` has not yet been called in the current session.

`setval`

Reset the sequence object's counter value. The two-parameter form sets the sequence's `last_value` field to the specified value and sets its `is_called` field to `true`, meaning that the next `nextval` will advance the sequence before returning a value. In the three-parameter form, `is_called` may be set either `true` or `false`. If it's set to `false`, the next `nextval` will return exactly the specified value, and sequence advancement commences with the following `nextval`. For example,

```
SELECT setval('foo', 42);           Next nextval will return 43
SELECT setval('foo', 42, true);     Same as above
SELECT setval('foo', 42, false);    Next nextval will return 42
```

The result returned by `setval` is just the value of its second argument.

If a sequence object has been created with default parameters, `nextval` calls on it will return successive values beginning with 1. Other behaviors can be obtained by using special parameters in the *CREATE SEQUENCE* command; see its command reference page for more information.

Important: To avoid blocking of concurrent transactions that obtain numbers from the same sequence, a `nextval` operation is never rolled back; that is, once a value has been fetched it is considered used, even if the transaction that did the `nextval` later aborts. This means that aborted transactions may leave unused "holes" in the sequence of assigned values. `setval` operations are never rolled back, either.

9.13. Conditional Expressions

This section describes the SQL-compliant conditional expressions available in PostgreSQL.

Tip: If your needs go beyond the capabilities of these conditional expressions you might want to consider writing a stored procedure in a more expressive programming language.

9.13.1. CASE

The SQL `CASE` expression is a generic conditional expression, similar to if/else statements in other languages:

```
CASE WHEN condition THEN result
     [WHEN ...]
     [ELSE result]
END
```

`CASE` clauses can be used wherever an expression is valid. *condition* is an expression that returns a `boolean` result. If the result is true then the value of the `CASE` expression is the *result* that follows the condition. If the result is false any subsequent `WHEN` clauses are searched in the same manner. If no `WHEN` *condition* is true then the value of the case expression is the *result* in the `ELSE` clause. If the `ELSE` clause is omitted and no condition matches, the result is null.

An example:

```
SELECT * FROM test;

 a
---
 1
 2
 3


SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
       END
    FROM test;
```

```
 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

The data types of all the *result* expressions must be convertible to a single output type. See Section 10.5 for more detail.

The following "simple" CASE expression is a specialized variant of the general form above:

```
CASE expression
    WHEN value THEN result
    [WHEN ...]
    [ELSE result]
END
```

The *expression* is computed and compared to all the *value* specifications in the WHEN clauses until one is found that is equal. If no match is found, the *result* in the ELSE clause (or a null value) is returned. This is similar to the switch statement in C.

The example above can be written using the simple CASE syntax:

```
SELECT a,
       CASE a WHEN 1 THEN 'one'
              WHEN 2 THEN 'two'
              ELSE 'other'
       END
    FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

A CASE expression does not evaluate any subexpressions that are not needed to determine the result. For example, this is a possible way of avoiding a division-by-zero failure:

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

9.13.2. COALESCE

```
COALESCE(value [, ...])
```

The COALESCE function returns the first of its arguments that is not null. Null is returned only if all arguments are null. It is often used to substitute a default value for null values when data is retrieved for display, for example:

```
SELECT COALESCE(description, short_description, '(none)') ...
```

Like a CASE expression, COALESCE will not evaluate arguments that are not needed to determine the result; that is, arguments to the right of the first non-null argument are not evaluated. This SQL-standard function provides capabilities similar to NVL and IFNULL, which are used in some other database systems.

9.13.3. NULLIF

```
NULLIF(value1, value2)
```

The NULLIF function returns a null value if *value1* and *value2* are equal; otherwise it returns *value1*. This can be used to perform the inverse operation of the COALESCE example given above:

```
SELECT NULLIF(value, '(none)') ...
```

If *value1* is (none), return a null, otherwise return *value1*.

9.13.4. GREATEST and LEAST

```
GREATEST(value [, ...])
```

```
LEAST(value [, ...])
```

The GREATEST and LEAST functions select the largest or smallest value from a list of any number of expressions. The expressions must all be convertible to a common data type, which will be the type of the result (see Section 10.5 for details). NULL values in the list are ignored. The result will be NULL only if all the expressions evaluate to NULL.

Note that GREATEST and LEAST are not in the SQL standard, but are a common extension.

9.14. Array Functions and Operators

Table 9-35 shows the operators available for array types.

Table 9-35. array Operators

Operator	Description	Example	Result

Operator	Description	Example	Result
=	equal	ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3]	t
<>	not equal	ARRAY[1,2,3] <> ARRAY[1,2,4]	t
<	less than	ARRAY[1,2,3] < ARRAY[1,2,4]	t
>	greater than	ARRAY[1,4,3] > ARRAY[1,2,4]	t
<=	less than or equal	ARRAY[1,2,3] <= ARRAY[1,2,3]	t
>=	greater than or equal	ARRAY[1,4,3] >= ARRAY[1,4,3]	t
@>	contains	ARRAY[1,4,3] @> ARRAY[3,1]	t
<@	is contained by	ARRAY[2,7] <@ ARRAY[1,7,4,2,6]	t
&&	overlap (have elements in common)	ARRAY[1,4,3] && ARRAY[2,1]	t
||	array-to-array concatenation	ARRAY[1,2,3] || ARRAY[4,5,6]	{1,2,3,4,5,6}
||	array-to-array concatenation	ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9]]	{{1,2,3},{4,5,6},{7,8,9}}
||	element-to-array concatenation	3 || ARRAY[4,5,6]	{3,4,5,6}
||	array-to-element concatenation	ARRAY[4,5,6] || 7	{4,5,6,7}

Array comparisons compare the array contents element-by-element, using the default B-Tree comparison function for the element data type. In multidimensional arrays the elements are visited in row-major order (last subscript varies most rapidly). If the contents of two arrays are equal but the dimensionality is different, the first difference in the dimensionality information determines the sort order. (This is a change from versions of PostgreSQL prior to 8.2: older versions would claim that two arrays with the same contents were equal, even if the number of dimensions or subscript ranges were different.)

See Section 8.10 for more details about array operator behavior.

Table 9-36 shows the functions available for use with array types. See Section 8.10 for more discussion and examples of the use of these functions.

Table 9-36. `array` Functions

Function	Return Type	Description	Example	Result

Function	Return Type	Description	Example	Result
`array_append(anyarray, anyelement)`	`anyarray`	append an element to the end of an array	`array_append(ARRAY[1,2], 3)`	`{1,2,3}`
`array_cat(anyarray, anyarray)`	`anyarray`	concatenate two arrays	`array_cat(ARRAY[1,2,3], ARRAY[4,5])`	`{1,2,3,4,5}`
`array_dims(anyarray)`	`text`	returns a text representation of array's dimensions	`array_dims(ARRAY[[1,2,3], [4,5,6]])`	`[1:2][1:3]`
`array_lower(anyarray, int)`	`int`	returns lower bound of the requested array dimension	`array_lower('[0:2]={1,2,3}'::int[], 1)`	`0`
`array_prepend(anyelement, anyarray)`	`anyarray`	append an element to the beginning of an array	`array_prepend(1, ARRAY[2,3])`	`{1,2,3}`
`array_to_string(anyarray, text)`	`text`	concatenates array elements using provided delimiter	`array_to_string(ARRAY[1, 2, 3], '~^~')`	`1~^~2~^~3`
`array_upper(anyarray, int)`	`int`	returns upper bound of the requested array dimension	`array_upper(ARRAY[1,2,3,4], 1)`	`4`
`string_to_array(text, text)`	`text[]`	splits string into array elements using provided delimiter	`string_to_array('xx~^~yy~^~zz', '~^~')`	`{xx,yy,zz}`

9.15. Aggregate Functions

Aggregate functions compute a single result value from a set of input values. The built-in aggregate functions are listed in Table 9-37 and Table 9-38. The special syntax considerations for aggregate functions are explained in Section 4.2.7. Consult Section 2.7 for additional introductory information.

Table 9-37. General-Purpose Aggregate Functions

Function	Argument Type	Return Type	Description

Function	Argument Type	Return Type	Description
avg(*expression*)	smallint, int, bigint, real, double precision, numeric, or interval	numeric for any integer type argument, double precision for a floating-point argument, otherwise the same as the argument data type	the average (arithmetic mean) of all input values
bit_and(*expression*)	smallint, int, bigint, or bit	same as argument data type	the bitwise AND of all non-null input values, or null if none
bit_or(*expression*)	smallint, int, bigint, or bit	same as argument data type	the bitwise OR of all non-null input values, or null if none
bool_and(*expression*)	bool	bool	true if all input values are true, otherwise false
bool_or(*expression*)	bool	bool	true if at least one input value is true, otherwise false
count(*)		bigint	number of input rows
count(*expression*)	any	bigint	number of input rows for which the value of *expression* is not null
every(*expression*)	bool	bool	equivalent to bool_and
max(*expression*)	any array, numeric, string, or date/time type	same as argument type	maximum value of *expression* across all input values
min(*expression*)	any array, numeric, string, or date/time type	same as argument type	minimum value of *expression* across all input values
sum(*expression*)	smallint, int, bigint, real, double precision, numeric, or interval	bigint for smallint or int arguments, numeric for bigint arguments, double precision for floating-point arguments, otherwise the same as the argument data type	sum of *expression* across all input values

It should be noted that except for count, these functions return a null value when no rows are selected. In particular, sum of no rows returns null, not zero as one might expect. The coalesce function may be used to substitute zero for null when necessary.

Note: Boolean aggregates `bool_and` and `bool_or` correspond to standard SQL aggregates `every` and `any` or `some`. As for `any` and `some`, it seems that there is an ambiguity built into the standard syntax:

```
SELECT b1 = ANY((SELECT b2 FROM t2 ...)) FROM t1 ...;
```

Here `ANY` can be considered both as leading to a subquery or as an aggregate if the select expression returns 1 row. Thus the standard name cannot be given to these aggregates.

Note: Users accustomed to working with other SQL database management systems may be surprised by the performance of the `count` aggregate when it is applied to the entire table. A query like:

```
SELECT count(*) FROM sometable;
```

will be executed by PostgreSQL using a sequential scan of the entire table.

Table 9-38 shows aggregate functions typically used in statistical analysis. (These are separated out merely to avoid cluttering the listing of more-commonly-used aggregates.) Where the description mentions *N*, it means the number of input rows for which all the input expressions are non-null. In all cases, null is returned if the computation is meaningless, for example when *N* is zero.

Table 9-38. Aggregate Functions for Statistics

Function	Argument Type	Return Type	Description
corr(*Y*, *X*)	double precision	double precision	correlation coefficient
covar_pop(*Y*, *X*)	double precision	double precision	population covariance
covar_samp(*Y*, *X*)	double precision	double precision	sample covariance
regr_avgx(*Y*, *X*)	double precision	double precision	average of the independent variable (sum(*X*)/*N*)
regr_avgy(*Y*, *X*)	double precision	double precision	average of the dependent variable (sum(*Y*)/*N*)
regr_count(*Y*, *X*)	double precision	bigint	number of input rows in which both expressions are nonnull
regr_intercept(*Y*, *X*)	double precision	double precision	y-intercept of the least-squares-fit linear equation determined by the (*X*, *Y*) pairs
regr_r2(*Y*, *X*)	double precision	double precision	square of the correlation coefficient

Function	Argument Type	Return Type	Description
regr_slope(*Y*, *X*)	double precision	double precision	slope of the least-squares-fit linear equation determined by the (*X*, *Y*) pairs
regr_sxx(*Y*, *X*)	double precision	double precision	sum(*X*^2) - sum(*X*)^2/*N* ("sum of squares" of the independent variable)
regr_sxy(*Y*, *X*)	double precision	double precision	sum(*X***Y*) - sum(*X*) * sum(*Y*)/*N* ("sum of products" of independent times dependent variable)
regr_syy(*Y*, *X*)	double precision	double precision	sum(*Y*^2) - sum(*Y*)^2/*N* ("sum of squares" of the dependent variable)
stddev(*expression*)	smallint, int, bigint, real, double precision, or numeric	double precision for floating-point arguments, otherwise numeric	historical alias for stddev_samp
stddev_pop(*expression*)	smallint, int, bigint, real, double precision, or numeric	double precision for floating-point arguments, otherwise numeric	population standard deviation of the input values
stddev_samp(*expression*)	smallint, int, bigint, real, double precision, or numeric	double precision for floating-point arguments, otherwise numeric	sample standard deviation of the input values
variance(*expression*)	smallint, int, bigint, real, double precision, or numeric	double precision for floating-point arguments, otherwise numeric	historical alias for var_samp
var_pop(*expression*)	smallint, int, bigint, real, double precision, or numeric	double precision for floating-point arguments, otherwise numeric	population variance of the input values (square of the population standard deviation)
var_samp(*expression*)	smallint, int, bigint, real, double precision, or numeric	double precision for floating-point arguments, otherwise numeric	sample variance of the input values (square of the sample standard deviation)

9.16. Subquery Expressions

This section describes the SQL-compliant subquery expressions available in PostgreSQL. All of the expression forms documented in this section return Boolean (true/false) results.

9.16.1. EXISTS

```
EXISTS (subquery)
```

The argument of EXISTS is an arbitrary SELECT statement, or *subquery*. The subquery is evaluated to determine whether it returns any rows. If it returns at least one row, the result of EXISTS is "true"; if the subquery returns no rows, the result of EXISTS is "false".

The subquery can refer to variables from the surrounding query, which will act as constants during any one evaluation of the subquery.

The subquery will generally only be executed far enough to determine whether at least one row is returned, not all the way to completion. It is unwise to write a subquery that has any side effects (such as calling sequence functions); whether the side effects occur or not may be difficult to predict.

Since the result depends only on whether any rows are returned, and not on the contents of those rows, the output list of the subquery is normally uninteresting. A common coding convention is to write all EXISTS tests in the form EXISTS(SELECT 1 WHERE ...). There are exceptions to this rule however, such as subqueries that use INTERSECT.

This simple example is like an inner join on col2, but it produces at most one output row for each tab1 row, even if there are multiple matching tab2 rows:

```
SELECT col1 FROM tab1
    WHERE EXISTS(SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

9.16.2. IN

```
expression IN (subquery)
```

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of IN is "true" if any equal subquery row is found. The result is "false" if no equal row is found (including the special case where the subquery returns no rows).

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand row yields null, the result of the IN construct will be null, not false. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

```
row_constructor IN (subquery)
```

The left-hand side of this form of IN is a row constructor, as described in Section 4.2.11. The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result. The result of IN is "true" if any equal subquery row is found. The result is "false" if no equal row is found (including the special case where the subquery returns no rows).

As usual, null values in the rows are combined per the normal rules of SQL Boolean expressions. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of that row comparison is unknown (null). If all the per-row results are either unequal or null, with at least one null, then the result of IN is null.

9.16.3. NOT IN

```
expression NOT IN (subquery)
```

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of NOT IN is "true" if only unequal subquery rows are found (including the special case where the subquery returns no rows). The result is "false" if any equal row is found.

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand row yields null, the result of the NOT IN construct will be null, not true. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

```
row_constructor NOT IN (subquery)
```

The left-hand side of this form of NOT IN is a row constructor, as described in Section 4.2.11. The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result. The result of NOT IN is "true" if only unequal subquery rows are found (including the special case where the subquery returns no rows). The result is "false" if any equal row is found.

As usual, null values in the rows are combined per the normal rules of SQL Boolean expressions. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of that row comparison is unknown (null). If all the per-row results are either unequal or null, with at least one null, then the result of NOT IN is null.

9.16.4. ANY/SOME

```
expression operator ANY (subquery)
expression operator SOME (subquery)
```

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given *operator*,

which must yield a Boolean result. The result of ANY is "true" if any true result is obtained. The result is "false" if no true result is found (including the special case where the subquery returns no rows).

SOME is a synonym for ANY. IN is equivalent to = ANY.

Note that if there are no successes and at least one right-hand row yields null for the operator's result, the result of the ANY construct will be null, not false. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

```
row_constructor operator ANY (subquery)
row_constructor operator SOME (subquery)
```

The left-hand side of this form of ANY is a row constructor, as described in Section 4.2.11. The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result, using the given *operator*. The result of ANY is "true" if the comparison returns true for any subquery row. The result is "false" if the comparison returns false for every subquery row (including the special case where the subquery returns no rows). The result is NULL if the comparison does not return true for any row, and it returns NULL for at least one row.

See Section 9.17.5 for details about the meaning of a row-wise comparison.

9.16.5. ALL

```
expression operator ALL (subquery)
```

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given *operator*, which must yield a Boolean result. The result of ALL is "true" if all rows yield true (including the special case where the subquery returns no rows). The result is "false" if any false result is found. The result is NULL if the comparison does not return false for any row, and it returns NULL for at least one row.

NOT IN is equivalent to <> ALL.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

```
row_constructor operator ALL (subquery)
```

The left-hand side of this form of ALL is a row constructor, as described in Section 4.2.11. The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result, using the given *operator*. The result of ALL is "true" if the comparison returns true for all subquery rows (including the special case where the subquery returns no rows). The result is "false" if the comparison returns false for any subquery row. The result is NULL if the comparison does not return false for any subquery row, and it returns NULL for at least one row.

See Section 9.17.5 for details about the meaning of a row-wise comparison.

9.16.6. Row-wise Comparison

```
row_constructor operator (subquery)
```

The left-hand side is a row constructor, as described in Section 4.2.11. The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. Furthermore, the subquery cannot return more than one row. (If it returns zero rows, the result is taken to be null.) The left-hand side is evaluated and compared row-wise to the single subquery result row.

See Section 9.17.5 for details about the meaning of a row-wise comparison.

9.17. Row and Array Comparisons

This section describes several specialized constructs for making multiple comparisons between groups of values. These forms are syntactically related to the subquery forms of the previous section, but do not involve subqueries. The forms involving array subexpressions are PostgreSQL extensions; the rest are SQL-compliant. All of the expression forms documented in this section return Boolean (true/false) results.

9.17.1. IN

```
expression IN (value [, ...])
```

The right-hand side is a parenthesized list of scalar expressions. The result is "true" if the left-hand expression's result is equal to any of the right-hand expressions. This is a shorthand notation for

```
expression = value1
OR
expression = value2
OR
...
```

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand expression yields null, the result of the IN construct will be null, not false. This is in accordance with SQL's normal rules for Boolean combinations of null values.

9.17.2. NOT IN

```
expression NOT IN (value [, ...])
```

The right-hand side is a parenthesized list of scalar expressions. The result is "true" if the left-hand expression's result is unequal to all of the right-hand expressions. This is a shorthand notation for

```
expression <> value1
AND
expression <> value2
```

```
AND
...
```

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand expression yields null, the result of the `NOT IN` construct will be null, not true as one might naively expect. This is in accordance with SQL's normal rules for Boolean combinations of null values.

> **Tip:** `x NOT IN y` is equivalent to `NOT (x IN y)` in all cases. However, null values are much more likely to trip up the novice when working with `NOT IN` than when working with `IN`. It's best to express your condition positively if possible.

9.17.3. `ANY`/`SOME` (array)

```
expression operator ANY (array expression)
expression operator SOME (array expression)
```

The right-hand side is a parenthesized expression, which must yield an array value. The left-hand expression is evaluated and compared to each element of the array using the given *`operator`*, which must yield a Boolean result. The result of `ANY` is "true" if any true result is obtained. The result is "false" if no true result is found (including the special case where the array has zero elements).

If the array expression yields a null array, the result of `ANY` will be null. If the left-hand expression yields null, the result of `ANY` is ordinarily null (though a non-strict comparison operator could possibly yield a different result). Also, if the right-hand array contains any null elements and no true comparison result is obtained, the result of `ANY` will be null, not false (again, assuming a strict comparison operator). This is in accordance with SQL's normal rules for Boolean combinations of null values.

`SOME` is a synonym for `ANY`.

9.17.4. `ALL` (array)

```
expression operator ALL (array expression)
```

The right-hand side is a parenthesized expression, which must yield an array value. The left-hand expression is evaluated and compared to each element of the array using the given *`operator`*, which must yield a Boolean result. The result of `ALL` is "true" if all comparisons yield true (including the special case where the array has zero elements). The result is "false" if any false result is found.

If the array expression yields a null array, the result of `ALL` will be null. If the left-hand expression yields null, the result of `ALL` is ordinarily null (though a non-strict comparison operator could possibly yield a different result). Also, if the right-hand array contains any null elements and no false comparison result is obtained, the result of `ALL` will be null, not true (again, assuming a strict comparison operator). This is in accordance with SQL's normal rules for Boolean combinations of null values.

9.17.5. Row-wise Comparison

row_constructor operator row_constructor

Each side is a row constructor, as described in Section 4.2.11. The two row values must have the same number of fields. Each side is evaluated and they are compared row-wise. Row comparisons are allowed when the *operator* is =, <>, <, <=, > or >=, or has semantics similar to one of these. (To be specific, an operator can be a row comparison operator if it is a member of a B-Tree operator class, or is the negator of the = member of a B-Tree operator class.)

The = and <> cases work slightly differently from the others. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of the row comparison is unknown (null).

For the <, <=, > and >= cases, the row elements are compared left-to-right, stopping as soon as an unequal or null pair of elements is found. If either of this pair of elements is null, the result of the row comparison is unknown (null); otherwise comparison of this pair of elements determines the result. For example, `ROW(1,2,NULL) < ROW(1,3,0)` yields true, not null, because the third pair of elements are not considered.

> **Note:** Prior to PostgreSQL 8.2, the <, <=, > and >= cases were not handled per SQL specification. A comparison like `ROW(a,b) < ROW(c,d)` was implemented as `a < c AND b < d` whereas the correct behavior is equivalent to `a < c OR (a = c AND b < d)`.

row_constructor IS DISTINCT FROM *row_constructor*

This construct is similar to a <> row comparison, but it does not yield null for null inputs. Instead, any null value is considered unequal to (distinct from) any non-null value, and any two nulls are considered equal (not distinct). Thus the result will always be either true or false, never null.

row_constructor IS NOT DISTINCT FROM *row_constructor*

This construct is similar to a = row comparison, but it does not yield null for null inputs. Instead, any null value is considered unequal to (distinct from) any non-null value, and any two nulls are considered equal (not distinct). Thus the result will always be either true or false, never null.

9.18. Set Returning Functions

This section describes functions that possibly return more than one row. Currently the only functions in this class are series generating functions, as detailed in Table 9-39.

Table 9-39. Series Generating Functions

Function	Argument Type	Return Type	Description

Function	Argument Type	Return Type	Description
`generate_series(start, stop)`	`int` or `bigint`	`setof int` or `setof bigint` (same as argument type)	Generate a series of values, from `start` to `stop` with a step size of one
`generate_series(start, stop, step)`	`int` or `bigint`	`setof int` or `setof bigint` (same as argument type)	Generate a series of values, from `start` to `stop` with a step size of `step`

When `step` is positive, zero rows are returned if `start` is greater than `stop`. Conversely, when `step` is negative, zero rows are returned if `start` is less than `stop`. Zero rows are also returned for `NULL` inputs. It is an error for `step` to be zero. Some examples follow:

```
select * from generate_series(2,4);
 generate_series
-----------------
               2
               3
               4
(3 rows)

select * from generate_series(5,1,-2);
 generate_series
-----------------
               5
               3
               1
(3 rows)

select * from generate_series(4,3);
 generate_series
-----------------
(0 rows)

select current_date + s.a as dates from generate_series(0,14,7) as s(a);
   dates
------------
 2004-02-05
 2004-02-12
 2004-02-19
(3 rows)
```

9.19. System Information Functions

Table 9-40 shows several functions that extract session and system information.

Table 9-40. Session Information Functions

Name	Return Type	Description
`current_database()`	`name`	name of current database
`current_schema()`	`name`	name of current schema
`current_schemas(boolean)`	`name[]`	names of schemas in search path optionally including implicit schemas
`current_user`	`name`	user name of current execution context
`inet_client_addr()`	`inet`	address of the remote connection
`inet_client_port()`	`int`	port of the remote connection
`inet_server_addr()`	`inet`	address of the local connection
`inet_server_port()`	`int`	port of the local connection
`pg_my_temp_schema()`	`oid`	OID of session's temporary schema, or 0 if none
`pg_is_other_temp_schema(oid)`	`boolean`	is schema another session's temporary schema?
`pg_postmaster_start_time()`	`timestamp with time zone`	server start time
`session_user`	`name`	session user name
`user`	`name`	equivalent to `current_user`
`version()`	`text`	PostgreSQL version information

The `session_user` is normally the user who initiated the current database connection; but superusers can change this setting with *SET SESSION AUTHORIZATION*. The `current_user` is the user identifier that is applicable for permission checking. Normally, it is equal to the session user, but it can be changed with *SET ROLE*. It also changes during the execution of functions with the attribute `SECURITY DEFINER`. In Unix parlance, the session user is the "real user" and the current user is the "effective user".

> **Note:** `current_user`, `session_user`, and `user` have special syntactic status in SQL: they must be called without trailing parentheses.

`current_schema` returns the name of the schema that is at the front of the search path (or a null value if the search path is empty). This is the schema that will be used for any tables or other named objects that are created without specifying a target schema. `current_schemas(boolean)` returns an array of the names of all schemas presently in the search path. The Boolean option determines whether or not implicitly included system schemas such as `pg_catalog` are included in the search path returned.

> **Note:** The search path may be altered at run time. The command is:
>
> `SET search_path TO` *schema* `[,` *schema*`, ...]`

`inet_client_addr` returns the IP address of the current client, and `inet_client_port` returns the port number. `inet_server_addr` returns the IP address on which the server accepted the current connection, and `inet_server_port` returns the port number. All these functions return NULL if the current connection is via a Unix-domain socket.

`pg_my_temp_schema` returns the OID of the current session's temporary schema, or 0 if it has none (because it has not created any temporary tables). `pg_is_other_temp_schema` returns true if the given OID is the OID of any other session's temporary schema. (This can be useful, for example, to exclude other sessions' temporary tables from a catalog display.)

`pg_postmaster_start_time` returns the `timestamp with time zone` when the server started.

`version` returns a string describing the PostgreSQL server's version.

Table 9-41 lists functions that allow the user to query object access privileges programmatically. See Section 5.6 for more information about privileges.

Table 9-41. Access Privilege Inquiry Functions

Name	Return Type	Description
`has_database_privilege(user, database, privilege)`	`boolean`	does user have privilege for database
`has_database_privilege(database, privilege)`	`boolean`	does current user have privilege for database
`has_function_privilege(user, function, privilege)`	`boolean`	does user have privilege for function
`has_function_privilege(function, privilege)`	`boolean`	does current user have privilege for function
`has_language_privilege(user, language, privilege)`	`boolean`	does user have privilege for language
`has_language_privilege(language, privilege)`	`boolean`	does current user have privilege for language
`has_schema_privilege(user, schema, privilege)`	`boolean`	does user have privilege for schema
`has_schema_privilege(schema, privilege)`	`boolean`	does current user have privilege for schema
`has_table_privilege(user, table, privilege)`	`boolean`	does user have privilege for table
`has_table_privilege(table, privilege)`	`boolean`	does current user have privilege for table
`has_tablespace_privilege(user, tablespace, privilege)`	`boolean`	does user have privilege for tablespace
`has_tablespace_privilege(tablespace, privilege)`	`boolean`	does current user have privilege for tablespace
`pg_has_role(user, role, privilege)`	`boolean`	does user have privilege for role

Name	Return Type	Description
pg_has_role(role, privilege)	boolean	does current user have privilege for role

has_database_privilege checks whether a user can access a database in a particular way. The possibilities for its arguments are analogous to has_table_privilege. The desired access privilege type must evaluate to CREATE, CONNECT, TEMPORARY, or TEMP (which is equivalent to TEMPORARY).

has_function_privilege checks whether a user can access a function in a particular way. The possibilities for its arguments are analogous to has_table_privilege. When specifying a function by a text string rather than by OID, the allowed input is the same as for the regprocedure data type (see Section 8.12). The desired access privilege type must evaluate to EXECUTE. An example is:

```
SELECT has_function_privilege('joeuser', 'myfunc(int, text)', 'execute');
```

has_language_privilege checks whether a user can access a procedural language in a particular way. The possibilities for its arguments are analogous to has_table_privilege. The desired access privilege type must evaluate to USAGE.

has_schema_privilege checks whether a user can access a schema in a particular way. The possibilities for its arguments are analogous to has_table_privilege. The desired access privilege type must evaluate to CREATE or USAGE.

has_table_privilege checks whether a user can access a table in a particular way. The user can be specified by name or by OID (pg_authid.oid), or if the argument is omitted current_user is assumed. The table can be specified by name or by OID. (Thus, there are actually six variants of has_table_privilege, which can be distinguished by the number and types of their arguments.) When specifying by name, the name can be schema-qualified if necessary. The desired access privilege type is specified by a text string, which must evaluate to one of the values SELECT, INSERT, UPDATE, DELETE, REFERENCES, or TRIGGER. (Case of the string is not significant, however.) An example is:

```
SELECT has_table_privilege('myschema.mytable', 'select');
```

has_tablespace_privilege checks whether a user can access a tablespace in a particular way. The possibilities for its arguments are analogous to has_table_privilege. The desired access privilege type must evaluate to CREATE.

pg_has_role checks whether a user can access a role in a particular way. The possibilities for its arguments are analogous to has_table_privilege. The desired access privilege type must evaluate to MEMBER or USAGE. MEMBER denotes direct or indirect membership in the role (that is, the right to do SET ROLE), while USAGE denotes whether the privileges of the role are immediately available without doing SET ROLE.

To test whether a user holds a grant option on the privilege, append WITH GRANT OPTION to the privilege key word; for example 'UPDATE WITH GRANT OPTION'.

Table 9-42 shows functions that determine whether a certain object is *visible* in the current schema search path. A table is said to be visible if its containing schema is in the search path and no table of the same name appears earlier in the search path. This is equivalent to the statement that the table can be referenced by name without explicit schema qualification. For example, to list the names of all visible tables:

```
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

Table 9-42. Schema Visibility Inquiry Functions

Name	Return Type	Description
pg_conversion_is_visible(conversion_oid)	boolean	is conversion visible in search path
pg_function_is_visible(function_oid)	boolean	is function visible in search path
pg_operator_is_visible(operator_oid)	boolean	is operator visible in search path
pg_opclass_is_visible(opclass_oid)	boolean	is operator class visible in search path
pg_table_is_visible(table_oid)	boolean	is table visible in search path
pg_type_is_visible(type_oid)	boolean	is type (or domain) visible in search path

pg_conversion_is_visible, pg_function_is_visible, pg_operator_is_visible, pg_opclass_is_visible, pg_table_is_visible, and pg_type_is_visible perform the visibility check for conversions, functions, operators, operator classes, tables, and types. Note that pg_table_is_visible can also be used with views, indexes and sequences; pg_type_is_visible can also be used with domains. For functions and operators, an object in the search path is visible if there is no object of the same name *and argument data type(s)* earlier in the path. For operator classes, both name and associated index access method are considered.

All these functions require object OIDs to identify the object to be checked. If you want to test an object by name, it is convenient to use the OID alias types (regclass, regtype, regprocedure, or regoperator), for example

```
SELECT pg_type_is_visible('myschema.widget'::regtype);
```

Note that it would not make much sense to test an unqualified name in this way — if the name can be recognized at all, it must be visible.

Table 9-43 lists functions that extract information from the system catalogs.

Table 9-43. System Catalog Information Functions

Name	Return Type	Description
format_type(type_oid, typemod)	text	get SQL name of a data type
pg_get_constraintdef(constraint_oid)	text	get definition of a constraint
pg_get_constraintdef(constraint_oid, pretty_bool)	text	get definition of a constraint

Name	Return Type	Description
pg_get_expr(expr_text, relation_oid)	text	decompile internal form of an expression, assuming that any Vars in it refer to the relation indicated by the second parameter
pg_get_expr(expr_text, relation_oid, pretty_bool)	text	decompile internal form of an expression, assuming that any Vars in it refer to the relation indicated by the second parameter
pg_get_indexdef(index_oid)	text	get CREATE INDEX command for index
pg_get_indexdef(index_oid, column_no, pretty_bool)	text	get CREATE INDEX command for index, or definition of just one index column when column_no is not zero
pg_get_ruledef(rule_oid)	text	get CREATE RULE command for rule
pg_get_ruledef(rule_oid, pretty_bool)	text	get CREATE RULE command for rule
pg_get_serial_sequence(table_text, column_name)	text	get name of the sequence that a serial or bigserial column uses
pg_get_triggerdef(trigger_oid)	text	get CREATE [CONSTRAINT] TRIGGER command for trigger
pg_get_userbyid(roleid)	name	get role name with given ID
pg_get_viewdef(view_name)	text	get underlying SELECT command for view (*deprecated*)
pg_get_viewdef(view_name, pretty_bool)	text	get underlying SELECT command for view (*deprecated*)
pg_get_viewdef(view_oid)	text	get underlying SELECT command for view
pg_get_viewdef(view_oid, pretty_bool)	text	get underlying SELECT command for view
pg_tablespace_databases(tablespace_oid)	setof oid	get the set of database OIDs that have objects in the tablespace

format_type returns the SQL name of a data type that is identified by its type OID and possibly a type modifier. Pass NULL for the type modifier if no specific modifier is known.

pg_get_constraintdef, pg_get_indexdef, pg_get_ruledef, and pg_get_triggerdef, respectively reconstruct the creating command for a constraint, index, rule, or trigger. (Note that this is a decompiled reconstruction, not the original text of the command.) pg_get_expr decompiles the internal form of an individual expression, such as the default value for a column. It may be useful when examining the contents of system catalogs. pg_get_viewdef reconstructs the SELECT query that defines a view. Most

of these functions come in two variants, one of which can optionally "pretty-print" the result. The pretty-printed format is more readable, but the default format is more likely to be interpreted the same way by future versions of PostgreSQL; avoid using pretty-printed output for dump purposes. Passing `false` for the pretty-print parameter yields the same result as the variant that does not have the parameter at all.

`pg_get_serial_sequence` returns the name of the sequence associated with a column, or NULL if no sequence is associated with the column. The first input parameter is a table name with optional schema, and the second parameter is a column name. Because the first parameter is potentially a schema and table, it is not treated as a double-quoted identifier, meaning it is lowercased by default, while the second parameter, being just a column name, is treated as double-quoted and has its case preserved. The function returns a value suitably formatted for passing to the sequence functions (see Section 9.12). This association can be modified or removed with `ALTER SEQUENCE OWNED BY`. (The function probably should have been called `pg_get_owned_sequence`; its name reflects the fact that it's typically used with `serial` or `bigserial` columns.)

`pg_get_userbyid` extracts a role's name given its OID.

`pg_tablespace_databases` allows a tablespace to be examined. It returns the set of OIDs of databases that have objects stored in the tablespace. If this function returns any rows, the tablespace is not empty and cannot be dropped. To display the specific objects populating the tablespace, you will need to connect to the databases identified by `pg_tablespace_databases` and query their `pg_class` catalogs.

The functions shown in Table 9-44 extract comments previously stored with the *COMMENT* command. A null value is returned if no comment could be found matching the specified parameters.

Table 9-44. Comment Information Functions

Name	Return Type	Description
`col_description(table_oid, column_number)`	`text`	get comment for a table column
`obj_description(object_oid, catalog_name)`	`text`	get comment for a database object
`obj_description(object_oid)`	`text`	get comment for a database object (*deprecated*)
`shobj_description(object_oid, catalog_name)`	`text`	get comment for a shared database object

`col_description` returns the comment for a table column, which is specified by the OID of its table and its column number. `obj_description` cannot be used for table columns since columns do not have OIDs of their own.

The two-parameter form of `obj_description` returns the comment for a database object specified by its OID and the name of the containing system catalog. For example, `obj_description(123456,'pg_class')` would retrieve the comment for a table with OID 123456. The one-parameter form of `obj_description` requires only the object OID. It is now deprecated since there is no guarantee that OIDs are unique across different system catalogs; therefore, the wrong comment could be returned.

`shobj_description` is used just like `obj_description` only that it is used for retrieving comments on shared objects. Some system catalogs are global to all databases within each cluster and their descriptions are stored globally as well.

9.20. System Administration Functions

Table 9-45 shows the functions available to query and alter run-time configuration parameters.

Table 9-45. Configuration Settings Functions

Name	Return Type	Description
current_setting(setting_name)	text	current value of setting
set_config(setting_name, new_value, is_local)	text	set parameter and return new value

The function current_setting yields the current value of the setting setting_name. It corresponds to the SQL command SHOW. An example:

```
SELECT current_setting('datestyle');

 current_setting
-----------------
 ISO, MDY
(1 row)
```

set_config sets the parameter setting_name to new_value. If is_local is true, the new value will only apply to the current transaction. If you want the new value to apply for the current session, use false instead. The function corresponds to the SQL command SET. An example:

```
SELECT set_config('log_statement_stats', 'off', false);

 set_config
------------
 off
(1 row)
```

The functions shown in Table 9-46 send control signals to other server processes. Use of these functions is restricted to superusers.

Table 9-46. Server Signalling Functions

Name	Return Type	Description
pg_cancel_backend(pid int)	boolean	Cancel a backend's current query
pg_reload_conf()	boolean	Cause server processes to reload their configuration files
pg_rotate_logfile()	boolean	Rotate server's log file

Each of these functions returns true if successful and false otherwise.

pg_cancel_backend sends a query cancel (SIGINT) signal to a backend process identified by process ID. The process ID of an active backend can be found from the procpid column in the pg_stat_activity view, or by listing the postgres processes on the server with ps.

pg_reload_conf sends a SIGHUP signal to the server, causing the configuration files to be reloaded by all server processes.

pg_rotate_logfile signals the log-file manager to switch to a new output file immediately. This works only when redirect_stderr is used for logging, since otherwise there is no log-file manager subprocess.

The functions shown in Table 9-47 assist in making on-line backups. Use of the first three functions is restricted to superusers.

Table 9-47. Backup Control Functions

Name	Return Type	Description
pg_start_backup(label text)	text	Set up for performing on-line backup
pg_stop_backup()	text	Finish performing on-line backup
pg_switch_xlog()	text	Force switch to a new transaction log file
pg_current_xlog_location()	text	Get current transaction log write location
pg_current_xlog_insert_location()	text	Get current transaction log insert location
pg_xlogfile_name_offset(location text)	text, integer	Convert transaction log location string to file name and decimal byte offset within file
pg_xlogfile_name(location text)	text	Convert transaction log location string to file name

pg_start_backup accepts a single parameter which is an arbitrary user-defined label for the backup. (Typically this would be the name under which the backup dump file will be stored.) The function writes a backup label file into the database cluster's data directory, and then returns the backup's starting transaction log location as text. The user need not pay any attention to this result value, but it is provided in case it is of use.

```
postgres=# select pg_start_backup('label_goes_here');
 pg_start_backup
-----------------
 0/D4445B8
(1 row)
```

pg_stop_backup removes the label file created by pg_start_backup, and instead creates a backup history file in the transaction log archive area. The history file includes the label given to pg_start_backup,

the starting and ending transaction log locations for the backup, and the starting and ending times of the backup. The return value is the backup's ending transaction log location (which again may be of little interest). After noting the ending location, the current transaction log insertion point is automatically advanced to the next transaction log file, so that the ending transaction log file can be archived immediately to complete the backup.

`pg_switch_xlog` moves to the next transaction log file, allowing the current file to be archived (assuming you are using continuous archiving). The result is the ending transaction log location within the just-completed transaction log file. If there has been no transaction log activity since the last transaction log switch, `pg_switch_xlog` does nothing and returns the end location of the previous transaction log file.

`pg_current_xlog_location` displays the current transaction log write location in the same format used by the above functions. Similarly `pg_current_xlog_insert_location` displays the current transaction log insertion point. The insertion point is the "logical" end of transaction log at any instant, while the write location is the end of what has actually been written out from the server's internal buffers. The write location is the end of what can be examined from outside the server, and is usually what you want if you are interested in archiving partially-complete transaction log files. The insertion point is made available primarily for server debugging purposes. These are both read-only operations and do not require superuser permissions.

You can use `pg_xlogfile_name_offset` to extract the corresponding transaction log file name and byte offset from the results of any of the above functions. For example:

```
postgres=# select * from pg_xlogfile_name_offset(pg_stop_backup());
         file_name         | file_offset
---------------------------+-------------
 00000001000000000000000D |     4039624
(1 row)
```

Similarly, `pg_xlogfile_name` extracts just the transaction log file name. When the given transction log location is exactly at an transaction log file boundary, both these functions return the name of the preceding transaction log file. This is usually the desired behavior for managing transaction log archiving behavior, since the preceding file is the last one that currently needs to be archived.

For details about proper usage of these functions, see Section 23.3.

The functions shown in Table 9-48 calculate the actual disk space usage of database objects.

Table 9-48. Database Object Size Functions

Name	Return Type	Description
`pg_column_size(any)`	`int`	Number of bytes used to store a particular value (possibly compressed)
`pg_database_size(oid)`	`bigint`	Disk space used by the database with the specified OID
`pg_database_size(name)`	`bigint`	Disk space used by the database with the specified name
`pg_relation_size(oid)`	`bigint`	Disk space used by the table or index with the specified OID

Name	Return Type	Description
pg_relation_size(text)	bigint	Disk space used by the table or index with the specified name. The table name may be qualified with a schema name
pg_size_pretty(bigint)	text	Converts a size in bytes into a human-readable format with size units
pg_tablespace_size(oid)	bigint	Disk space used by the tablespace with the specified OID
pg_tablespace_size(name)	bigint	Disk space used by the tablespace with the specified name
pg_total_relation_size(oid)	bigint	Total disk space used by the table with the specified OID, including indexes and toasted data
pg_total_relation_size(text)	bigint	Total disk space used by the table with the specified name, including indexes and toasted data. The table name may be qualified with a schema name

pg_column_size shows the space used to store any individual data value.

pg_database_size and pg_tablespace_size accept the OID or name of a database or tablespace, and return the total disk space used therein.

pg_relation_size accepts the OID or name of a table, index or toast table, and returns the size in bytes.

pg_size_pretty can be used to format the result of one of the other functions in a human-readable way, using kB, MB, GB or TB as appropriate.

pg_total_relation_size accepts the OID or name of a table or toast table, and returns the size in bytes of the data and all associated indexes and toast tables.

The functions shown in Table 9-49 provide native file access to files on the machine hosting the server. Only files within the database cluster directory and the log_directory may be accessed. Use a relative path for files within the cluster directory, and a path matching the log_directory configuration setting for log files. Use of these functions is restricted to superusers.

Table 9-49. Generic File Access Functions

Name	Return Type	Description
pg_ls_dir(dirname text)	setof text	List the contents of a directory

Name	Return Type	Description
pg_read_file(filename text, offset bigint, length bigint)	text	Return the contents of a text file
pg_stat_file(filename text)	record	Return information about a file

pg_ls_dir returns all the names in the specified directory, except the special entries "." and "..".

pg_read_file returns part of a text file, starting at the given offset, returning at most length bytes (less if the end of file is reached first). If offset is negative, it is relative to the end of the file.

pg_stat_file returns a record containing the file size, last accessed time stamp, last modified time stamp, last file status change time stamp (Unix platforms only), file creation time stamp (Windows only), and a boolean indicating if it is a directory. Typical usages include:

```
SELECT * FROM pg_stat_file('filename');
SELECT (pg_stat_file('filename')).modification;
```

The functions shown in Table 9-50 manage advisory locks. For details about proper usage of these functions, see Section 12.3.4.

Table 9-50. Advisory Lock Functions

Name	Return Type	Description
pg_advisory_lock(key bigint)	void	Obtain exclusive advisory lock
pg_advisory_lock(key1 int, key2 int)	void	Obtain exclusive advisory lock
pg_advisory_lock_shared(key bigint)	void	Obtain shared advisory lock
pg_advisory_lock_shared(key1 int, key2 int)	void	Obtain shared advisory lock
pg_try_advisory_lock(key bigint)	boolean	Obtain exclusive advisory lock if available
pg_try_advisory_lock(key1 int, key2 int)	boolean	Obtain exclusive advisory lock if available
pg_try_advisory_lock_shared(key bigint)	boolean	Obtain shared advisory lock if available
pg_try_advisory_lock_shared(key1 int, key2 int)	boolean	Obtain shared advisory lock if available

Name	Return Type	Description
pg_advisory_unlock(key bigint)	boolean	Release an exclusive advisory lock
pg_advisory_unlock(key1 int, key2 int)	boolean	Release an exclusive advisory lock
pg_advisory_unlock_shared(key bigint)	boolean	Release a shared advisory lock
pg_advisory_unlock_shared(key1 int, key2 int)	boolean	Release a shared advisory lock
pg_advisory_unlock_all()	void	Release all advisory locks held by the current session

pg_advisory_lock locks an application-defined resource, which may be identified either by a single 64-bit key value or two 32-bit key values (note that these two key spaces do not overlap). If another session already holds a lock on the same resource, the function will wait until the resource becomes available. The lock is exclusive. Multiple lock requests stack, so that if the same resource is locked three times it must be also unlocked three times to be released for other sessions' use.

pg_advisory_lock_shared works the same as pg_advisory_lock, except the lock can be shared with other sessions requesting shared locks. Only would-be exclusive lockers are locked out.

pg_try_advisory_lock is similar to pg_advisory_lock, except the function will not wait for the lock to become available. It will either obtain the lock immediately and return true, or return false if the lock cannot be acquired now.

pg_try_advisory_lock_shared works the same as pg_try_advisory_lock, except it attempts to acquire shared rather than exclusive lock.

pg_advisory_unlock will release a previously-acquired exclusive advisory lock. It will return true if the lock is successfully released. If the lock was in fact not held, it will return false, and in addition, an SQL warning will be raised by the server.

pg_advisory_unlock_shared works the same as pg_advisory_unlock, except to release a shared advisory lock.

pg_advisory_unlock_all will release all advisory locks held by the current session. (This function is implicitly invoked at session end, even if the client disconnects ungracefully.)

Chapter 10. Type Conversion

SQL statements can, intentionally or not, require mixing of different data types in the same expression. PostgreSQL has extensive facilities for evaluating mixed-type expressions.

In many cases a user will not need to understand the details of the type conversion mechanism. However, the implicit conversions done by PostgreSQL can affect the results of a query. When necessary, these results can be tailored by using *explicit* type conversion.

This chapter introduces the PostgreSQL type conversion mechanisms and conventions. Refer to the relevant sections in Chapter 8 and Chapter 9 for more information on specific data types and allowed functions and operators.

10.1. Overview

SQL is a strongly typed language. That is, every data item has an associated data type which determines its behavior and allowed usage. PostgreSQL has an extensible type system that is much more general and flexible than other SQL implementations. Hence, most type conversion behavior in PostgreSQL is governed by general rules rather than by *ad hoc* heuristics. This allows mixed-type expressions to be meaningful even with user-defined types.

The PostgreSQL scanner/parser divides lexical elements into only five fundamental categories: integers, non-integer numbers, strings, identifiers, and key words. Constants of most non-numeric types are first classified as strings. The SQL language definition allows specifying type names with strings, and this mechanism can be used in PostgreSQL to start the parser down the correct path. For example, the query

```
SELECT text 'Origin' AS "label", point '(0,0)' AS "value";

 label  | value
--------+-------
 Origin | (0,0)
(1 row)
```

has two literal constants, of type `text` and `point`. If a type is not specified for a string literal, then the placeholder type `unknown` is assigned initially, to be resolved in later stages as described below.

There are four fundamental SQL constructs requiring distinct type conversion rules in the PostgreSQL parser:

Function calls

Much of the PostgreSQL type system is built around a rich set of functions. Functions can have one or more arguments. Since PostgreSQL permits function overloading, the function name alone does not uniquely identify the function to be called; the parser must select the right function based on the data types of the supplied arguments.

Operators

PostgreSQL allows expressions with prefix and postfix unary (one-argument) operators, as well as binary (two-argument) operators. Like functions, operators can be overloaded, and so the same problem of selecting the right operator exists.

Value Storage

SQL `INSERT` and `UPDATE` statements place the results of expressions into a table. The expressions in the statement must be matched up with, and perhaps converted to, the types of the target columns.

`UNION`, `CASE`, and related constructs

Since all query results from a unionized `SELECT` statement must appear in a single set of columns, the types of the results of each `SELECT` clause must be matched up and converted to a uniform set. Similarly, the result expressions of a `CASE` construct must be converted to a common type so that the `CASE` expression as a whole has a known output type. The same holds for `ARRAY` constructs, and for the `GREATEST` and `LEAST` functions.

The system catalogs store information about which conversions, called *casts*, between data types are valid, and how to perform those conversions. Additional casts can be added by the user with the `CREATE CAST` command. (This is usually done in conjunction with defining new data types. The set of casts between the built-in types has been carefully crafted and is best not altered.)

An additional heuristic is provided in the parser to allow better guesses at proper behavior for SQL standard types. There are several basic *type categories* defined: `boolean`, `numeric`, `string`, `bitstring`, `datetime`, `timespan`, `geometric`, `network`, and user-defined. Each category, with the exception of user-defined, has one or more *preferred types* which are preferentially selected when there is ambiguity. In the user-defined category, each type is its own preferred type. Ambiguous expressions (those with multiple candidate parsing solutions) can therefore often be resolved when there are multiple possible built-in types, but they will raise an error when there are multiple choices for user-defined types.

All type conversion rules are designed with several principles in mind:

- Implicit conversions should never have surprising or unpredictable outcomes.
- User-defined types, of which the parser has no *a priori* knowledge, should be "higher" in the type hierarchy. In mixed-type expressions, native types shall always be converted to a user-defined type (of course, only if conversion is necessary).
- User-defined types are not related. Currently, PostgreSQL does not have information available to it on relationships between types, other than hardcoded heuristics for built-in types and implicit relationships based on available functions and casts.
- There should be no extra overhead from the parser or executor if a query does not need implicit type conversion. That is, if a query is well formulated and the types already match up, then the query should proceed without spending extra time in the parser and without introducing unnecessary implicit conversion calls into the query.

  Additionally, if a query usually requires an implicit conversion for a function, and if then the user defines a new function with the correct argument types, the parser should use this new function and will no longer do the implicit conversion using the old function.

10.2. Operators

The specific operator to be used in an operator invocation is determined by following the procedure below. Note that this procedure is indirectly affected by the precedence of the involved operators. See Section 4.1.6 for more information.

Operator Type Resolution

1. Select the operators to be considered from the `pg_operator` system catalog. If an unqualified operator name was used (the usual case), the operators considered are those of the right name and argument count that are visible in the current search path (see Section 5.7.3). If a qualified operator name was given, only operators in the specified schema are considered.

    a. If the search path finds multiple operators of identical argument types, only the one appearing earliest in the path is considered. But operators of different argument types are considered on an equal footing regardless of search path position.

2. Check for an operator accepting exactly the input argument types. If one exists (there can be only one exact match in the set of operators considered), use it.

    a. If one argument of a binary operator invocation is of the `unknown` type, then assume it is the same type as the other argument for this check. Other cases involving `unknown` will never find a match at this step.

3. Look for the best match.

    a. Discard candidate operators for which the input types do not match and cannot be converted (using an implicit conversion) to match. `unknown` literals are assumed to be convertible to anything for this purpose. If only one candidate remains, use it; else continue to the next step.

    b. Run through all candidates and keep those with the most exact matches on input types. (Domains are considered the same as their base type for this purpose.) Keep all candidates if none have any exact matches. If only one candidate remains, use it; else continue to the next step.

    c. Run through all candidates and keep those that accept preferred types (of the input data type's type category) at the most positions where type conversion will be required. Keep all candidates if none accept preferred types. If only one candidate remains, use it; else continue to the next step.

    d. If any input arguments are `unknown`, check the type categories accepted at those argument positions by the remaining candidates. At each position, select the `string` category if any candidate accepts that category. (This bias towards string is appropriate since an unknown-type literal does look like a string.) Otherwise, if all the remaining candidates accept the same type category, select that category; otherwise fail because the correct choice cannot be deduced without more clues. Now discard candidates that do not accept the selected type category. Furthermore, if any candidate accepts a preferred type at a given argument position, discard candidates that accept non-preferred types for that argument.

    e. If only one candidate remains, use it. If no candidate or more than one candidate remains, then fail.

Some examples follow.

Example 10-1. Exponentiation Operator Type Resolution

There is only one exponentiation operator defined in the catalog, and it takes arguments of type `double precision`. The scanner assigns an initial type of `integer` to both arguments of this query expression:

```
SELECT 2 ^ 3 AS "exp";

 exp
-----
   8
(1 row)
```

So the parser does a type conversion on both operands and the query is equivalent to

```
SELECT CAST(2 AS double precision) ^ CAST(3 AS double precision) AS "exp";
```

Example 10-2. String Concatenation Operator Type Resolution

A string-like syntax is used for working with string types as well as for working with complex extension types. Strings with unspecified type are matched with likely operator candidates.

An example with one unspecified argument:

```
SELECT text 'abc' || 'def' AS "text and unknown";

 text and unknown
------------------
 abcdef
(1 row)
```

In this case the parser looks to see if there is an operator taking `text` for both arguments. Since there is, it assumes that the second argument should be interpreted as of type `text`.

Here is a concatenation on unspecified types:

```
SELECT 'abc' || 'def' AS "unspecified";

 unspecified
-------------
 abcdef
(1 row)
```

In this case there is no initial hint for which type to use, since no types are specified in the query. So, the parser looks for all candidate operators and finds that there are candidates accepting both string-category and bit-string-category inputs. Since string category is preferred when available, that category is selected, and then the preferred type for strings, `text`, is used as the specific type to resolve the unknown literals to.

Example 10-3. Absolute-Value and Negation Operator Type Resolution

The PostgreSQL operator catalog has several entries for the prefix operator `@`, all of which implement absolute-value operations for various numeric data types. One of these entries is for type `float8`, which is the preferred type in the numeric category. Therefore, PostgreSQL will use that entry when faced with a non-numeric input:

```
SELECT @ '-4.5' AS "abs";
 abs
-----
 4.5
(1 row)
```

Here the system has performed an implicit conversion from `text` to `float8` before applying the chosen operator. We can verify that `float8` and not some other type was used:

```
SELECT @ '-4.5e500' AS "abs";

ERROR:  "-4.5e500" is out of range for type double precision
```

On the other hand, the prefix operator ~ (bitwise negation) is defined only for integer data types, not for `float8`. So, if we try a similar case with ~, we get:

```
SELECT ~ '20' AS "negation";

ERROR:  operator is not unique: ~ "unknown"
HINT:  Could not choose a best candidate operator. You may need to add explicit
type casts.
```

This happens because the system can't decide which of the several possible ~ operators should be preferred. We can help it out with an explicit cast:

```
SELECT ~ CAST('20' AS int8) AS "negation";

 negation
----------
      -21
(1 row)
```

10.3. Functions

The specific function to be used in a function invocation is determined according to the following steps.

Function Type Resolution

1. Select the functions to be considered from the `pg_proc` system catalog. If an unqualified function name was used, the functions considered are those of the right name and argument count that are visible in the current search path (see Section 5.7.3). If a qualified function name was given, only functions in the specified schema are considered.

    a. If the search path finds multiple functions of identical argument types, only the one appearing earliest in the path is considered. But functions of different argument types are considered on an equal footing regardless of search path position.

2. Check for a function accepting exactly the input argument types. If one exists (there can be only one exact match in the set of functions considered), use it. (Cases involving `unknown` will never find a match at this step.)
3. If no exact match is found, see whether the function call appears to be a trivial type conversion request. This happens if the function call has just one argument and the function name is the same as the (internal) name of some data type. Furthermore, the function argument must be either an unknown-type literal or a type that is binary-compatible with the named data type. When these conditions are met, the function argument is converted to the named data type without any actual function call.
4. Look for the best match.

    a. Discard candidate functions for which the input types do not match and cannot be converted (using an implicit conversion) to match. `unknown` literals are assumed to be convertible to anything for this purpose. If only one candidate remains, use it; else continue to the next step.

    b. Run through all candidates and keep those with the most exact matches on input types. (Domains are considered the same as their base type for this purpose.) Keep all candidates if none have any exact matches. If only one candidate remains, use it; else continue to the next step.

    c. Run through all candidates and keep those that accept preferred types (of the input data type's type category) at the most positions where type conversion will be required. Keep all candidates if none accept preferred types. If only one candidate remains, use it; else continue to the next step.

    d. If any input arguments are `unknown`, check the type categories accepted at those argument positions by the remaining candidates. At each position, select the `string` category if any candidate accepts that category. (This bias towards string is appropriate since an unknown-type literal does look like a string.) Otherwise, if all the remaining candidates accept the same type category, select that category; otherwise fail because the correct choice cannot be deduced without more clues. Now discard candidates that do not accept the selected type category. Furthermore, if any candidate accepts a preferred type at a given argument position, discard candidates that accept non-preferred types for that argument.

    e. If only one candidate remains, use it. If no candidate or more than one candidate remains, then fail.

Note that the "best match" rules are identical for operator and function type resolution. Some examples follow.

Example 10-4. Rounding Function Argument Type Resolution

There is only one `round` function with two arguments. (The first is `numeric`, the second is `integer`.) So the following query automatically converts the first argument of type `integer` to `numeric`:

```
SELECT round(4, 4);

 round
--------
 4.0000
(1 row)
```

That query is actually transformed by the parser to
```
SELECT round(CAST (4 AS numeric), 4);
```

Since numeric constants with decimal points are initially assigned the type `numeric`, the following query will require no type conversion and may therefore be slightly more efficient:

```
SELECT round(4.0, 4);
```

Example 10-5. Substring Function Type Resolution

There are several `substr` functions, one of which takes types `text` and `integer`. If called with a string constant of unspecified type, the system chooses the candidate function that accepts an argument of the preferred category `string` (namely of type `text`).

```
SELECT substr('1234', 3);

 substr
--------
     34
(1 row)
```

If the string is declared to be of type `varchar`, as might be the case if it comes from a table, then the parser will try to convert it to become `text`:

```
SELECT substr(varchar '1234', 3);

 substr
--------
     34
(1 row)
```

This is transformed by the parser to effectively become
```
SELECT substr(CAST (varchar '1234' AS text), 3);
```

> **Note:** The parser learns from the `pg_cast` catalog that `text` and `varchar` are binary-compatible, meaning that one can be passed to a function that accepts the other without doing any physical conversion. Therefore, no explicit type conversion call is really inserted in this case.

And, if the function is called with an argument of type `integer`, the parser will try to convert that to `text`:

```
SELECT substr(1234, 3);

 substr
--------
     34
(1 row)
```

This actually executes as
```
SELECT substr(CAST (1234 AS text), 3);
```

This automatic transformation can succeed because there is an implicitly invocable cast from `integer` to `text`.

10.4. Value Storage

Values to be inserted into a table are converted to the destination column's data type according to the following steps.

Value Storage Type Conversion

1. Check for an exact match with the target.
2. Otherwise, try to convert the expression to the target type. This will succeed if there is a registered cast between the two types. If the expression is an unknown-type literal, the contents of the literal string will be fed to the input conversion routine for the target type.
3. Check to see if there is a sizing cast for the target type. A sizing cast is a cast from that type to itself. If one is found in the `pg_cast` catalog, apply it to the expression before storing into the destination column. The implementation function for such a cast always takes an extra parameter of type `integer`, which receives the destination column's declared length (actually, its `atttypmod` value; the interpretation of `atttypmod` varies for different data types). The cast function is responsible for applying any length-dependent semantics such as size checking or truncation.

Example 10-6. `character` Storage Type Conversion

For a target column declared as `character(20)` the following statement ensures that the stored value is sized correctly:

```
CREATE TABLE vv (v character(20));
INSERT INTO vv SELECT 'abc' || 'def';
SELECT v, length(v) FROM vv;

          v           | length
----------------------+--------
 abcdef               |     20
(1 row)
```

What has really happened here is that the two unknown literals are resolved to `text` by default, allowing the `||` operator to be resolved as `text` concatenation. Then the `text` result of the operator is converted to `bpchar` ("blank-padded char", the internal name of the `character` data type) to match the target column type. (Since the types `text` and `bpchar` are binary-compatible, this conversion does not insert any real function call.) Finally, the sizing function `bpchar(bpchar, integer)` is found in the system catalog and applied to the operator's result and the stored column length. This type-specific function performs the required length check and addition of padding spaces.

10.5. UNION, CASE, and Related Constructs

SQL UNION constructs must match up possibly dissimilar types to become a single result set. The resolution algorithm is applied separately to each output column of a union query. The INTERSECT and EXCEPT constructs resolve dissimilar types in the same way as UNION. The CASE, ARRAY, VALUES, GREATEST and LEAST constructs use the identical algorithm to match up their component expressions and select a result data type.

Type Resolution for UNION, CASE, and Related Constructs

1. If all inputs are of type unknown, resolve as type text (the preferred type of the string category). Otherwise, ignore the unknown inputs while choosing the result type.
2. If the non-unknown inputs are not all of the same type category, fail.
3. Choose the first non-unknown input type which is a preferred type in that category or allows all the non-unknown inputs to be implicitly converted to it.
4. Convert all inputs to the selected type.

Some examples follow.

Example 10-7. Type Resolution with Underspecified Types in a Union

```
SELECT text 'a' AS "text" UNION SELECT 'b';

 text
------
 a
 b
(2 rows)
```

Here, the unknown-type literal 'b' will be resolved as type text.

Example 10-8. Type Resolution in a Simple Union

```
SELECT 1.2 AS "numeric" UNION SELECT 1;

 numeric
---------
       1
     1.2
(2 rows)
```

The literal 1.2 is of type numeric, and the integer value 1 can be cast implicitly to numeric, so that type is used.

Example 10-9. Type Resolution in a Transposed Union

```
SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);

 real
------
    1
  2.2
(2 rows)
```

Here, since type `real` cannot be implicitly cast to `integer`, but `integer` can be implicitly cast to `real`, the union result type is resolved as `real`.

Chapter 11. Indexes

Indexes are a common way to enhance database performance. An index allows the database server to find and retrieve specific rows much faster than it could do without an index. But indexes also add overhead to the database system as a whole, so they should be used sensibly.

11.1. Introduction

Suppose we have a table similar to this:

```
CREATE TABLE test1 (
    id integer,
    content varchar
);
```

and the application requires a lot of queries of the form

```
SELECT content FROM test1 WHERE id = constant;
```

With no advance preparation, the system would have to scan the entire `test1` table, row by row, to find all matching entries. If there are a lot of rows in `test1` and only a few rows (perhaps only zero or one) that would be returned by such a query, then this is clearly an inefficient method. But if the system has been instructed to maintain an index on the `id` column, then it can use a more efficient method for locating matching rows. For instance, it might only have to walk a few levels deep into a search tree.

A similar approach is used in most books of non-fiction: terms and concepts that are frequently looked up by readers are collected in an alphabetic index at the end of the book. The interested reader can scan the index relatively quickly and flip to the appropriate page(s), rather than having to read the entire book to find the material of interest. Just as it is the task of the author to anticipate the items that the readers are likely to look up, it is the task of the database programmer to foresee which indexes will be of advantage.

The following command would be used to create the index on the `id` column, as discussed:

```
CREATE INDEX test1_id_index ON test1 (id);
```

The name `test1_id_index` can be chosen freely, but you should pick something that enables you to remember later what the index was for.

To remove an index, use the `DROP INDEX` command. Indexes can be added to and removed from tables at any time.

Once an index is created, no further intervention is required: the system will update the index when the table is modified, and it will use the index in queries when it thinks this would be more efficient than a sequential table scan. But you may have to run the `ANALYZE` command regularly to update statistics to allow the query planner to make educated decisions. See Chapter 13 for information about how to find out whether an index is used and when and why the planner may choose *not* to use an index.

Indexes can also benefit `UPDATE` and `DELETE` commands with search conditions. Indexes can moreover be used in join searches. Thus, an index defined on a column that is part of a join condition can significantly speed up queries with joins.

Creating an index on a large table can take a long time. By default, PostgreSQL allows reads (selects) to occur on the table in parallel with creation of an index, but writes (inserts, updates, deletes) are blocked until the index build is finished. In production environments this is often unacceptable. It is possible to allow writes to occur in parallel with index creation, but there are several caveats to be aware of — for more information see *Building Indexes Concurrently*.

After an index is created, the system has to keep it synchronized with the table. This adds overhead to data manipulation operations. Therefore indexes that are seldom or never used in queries should be removed.

11.2. Index Types

PostgreSQL provides several index types: B-tree, Hash, GiST and GIN. Each index type uses a different algorithm that is best suited to different types of queries. By default, the `CREATE INDEX` command will create a B-tree index, which fits the most common situations.

B-trees can handle equality and range queries on data that can be sorted into some ordering. In particular, the PostgreSQL query planner will consider using a B-tree index whenever an indexed column is involved in a comparison using one of these operators:

```
<
<=
=
>=
>
```

Constructs equivalent to combinations of these operators, such as `BETWEEN` and `IN`, can also be implemented with a B-tree index search. (But note that `IS NULL` is not equivalent to = and is not indexable.)

The optimizer can also use a B-tree index for queries involving the pattern matching operators `LIKE` and `~` *if* the pattern is a constant and is anchored to the beginning of the string — for example, `col LIKE 'foo%'` or `col ~ '^foo'`, but not `col LIKE '%bar'`. However, if your server does not use the C locale you will need to create the index with a special operator class to support indexing of pattern-matching queries. See Section 11.8 below. It is also possible to use B-tree indexes for `ILIKE` and `~*`, but only if the pattern starts with non-alphabetic characters, i.e. characters that are not affected by upper/lower case conversion.

Hash indexes can only handle simple equality comparisons. The query planner will consider using a hash index whenever an indexed column is involved in a comparison using the = operator. The following command is used to create a hash index:

```
CREATE INDEX name ON table USING hash (column);
```

> **Note:** Testing has shown PostgreSQL's hash indexes to perform no better than B-tree indexes, and the index size and build time for hash indexes is much worse. Furthermore, hash index operations are not presently WAL-logged, so hash indexes may need to be rebuilt with `REINDEX` after a database crash. For these reasons, hash index use is presently discouraged.

GiST indexes are not a single kind of index, but rather an infrastructure within which many different indexing strategies can be implemented. Accordingly, the particular operators with which a GiST index can be used vary depending on the indexing strategy (the *operator class*). As an example, the standard distribution of PostgreSQL includes GiST operator classes for several two-dimensional geometric data types, which support indexed queries using these operators:

```
<<
&<
&>
>>
<<|
&<|
|&>
|>>
@>
<@
~=
&&
```

(See Section 9.10 for the meaning of these operators.) Many other GiST operator classes are available in the `contrib` collection or as separate projects. For more information see Chapter 50.

GIN indexes are inverted indexes which can handle values that contain more than one key, arrays for example. Like GiST, GIN can support many different user-defined indexing strategies and the particular operators with which a GIN index can be used vary depending on the indexing strategy. As an example, the standard distribution of PostgreSQL includes GIN operator classes for one-dimensional arrays, which support indexed queries using these operators:

```
<@
@>
=
&&
```

(See Section 9.14 for the meaning of these operators.) Other GIN operator classes are available in the `contrib` `tsearch2` and `intarray` modules. For more information see Chapter 51.

11.3. Multicolumn Indexes

An index can be defined on more than one column of a table. For example, if you have a table of this form:

```
CREATE TABLE test2 (
  major int,
  minor int,
  name varchar
);
```

(say, you keep your `/dev` directory in a database...) and you frequently make queries like

```
SELECT name FROM test2 WHERE major = constant AND minor = constant;
```

then it may be appropriate to define an index on the columns `major` and `minor` together, e.g.,

```
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```

Currently, only the B-tree and GiST index types support multicolumn indexes. Up to 32 columns may be specified. (This limit can be altered when building PostgreSQL; see the file `pg_config_manual.h`.)

A multicolumn B-tree index can be used with query conditions that involve any subset of the index's columns, but the index is most efficient when there are constraints on the leading (leftmost) columns. The exact rule is that equality constraints on leading columns, plus any inequality constraints on the first column that does not have an equality constraint, will be used to limit the portion of the index that is scanned. Constraints on columns to the right of these columns are checked in the index, so they save visits to the table proper, but they do not reduce the portion of the index that has to be scanned. For example, given an index on `(a, b, c)` and a query condition `WHERE a = 5 AND b >= 42 AND c < 77`, the index would have to be scanned from the first entry with `a` = 5 and `b` = 42 up through the last entry with `a` = 5. Index entries with `c` >= 77 would be skipped, but they'd still have to be scanned through. This index could in principle be used for queries that have constraints on `b` and/or `c` with no constraint on `a` — but the entire index would have to be scanned, so in most cases the planner would prefer a sequential table scan over using the index.

A multicolumn GiST index can be used with query conditions that involve any subset of the index's columns. Conditions on additional columns restrict the entries returned by the index, but the condition on the first column is the most important one for determining how much of the index needs to be scanned. A GiST index will be relatively ineffective if its first column has only a few distinct values, even if there are many distinct values in additional columns.

Of course, each column must be used with operators appropriate to the index type; clauses that involve other operators will not be considered.

Multicolumn indexes should be used sparingly. In most situations, an index on a single column is sufficient and saves space and time. Indexes with more than three columns are unlikely to be helpful unless the usage of the table is extremely stylized. See also Section 11.4 for some discussion of the merits of different index setups.

11.4. Combining Multiple Indexes

A single index scan can only use query clauses that use the index's columns with operators of its operator class and are joined with `AND`. For example, given an index on `(a, b)` a query condition like `WHERE a = 5 AND b = 6` could use the index, but a query like `WHERE a = 5 OR b = 6` could not directly use the index.

Beginning in release 8.1, PostgreSQL has the ability to combine multiple indexes (including multiple uses of the same index) to handle cases that cannot be implemented by single index scans. The system can form `AND` and `OR` conditions across several index scans. For example, a query like `WHERE x = 42 OR x = 47 OR x = 53 OR x = 99` could be broken down into four separate scans of an index on `x`, each scan using one of the query clauses. The results of these scans are then ORed together to produce the result. Another example is that if we have separate indexes on `x` and `y`, one possible implementation of

a query like `WHERE x = 5 AND y = 6` is to use each index with the appropriate query clause and then AND together the index results to identify the result rows.

To combine multiple indexes, the system scans each needed index and prepares a *bitmap* in memory giving the locations of table rows that are reported as matching that index's conditions. The bitmaps are then ANDed and ORed together as needed by the query. Finally, the actual table rows are visited and returned. The table rows are visited in physical order, because that is how the bitmap is laid out; this means that any ordering of the original indexes is lost, and so a separate sort step will be needed if the query has an `ORDER BY` clause. For this reason, and because each additional index scan adds extra time, the planner will sometimes choose to use a simple index scan even though additional indexes are available that could have been used as well.

In all but the simplest applications, there are various combinations of indexes that may be useful, and the database developer must make trade-offs to decide which indexes to provide. Sometimes multicolumn indexes are best, but sometimes it's better to create separate indexes and rely on the index-combination feature. For example, if your workload includes a mix of queries that sometimes involve only column `x`, sometimes only column `y`, and sometimes both columns, you might choose to create two separate indexes on `x` and `y`, relying on index combination to process the queries that use both columns. You could also create a multicolumn index on `(x, y)`. This index would typically be more efficient than index combination for queries involving both columns, but as discussed in Section 11.3, it would be almost useless for queries involving only `y`, so it could not be the only index. A combination of the multicolumn index and a separate index on `y` would serve reasonably well. For queries involving only `x`, the multicolumn index could be used, though it would be larger and hence slower than an index on `x` alone. The last alternative is to create all three indexes, but this is probably only reasonable if the table is searched much more often than it is updated and all three types of query are common. If one of the types of query is much less common than the others, you'd probably settle for creating just the two indexes that best match the common types.

11.5. Unique Indexes

Indexes may also be used to enforce uniqueness of a column's value, or the uniqueness of the combined values of more than one column.

```
CREATE UNIQUE INDEX name ON table (column [, ...]);
```

Currently, only B-tree indexes can be declared unique.

When an index is declared unique, multiple table rows with equal indexed values will not be allowed. Null values are not considered equal. A multicolumn unique index will only reject cases where all of the indexed columns are equal in two rows.

PostgreSQL automatically creates a unique index when a unique constraint or a primary key is defined for a table. The index covers the columns that make up the primary key or unique columns (a multicolumn index, if appropriate), and is the mechanism that enforces the constraint.

> **Note:** The preferred way to add a unique constraint to a table is `ALTER TABLE ... ADD CONSTRAINT`. The use of indexes to enforce unique constraints could be considered an implementation detail that should not be accessed directly. One should, however, be aware that there's no need to manually create indexes on unique columns; doing so would just duplicate the automatically-created index.

11.6. Indexes on Expressions

An index column need not be just a column of the underlying table, but can be a function or scalar expression computed from one or more columns of the table. This feature is useful to obtain fast access to tables based on the results of computations.

For example, a common way to do case-insensitive comparisons is to use the `lower` function:

```
SELECT * FROM test1 WHERE lower(col1) = 'value';
```

This query can use an index, if one has been defined on the result of the `lower(col1)` operation:

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

If we were to declare this index `UNIQUE`, it would prevent creation of rows whose `col1` values differ only in case, as well as rows whose `col1` values are actually identical. Thus, indexes on expressions can be used to enforce constraints that are not definable as simple unique constraints.

As another example, if one often does queries like this:

```
SELECT * FROM people WHERE (first_name || ' ' || last_name) = 'John Smith';
```

then it might be worth creating an index like this:

```
CREATE INDEX people_names ON people ((first_name || ' ' || last_name));
```

The syntax of the `CREATE INDEX` command normally requires writing parentheses around index expressions, as shown in the second example. The parentheses may be omitted when the expression is just a function call, as in the first example.

Index expressions are relatively expensive to maintain, because the derived expression(s) must be computed for each row upon insertion and whenever it is updated. However, the index expressions are *not* recomputed during an indexed search, since they are already stored in the index. In both examples above, the system sees the query as just `WHERE indexedcolumn = 'constant'` and so the speed of the search is equivalent to any other simple index query. Thus, indexes on expressions are useful when retrieval speed is more important than insertion and update speed.

11.7. Partial Indexes

A *partial index* is an index built over a subset of a table; the subset is defined by a conditional expression (called the *predicate* of the partial index). The index contains entries for only those table rows that satisfy the predicate. Partial indexes are a specialized feature, but there are several situations in which they are useful.

One major reason for using a partial index is to avoid indexing common values. Since a query searching for a common value (one that accounts for more than a few percent of all the table rows) will not use the index anyway, there is no point in keeping those rows in the index at all. This reduces the size of the index, which will speed up queries that do use the index. It will also speed up many table update operations because the index does not need to be updated in all cases. Example 11-1 shows a possible application of this idea.

Example 11-1. Setting up a Partial Index to Exclude Common Values

Suppose you are storing web server access logs in a database. Most accesses originate from the IP address range of your organization but some are from elsewhere (say, employees on dial-up connections). If your searches by IP are primarily for outside accesses, you probably do not need to index the IP range that corresponds to your organization's subnet.

Assume a table like this:

```
CREATE TABLE access_log (
    url varchar,
    client_ip inet,
    ...
);
```

To create a partial index that suits our example, use a command such as this:

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
    WHERE NOT (client_ip > inet '192.168.100.0' AND client_ip < inet '192.168.100.255');
```

A typical query that can use this index would be:

```
SELECT * FROM access_log WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

A query that cannot use this index is:

```
SELECT * FROM access_log WHERE client_ip = inet '192.168.100.23';
```

Observe that this kind of partial index requires that the common values be predetermined. If the distribution of values is inherent (due to the nature of the application) and static (not changing over time), this is not difficult, but if the common values are merely due to the coincidental data load this can require a lot of maintenance work to change the index definition from time to time.

Another possible use for a partial index is to exclude values from the index that the typical query workload is not interested in; this is shown in Example 11-2. This results in the same advantages as listed above, but it prevents the "uninteresting" values from being accessed via that index at all, even if an index scan might be profitable in that case. Obviously, setting up partial indexes for this kind of scenario will require a lot of care and experimentation.

Example 11-2. Setting up a Partial Index to Exclude Uninteresting Values

If you have a table that contains both billed and unbilled orders, where the unbilled orders take up a small fraction of the total table and yet those are the most-accessed rows, you can improve performance by creating an index on just the unbilled rows. The command to create the index would look like this:

```
CREATE INDEX orders_unbilled_index ON orders (order_nr)
    WHERE billed is not true;
```

A possible query to use this index would be

```
SELECT * FROM orders WHERE billed is not true AND order_nr < 10000;
```

However, the index can also be used in queries that do not involve `order_nr` at all, e.g.,

```
SELECT * FROM orders WHERE billed is not true AND amount > 5000.00;
```

This is not as efficient as a partial index on the `amount` column would be, since the system has to scan the entire index. Yet, if there are relatively few unbilled orders, using this partial index just to find the unbilled orders could be a win.

Note that this query cannot use this index:

```
SELECT * FROM orders WHERE order_nr = 3501;
```

The order 3501 may be among the billed or among the unbilled orders.

Example 11-2 also illustrates that the indexed column and the column used in the predicate do not need to match. PostgreSQL supports partial indexes with arbitrary predicates, so long as only columns of the table being indexed are involved. However, keep in mind that the predicate must match the conditions used in the queries that are supposed to benefit from the index. To be precise, a partial index can be used in a query only if the system can recognize that the `WHERE` condition of the query mathematically implies the predicate of the index. PostgreSQL does not have a sophisticated theorem prover that can recognize mathematically equivalent expressions that are written in different forms. (Not only is such a general theorem prover extremely difficult to create, it would probably be too slow to be of any real use.) The system can recognize simple inequality implications, for example "x < 1" implies "x < 2"; otherwise the predicate condition must exactly match part of the query's `WHERE` condition or the index will not be recognized to be usable. Matching takes place at query planning time, not at run time. As a result, parameterized query clauses will not work with a partial index. For example a prepared query with a parameter might specify "x < ?" which will never imply "x < 2" for all possible values of the parameter.

A third possible use for partial indexes does not require the index to be used in queries at all. The idea here is to create a unique index over a subset of a table, as in Example 11-3. This enforces uniqueness among the rows that satisfy the index predicate, without constraining those that do not.

Example 11-3. Setting up a Partial Unique Index

Suppose that we have a table describing test outcomes. We wish to ensure that there is only one "successful" entry for a given subject and target combination, but there might be any number of "unsuccessful" entries. Here is one way to do it:

```
CREATE TABLE tests (
    subject text,
    target text,
    success boolean,
    ...
);

CREATE UNIQUE INDEX tests_success_constraint ON tests (subject, target)
    WHERE success;
```

This is a particularly efficient way of doing it when there are few successful tests and many unsuccessful ones.

Finally, a partial index can also be used to override the system's query plan choices. It may occur that data sets with peculiar distributions will cause the system to use an index when it really should not. In that case the index can be set up so that it is not available for the offending query. Normally, PostgreSQL makes reasonable choices about index usage (e.g., it avoids them when retrieving common values, so the earlier example really only saves index size, it is not required to avoid index usage), and grossly incorrect plan choices are cause for a bug report.

Keep in mind that setting up a partial index indicates that you know at least as much as the query planner knows, in particular you know when an index might be profitable. Forming this knowledge requires experience and understanding of how indexes in PostgreSQL work. In most cases, the advantage of a partial index over a regular index will not be much.

More information about partial indexes can be found in *The case for partial indexes* , *Partial indexing in POSTGRES: research project*, and *Generalized Partial Indexes (cached version)* .

11.8. Operator Classes

An index definition may specify an *operator class* for each column of an index.

```
CREATE INDEX name ON table (column opclass [, ...]);
```

The operator class identifies the operators to be used by the index for that column. For example, a B-tree index on the type `int4` would use the `int4_ops` class; this operator class includes comparison functions for values of type `int4`. In practice the default operator class for the column's data type is usually sufficient. The main point of having operator classes is that for some data types, there could be more than one meaningful index behavior. For example, we might want to sort a complex-number data type either by absolute value or by real part. We could do this by defining two operator classes for the data type and then selecting the proper class when making an index.

There are also some built-in operator classes besides the default ones:

- The operator classes `text_pattern_ops`, `varchar_pattern_ops`, `bpchar_pattern_ops`, and `name_pattern_ops` support B-tree indexes on the types `text`, `varchar`, `char`, and `name`, respectively. The difference from the default operator classes is that the values are compared strictly character by character rather than according to the locale-specific collation rules. This makes these operator classes suitable for use by queries involving pattern matching expressions (`LIKE` or POSIX regular expressions) when the server does not use the standard "C" locale. As an example, you might index a `varchar` column like this:

  ```
  CREATE INDEX test_index ON test_table (col varchar_pattern_ops);
  ```

  Note that you should also create an index with the default operator class if you want queries involving ordinary comparisons to use an index. Such queries cannot use the *xxx*`_pattern_ops` operator classes. It is allowed to create multiple indexes on the same column with different operator classes. If you do use the C locale, you do not need the *xxx*`_pattern_ops` operator classes, because an index with the default operator class is usable for pattern-matching queries in the C locale.

The following query shows all defined operator classes:

```
SELECT am.amname AS index_method,
```

```
       opc.opcname AS opclass_name
    FROM pg_am am, pg_opclass opc
    WHERE opc.opcamid = am.oid
    ORDER BY index_method, opclass_name;
```

It can be extended to show all the operators included in each class:

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name,
       opr.oid::regoperator AS opclass_operator
    FROM pg_am am, pg_opclass opc, pg_amop amop, pg_operator opr
    WHERE opc.opcamid = am.oid AND
          amop.amopclaid = opc.oid AND
          amop.amopopr = opr.oid
    ORDER BY index_method, opclass_name, opclass_operator;
```

11.9. Examining Index Usage

Although indexes in PostgreSQL do not need maintenance and tuning, it is still important to check which indexes are actually used by the real-life query workload. Examining index usage for an individual query is done with the *EXPLAIN* command; its application for this purpose is illustrated in Section 13.1. It is also possible to gather overall statistics about index usage in a running server, as described in Section 25.2.

It is difficult to formulate a general procedure for determining which indexes to set up. There are a number of typical cases that have been shown in the examples throughout the previous sections. A good deal of experimentation will be necessary in most cases. The rest of this section gives some tips for that.

- Always run *ANALYZE* first. This command collects statistics about the distribution of the values in the table. This information is required to guess the number of rows returned by a query, which is needed by the planner to assign realistic costs to each possible query plan. In absence of any real statistics, some default values are assumed, which are almost certain to be inaccurate. Examining an application's index usage without having run `ANALYZE` is therefore a lost cause.
- Use real data for experimentation. Using test data for setting up indexes will tell you what indexes you need for the test data, but that is all.

  It is especially fatal to use very small test data sets. While selecting 1000 out of 100000 rows could be a candidate for an index, selecting 1 out of 100 rows will hardly be, because the 100 rows will probably fit within a single disk page, and there is no plan that can beat sequentially fetching 1 disk page.

  Also be careful when making up test data, which is often unavoidable when the application is not in production use yet. Values that are very similar, completely random, or inserted in sorted order will skew the statistics away from the distribution that real data would have.
- When indexes are not used, it can be useful for testing to force their use. There are run-time parameters that can turn off various plan types (see Section 17.6.1). For instance, turning off sequential scans (`enable_seqscan`) and nested-loop joins (`enable_nestloop`), which are the most basic plans, will force the system to use a different plan. If the system still chooses a sequential scan or nested-loop join

then there is probably a more fundamental reason why the index is not used; for example, the query condition does not match the index. (What kind of query can use what kind of index is explained in the previous sections.)

- If forcing index usage does use the index, then there are two possibilities: Either the system is right and using the index is indeed not appropriate, or the cost estimates of the query plans are not reflecting reality. So you should time your query with and without indexes. The `EXPLAIN ANALYZE` command can be useful here.
- If it turns out that the cost estimates are wrong, there are, again, two possibilities. The total cost is computed from the per-row costs of each plan node times the selectivity estimate of the plan node. The costs estimated for the plan nodes can be adjusted via run-time parameters (described in Section 17.6.2). An inaccurate selectivity estimate is due to insufficient statistics. It may be possible to improve this by tuning the statistics-gathering parameters (see *ALTER TABLE*).

  If you do not succeed in adjusting the costs to be more appropriate, then you may have to resort to forcing index usage explicitly. You may also want to contact the PostgreSQL developers to examine the issue.

Chapter 12. Concurrency Control

This chapter describes the behavior of the PostgreSQL database system when two or more sessions try to access the same data at the same time. The goals in that situation are to allow efficient access for all sessions while maintaining strict data integrity. Every developer of database applications should be familiar with the topics covered in this chapter.

12.1. Introduction

PostgreSQL provides a rich set of tools for developers to manage concurrent access to data. Internally, data consistency is maintained by using a multiversion model (Multiversion Concurrency Control, MVCC). This means that while querying a database each transaction sees a snapshot of data (a *database version*) as it was some time ago, regardless of the current state of the underlying data. This protects the transaction from viewing inconsistent data that could be caused by (other) concurrent transaction updates on the same data rows, providing *transaction isolation* for each database session. MVCC, by eschewing explicit locking methodologies of traditional database systems, minimizes lock contention in order to allow for reasonable performance in multiuser environments.

The main advantage to using the MVCC model of concurrency control rather than locking is that in MVCC locks acquired for querying (reading) data do not conflict with locks acquired for writing data, and so reading never blocks writing and writing never blocks reading.

Table- and row-level locking facilities are also available in PostgreSQL for applications that cannot adapt easily to MVCC behavior. However, proper use of MVCC will generally provide better performance than locks. In addition, application-defined advisory locks provide a mechanism for acquiring locks that are not tied to a single transaction.

12.2. Transaction Isolation

The SQL standard defines four levels of transaction isolation in terms of three phenomena that must be prevented between concurrent transactions. These undesirable phenomena are:

dirty read

A transaction reads data written by a concurrent uncommitted transaction.

nonrepeatable read

A transaction re-reads data it has previously read and finds that data has been modified by another transaction (that committed since the initial read).

phantom read

A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction.

The four transaction isolation levels and the corresponding behaviors are described in Table 12-1.

Table 12-1. SQL Transaction Isolation Levels

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

In PostgreSQL, you can request any of the four standard transaction isolation levels. But internally, there are only two distinct isolation levels, which correspond to the levels Read Committed and Serializable. When you select the level Read Uncommitted you really get Read Committed, and when you select Repeatable Read you really get Serializable, so the actual isolation level may be stricter than what you select. This is permitted by the SQL standard: the four isolation levels only define which phenomena must not happen, they do not define which phenomena must happen. The reason that PostgreSQL only provides two isolation levels is that this is the only sensible way to map the standard isolation levels to the multiversion concurrency control architecture. The behavior of the available isolation levels is detailed in the following subsections.

To set the transaction isolation level of a transaction, use the command *SET TRANSACTION*.

12.2.1. Read Committed Isolation Level

Read Committed is the default isolation level in PostgreSQL. When a transaction runs on this isolation level, a `SELECT` query sees only data committed before the query began; it never sees either uncommitted data or changes committed during query execution by concurrent transactions. (However, the `SELECT` does see the effects of previous updates executed within its own transaction, even though they are not yet committed.) In effect, a `SELECT` query sees a snapshot of the database as of the instant that that query begins to run. Notice that two successive `SELECT` commands can see different data, even though they are within a single transaction, if other transactions commit changes during execution of the first `SELECT`.

`UPDATE`, `DELETE`, `SELECT FOR UPDATE`, and `SELECT FOR SHARE` commands behave the same as `SELECT` in terms of searching for target rows: they will only find target rows that were committed as of the command start time. However, such a target row may have already been updated (or deleted or locked) by another concurrent transaction by the time it is found. In this case, the would-be updater will wait for the first updating transaction to commit or roll back (if it is still in progress). If the first updater rolls back, then its effects are negated and the second updater can proceed with updating the originally found row. If the first updater commits, the second updater will ignore the row if the first updater deleted it, otherwise it will attempt to apply its operation to the updated version of the row. The search condition of the command (the `WHERE` clause) is re-evaluated to see if the updated version of the row still matches the search condition. If so, the second updater proceeds with its operation, starting from the updated version of the row. (In the case of `SELECT FOR UPDATE` and `SELECT FOR SHARE`, that means it is the updated version of the row that is locked and returned to the client.)

Because of the above rule, it is possible for an updating command to see an inconsistent snapshot: it can see the effects of concurrent updating commands that affected the same rows it is trying to update, but it does not see effects of those commands on other rows in the database. This behavior makes Read Committed mode unsuitable for commands that involve complex search conditions. However, it is just right for simpler cases. For example, consider updating bank balances with transactions like

```
BEGIN;
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 12345;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 7534;
COMMIT;
```

If two such transactions concurrently try to change the balance of account 12345, we clearly want the second transaction to start from the updated version of the account's row. Because each command is affecting only a predetermined row, letting it see the updated version of the row does not create any troublesome inconsistency.

Since in Read Committed mode each new command starts with a new snapshot that includes all transactions committed up to that instant, subsequent commands in the same transaction will see the effects of the committed concurrent transaction in any case. The point at issue here is whether or not within a *single* command we see an absolutely consistent view of the database.

The partial transaction isolation provided by Read Committed mode is adequate for many applications, and this mode is fast and simple to use. However, for applications that do complex queries and updates, it may be necessary to guarantee a more rigorously consistent view of the database than the Read Committed mode provides.

12.2.2. Serializable Isolation Level

The level *Serializable* provides the strictest transaction isolation. This level emulates serial transaction execution, as if transactions had been executed one after another, serially, rather than concurrently. However, applications using this level must be prepared to retry transactions due to serialization failures.

When a transaction is on the serializable level, a `SELECT` query sees only data committed before the transaction began; it never sees either uncommitted data or changes committed during transaction execution by concurrent transactions. (However, the `SELECT` does see the effects of previous updates executed within its own transaction, even though they are not yet committed.) This is different from Read Committed in that the `SELECT` sees a snapshot as of the start of the transaction, not as of the start of the current query within the transaction. Thus, successive `SELECT` commands within a single transaction always see the same data.

`UPDATE`, `DELETE`, `SELECT FOR UPDATE`, and `SELECT FOR SHARE` commands behave the same as `SELECT` in terms of searching for target rows: they will only find target rows that were committed as of the transaction start time. However, such a target row may have already been updated (or deleted or locked) by another concurrent transaction by the time it is found. In this case, the serializable transaction will wait for the first updating transaction to commit or roll back (if it is still in progress). If the first updater rolls back, then its effects are negated and the serializable transaction can proceed with updating the originally found row. But if the first updater commits (and actually updated or deleted the row, not just locked it) then the serializable transaction will be rolled back with the message

```
ERROR:  could not serialize access due to concurrent update
```

because a serializable transaction cannot modify or lock rows changed by other transactions after the serializable transaction began.

When the application receives this error message, it should abort the current transaction and then retry the whole transaction from the beginning. The second time through, the transaction sees the previously-

committed change as part of its initial view of the database, so there is no logical conflict in using the new version of the row as the starting point for the new transaction's update.

Note that only updating transactions may need to be retried; read-only transactions will never have serialization conflicts.

The Serializable mode provides a rigorous guarantee that each transaction sees a wholly consistent view of the database. However, the application has to be prepared to retry transactions when concurrent updates make it impossible to sustain the illusion of serial execution. Since the cost of redoing complex transactions may be significant, this mode is recommended only when updating transactions contain logic sufficiently complex that they may give wrong answers in Read Committed mode. Most commonly, Serializable mode is necessary when a transaction executes several successive commands that must see identical views of the database.

12.2.2.1. Serializable Isolation versus True Serializability

The intuitive meaning (and mathematical definition) of "serializable" execution is that any two successfully committed concurrent transactions will appear to have executed strictly serially, one after the other — although which one appeared to occur first may not be predictable in advance. It is important to realize that forbidding the undesirable behaviors listed in Table 12-1 is not sufficient to guarantee true serializability, and in fact PostgreSQL's Serializable mode *does not guarantee serializable execution in this sense*. As an example, consider a table `mytab`, initially containing

```
 class | value
-------+-------
     1 |    10
     1 |    20
     2 |   100
     2 |   200
```

Suppose that serializable transaction A computes

```
SELECT SUM(value) FROM mytab WHERE class = 1;
```

and then inserts the result (30) as the `value` in a new row with `class` = 2. Concurrently, serializable transaction B computes

```
SELECT SUM(value) FROM mytab WHERE class = 2;
```

and obtains the result 300, which it inserts in a new row with `class` = 1. Then both transactions commit. None of the listed undesirable behaviors have occurred, yet we have a result that could not have occurred in either order serially. If A had executed before B, B would have computed the sum 330, not 300, and similarly the other order would have resulted in a different sum computed by A.

To guarantee true mathematical serializability, it is necessary for a database system to enforce *predicate locking*, which means that a transaction cannot insert or modify a row that would have matched the `WHERE` condition of a query in another concurrent transaction. For example, once transaction A has executed the query `SELECT ... WHERE class = 1`, a predicate-locking system would forbid transaction B from inserting any new row with class 1 until A has committed. [1] Such a locking system is complex to im-

1. Essentially, a predicate-locking system prevents phantom reads by restricting what is written, whereas MVCC prevents them by restricting what is read.

plement and extremely expensive in execution, since every session must be aware of the details of every query executed by every concurrent transaction. And this large expense is mostly wasted, since in practice most applications do not do the sorts of things that could result in problems. (Certainly the example above is rather contrived and unlikely to represent real software.) For these reasons, PostgreSQL does not implement predicate locking.

In those cases where the possibility of nonserializable execution is a real hazard, problems can be prevented by appropriate use of explicit locking. Further discussion appears in the following sections.

12.3. Explicit Locking

PostgreSQL provides various lock modes to control concurrent access to data in tables. These modes can be used for application-controlled locking in situations where MVCC does not give the desired behavior. Also, most PostgreSQL commands automatically acquire locks of appropriate modes to ensure that referenced tables are not dropped or modified in incompatible ways while the command executes. (For example, `ALTER TABLE` cannot safely be executed concurrently with other operations on the same table, so it obtains an exclusive lock on the table to enforce that.)

To examine a list of the currently outstanding locks in a database server, use the `pg_locks` system view. For more information on monitoring the status of the lock manager subsystem, refer to Chapter 25.

12.3.1. Table-Level Locks

The list below shows the available lock modes and the contexts in which they are used automatically by PostgreSQL. You can also acquire any of these locks explicitly with the command *LOCK*. Remember that all of these lock modes are table-level locks, even if the name contains the word "row"; the names of the lock modes are historical. To some extent the names reflect the typical usage of each lock mode — but the semantics are all the same. The only real difference between one lock mode and another is the set of lock modes with which each conflicts. Two transactions cannot hold locks of conflicting modes on the same table at the same time. (However, a transaction never conflicts with itself. For example, it may acquire `ACCESS EXCLUSIVE` lock and later acquire `ACCESS SHARE` lock on the same table.) Nonconflicting lock modes may be held concurrently by many transactions. Notice in particular that some lock modes are self-conflicting (for example, an `ACCESS EXCLUSIVE` lock cannot be held by more than one transaction at a time) while others are not self-conflicting (for example, an `ACCESS SHARE` lock can be held by multiple transactions).

Table-level lock modes

`ACCESS SHARE`

Conflicts with the `ACCESS EXCLUSIVE` lock mode only.

The `SELECT` command acquires a lock of this mode on referenced tables. In general, any query that only reads a table and does not modify it will acquire this lock mode.

`ROW SHARE`

Conflicts with the `EXCLUSIVE` and `ACCESS EXCLUSIVE` lock modes.

The SELECT FOR UPDATE and SELECT FOR SHARE commands acquire a lock of this mode on the target table(s) (in addition to ACCESS SHARE locks on any other tables that are referenced but not selected FOR UPDATE/FOR SHARE).

ROW EXCLUSIVE

Conflicts with the SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes.

The commands UPDATE, DELETE, and INSERT acquire this lock mode on the target table (in addition to ACCESS SHARE locks on any other referenced tables). In general, this lock mode will be acquired by any command that modifies the data in a table.

SHARE UPDATE EXCLUSIVE

Conflicts with the SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This mode protects a table against concurrent schema changes and VACUUM runs.

Acquired by VACUUM (without FULL), ANALYZE, and CREATE INDEX CONCURRENTLY.

SHARE

Conflicts with the ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This mode protects a table against concurrent data changes.

Acquired by CREATE INDEX (without CONCURRENTLY).

SHARE ROW EXCLUSIVE

Conflicts with the ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes.

This lock mode is not automatically acquired by any PostgreSQL command.

EXCLUSIVE

Conflicts with the ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This mode allows only concurrent ACCESS SHARE locks, i.e., only reads from the table can proceed in parallel with a transaction holding this lock mode.

This lock mode is not automatically acquired on user tables by any PostgreSQL command. However it is acquired on certain system catalogs in some operations.

ACCESS EXCLUSIVE

Conflicts with locks of all modes (ACCESS SHARE, ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE). This mode guarantees that the holder is the only transaction accessing the table in any way.

Acquired by the ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER, and VACUUM FULL commands. This is also the default lock mode for LOCK TABLE statements that do not specify a mode explicitly.

Tip: Only an ACCESS EXCLUSIVE lock blocks a SELECT (without FOR UPDATE/SHARE) statement.

Once acquired, a lock is normally held till end of transaction. But if a lock is acquired after establishing a savepoint, the lock is released immediately if the savepoint is rolled back to. This is consistent with the principle that `ROLLBACK` cancels all effects of the commands since the savepoint. The same holds for locks acquired within a PL/pgSQL exception block: an error escape from the block releases locks acquired within it.

12.3.2. Row-Level Locks

In addition to table-level locks, there are row-level locks, which can be exclusive or shared locks. An exclusive row-level lock on a specific row is automatically acquired when the row is updated or deleted. The lock is held until the transaction commits or rolls back, in just the same way as for table-level locks. Row-level locks do not affect data querying; they block *writers to the same row* only.

To acquire an exclusive row-level lock on a row without actually modifying the row, select the row with `SELECT FOR UPDATE`. Note that once the row-level lock is acquired, the transaction may update the row multiple times without fear of conflicts.

To acquire a shared row-level lock on a row, select the row with `SELECT FOR SHARE`. A shared lock does not prevent other transactions from acquiring the same shared lock. However, no transaction is allowed to update, delete, or exclusively lock a row on which any other transaction holds a shared lock. Any attempt to do so will block until the shared lock(s) have been released.

PostgreSQL doesn't remember any information about modified rows in memory, so it has no limit to the number of rows locked at one time. However, locking a row may cause a disk write; thus, for example, `SELECT FOR UPDATE` will modify selected rows to mark them locked, and so will result in disk writes.

In addition to table and row locks, page-level share/exclusive locks are used to control read/write access to table pages in the shared buffer pool. These locks are released immediately after a row is fetched or updated. Application developers normally need not be concerned with page-level locks, but we mention them for completeness.

12.3.3. Deadlocks

The use of explicit locking can increase the likelihood of *deadlocks*, wherein two (or more) transactions each hold locks that the other wants. For example, if transaction 1 acquires an exclusive lock on table A and then tries to acquire an exclusive lock on table B, while transaction 2 has already exclusive-locked table B and now wants an exclusive lock on table A, then neither one can proceed. PostgreSQL automatically detects deadlock situations and resolves them by aborting one of the transactions involved, allowing the other(s) to complete. (Exactly which transaction will be aborted is difficult to predict and should not be relied on.)

Note that deadlocks can also occur as the result of row-level locks (and thus, they can occur even if explicit locking is not used). Consider the case in which there are two concurrent transactions modifying a table. The first transaction executes:

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 11111;
```

This acquires a row-level lock on the row with the specified account number. Then, the second transaction executes:

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 22222;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 11111;
```

The first `UPDATE` statement successfully acquires a row-level lock on the specified row, so it succeeds in updating that row. However, the second `UPDATE` statement finds that the row it is attempting to update has already been locked, so it waits for the transaction that acquired the lock to complete. Transaction two is now waiting on transaction one to complete before it continues execution. Now, transaction one executes:

```
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 22222;
```

Transaction one attempts to acquire a row-level lock on the specified row, but it cannot: transaction two already holds such a lock. So it waits for transaction two to complete. Thus, transaction one is blocked on transaction two, and transaction two is blocked on transaction one: a deadlock condition. PostgreSQL will detect this situation and abort one of the transactions.

The best defense against deadlocks is generally to avoid them by being certain that all applications using a database acquire locks on multiple objects in a consistent order. In the example above, if both transactions had updated the rows in the same order, no deadlock would have occurred. One should also ensure that the first lock acquired on an object in a transaction is the highest mode that will be needed for that object. If it is not feasible to verify this in advance, then deadlocks may be handled on-the-fly by retrying transactions that are aborted due to deadlock.

So long as no deadlock situation is detected, a transaction seeking either a table-level or row-level lock will wait indefinitely for conflicting locks to be released. This means it is a bad idea for applications to hold transactions open for long periods of time (e.g., while waiting for user input).

12.3.4. Advisory Locks

PostgreSQL provides a means for creating locks that have application-defined meanings. These are called *advisory locks*, because the system does not enforce their use — it is up to the application to use them correctly. Advisory locks can be useful for locking strategies that are an awkward fit for the MVCC model. Once acquired, an advisory lock is held until explicitly released or the session ends. Unlike standard locks, advisory locks do not honor transaction semantics: a lock acquired during a transaction that is later rolled back will still be held following the rollback, and likewise an unlock is effective even if the calling transaction fails later. The same lock can be acquired multiple times by its owning process: for each lock request there must be a corresponding unlock request before the lock is actually released. (If a session already holds a given lock, additional requests will always succeed, even if other sessions are awaiting the lock.) Like all locks in PostgreSQL, a complete list of advisory locks currently held by any session can be found in the `pg_locks` system view.

Advisory locks are allocated out of a shared memory pool whose size is defined by the configuration variables max_locks_per_transaction and max_connections. Care must be taken not to exhaust this memory or the server will not be able to grant any locks at all. This imposes an upper limit on the number of advisory locks grantable by the server, typically in the tens to hundreds of thousands depending on how the server is configured.

A common use of advisory locks is to emulate pessimistic locking strategies typical of so called “flat file” data management systems. While a flag stored in a table could be used for the same purpose, advisory locks are faster, avoid MVCC bloat, and are automatically cleaned up by the server at the end of the session. In certain cases using this method, especially in queries involving explicit ordering and `LIMIT`

clauses, care must be taken to control the locks acquired because of the order in which SQL expressions are evaluated. For example:

```
SELECT pg_advisory_lock(id) FROM foo WHERE id = 12345; -- ok
SELECT pg_advisory_lock(id) FROM foo WHERE id > 12345 LIMIT 100; -- danger!
SELECT pg_advisory_lock(q.id) FROM
(
  SELECT id FROM foo WHERE id > 12345 LIMIT 100;
) q; -- ok
```

In the above queries, the second form is dangerous because the `LIMIT` is not guaranteed to be applied before the locking function is executed. This might cause some locks to be acquired that the application was not expecting, and hence would fail to release (until it ends the session). From the point of view of the application, such locks would be dangling, although still viewable in `pg_locks`.

The functions provided to manipulate advisory locks are described in Table 9-50.

12.4. Data Consistency Checks at the Application Level

Because readers in PostgreSQL do not lock data, regardless of transaction isolation level, data read by one transaction can be overwritten by another concurrent transaction. In other words, if a row is returned by `SELECT` it doesn't mean that the row is still current at the instant it is returned (i.e., sometime after the current query began). The row might have been modified or deleted by an already-committed transaction that committed after this one started. Even if the row is still valid "now", it could be changed or deleted before the current transaction does a commit or rollback.

Another way to think about it is that each transaction sees a snapshot of the database contents, and concurrently executing transactions may very well see different snapshots. So the whole concept of "now" is somewhat ill-defined anyway. This is not normally a big problem if the client applications are isolated from each other, but if the clients can communicate via channels outside the database then serious confusion may ensue.

To ensure the current validity of a row and protect it against concurrent updates one must use `SELECT FOR UPDATE`, `SELECT FOR SHARE`, or an appropriate `LOCK TABLE` statement. (`SELECT FOR UPDATE` or `SELECT FOR SHARE` locks just the returned rows against concurrent updates, while `LOCK TABLE` locks the whole table.) This should be taken into account when porting applications to PostgreSQL from other environments.

Global validity checks require extra thought under MVCC. For example, a banking application might wish to check that the sum of all credits in one table equals the sum of debits in another table, when both tables are being actively updated. Comparing the results of two successive `SELECT sum(...)` commands will not work reliably under Read Committed mode, since the second query will likely include the results of transactions not counted by the first. Doing the two sums in a single serializable transaction will give an accurate picture of the effects of transactions that committed before the serializable transaction started — but one might legitimately wonder whether the answer is still relevant by the time it is delivered. If the serializable transaction itself applied some changes before trying to make the consistency check, the usefulness of the check becomes even more debatable, since now it includes some but not all post-transaction-start changes. In such cases a careful person might wish to lock all tables needed for the check,

in order to get an indisputable picture of current reality. A SHARE mode (or higher) lock guarantees that there are no uncommitted changes in the locked table, other than those of the current transaction.

Note also that if one is relying on explicit locking to prevent concurrent changes, one should use Read Committed mode, or in Serializable mode be careful to obtain the lock(s) before performing queries. A lock obtained by a serializable transaction guarantees that no other transactions modifying the table are still running, but if the snapshot seen by the transaction predates obtaining the lock, it may predate some now-committed changes in the table. A serializable transaction's snapshot is actually frozen at the start of its first query or data-modification command (SELECT, INSERT, UPDATE, or DELETE), so it's possible to obtain locks explicitly before the snapshot is frozen.

12.5. Locking and Indexes

Though PostgreSQL provides nonblocking read/write access to table data, nonblocking read/write access is not currently offered for every index access method implemented in PostgreSQL. The various index types are handled as follows:

B-tree and GiST indexes

: Short-term share/exclusive page-level locks are used for read/write access. Locks are released immediately after each index row is fetched or inserted. These index types provide the highest concurrency without deadlock conditions.

Hash indexes

: Share/exclusive hash-bucket-level locks are used for read/write access. Locks are released after the whole bucket is processed. Bucket-level locks provide better concurrency than index-level ones, but deadlock is possible since the locks are held longer than one index operation.

GIN indexes

: Short-term share/exclusive page-level locks are used for read/write access. Locks are released immediately after each index row is fetched or inserted. But note that a GIN-indexed value insertion usually produces several index key insertions per row, so GIN may do substantial work for a single value's insertion.

Currently, B-tree indexes offer the best performance for concurrent applications; since they also have more features than hash indexes, they are the recommended index type for concurrent applications that need to index scalar data. When dealing with non-scalar data, B-trees are not useful, and GiST or GIN indexes should be used instead.

Chapter 13. Performance Tips

Query performance can be affected by many things. Some of these can be manipulated by the user, while others are fundamental to the underlying design of the system. This chapter provides some hints about understanding and tuning PostgreSQL performance.

13.1. Using EXPLAIN

PostgreSQL devises a *query plan* for each query it is given. Choosing the right plan to match the query structure and the properties of the data is absolutely critical for good performance, so the system includes a complex *planner* that tries to select good plans. You can use the *EXPLAIN* command to see what query plan the planner creates for any query. Plan-reading is an art that deserves an extensive tutorial, which this is not; but here is some basic information.

The structure of a query plan is a tree of *plan nodes*. Nodes at the bottom level are table scan nodes: they return raw rows from a table. There are different types of scan nodes for different table access methods: sequential scans, index scans, and bitmap index scans. If the query requires joining, aggregation, sorting, or other operations on the raw rows, then there will be additional nodes "atop" the scan nodes to perform these operations. Again, there is usually more than one possible way to do these operations, so different node types can appear here too. The output of EXPLAIN has one line for each node in the plan tree, showing the basic node type plus the cost estimates that the planner made for the execution of that plan node. The first line (topmost node) has the estimated total execution cost for the plan; it is this number that the planner seeks to minimize.

Here is a trivial example, just to show what the output looks like. [1]

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

The numbers that are quoted by EXPLAIN are:

- Estimated start-up cost (Time expended before output scan can start, e.g., time to do the sorting in a sort node.)
- Estimated total cost (If all rows were to be retrieved, which they may not be: for example, a query with a LIMIT clause will stop short of paying the total cost of the Limit plan node's input node.)
- Estimated number of rows output by this plan node (Again, only if executed to completion.)
- Estimated average width (in bytes) of rows output by this plan node

1. Examples in this section are drawn from the regression test database after doing a VACUUM ANALYZE, using 8.2 development sources. You should be able to get similar results if you try the examples yourself, but your estimated costs and row counts will probably vary slightly because ANALYZE's statistics are random samples rather than being exact.

The costs are measured in arbitrary units determined by the planner's cost parameters (see Section 17.6.2). Traditional practice is to measure the costs in units of disk page fetches; that is, seq_page_cost is conventionally set to `1.0` and the other cost parameters are set relative to that. The examples in this section are run with the default cost parameters.

It's important to note that the cost of an upper-level node includes the cost of all its child nodes. It's also important to realize that the cost only reflects things that the planner cares about. In particular, the cost does not consider the time spent transmitting result rows to the client, which could be an important factor in the true elapsed time; but the planner ignores it because it cannot change it by altering the plan. (Every correct plan will output the same row set, we trust.)

Rows output is a little tricky because it is *not* the number of rows processed or scanned by the plan node. It is usually less, reflecting the estimated selectivity of any `WHERE`-clause conditions that are being applied at the node. Ideally the top-level rows estimate will approximate the number of rows actually returned, updated, or deleted by the query.

Returning to our example:

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

This is about as straightforward as it gets. If you do

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';
```

you will find out that `tenk1` has 358 disk pages and 10000 rows. So the cost is estimated at 358 page reads, costing seq_page_cost apiece (1.0 by default), plus 10000 * cpu_tuple_cost which is 0.01 by default.

Now let's modify the query to add a `WHERE` condition:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 7000;

                         QUERY PLAN
------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7033 width=244)
   Filter: (unique1 < 7000)
```

Notice that the `EXPLAIN` output shows the `WHERE` clause being applied as a "filter" condition; this means that the plan node checks the condition for each row it scans, and outputs only the ones that pass the condition. The estimate of output rows has gone down because of the `WHERE` clause. However, the scan will still have to visit all 10000 rows, so the cost hasn't decreased; in fact it has gone up a bit to reflect the extra CPU time spent checking the `WHERE` condition.

The actual number of rows this query would select is 7000, but the rows estimate is only approximate. If you try to duplicate this experiment, you will probably get a slightly different estimate; moreover, it will change after each `ANALYZE` command, because the statistics produced by `ANALYZE` are taken from a randomized sample of the table.

Now, let's make the condition more restrictive:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100;

                                 QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=2.37..232.35 rows=106 width=244)
   Recheck Cond: (unique1 < 100)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..2.37 rows=106 width=0)
         Index Cond: (unique1 < 100)
```

Here the planner has decided to use a two-step plan: the bottom plan node visits an index to find the locations of rows matching the index condition, and then the upper plan node actually fetches those rows from the table itself. Fetching the rows separately is much more expensive than sequentially reading them, but because not all the pages of the table have to be visited, this is still cheaper than a sequential scan. (The reason for using two levels of plan is that the upper plan node sorts the row locations identified by the index into physical order before reading them, so as to minimize the costs of the separate fetches. The "bitmap" mentioned in the node names is the mechanism that does the sorting.)

If the `WHERE` condition is selective enough, the planner may switch to a "simple" index scan plan:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 3;

                                 QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.00..10.00 rows=2 width=244)
   Index Cond: (unique1 < 3)
```

In this case the table rows are fetched in index order, which makes them even more expensive to read, but there are so few that the extra cost of sorting the row locations is not worth it. You'll most often see this plan type for queries that fetch just a single row, and for queries that request an `ORDER BY` condition that matches the index order.

Add another condition to the `WHERE` clause:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 3 AND stringu1 = 'xxx';

                                 QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.00..10.01 rows=1 width=244)
   Index Cond: (unique1 < 3)
   Filter: (stringu1 = 'xxx'::name)
```

The added condition `stringu1 = 'xxx'` reduces the output-rows estimate, but not the cost because we still have to visit the same set of rows. Notice that the `stringu1` clause cannot be applied as an index condition (since this index is only on the `unique1` column). Instead it is applied as a filter on the rows retrieved by the index. Thus the cost has actually gone up a little bit to reflect this extra checking.

If there are indexes on several columns used in `WHERE`, the planner might choose to use an AND or OR combination of the indexes:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                     QUERY PLAN
-------------------------------------------------------------------------------
```

```
 Bitmap Heap Scan on tenk1  (cost=11.27..49.11 rows=11 width=244)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   ->  BitmapAnd  (cost=11.27..11.27 rows=11 width=0)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..2.37 rows=106 width=0)
               Index Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..8.65 rows=1042 width=0)
               Index Cond: (unique2 > 9000)
```

But this requires visiting both indexes, so it's not necessarily a win compared to using just one index and treating the other condition as a filter. If you vary the ranges involved you'll see the plan change accordingly.

Let's try joining two tables, using the columns we have been discussing:

```
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2

                                      QUERY PLAN
------------------------------------------------------------------------------------
 Nested Loop  (cost=2.37..553.11 rows=106 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=2.37..232.35 rows=106 width=244)
         Recheck Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..2.37 rows=106 width=0)
               Index Cond: (unique1 < 100)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.00..3.01 rows=1 width=244)
         Index Cond: ("outer".unique2 = t2.unique2)
```

In this nested-loop join, the outer scan is the same bitmap index scan we saw earlier, and so its cost and row count are the same because we are applying the `WHERE` clause `unique1 < 100` at that node. The `t1.unique2 = t2.unique2` clause is not relevant yet, so it doesn't affect row count of the outer scan. For the inner scan, the `unique2` value of the current outer-scan row is plugged into the inner index scan to produce an index condition like `t2.unique2 = constant`. So we get the same inner-scan plan and costs that we'd get from, say, `EXPLAIN SELECT * FROM tenk2 WHERE unique2 = 42`. The costs of the loop node are then set on the basis of the cost of the outer scan, plus one repetition of the inner scan for each outer row (106 * 3.01, here), plus a little CPU time for join processing.

In this example the join's output row count is the same as the product of the two scans' row counts, but that's not true in general, because in general you can have `WHERE` clauses that mention both tables and so can only be applied at the join point, not to either input scan. For example, if we added `WHERE ... AND t1.hundred < t2.hundred`, that would decrease the output row count of the join node, but not change either input scan.

One way to look at variant plans is to force the planner to disregard whatever strategy it thought was the winner, using the enable/disable flags described in Section 17.6.1. (This is a crude tool, but useful. See also Section 13.3.)

```
SET enable_nestloop = off;
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2

                                      QUERY PLAN
------------------------------------------------------------------------------------
 Hash Join  (cost=232.61..741.67 rows=106 width=488)
   Hash Cond: ("outer".unique2 = "inner".unique2)
```

```
   ->  Seq Scan on tenk2 t2  (cost=0.00..458.00 rows=10000 width=244)
   ->  Hash  (cost=232.35..232.35 rows=106 width=244)
         ->  Bitmap Heap Scan on tenk1 t1  (cost=2.37..232.35 rows=106 width=244)
               Recheck Cond: (unique1 < 100)
               ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..2.37 rows=106 width=0)
                     Index Cond: (unique1 < 100)
```

This plan proposes to extract the 100 interesting rows of `tenk1` using that same old index scan, stash them into an in-memory hash table, and then do a sequential scan of `tenk2`, probing into the hash table for possible matches of `t1.unique2 = t2.unique2` at each `tenk2` row. The cost to read `tenk1` and set up the hash table is entirely start-up cost for the hash join, since we won't get any rows out until we can start reading `tenk2`. The total time estimate for the join also includes a hefty charge for the CPU time to probe the hash table 10000 times. Note, however, that we are *not* charging 10000 times 232.35; the hash table setup is only done once in this plan type.

It is possible to check on the accuracy of the planner's estimated costs by using `EXPLAIN ANALYZE`. This command actually executes the query, and then displays the true run time accumulated within each plan node along with the same estimated costs that a plain `EXPLAIN` shows. For example, we might get a result like this:

```
EXPLAIN ANALYZE SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 100 AND t1.unique2 = t2

                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------
 Nested Loop  (cost=2.37..553.11 rows=106 width=488) (actual time=1.392..12.700 rows=100 lo
   ->  Bitmap Heap Scan on tenk1 t1  (cost=2.37..232.35 rows=106 width=244) (actual time=0.
         Recheck Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..2.37 rows=106 width=0) (actual
               Index Cond: (unique1 < 100)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.00..3.01 rows=1 width=244) (actu
         Index Cond: ("outer".unique2 = t2.unique2)
 Total runtime: 14.452 ms
```

Note that the "actual time" values are in milliseconds of real time, whereas the "cost" estimates are expressed in arbitrary units; so they are unlikely to match up. The thing to pay attention to is whether the ratios of actual time and estimated costs are consistent.

In some query plans, it is possible for a subplan node to be executed more than once. For example, the inner index scan is executed once per outer row in the above nested-loop plan. In such cases, the "loops" value reports the total number of executions of the node, and the actual time and rows values shown are averages per-execution. This is done to make the numbers comparable with the way that the cost estimates are shown. Multiply by the "loops" value to get the total time actually spent in the node.

The `Total runtime` shown by `EXPLAIN ANALYZE` includes executor start-up and shut-down time, as well as time spent processing the result rows. It does not include parsing, rewriting, or planning time. For a `SELECT` query, the total run time will normally be just a little larger than the total time reported for the top-level plan node. For `INSERT`, `UPDATE`, and `DELETE` commands, the total run time may be considerably larger, because it includes the time spent processing the result rows. In these commands, the time for the top plan node essentially is the time spent computing the new rows and/or locating the old ones, but it doesn't include the time spent applying the changes. Time spent firing triggers, if any, is also outside the top plan node, and is shown separately for each trigger.

It is worth noting that EXPLAIN results should not be extrapolated to situations other than the one you are actually testing; for example, results on a toy-sized table can't be assumed to apply to large tables. The planner's cost estimates are not linear and so it may well choose a different plan for a larger or smaller table. An extreme example is that on a table that only occupies one disk page, you'll nearly always get a sequential scan plan whether indexes are available or not. The planner realizes that it's going to take one disk page read to process the table in any case, so there's no value in expending additional page reads to look at an index.

13.2. Statistics Used by the Planner

As we saw in the previous section, the query planner needs to estimate the number of rows retrieved by a query in order to make good choices of query plans. This section provides a quick look at the statistics that the system uses for these estimates.

One component of the statistics is the total number of entries in each table and index, as well as the number of disk blocks occupied by each table and index. This information is kept in the table pg_class, in the columns reltuples and relpages. We can look at it with queries similar to this one:

```
SELECT relname, relkind, reltuples, relpages FROM pg_class WHERE relname LIKE 'tenk1%';

       relname        | relkind | reltuples | relpages
----------------------+---------+-----------+----------
 tenk1                | r       |     10000 |      358
 tenk1_hundred        | i       |     10000 |       30
 tenk1_thous_tenthous | i       |     10000 |       30
 tenk1_unique1        | i       |     10000 |       30
 tenk1_unique2        | i       |     10000 |       30
(5 rows)
```

Here we can see that tenk1 contains 10000 rows, as do its indexes, but the indexes are (unsurprisingly) much smaller than the table.

For efficiency reasons, reltuples and relpages are not updated on-the-fly, and so they usually contain somewhat out-of-date values. They are updated by VACUUM, ANALYZE, and a few DDL commands such as CREATE INDEX. A stand-alone ANALYZE, that is one not part of VACUUM, generates an approximate reltuples value since it does not read every row of the table. The planner will scale the values it finds in pg_class to match the current physical table size, thus obtaining a closer approximation.

Most queries retrieve only a fraction of the rows in a table, due to having WHERE clauses that restrict the rows to be examined. The planner thus needs to make an estimate of the *selectivity* of WHERE clauses, that is, the fraction of rows that match each condition in the WHERE clause. The information used for this task is stored in the pg_statistic system catalog. Entries in pg_statistic are updated by the ANALYZE and VACUUM ANALYZE commands, and are always approximate even when freshly updated.

Rather than look at pg_statistic directly, it's better to look at its view pg_stats when examining the statistics manually. pg_stats is designed to be more easily readable. Furthermore, pg_stats is readable by all, whereas pg_statistic is only readable by a superuser. (This prevents unprivileged users from learning something about the contents of other people's tables from the statistics. The pg_stats view is restricted to show only rows about tables that the current user can read.) For example, we might do:

```
SELECT attname, n_distinct, most_common_vals FROM pg_stats WHERE tablename = 'road';

 attname | n_distinct |
---------+------------+------------------------------------------------------------------------
 name    |  -0.467008 | {"I- 580                         Ramp","I- 880
 thepath |         20 | {"[(-122.089,37.71),(-122.0886,37.711)]"}
(2 rows)
```

`pg_stats` is described in detail in Section 43.46.

The amount of information stored in `pg_statistic`, in particular the maximum number of entries in the `most_common_vals` and `histogram_bounds` arrays for each column, can be set on a column-by-column basis using the `ALTER TABLE SET STATISTICS` command, or globally by setting the default_statistics_target configuration variable. The default limit is presently 10 entries. Raising the limit may allow more accurate planner estimates to be made, particularly for columns with irregular data distributions, at the price of consuming more space in `pg_statistic` and slightly more time to compute the estimates. Conversely, a lower limit may be appropriate for columns with simple data distributions.

13.3. Controlling the Planner with Explicit `JOIN` Clauses

It is possible to control the query planner to some extent by using the explicit `JOIN` syntax. To see why this matters, we first need some background.

In a simple join query, such as

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

the planner is free to join the given tables in any order. For example, it could generate a query plan that joins A to B, using the `WHERE` condition `a.id = b.id`, and then joins C to this joined table, using the other `WHERE` condition. Or it could join B to C and then join A to that result. Or it could join A to C and then join them with B — but that would be inefficient, since the full Cartesian product of A and C would have to be formed, there being no applicable condition in the `WHERE` clause to allow optimization of the join. (All joins in the PostgreSQL executor happen between two input tables, so it's necessary to build up the result in one or another of these fashions.) The important point is that these different join possibilities give semantically equivalent results but may have hugely different execution costs. Therefore, the planner will explore all of them to try to find the most efficient query plan.

When a query only involves two or three tables, there aren't many join orders to worry about. But the number of possible join orders grows exponentially as the number of tables expands. Beyond ten or so input tables it's no longer practical to do an exhaustive search of all the possibilities, and even for six or seven tables planning may take an annoyingly long time. When there are too many input tables, the PostgreSQL planner will switch from exhaustive search to a *genetic* probabilistic search through a limited number of possibilities. (The switch-over threshold is set by the geqo_threshold run-time parameter.) The genetic search takes less time, but it won't necessarily find the best possible plan.

When the query involves outer joins, the planner has less freedom than it does for plain (inner) joins. For example, consider

```
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

Although this query's restrictions are superficially similar to the previous example, the semantics are different because a row must be emitted for each row of A that has no matching row in the join of B and C. Therefore the planner has no choice of join order here: it must join B to C and then join A to that result. Accordingly, this query takes less time to plan than the previous query. In other cases, the planner may be able to determine that more than one join order is safe. For example, given

```
SELECT * FROM a LEFT JOIN b ON (a.bid = b.id) LEFT JOIN c ON (a.cid = c.id);
```

it is valid to join A to either B or C first. Currently, only `FULL JOIN` completely constrains the join order. Most practical cases involving `LEFT JOIN` or `RIGHT JOIN` can be rearranged to some extent.

Explicit inner join syntax (`INNER JOIN`, `CROSS JOIN`, or unadorned `JOIN`) is semantically the same as listing the input relations in `FROM`, so it does not constrain the join order.

Even though most kinds of `JOIN` don't completely constrain the join order, it is possible to instruct the PostgreSQL query planner to treat all `JOIN` clauses as constraining the join order anyway. For example, these three queries are logically equivalent:

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

But if we tell the planner to honor the `JOIN` order, the second and third take less time to plan than the first. This effect is not worth worrying about for only three tables, but it can be a lifesaver with many tables.

To force the planner to follow the join order laid out by explicit `JOIN`s, set the join_collapse_limit run-time parameter to 1. (Other possible values are discussed below.)

You do not need to constrain the join order completely in order to cut search time, because it's OK to use `JOIN` operators within items of a plain `FROM` list. For example, consider

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

With `join_collapse_limit` = 1, this forces the planner to join A to B before joining them to other tables, but doesn't constrain its choices otherwise. In this example, the number of possible join orders is reduced by a factor of 5.

Constraining the planner's search in this way is a useful technique both for reducing planning time and for directing the planner to a good query plan. If the planner chooses a bad join order by default, you can force it to choose a better order via `JOIN` syntax — assuming that you know of a better order, that is. Experimentation is recommended.

A closely related issue that affects planning time is collapsing of subqueries into their parent query. For example, consider

```
SELECT *
FROM x, y,
    (SELECT * FROM a, b, c WHERE something) AS ss
WHERE somethingelse;
```

This situation might arise from use of a view that contains a join; the view's `SELECT` rule will be inserted in place of the view reference, yielding a query much like the above. Normally, the planner will try to collapse the subquery into the parent, yielding

```
SELECT * FROM x, y, a, b, c WHERE something AND somethingelse;
```

This usually results in a better plan than planning the subquery separately. (For example, the outer WHERE conditions might be such that joining X to A first eliminates many rows of A, thus avoiding the need to form the full logical output of the subquery.) But at the same time, we have increased the planning time; here, we have a five-way join problem replacing two separate three-way join problems. Because of the exponential growth of the number of possibilities, this makes a big difference. The planner tries to avoid getting stuck in huge join search problems by not collapsing a subquery if more than from_collapse_limit FROM items would result in the parent query. You can trade off planning time against quality of plan by adjusting this run-time parameter up or down.

from_collapse_limit and join_collapse_limit are similarly named because they do almost the same thing: one controls when the planner will "flatten out" subselects, and the other controls when it will flatten out explicit joins. Typically you would either set join_collapse_limit equal to from_collapse_limit (so that explicit joins and subselects act similarly) or set join_collapse_limit to 1 (if you want to control join order with explicit joins). But you might set them differently if you are trying to fine-tune the trade-off between planning time and run time.

13.4. Populating a Database

One may need to insert a large amount of data when first populating a database. This section contains some suggestions on how to make this process as efficient as possible.

13.4.1. Disable Autocommit

Turn off autocommit and just do one commit at the end. (In plain SQL, this means issuing BEGIN at the start and COMMIT at the end. Some client libraries may do this behind your back, in which case you need to make sure the library does it when you want it done.) If you allow each insertion to be committed separately, PostgreSQL is doing a lot of work for each row that is added. An additional benefit of doing all insertions in one transaction is that if the insertion of one row were to fail then the insertion of all rows inserted up to that point would be rolled back, so you won't be stuck with partially loaded data.

13.4.2. Use COPY

Use *COPY* to load all the rows in one command, instead of using a series of INSERT commands. The COPY command is optimized for loading large numbers of rows; it is less flexible than INSERT, but incurs significantly less overhead for large data loads. Since COPY is a single command, there is no need to disable autocommit if you use this method to populate a table.

If you cannot use COPY, it may help to use *PREPARE* to create a prepared INSERT statement, and then use EXECUTE as many times as required. This avoids some of the overhead of repeatedly parsing and planning INSERT.

Note that loading a large number of rows using COPY is almost always faster than using INSERT, even if PREPARE is used and multiple insertions are batched into a single transaction.

13.4.3. Remove Indexes

If you are loading a freshly created table, the fastest way is to create the table, bulk load the table's data using `COPY`, then create any indexes needed for the table. Creating an index on pre-existing data is quicker than updating it incrementally as each row is loaded.

If you are adding large amounts of data to an existing table, it may be a win to drop the index, load the table, and then recreate the index. Of course, the database performance for other users may be adversely affected during the time that the index is missing. One should also think twice before dropping unique indexes, since the error checking afforded by the unique constraint will be lost while the index is missing.

13.4.4. Remove Foreign Key Constraints

Just as with indexes, a foreign key constraint can be checked "in bulk" more efficiently than row-by-row. So it may be useful to drop foreign key constraints, load data, and re-create the constraints. Again, there is a trade-off between data load speed and loss of error checking while the constraint is missing.

13.4.5. Increase `maintenance_work_mem`

Temporarily increasing the maintenance_work_mem configuration variable when loading large amounts of data can lead to improved performance. This will help to speed up `CREATE INDEX` commands and `ALTER TABLE ADD FOREIGN KEY` commands. It won't do much for `COPY` itself, so this advice is only useful when you are using one or both of the above techniques.

13.4.6. Increase `checkpoint_segments`

Temporarily increasing the checkpoint_segments configuration variable can also make large data loads faster. This is because loading a large amount of data into PostgreSQL will cause checkpoints to occur more often than the normal checkpoint frequency (specified by the `checkpoint_timeout` configuration variable). Whenever a checkpoint occurs, all dirty pages must be flushed to disk. By increasing `checkpoint_segments` temporarily during bulk data loads, the number of checkpoints that are required can be reduced.

13.4.7. Run `ANALYZE` Afterwards

Whenever you have significantly altered the distribution of data within a table, running *ANALYZE* is strongly recommended. This includes bulk loading large amounts of data into the table. Running `ANALYZE` (or `VACUUM ANALYZE`) ensures that the planner has up-to-date statistics about the table. With no statistics or obsolete statistics, the planner may make poor decisions during query planning, leading to poor performance on any tables with inaccurate or nonexistent statistics.

13.4.8. Some Notes About pg_dump

Dump scripts generated by pg_dump automatically apply several, but not all, of the above guidelines. To reload a pg_dump dump as quickly as possible, you need to do a few extra things manually. (Note that these points apply while *restoring* a dump, not while *creating* it. The same points apply when using pg_restore to load from a pg_dump archive file.)

By default, pg_dump uses `COPY`, and when it is generating a complete schema-and-data dump, it is careful to load data before creating indexes and foreign keys. So in this case the first several guidelines are handled automatically. What is left for you to do is to set appropriate (i.e., larger than normal) values for `maintenance_work_mem` and `checkpoint_segments` before loading the dump script, and then to run `ANALYZE` afterwards.

A data-only dump will still use `COPY`, but it does not drop or recreate indexes, and it does not normally touch foreign keys. [2] So when loading a data-only dump, it is up to you to drop and recreate indexes and foreign keys if you wish to use those techniques. It's still useful to increase `checkpoint_segments` while loading the data, but don't bother increasing `maintenance_work_mem`; rather, you'd do that while manually recreating indexes and foreign keys afterwards. And don't forget to `ANALYZE` when you're done.

2. You can get the effect of disabling foreign keys by using the `--disable-triggers` option — but realize that that eliminates, rather than just postponing, foreign key validation, and so it is possible to insert bad data if you use it.

III. Server Administration

This part covers topics that are of interest to a PostgreSQL database administrator. This includes installation of the software, set up and configuration of the server, management of users and databases, and maintenance tasks. Anyone who runs a PostgreSQL server, even for personal use, but especially in production, should be familiar with the topics covered in this part.

The information in this part is arranged approximately in the order in which a new user should read it. But the chapters are self-contained and can be read individually as desired. The information in this part is presented in a narrative fashion in topical units. Readers looking for a complete description of a particular command should look into Part VI.

The first few chapters are written so that they can be understood without prerequisite knowledge, so that new users who need to set up their own server can begin their exploration with this part. The rest of this part is about tuning and management; that material assumes that the reader is familiar with the general use of the PostgreSQL database system. Readers are encouraged to look at Part I and Part II for additional information.

Chapter 14. Installation Instructions

This chapter describes the installation of PostgreSQL from the source code distribution. (If you are installing a pre-packaged distribution, such as an RPM or Debian package, ignore this chapter and read the packager's instructions instead.)

14.1. Short Version

```
./configure
gmake
su
gmake install
adduser postgres
mkdir /usr/local/pgsql/data
chown postgres /usr/local/pgsql/data
su - postgres
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data >logfile 2>&1 &
/usr/local/pgsql/bin/createdb test
/usr/local/pgsql/bin/psql test
```

The long version is the rest of this chapter.

14.2. Requirements

In general, a modern Unix-compatible platform should be able to run PostgreSQL. The platforms that had received specific testing at the time of release are listed in Section 14.7 below. In the `doc` subdirectory of the distribution there are several platform-specific FAQ documents you might wish to consult if you are having trouble.

The following software packages are required for building PostgreSQL:

- GNU make is required; other make programs will *not* work. GNU make is often installed under the name `gmake`; this document will always refer to it by that name. (On some systems GNU make is the default tool with the name `make`.) To test for GNU make enter

  `gmake --version`

  It is recommended to use version 3.76.1 or later.
- You need an ISO/ANSI C compiler. Recent versions of GCC are recommendable, but PostgreSQL is known to build with a wide variety of compilers from different vendors.
- tar is required to unpack the source distribution in the first place, in addition to either gzip or bzip2.
- The GNU Readline library (for simple line editing and command history retrieval) is used by default. If you don't want to use it then you must specify the `--without-readline` option for `configure`. As an alternative, you can often use the BSD-licensed `libedit` library, originally developed on NetBSD. The `libedit` library is GNU Readline-compatible and is used if `libreadline` is not found, or if

`--with-libedit-preferred` is used as an option to `configure`. If you are using a package-based Linux distribution, be aware that you need both the `readline` and `readline-devel` packages, if those are separate in your distribution.

- The zlib compression library will be used by default. If you don't want to use it then you must specify the `--without-zlib` option for `configure`. Using this option disables support for compressed archives in pg_dump and pg_restore.
- Additional software is needed to build PostgreSQL on Windows. You can build PostgreSQL for NT-based versions of Windows (like Windows XP and 2003) using MinGW; see `doc/FAQ_MINGW` for details. You can also build PostgreSQL using Cygwin; see `doc/FAQ_CYGWIN`. A Cygwin-based build will work on older versions of Windows, but if you have a choice, we recommend the MinGW approach. While these are the only tool sets recommended for a complete build, it is possible to build just the C client library (libpq) and the interactive terminal (psql) using other Windows tool sets. For details of that see Chapter 15.

The following packages are optional. They are not required in the default configuration, but they are needed when certain build options are enabled, as explained below.

- To build the server programming language PL/Perl you need a full Perl installation, including the `libperl` library and the header files. Since PL/Perl will be a shared library, the `libperl` library must be a shared library also on most platforms. This appears to be the default in recent Perl versions, but it was not in earlier versions, and in any case it is the choice of whomever installed Perl at your site.

  If you don't have the shared library but you need one, a message like this will appear during the build to point out this fact:

  ```
  *** Cannot build PL/Perl because libperl is not a shared library.
  *** You might have to rebuild your Perl installation.  Refer to
  *** the documentation for details.
  ```

  (If you don't follow the on-screen output you will merely notice that the PL/Perl library object, `plperl.so` or similar, will not be installed.) If you see this, you will have to rebuild and install Perl manually to be able to build PL/Perl. During the configuration process for Perl, request a shared library.
- To build the PL/Python server programming language, you need a Python installation with the header files and the distutils module. The distutils module is included by default with Python 1.6 and later; users of earlier versions of Python will need to install it.

  Since PL/Python will be a shared library, the `libpython` library must be a shared library also on most platforms. This is not the case in a default Python installation. If after building and installing you have a file called `plpython.so` (possibly a different extension), then everything went well. Otherwise you should have seen a notice like this flying by:

  ```
  *** Cannot build PL/Python because libpython is not a shared library.
  *** You might have to rebuild your Python installation.  Refer to
  *** the documentation for details.
  ```

  That means you have to rebuild (part of) your Python installation to supply this shared library.

  If you have problems, run Python 2.3 or later's configure using the `--enable-shared` flag. On some operating systems you don't have to build a shared library, but you will have to convince the PostgreSQL build system of this. Consult the `Makefile` in the `src/pl/plpython` directory for details.

- If you want to build the PL/Tcl procedural language, you of course need a Tcl installation.
- To enable Native Language Support (NLS), that is, the ability to display a program's messages in a language other than English, you need an implementation of the Gettext API. Some operating systems have this built-in (e.g., Linux, NetBSD, Solaris), for other systems you can download an add-on package from http://developer.postgresql.org/~petere/bsd-gettext/. If you are using the Gettext implementation in the GNU C library then you will additionally need the GNU Gettext package for some utility programs. For any of the other implementations you will not need it.
- Kerberos, OpenSSL, OpenLDAP, and/or PAM, if you want to support authentication or encryption using these services.

If you are building from a CVS tree instead of using a released source package, or if you want to do development, you also need the following packages:

- GNU Flex and Bison are needed to build a CVS checkout or if you changed the actual scanner and parser definition files. If you need them, be sure to get Flex 2.5.4 or later and Bison 1.875 or later. Other yacc programs can sometimes be used, but doing so requires extra effort and is not recommended. Other lex programs will definitely not work.

If you need to get a GNU package, you can find it at your local GNU mirror site (see http://www.gnu.org/order/ftp.html for a list) or at ftp://ftp.gnu.org/gnu/.

Also check that you have sufficient disk space. You will need about 65 MB for the source tree during compilation and about 15 MB for the installation directory. An empty database cluster takes about 25 MB, databases take about five times the amount of space that a flat text file with the same data would take. If you are going to run the regression tests you will temporarily need up to an extra 90 MB. Use the `df` command to check free disk space.

14.3. Getting The Source

The PostgreSQL 8.2.6 sources can be obtained by anonymous FTP from ftp://ftp.postgresql.org/pub/source/v8.2.6/postgresql-8.2.6.tar.gz. Other download options can be found on our website: http://www.postgresql.org/download/. After you have obtained the file, unpack it:

```
gunzip postgresql-8.2.6.tar.gz
tar xf postgresql-8.2.6.tar
```

This will create a directory `postgresql-8.2.6` under the current directory with the PostgreSQL sources. Change into that directory for the rest of the installation procedure.

14.4. If You Are Upgrading

The internal data storage format changes with new releases of PostgreSQL. Therefore, if you are upgrading an existing installation that does not have a version number "8.2.x", you must back up and

restore your data as shown here. These instructions assume that your existing installation is under the /usr/local/pgsql directory, and that the data area is in /usr/local/pgsql/data. Substitute your paths appropriately.

1. Make sure that your database is not updated during or after the backup. This does not affect the integrity of the backup, but the changed data would of course not be included. If necessary, edit the permissions in the file /usr/local/pgsql/data/pg_hba.conf (or equivalent) to disallow access from everyone except you.

2. To back up your database installation, type:

   pg_dumpall > ***outputfile***

   If you need to preserve OIDs (such as when using them as foreign keys), then use the -o option when running pg_dumpall.

   To make the backup, you can use the pg_dumpall command from the version you are currently running. For best results, however, try to use the pg_dumpall command from PostgreSQL 8.2.6, since this version contains bug fixes and improvements over older versions. While this advice might seem idiosyncratic since you haven't installed the new version yet, it is advisable to follow it if you plan to install the new version in parallel with the old version. In that case you can complete the installation normally and transfer the data later. This will also decrease the downtime.

3. If you are installing the new version at the same location as the old one then shut down the old server, at the latest before you install the new files:

   pg_ctl stop

   On systems that have PostgreSQL started at boot time, there is probably a start-up file that will accomplish the same thing. For example, on a Red Hat Linux system one might find that

   /etc/rc.d/init.d/postgresql stop

   works.

4. If you are installing in the same place as the old version then it is also a good idea to move the old installation out of the way, in case you have trouble and need to revert to it. Use a command like this:

   mv /usr/local/pgsql /usr/local/pgsql.old

After you have installed PostgreSQL 8.2.6, create a new database directory and start the new server. Remember that you must execute these commands while logged in to the special database user account (which you already have if you are upgrading).

```
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
```

Finally, restore your data with

/usr/local/pgsql/bin/psql -d postgres -f *outputfile*

using the *new* psql.

Further discussion appears in Section 23.5, which you are encouraged to read in any case.

14.5. Installation Procedure

1. Configuration

   The first step of the installation procedure is to configure the source tree for your system and choose the options you would like. This is done by running the `configure` script. For a default installation simply enter

   ```
   ./configure
   ```

   This script will run a number of tests to guess values for various system dependent variables and detect some quirks of your operating system, and finally will create several files in the build tree to record what it found. (You can also run `configure` in a directory outside the source tree if you want to keep the build directory separate.)

   The default configuration will build the server and utilities, as well as all client applications and interfaces that require only a C compiler. All files will be installed under `/usr/local/pgsql` by default.

   You can customize the build and installation process by supplying one or more of the following command line options to `configure`:

   `--prefix=`*`PREFIX`*

   Install all files under the directory *`PREFIX`* instead of `/usr/local/pgsql`. The actual files will be installed into various subdirectories; no files will ever be installed directly into the *`PREFIX`* directory.

   If you have special needs, you can also customize the individual subdirectories with the following options. However, if you leave these with their defaults, the installation will be relocatable, meaning you can move the directory after installation. (The `man` and `doc` locations are not affected by this.)

   For relocatable installs, you might want to use `configure`'s `--disable-rpath` option. Also, you will need to tell the operating system how to find the shared libraries.

   `--exec-prefix=`*`EXEC-PREFIX`*

   You can install architecture-dependent files under a different prefix, *`EXEC-PREFIX`*, than what *`PREFIX`* was set to. This can be useful to share architecture-independent files between hosts. If you omit this, then *`EXEC-PREFIX`* is set equal to *`PREFIX`* and both architecture-dependent and independent files will be installed under the same tree, which is probably what you want.

   `--bindir=`*`DIRECTORY`*

   Specifies the directory for executable programs. The default is *`EXEC-PREFIX`*`/bin`, which normally means `/usr/local/pgsql/bin`.

   `--datadir=`*`DIRECTORY`*

   Sets the directory for read-only data files used by the installed programs. The default is *`PREFIX`*`/share`. Note that this has nothing to do with where your database files will be placed.

   `--sysconfdir=`*`DIRECTORY`*

   The directory for various configuration files, *`PREFIX`*`/etc` by default.

`--libdir=`*`DIRECTORY`*

The location to install libraries and dynamically loadable modules. The default is *`EXEC-PREFIX`*`/lib`.

`--includedir=`*`DIRECTORY`*

The directory for installing C and C++ header files. The default is *`PREFIX`*`/include`.

`--mandir=`*`DIRECTORY`*

The man pages that come with PostgreSQL will be installed under this directory, in their respective `man`*`x`* subdirectories. The default is *`PREFIX`*`/man`.

`--with-docdir=`*`DIRECTORY`*
`--without-docdir`

Documentation files, except "man" pages, will be installed into this directory. The default is *`PREFIX`*`/doc`. If the option `--without-docdir` is specified, the documentation will not be installed by `make install`. This is intended for packaging scripts that have special methods for installing documentation.

Note: Care has been taken to make it possible to install PostgreSQL into shared installation locations (such as `/usr/local/include`) without interfering with the namespace of the rest of the system. First, the string "`/postgresql`" is automatically appended to `datadir`, `sysconfdir`, and `docdir`, unless the fully expanded directory name already contains the string "`postgres`" or "`pgsql`". For example, if you choose `/usr/local` as prefix, the documentation will be installed in `/usr/local/doc/postgresql`, but if the prefix is `/opt/postgres`, then it will be in `/opt/postgres/doc`. The public C header files of the client interfaces are installed into `includedir` and are namespace-clean. The internal header files and the server header files are installed into private directories under `includedir`. See the documentation of each interface for information about how to get at the its header files. Finally, a private subdirectory will also be created, if appropriate, under `libdir` for dynamically loadable modules.

`--with-includes=`*`DIRECTORIES`*

`DIRECTORIES` is a colon-separated list of directories that will be added to the list the compiler searches for header files. If you have optional packages (such as GNU Readline) installed in a non-standard location, you have to use this option and probably also the corresponding `--with-libraries` option.

Example: `--with-includes=/opt/gnu/include:/usr/sup/include`.

`--with-libraries=`*`DIRECTORIES`*

`DIRECTORIES` is a colon-separated list of directories to search for libraries. You will probably have to use this option (and the corresponding `--with-includes` option) if you have packages installed in non-standard locations.

Example: `--with-libraries=/opt/gnu/lib:/usr/sup/lib`.

`--enable-nls[=LANGUAGES]`

Enables Native Language Support (NLS), that is, the ability to display a program's messages in a language other than English. *LANGUAGES* is a space-separated list of codes of the languages that you want supported, for example `--enable-nls='de fr'`. (The intersection between your list and the set of actually provided translations will be computed automatically.) If you do not specify a list, then all available translations are installed.

To use this option, you will need an implementation of the Gettext API; see above.

`--with-pgport=NUMBER`

Set *NUMBER* as the default port number for server and clients. The default is 5432. The port can always be changed later on, but if you specify it here then both server and clients will have the same default compiled in, which can be very convenient. Usually the only good reason to select a non-default value is if you intend to run multiple PostgreSQL servers on the same machine.

`--with-perl`

Build the PL/Perl server-side language.

`--with-python`

Build the PL/Python server-side language.

`--with-tcl`

Build the PL/Tcl server-side language.

`--with-tclconfig=DIRECTORY`

Tcl installs the file `tclConfig.sh`, which contains configuration information needed to build modules interfacing to Tcl. This file is normally found automatically at a well-known location, but if you want to use a different version of Tcl you can specify the directory in which to look for it.

`--with-krb5`

Build with support for Kerberos 5 authentication. On many systems, the Kerberos system is not installed in a location that is searched by default (e.g., `/usr/include`, `/usr/lib`), so you must use the options `--with-includes` and `--with-libraries` in addition to this option. `configure` will check for the required header files and libraries to make sure that your Kerberos installation is sufficient before proceeding.

`--with-krb-srvnam=NAME`

The default name of the Kerberos service principal. `postgres` is the default. There's usually no reason to change this.

`--with-openssl`

Build with support for SSL (encrypted) connections. This requires the OpenSSL package to be installed. `configure` will check for the required header files and libraries to make sure that your OpenSSL installation is sufficient before proceeding.

`--with-pam`

Build with PAM (Pluggable Authentication Modules) support.

`--with-ldap`

Build with LDAP support for authentication and connection parameter lookup (see Section 29.15 and Section 20.2.5 for more information). On Unix, this requires the OpenLDAP package to be installed. `configure` will check for the required header files and libraries to make sure that your OpenLDAP installation is sufficient before proceeding. On Windows, the default WinLDAP library is used.

`--without-readline`

Prevents use of the Readline library (and libedit as well). This option disables command-line editing and history in psql, so it is not recommended.

`--with-libedit-preferred`

Favors the use of the BSD-licensed libedit library rather than GPL-licensed Readline. This option is significant only if you have both libraries installed; the default in that case is to use Readline.

`--with-bonjour`

Build with Bonjour support. This requires Bonjour support in your operating system. Recommended on Mac OS X.

`--enable-integer-datetimes`

Use 64-bit integer storage for datetimes and intervals, rather than the default floating-point storage. This reduces the range of representable values but guarantees microsecond precision across the full range (see Section 8.5 for more information). Note also that the integer datetimes code is newer than the floating-point code, and we still find bugs in it from time to time.

`--disable-spinlocks`

Allow the build to succeed even if PostgreSQL has no CPU spinlock support for the platform. The lack of spinlock support will result in poor performance; therefore, this option should only be used if the build aborts and informs you that the platform lacks spinlock support. If this option is required to build PostgreSQL on your platform, please report the problem to the PostgreSQL developers.

`--enable-thread-safety`

Make the client libraries thread-safe. This allows concurrent threads in libpq and ECPG programs to safely control their private connection handles. This option requires adequate threading support in your operating system.

`--without-zlib`

Prevents use of the Zlib library. This disables support for compressed archives in pg_dump and pg_restore. This option is only intended for those rare systems where this library is not available.

`--enable-debug`

Compiles all programs and libraries with debugging symbols. This means that you can run the programs through a debugger to analyze problems. This enlarges the size of the installed executables considerably, and on non-GCC compilers it usually also disables compiler optimization, causing slowdowns. However, having the symbols available is extremely helpful for dealing with any problems that may arise. Currently, this option is recommended for production installations

only if you use GCC. But you should always have it on if you are doing development work or running a beta version.

`--enable-cassert`

Enables *assertion* checks in the server, which test for many "can't happen" conditions. This is invaluable for code development purposes, but the tests slow things down a little. Also, having the tests turned on won't necessarily enhance the stability of your server! The assertion checks are not categorized for severity, and so what might be a relatively harmless bug will still lead to server restarts if it triggers an assertion failure. Currently, this option is not recommended for production use, but you should have it on for development work or when running a beta version.

`--enable-depend`

Enables automatic dependency tracking. With this option, the makefiles are set up so that all affected object files will be rebuilt when any header file is changed. This is useful if you are doing development work, but is just wasted overhead if you intend only to compile once and install. At present, this option will work only if you use GCC.

`--enable-dtrace`

Compiles with support for the dynamic tracing tool DTrace. Operating system support for DTrace is currently only available in Solaris.

To point to the `dtrace` program, the environment variable `DTRACE` can be set. This will often be necessary because `dtrace` is typically installed under `/usr/sbin`, which might not be in the path. Additional command-line options for the `dtrace` program can be specified in the environment variable `DTRACEFLAGS`.

To include DTrace support in a 64-bit binary, specify `DTRACEFLAGS="-64"` to configure. For example, using the GCC compiler:

```
./configure CC='gcc -m64' --enable-dtrace DTRACEFLAGS='-64' ...
```

Using Sun's compiler:

```
./configure CC='/opt/SUNWspro/bin/cc -xtarget=native64' --enable-dtrace DTRACEFLAGS=
```

If you prefer a C compiler different from the one `configure` picks, you can set the environment variable `CC` to the program of your choice. By default, `configure` will pick `gcc` if available, else the platform's default (usually `cc`). Similarly, you can override the default compiler flags if needed with the `CFLAGS` variable.

You can specify environment variables on the `configure` command line, for example:

```
./configure CC=/opt/bin/gcc CFLAGS='-O2 -pipe'
```

Here is a list of the significant variables that can be set in this manner:

`CC`

C compiler

`CFLAGS`

options to pass to the C compiler

`CPP`

C preprocessor

`CPPFLAGS`

options to pass to the C preprocessor

`DTRACE`

location of the `dtrace` program

`DTRACEFLAGS`

options to pass to the `dtrace` program

`LDFLAGS`

options to pass to the link editor

`LDFLAGS_SL`

linker options for shared library linking

`MSGFMT`

`msgfmt` program for native language support

`PERL`

Full path to the Perl interpreter. This will be used to determine the dependencies for building PL/Perl.

`PYTHON`

Full path to the Python interpreter. This will be used to determine the dependencies for building PL/Python.

`TCLSH`

Full path to the Tcl interpreter. This wil be used to determine the dependencies for building PL/Tcl.

`YACC`

Yacc program (`bison -y` if using Bison)

2. Build

   To start the build, type

   ```
   gmake
   ```

   (Remember to use GNU make.) The build may take anywhere from 5 minutes to half an hour depending on your hardware. The last line displayed should be

   ```
   All of PostgreSQL is successfully made. Ready to install.
   ```

3. Regression Tests

   If you want to test the newly built server before you install it, you can run the regression tests at this point. The regression tests are a test suite to verify that PostgreSQL runs on your machine in the way the developers expected it to. Type

   ```
   gmake check
   ```

(This won't work as root; do it as an unprivileged user.) Chapter 28 contains detailed information about interpreting the test results. You can repeat this test at any later time by issuing the same command.

4. Installing The Files

> **Note:** If you are upgrading an existing system and are going to install the new files over the old ones, be sure to back up your data and shut down the old server before proceeding, as explained in Section 14.4 above.

To install PostgreSQL enter

```
gmake install
```

This will install files into the directories that were specified in step 1. Make sure that you have appropriate permissions to write into that area. Normally you need to do this step as root. Alternatively, you could create the target directories in advance and arrange for appropriate permissions to be granted.

You can use `gmake install-strip` instead of `gmake install` to strip the executable files and libraries as they are installed. This will save some space. If you built with debugging support, stripping will effectively remove the debugging support, so it should only be done if debugging is no longer needed. `install-strip` tries to do a reasonable job saving space, but it does not have perfect knowledge of how to strip every unneeded byte from an executable file, so if you want to save all the disk space you possibly can, you will have to do manual work.

The standard installation provides all the header files needed for client application development as well as for server-side program development, such as custom functions or data types written in C. (Prior to PostgreSQL 8.0, a separate `gmake install-all-headers` command was needed for the latter, but this step has been folded into the standard install.)

Client-only installation: If you want to install only the client applications and interface libraries, then you can use these commands:

```
gmake -C src/bin install
gmake -C src/include install
gmake -C src/interfaces install
gmake -C doc install
```

`src/bin` has a few binaries for server-only use, but they are small.

Registering eventlog on Windows: To register a Windows eventlog library with the operating system, issue this command after installation:

```
regsvr32 pgsql_library_directory/pgevent.dll
```

This creates registry entries used by the event viewer.

Uninstallation: To undo the installation use the command `gmake uninstall`. However, this will not remove any created directories.

Cleaning: After the installation you can make room by removing the built files from the source tree with the command `gmake clean`. This will preserve the files made by the `configure` program, so that you can rebuild everything with `gmake` later on. To reset the source tree to the state in which it was distributed, use `gmake distclean`. If you are going to build for several platforms within the same source tree you

must do this and re-configure for each build. (Alternatively, use a separate build tree for each platform, so that the source tree remains unmodified.)

If you perform a build and then discover that your `configure` options were wrong, or if you change anything that `configure` investigates (for example, software upgrades), then it's a good idea to do `gmake distclean` before reconfiguring and rebuilding. Without this, your changes in configuration choices may not propagate everywhere they need to.

14.6. Post-Installation Setup

14.6.1. Shared Libraries

On some systems that have shared libraries (which most systems do) you need to tell your system how to find the newly installed shared libraries. The systems on which this is *not* necessary include BSD/OS, FreeBSD, HP-UX, IRIX, Linux, NetBSD, OpenBSD, Tru64 UNIX (formerly Digital UNIX), and Solaris.

The method to set the shared library search path varies between platforms, but the most widely usable method is to set the environment variable `LD_LIBRARY_PATH` like so: In Bourne shells (`sh`, `ksh`, `bash`, `zsh`)

```
LD_LIBRARY_PATH=/usr/local/pgsql/lib
export LD_LIBRARY_PATH
```

or in `csh` or `tcsh`

```
setenv LD_LIBRARY_PATH /usr/local/pgsql/lib
```

Replace `/usr/local/pgsql/lib` with whatever you set `--libdir` to in step 1. You should put these commands into a shell start-up file such as `/etc/profile` or `~/.bash_profile`. Some good information about the caveats associated with this method can be found at http://www.visi.com/~barr/ldpath.html.

On some systems it might be preferable to set the environment variable `LD_RUN_PATH` *before* building.

On Cygwin, put the library directory in the `PATH` or move the `.dll` files into the `bin` directory.

If in doubt, refer to the manual pages of your system (perhaps `ld.so` or `rld`). If you later on get a message like

```
psql: error in loading shared libraries
libpq.so.2.1: cannot open shared object file: No such file or directory
```

then this step was necessary. Simply take care of it then.

If you are on BSD/OS, Linux, or SunOS 4 and you have root access you can run

```
/sbin/ldconfig /usr/local/pgsql/lib
```

(or equivalent directory) after installation to enable the run-time linker to find the shared libraries faster. Refer to the manual page of `ldconfig` for more information. On FreeBSD, NetBSD, and OpenBSD the command is

```
/sbin/ldconfig -m /usr/local/pgsql/lib
```

instead. Other systems are not known to have an equivalent command.

14.6.2. Environment Variables

If you installed into `/usr/local/pgsql` or some other location that is not searched for programs by default, you should add `/usr/local/pgsql/bin` (or whatever you set `--bindir` to in step 1) into your `PATH`. Strictly speaking, this is not necessary, but it will make the use of PostgreSQL much more convenient.

To do this, add the following to your shell start-up file, such as `~/.bash_profile` (or `/etc/profile`, if you want it to affect every user):

```
PATH=/usr/local/pgsql/bin:$PATH
export PATH
```

If you are using `csh` or `tcsh`, then use this command:

```
set path = ( /usr/local/pgsql/bin $path )
```

To enable your system to find the man documentation, you need to add lines like the following to a shell start-up file unless you installed into a location that is searched by default.

```
MANPATH=/usr/local/pgsql/man:$MANPATH
export MANPATH
```

The environment variables `PGHOST` and `PGPORT` specify to client applications the host and port of the database server, overriding the compiled-in defaults. If you are going to run client applications remotely then it is convenient if every user that plans to use the database sets `PGHOST`. This is not required, however: the settings can be communicated via command line options to most client programs.

14.7. Supported Platforms

PostgreSQL has been verified by the developer community to work on the platforms listed below. A supported platform generally means that PostgreSQL builds and installs according to these instructions and that the regression tests pass. “Build farm” entries refer to active test machines in the PostgreSQL Build Farm[1]. Platform entries that show an older version of PostgreSQL are those that did not receive explicit testing at the time of release of version 8.2 but that we still expect to work.

> **Note:** If you are having problems with the installation on a supported platform, please write to <pgsql-bugs@postgresql.org> or <pgsql-ports@postgresql.org>, not to the people listed here.

1. http://buildfarm.postgresql.org/

OS	Processor	Version	Reported	Remarks
AIX	PowerPC	8.2.0	Build farm grebe (5.3, gcc 4.0.1); kookaburra (5.2, cc 6.0); asp (5.2, gcc 3.3.2)	see `doc/FAQ_AIX`, particularly if using AIX 5.3 ML3
AIX	RS6000	8.0.0	Hans-Jürgen Schönig (<hs@cybertec.at>), 2004-12-06	see `doc/FAQ_AIX`
BSD/OS	x86	8.1.0	Bruce Momjian (<pgman@candle.pha.pa.us>), 2005-10-26	4.3.1
Debian GNU/Linux	Alpha	8.2.0	Build farm hare (3.1, gcc 3.3.4)	
Debian GNU/Linux	AMD64	8.2.0	Build farm shad (4.0, gcc 4.1.2); kite (3.1, gcc 4.0); panda (sid, gcc 3.3.5)	
Debian GNU/Linux	ARM	8.2.0	Build farm penguin (3.1, gcc 3.3.4)	
Debian GNU/Linux	Athlon XP	8.2.0	Build farm rook (3.1, gcc 3.3.5)	
Debian GNU/Linux	IA64	8.2.0	Build farm dugong (unstable, icc 9.1.045)	
Debian GNU/Linux	m68k	8.0.0	Noèl Köthe (<noel@debian.org>), 2004-12-09	sid
Debian GNU/Linux	MIPS	8.2.0	Build farm otter (3.1, gcc 3.3.4)	
Debian GNU/Linux	MIPSEL	8.2.0	Build farm lionfish (3.1, gcc 3.3.4); corgi (3.1, gcc 3.3.4)	
Debian GNU/Linux	PA-RISC	8.2.0	Build farm manatee (3.1, gcc 4.0.1); kingfisher (3.1, gcc 3.3.5)	
Debian GNU/Linux	PowerPC	8.0.0	Noèl Köthe (<noel@debian.org>), 2004-12-15	sid

OS	Processor	Version	Reported	Remarks
Debian GNU/Linux	Sparc	8.1.0	Build farm dormouse (3.1, gcc 3.2.5; 64-bit)	
Debian GNU/Linux	x86	8.2.0	Build farm wildebeest (3.1, gcc 3.3.5)	
Fedora Linux	AMD64	8.2.0	Build farm impala (FC6, gcc 4.1.1); bustard (FC5, gcc 4.1.0); wasp (FC5, gcc 4.1.0); viper (FC3, gcc 3.4.4)	
Fedora Linux	PowerPC	8.2.0	Build farm sponge (FC5, gcc 4.1.0)	
Fedora Linux	x86	8.2.0	Build farm agouti (FC5, gcc 4.1.1); thrush (FC1, gcc 3.3.2)	
FreeBSD	AMD64	8.2.0	Build farm platypus (6, gcc 3.4.4); dove (6.1, gcc 3.4.4); ermine (6.1, gcc 3.4.4)	
FreeBSD	x86	8.2.0	Build farm minnow (6.1, gcc 3.4.4); echidna (6, gcc 3.4.2); herring (6, Intel cc 7.1)	
Gentoo Linux	AMD64	8.1.0	Build farm caribou (2.6.9, gcc 3.3.5)	
Gentoo Linux	IA64	8.2.0	Build farm stoat (2.6, gcc 3.3)	
Gentoo Linux	PowerPC 64	8.2.0	Build farm cobra (1.4.16, gcc 3.4.3)	
Gentoo Linux	x86	8.2.0	Build farm mongoose (1.6.14, icc 9.0.032)	
HP-UX	IA64	8.2.0	Tom Lane (<tgl@sss.pgh.pa.us>), 2006-10-23	11.23, `gcc` and `cc`; see `doc/FAQ_HPUX`
HP-UX	PA-RISC	8.2.0	Tom Lane (<tgl@sss.pgh.pa.us>), 2006-10-23	10.20 and 11.23, `gcc` and `cc`; see `doc/FAQ_HPUX`

OS	Processor	Version	Reported	Remarks
IRIX	MIPS	8.1.0	Kenneth Marshall (<ktm@is.rice.edu>), 2005-11-04	6.5, cc only
Kubuntu Linux	AMD64	8.2.0	Build farm rosella (5.10 “Breezy”, gcc 4.0)	
Mac OS X	PowerPC	8.2.0	Build farm tuna (10.4.2, gcc 4.0)	
Mac OS X	x86	8.2.0	Build farm jackal (10.4.8, gcc 4.0.1)	
Mandriva Linux	x86	8.2.0	Build farm gopher (Mandriva 2006, gcc 4.0.1)	
NetBSD	m68k	8.2.0	Build farm osprey (2.0, gcc 3.3.3)	
NetBSD	x86	8.2.0	Build farm gazelle (3.0, gcc 3.3.3); canary (1.6, gcc 2.95.3)	
OpenBSD	AMD64	8.2.0	Build farm zebra (4.0, gcc 3.3.5)	
OpenBSD	Sparc	8.0.0	Chris Mair (<list@1006.org>), 2005-01-10	3.3
OpenBSD	Sparc64	8.2.0	Build farm spoonbill (3.9, gcc 3.3.5)	
OpenBSD	x86	8.2.0	Build farm emu (4.0, gcc 3.3.5); guppy (3.8, gcc 3.3.5)	minor ecpg test failure on 3.8
Red Hat Linux	AMD64	8.1.0	Tom Lane (<tgl@sss.pgh.pa.us>), 2005-10-23	RHEL 4
Red Hat Linux	IA64	8.1.0	Tom Lane (<tgl@sss.pgh.pa.us>), 2005-10-23	RHEL 4
Red Hat Linux	PowerPC	8.1.0	Tom Lane (<tgl@sss.pgh.pa.us>), 2005-10-23	RHEL 4
Red Hat Linux	PowerPC 64	8.1.0	Tom Lane (<tgl@sss.pgh.pa.us>), 2005-10-23	RHEL 4

OS	Processor	Version	Reported	Remarks
Red Hat Linux	S/390	8.1.0	Tom Lane (<tgl@sss.pgh.pa.us>), 2005-10-23	RHEL 4
Red Hat Linux	S/390x	8.1.0	Tom Lane (<tgl@sss.pgh.pa.us>), 2005-10-23	RHEL 4
Red Hat Linux	x86	8.1.0	Tom Lane (<tgl@sss.pgh.pa.us>), 2005-10-23	RHEL 4
Slackware Linux	x86	8.1.0	Sergey Koposov (<math@sai.msu.ru>), 2005-10-24	10.0
Solaris	Sparc	8.2.0	Build farm hyena (Solaris 10, gcc 3.4.3)	see doc/FAQ_Solaris
Solaris	x86	8.2.0	Build farm dragonfly (Solaris 9, gcc 3.2.3); kudu (Solaris 9, cc 5.3)	see doc/FAQ_Solaris
SUSE Linux	AMD64	8.1.0	Josh Berkus (<josh@agliodbs.com>), 2005-10-23	SLES 9.3
SUSE Linux	IA64	8.0.0	Reinhard Max (<max@suse.de>), 2005-01-03	SLES 9
SUSE Linux	PowerPC	8.0.0	Reinhard Max (<max@suse.de>), 2005-01-03	SLES 9
SUSE Linux	PowerPC 64	8.0.0	Reinhard Max (<max@suse.de>), 2005-01-03	SLES 9
SUSE Linux	S/390	8.0.0	Reinhard Max (<max@suse.de>), 2005-01-03	SLES 9
SUSE Linux	S/390x	8.0.0	Reinhard Max (<max@suse.de>), 2005-01-03	SLES 9
SUSE Linux	x86	8.0.0	Reinhard Max (<max@suse.de>), 2005-01-03	9.0, 9.1, 9.2, SLES 9
Tru64 UNIX	Alpha	8.1.0	Honda Shigehiro (<fwif0083@mb.infoweb.ne.jp>), 2005-11-01	5.0, cc 6.1-011

OS	Processor	Version	Reported	Remarks
Ubuntu Linux	x86	8.2.0	Build farm caracara (6.06, gcc 4.0.3)	
UnixWare	x86	8.2.0	Build farm warthog (7.1.4, `cc` 4.2)	see `doc/FAQ_SCO`
Windows	x86	8.2.0	Build farm yak (XP SP2, gcc 3.4.2); bandicoot (Windows 2000 Pro, gcc 3.4.2); snake (Windows Server 2003 SP1, gcc 3.4.2); trout (Windows Server 2000 SP4, gcc 3.4.2)	see `doc/FAQ_MINGW`
Windows with Cygwin	x86	8.2.0	Build farm eel (W2K Server SP4, gcc 3.4.4)	see `doc/FAQ_CYGWIN`
Yellow Dog Linux	PowerPC	8.1.0	Build farm carp (4.0, gcc 3.3.3)	

Unsupported Platforms: The following platforms used to work but have not been tested recently. We include these here to let you know that these platforms *could* be supported if given some attention.

OS	Processor	Version	Reported	Remarks
Debian GNU/Linux	S/390	7.4	Noèl Köthe (<noel@debian.org>), 2003-10-25	
FreeBSD	Alpha	7.4	Peter Eisentraut (<peter_e@gmx.net>), 2003-10-25	4.8
Linux	PlayStation 2	8.0.0	Chris Mair (<list@1006.org>), 2005-01-09	requires `-disable-spinlocks` (works, but very slow)
NetBSD	Alpha	7.2	Thomas Thai (<tom@minnesota.com>), 2001-11-20	1.5W
NetBSD	arm32	7.4	Patrick Welche (<prlw1@newn.cam.ac.uk>), 2003-11-12	1.6ZE/acorn32

OS	Processor	Version	Reported	Remarks
NetBSD	MIPS	7.2.1	Warwick Hunter (<whunter@agile.tv>), 2002-06-13	1.5.3
NetBSD	PowerPC	7.2	Bill Studenmund (<wrstuden@netbsd.org>), 2001-11-28	1.5
NetBSD	Sparc	7.4.1	Peter Eisentraut (<peter_e@gmx.net>), 2003-11-26	1.6.1, 32-bit
NetBSD	VAX	7.1	Tom I. Helbekkmo (<tih@kpnQwest.no>), 2001-03-30	1.5
SCO OpenServer	x86	7.3.1	Shibashish Satpathy (<shib@postmark.net>), 2002-12-11	5.0.4, gcc; see also doc/FAQ_SCO
SunOS 4	Sparc	7.2	Tatsuo Ishii (<t-ishii@sra.co.jp>), 2001-12-04	

Chapter 15. Client-Only Installation on Windows

Although a complete PostgreSQL installation for Windows can only be built using MinGW or Cygwin, the C client library (libpq) and the interactive terminal (psql) can be compiled using other Windows tool sets. Makefiles are included in the source distribution for Microsoft Visual C++ and Borland C++. It should be possible to compile the libraries manually for other configurations.

> **Tip:** Using MinGW or Cygwin is preferred. If using one of those tool sets, see Chapter 14.

To build everything that you can on Windows using Microsoft Visual C++, change into the `src` directory and type the command

```
nmake /f win32.mak
```

This assumes that you have Visual C++ in your path.

To build everything using Borland C++, change into the `src` directory and type the command

```
make -N -DCFG=Release /f bcc32.mak
```

The following files will be built:

`interfaces\libpq\Release\libpq.dll`

The dynamically linkable frontend library

`interfaces\libpq\Release\libpqdll.lib`

Import library to link your programs to `libpq.dll`

`interfaces\libpq\Release\libpq.lib`

Static version of the frontend library

```
bin\pg_config\Release\pg_config.exe
bin\psql\Release\psql.exe
bin\pg_dump\Release\pg_dump.exe
bin\pg_dump\Release\pg_dumpall.exe
bin\pg_dump\Release\pg_restore.exe
bin\scripts\Release\clusterdb.exe
bin\scripts\Release\createdb.exe
bin\scripts\Release\createuser.exe
bin\scripts\Release\createlang.exe
bin\scripts\Release\dropdb.exe
bin\scripts\Release\dropuser.exe
bin\scripts\Release\droplang.exe
bin\scripts\Release\vacuumdb.exe
bin\scripts\Release\reindexdb.exe
```

The PostgreSQL client applications and utilities.

Normally you do not need to install any of the client files. You should place the `libpq.dll` file in the same directory as your applications .EXE-file. Only if this is for some reason not possible should you install it in the `WINNT\SYSTEM32` directory (or in `WINDOWS\SYSTEM` on a Windows 95/98/ME system). If this file is installed using a setup program, it should be installed with version checking using the `VERSIONINFO` resource included in the file, to ensure that a newer version of the library is not overwritten.

If you plan to do development using libpq on this machine, you will have to add the `src\include` and `src\interfaces\libpq` subdirectories of the source tree to the include path in your compiler's settings.

To use the library, you must add the `libpqdll.lib` file to your project. (In Visual C++, just right-click on the project and choose to add it.)

Free development tools from Microsoft can be downloaded from http://msdn.microsoft.com/visualc/vctoolkit2003/. You will also need `MSVCRT.lib` from the platform SDK from http://www.microsoft.com/msdownload/platformsdk/sdkupdate/. You can also download the .NET framework from http://msdn.microsoft.com/netframework/downloads/updates/default.aspx. Once installed, the toolkit binaries must be in your path, and you might need to add a `/lib:<libpath>` to point to `MSVCRT.lib`. Free Borland C++ compiler tools can be downloaded from http://www.borland.com/products/downloads/download_cbuilder.html#, and require similar setup.

Chapter 16. Operating System Environment

This chapter discusses how to set up and run the database server and its interactions with the operating system.

16.1. The PostgreSQL User Account

As with any other server daemon that is accessible to the outside world, it is advisable to run PostgreSQL under a separate user account. This user account should only own the data that is managed by the server, and should not be shared with other daemons. (For example, using the user `nobody` is a bad idea.) It is not advisable to install executables owned by this user because compromised systems could then modify their own binaries.

To add a Unix user account to your system, look for a command `useradd` or `adduser`. The user name postgres is often used, and is assumed throughout this book, but you can use another name if you like.

16.2. Creating a Database Cluster

Before you can do anything, you must initialize a database storage area on disk. We call this a *database cluster*. (SQL uses the term catalog cluster.) A database cluster is a collection of databases that is managed by a single instance of a running database server. After initialization, a database cluster will contain a database named `postgres`, which is meant as a default database for use by utilities, users and third party applications. The database server itself does not require the `postgres` database to exist, but many external utility programs assume it exists. Another database created within each cluster during initialization is called `template1`. As the name suggests, this will be used as a template for subsequently created databases; it should not be used for actual work. (See Chapter 19 for information about creating new databases within a cluster.)

In file system terms, a database cluster will be a single directory under which all data will be stored. We call this the *data directory* or *data area*. It is completely up to you where you choose to store your data. There is no default, although locations such as `/usr/local/pgsql/data` or `/var/lib/pgsql/data` are popular. To initialize a database cluster, use the command initdb, which is installed with PostgreSQL. The desired file system location of your database cluster is indicated by the `-D` option, for example

```
$ initdb -D /usr/local/pgsql/data
```

Note that you must execute this command while logged into the PostgreSQL user account, which is described in the previous section.

> **Tip:** As an alternative to the `-D` option, you can set the environment variable `PGDATA`.

`initdb` will attempt to create the directory you specify if it does not already exist. It is likely that it will not have the permission to do so (if you followed our advice and created an unprivileged account). In that case you should create the directory yourself (as root) and change the owner to be the PostgreSQL user. Here is how this might be done:

```
root# mkdir /usr/local/pgsql/data
root# chown postgres /usr/local/pgsql/data
root# su postgres
postgres$ initdb -D /usr/local/pgsql/data
```

`initdb` will refuse to run if the data directory looks like it has already been initialized.

Because the data directory contains all the data stored in the database, it is essential that it be secured from unauthorized access. `initdb` therefore revokes access permissions from everyone but the PostgreSQL user.

However, while the directory contents are secure, the default client authentication setup allows any local user to connect to the database and even become the database superuser. If you do not trust other local users, we recommend you use one of `initdb`'s `-W`, `--pwprompt` or `--pwfile` options to assign a password to the database superuser. Also, specify `-A md5` or `-A password` so that the default `trust` authentication mode is not used; or modify the generated `pg_hba.conf` file after running `initdb`, *before* you start the server for the first time. (Other reasonable approaches include using `ident` authentication or file system permissions to restrict connections. See Chapter 20 for more information.)

`initdb` also initializes the default locale for the database cluster. Normally, it will just take the locale settings in the environment and apply them to the initialized database. It is possible to specify a different locale for the database; more information about that can be found in Section 21.1. The sort order used within a particular database cluster is set by `initdb` and cannot be changed later, short of dumping all data, rerunning `initdb`, and reloading the data. There is also a performance impact for using locales other than `C` or `POSIX`. Therefore, it is important to make this choice correctly the first time.

`initdb` also sets the default character set encoding for the database cluster. Normally this should be chosen to match the locale setting. For details see Section 21.2.

16.3. Starting the Database Server

Before anyone can access the database, you must start the database server. The database server program is called `postgres`. The `postgres` program must know where to find the data it is supposed to use. This is done with the `-D` option. Thus, the simplest way to start the server is:

```
$ postgres -D /usr/local/pgsql/data
```

which will leave the server running in the foreground. This must be done while logged into the PostgreSQL user account. Without `-D`, the server will try to use the data directory named by the environment variable `PGDATA`. If that variable is not provided either, it will fail.

Normally it is better to start `postgres` in the background. For this, use the usual shell syntax:

```
$ postgres -D /usr/local/pgsql/data >logfile 2>&1 &
```

It is important to store the server's stdout and stderr output somewhere, as shown above. It will help for auditing purposes and to diagnose problems. (See Section 22.3 for a more thorough discussion of log file handling.)

The `postgres` program also takes a number of other command-line options. For more information, see the postgres reference page and Chapter 17 below.

This shell syntax can get tedious quickly. Therefore the wrapper program pg_ctl is provided to simplify some tasks. For example:

```
pg_ctl start -l logfile
```

will start the server in the background and put the output into the named log file. The `-D` option has the same meaning here as for `postgres`. `pg_ctl` is also capable of stopping the server.

Normally, you will want to start the database server when the computer boots. Autostart scripts are operating-system-specific. There are a few distributed with PostgreSQL in the `contrib/start-scripts` directory. Installing one will require root privileges.

Different systems have different conventions for starting up daemons at boot time. Many systems have a file `/etc/rc.local` or `/etc/rc.d/rc.local`. Others use `rc.d` directories. Whatever you do, the server must be run by the PostgreSQL user account *and not by root* or any other user. Therefore you probably should form your commands using `su -c '...' postgres`. For example:

```
su -c 'pg_ctl start -D /usr/local/pgsql/data -l serverlog' postgres
```

Here are a few more operating-system-specific suggestions. (In each case be sure to use the proper installation directory and user name where we show generic values.)

- For FreeBSD, look at the file `contrib/start-scripts/freebsd` in the PostgreSQL source distribution.
- On OpenBSD, add the following lines to the file `/etc/rc.local`:

  ```
  if [ -x /usr/local/pgsql/bin/pg_ctl -a -x /usr/local/pgsql/bin/postgres ]; then
      su - -c '/usr/local/pgsql/bin/pg_ctl start -l /var/postgresql/log -s' postgres
      echo -n ' postgresql'
  fi
  ```

- On Linux systems either add

  ```
  /usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/data
  ```

  to `/etc/rc.d/rc.local` or look at the file `contrib/start-scripts/linux` in the PostgreSQL source distribution.
- On NetBSD, either use the FreeBSD or Linux start scripts, depending on preference.
- On Solaris, create a file called `/etc/init.d/postgresql` that contains the following line:

  ```
  su - postgres -c "/usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/data"
  ```

  Then, create a symbolic link to it in `/etc/rc3.d` as `S99postgresql`.

While the server is running, its PID is stored in the file `postmaster.pid` in the data directory. This is used to prevent multiple server instances from running in the same data directory and can also be used for shutting down the server.

16.3.1. Server Start-up Failures

There are several common reasons the server might fail to start. Check the server's log file, or start it by hand (without redirecting standard output or standard error) and see what error messages appear. Below we explain some of the most common error messages in more detail.

```
LOG:  could not bind IPv4 socket: Address already in use
HINT:  Is another postmaster already running on port 5432? If not, wait a few seconds and r
FATAL:  could not create TCP/IP listen socket
```

This usually means just what it suggests: you tried to start another server on the same port where one is already running. However, if the kernel error message is not `Address already in use` or some variant of that, there may be a different problem. For example, trying to start a server on a reserved port number may draw something like:

```
$ postgres -p 666
LOG:  could not bind IPv4 socket: Permission denied
HINT:  Is another postmaster already running on port 666? If not, wait a few seconds and re
FATAL:  could not create TCP/IP listen socket
```

A message like

```
FATAL:  could not create shared memory segment: Invalid argument
DETAIL:  Failed system call was shmget(key=5440001, size=4011376640, 03600).
```

probably means your kernel's limit on the size of shared memory is smaller than the work area PostgreSQL is trying to create (4011376640 bytes in this example). Or it could mean that you do not have System-V-style shared memory support configured into your kernel at all. As a temporary workaround, you can try starting the server with a smaller-than-normal number of buffers (shared_buffers). You will eventually want to reconfigure your kernel to increase the allowed shared memory size. You may also see this message when trying to start multiple servers on the same machine, if their total space requested exceeds the kernel limit.

An error like

```
FATAL:  could not create semaphores: No space left on device
DETAIL:  Failed system call was semget(5440126, 17, 03600).
```

does *not* mean you've run out of disk space. It means your kernel's limit on the number of System V semaphores is smaller than the number PostgreSQL wants to create. As above, you may be able to work around the problem by starting the server with a reduced number of allowed connections (max_connections), but you'll eventually want to increase the kernel limit.

If you get an "illegal system call" error, it is likely that shared memory or semaphores are not supported in your kernel at all. In that case your only option is to reconfigure the kernel to enable these features.

Details about configuring System V IPC facilities are given in Section 16.4.1.

16.3.2. Client Connection Problems

Although the error conditions possible on the client side are quite varied and application-dependent, a few of them might be directly related to how the server was started up. Conditions other than those shown below should be documented with the respective client application.

```
psql: could not connect to server: Connection refused
        Is the server running on host "server.joe.com" and accepting
        TCP/IP connections on port 5432?
```

This is the generic "I couldn't find a server to talk to" failure. It looks like the above when TCP/IP communication is attempted. A common mistake is to forget to configure the server to allow TCP/IP connections.

Alternatively, you'll get this when attempting Unix-domain socket communication to a local server:

```
psql: could not connect to server: No such file or directory
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

The last line is useful in verifying that the client is trying to connect to the right place. If there is in fact no server running there, the kernel error message will typically be either `Connection refused` or `No such file or directory`, as illustrated. (It is important to realize that `Connection refused` in this context does *not* mean that the server got your connection request and rejected it. That case will produce a different message, as shown in Section 20.3.) Other error messages such as `Connection timed out` may indicate more fundamental problems, like lack of network connectivity.

16.4. Managing Kernel Resources

A large PostgreSQL installation can quickly exhaust various operating system resource limits. (On some systems, the factory defaults are so low that you don't even need a really "large" installation.) If you have encountered this kind of problem, keep reading.

16.4.1. Shared Memory and Semaphores

Shared memory and semaphores are collectively referred to as "System V IPC" (together with message queues, which are not relevant for PostgreSQL). Almost all modern operating systems provide these features, but not all of them have them turned on or sufficiently sized by default, especially systems with BSD heritage. (For the Windows port, PostgreSQL provides its own replacement implementation of these facilities.)

The complete lack of these facilities is usually manifested by an Illegal system call error upon server start. In that case there's nothing left to do but to reconfigure your kernel. PostgreSQL won't work without them.

When PostgreSQL exceeds one of the various hard IPC limits, the server will refuse to start and should leave an instructive error message describing the problem encountered and what to do about it. (See also

Section 16.3.1.) The relevant kernel parameters are named consistently across different systems; Table 16-1 gives an overview. The methods to set them, however, vary. Suggestions for some platforms are given below. Be warned that it is often necessary to reboot your machine, and possibly even recompile the kernel, to change these settings.

Table 16-1. System V IPC parameters

Name	Description	Reasonable values
SHMMAX	Maximum size of shared memory segment (bytes)	at least several megabytes (see text)
SHMMIN	Minimum size of shared memory segment (bytes)	1
SHMALL	Total amount of shared memory available (bytes or pages)	if bytes, same as SHMMAX; if pages, ceil(SHMMAX/PAGE_SIZE)
SHMSEG	Maximum number of shared memory segments per process	only 1 segment is needed, but the default is much higher
SHMMNI	Maximum number of shared memory segments system-wide	like SHMSEG plus room for other applications
SEMMNI	Maximum number of semaphore identifiers (i.e., sets)	at least ceil(max_connections / 16)
SEMMNS	Maximum number of semaphores system-wide	ceil(max_connections / 16) * 17 plus room for other applications
SEMMSL	Maximum number of semaphores per set	at least 17
SEMMAP	Number of entries in semaphore map	see text
SEMVMX	Maximum value of semaphore	at least 1000 (The default is often 32767, don't change unless forced to)

The most important shared memory parameter is SHMMAX, the maximum size, in bytes, of a shared memory segment. If you get an error message from shmget like Invalid argument, it is likely that this limit has been exceeded. The size of the required shared memory segment varies depending on several PostgreSQL configuration parameters, as shown in Table 16-2. You can, as a temporary solution, lower some of those settings to avoid the failure. As a rough approximation, you can estimate the required segment size as 700 kB plus the variable amounts shown in the table. (Any error message you might get will include the exact size of the failed allocation request.) While it is possible to get PostgreSQL to run with SHMMAX as small as 1 MB, you need at least 4 MB for acceptable performance, and desirable settings are in the tens of megabytes.

Some systems also have a limit on the total amount of shared memory in the system (SHMALL). Make sure this is large enough for PostgreSQL plus any other applications that are using shared memory segments. (Caution: SHMALL is measured in pages rather than bytes on many systems.)

Less likely to cause problems is the minimum size for shared memory segments (`SHMMIN`), which should be at most approximately 500 kB for PostgreSQL (it is usually just 1). The maximum number of segments system-wide (`SHMMNI`) or per-process (`SHMSEG`) are unlikely to cause a problem unless your system has them set to zero.

PostgreSQL uses one semaphore per allowed connection (max_connections), in sets of 16. Each such set will also contain a 17th semaphore which contains a "magic number", to detect collision with semaphore sets used by other applications. The maximum number of semaphores in the system is set by `SEMMNS`, which consequently must be at least as high as `max_connections` plus one extra for each 16 allowed connections (see the formula in Table 16-1). The parameter `SEMMNI` determines the limit on the number of semaphore sets that can exist on the system at one time. Hence this parameter must be at least `ceil(max_connections / 16)`. Lowering the number of allowed connections is a temporary workaround for failures, which are usually confusingly worded No space left on device, from the function `semget`.

In some cases it might also be necessary to increase `SEMMAP` to be at least on the order of `SEMMNS`. This parameter defines the size of the semaphore resource map, in which each contiguous block of available semaphores needs an entry. When a semaphore set is freed it is either added to an existing entry that is adjacent to the freed block or it is registered under a new map entry. If the map is full, the freed semaphores get lost (until reboot). Fragmentation of the semaphore space could over time lead to fewer available semaphores than there should be.

The `SEMMSL` parameter, which determines how many semaphores can be in a set, must be at least 17 for PostgreSQL.

Various other settings related to "semaphore undo", such as `SEMMNU` and `SEMUME`, are not of concern for PostgreSQL.

BSD/OS

Shared Memory. By default, only 4 MB of shared memory is supported. Keep in mind that shared memory is not pageable; it is locked in RAM. To increase the amount of shared memory supported by your system, add something like the following to your kernel configuration file:

```
options "SHMALL=8192"
options "SHMMAX=\(SHMALL*PAGE_SIZE\)"
```

`SHMALL` is measured in 4 kB pages, so a value of 1024 represents 4 MB of shared memory. Therefore the above increases the maximum shared memory area to 32 MB. For those running 4.3 or later, you will probably also need to increase `KERNEL_VIRTUAL_MB` above the default `248`. Once all changes have been made, recompile the kernel, and reboot.

For those running 4.0 and earlier releases, use `bpatch` to find the `sysptsize` value in the current kernel. This is computed dynamically at boot time.

```
$ bpatch -r sysptsize
0x9 = 9
```

Next, add `SYSPTSIZE` as a hard-coded value in the kernel configuration file. Increase the value you found using `bpatch`. Add 1 for every additional 4 MB of shared memory you desire.

```
options "SYSPTSIZE=16"
```

`sysptsize` cannot be changed by `sysctl`.

Semaphores. You will probably want to increase the number of semaphores as well; the default system total of 60 will only allow about 50 PostgreSQL connections. Set the values you want in your kernel configuration file, e.g.:

```
options "SEMMNI=40"
options "SEMMNS=240"
```

FreeBSD

The default settings are only suitable for small installations (for example, default SHMMAX is 32 MB). Changes can be made via the sysctl or loader interfaces. The following parameters can be set using sysctl:

```
$ sysctl -w kern.ipc.shmall=32768
$ sysctl -w kern.ipc.shmmax=134217728
$ sysctl -w kern.ipc.semmap=256
```

To have these settings persist over reboots, modify /etc/sysctl.conf.

The remaining semaphore settings are read-only as far as sysctl is concerned, but can be changed before boot using the loader prompt:

```
(loader) set kern.ipc.semmni=256
(loader) set kern.ipc.semmns=512
(loader) set kern.ipc.semmnu=256
```

Similarly these can be saved between reboots in /boot/loader.conf.

You might also want to configure your kernel to lock shared memory into RAM and prevent it from being paged out to swap. This can be accomplished using the sysctl setting kern.ipc.shm_use_phys.

If running in FreeBSD jails by enabling sysctl's security.jail.sysvipc_allowed, postmasters running in different jails should be run by different operating system users. This improves security because it prevents non-root users from interfering with shared memory or semaphores in a different jail, and it allows the PostgreSQL IPC cleanup code to function properly. (In FreeBSD 6.0 and later the IPC cleanup code doesn't properly detect processes in other jails, preventing the running of postmasters on the same port in different jails.)

FreeBSD versions before 4.0 work like NetBSD and OpenBSD (see below).

NetBSD
OpenBSD

The options SYSVSHM and SYSVSEM need to be enabled when the kernel is compiled. (They are by default.) The maximum size of shared memory is determined by the option SHMMAXPGS (in pages). The following shows an example of how to set the various parameters (OpenBSD uses option instead):

```
options         SYSVSHM
options         SHMMAXPGS=4096
options         SHMSEG=256

options         SYSVSEM
options         SEMMNI=256
options         SEMMNS=512
options         SEMMNU=256
options         SEMMAP=256
```

You might also want to configure your kernel to lock shared memory into RAM and prevent it from being paged out to swap. This can be accomplished using the sysctl setting kern.ipc.shm_use_phys.

HP-UX

The default settings tend to suffice for normal installations. On HP-UX 10, the factory default for `SEMMNS` is 128, which might be too low for larger database sites.

IPC parameters can be set in the System Administration Manager (SAM) under Kernel Configuration⟶Configurable Parameters. Hit Create A New Kernel when you're done.

Linux

The default settings are only suitable for small installations (the default max segment size is 32 MB). However the remaining defaults are quite generously sized, and usually do not require changes. The max segment size can be changed via the `sysctl` interface. For example, to allow 128 MB, and explicitly set the maximum total shared memory size to 2097152 pages (the default):

```
$ sysctl -w kernel.shmmax=134217728
$ sysctl -w kernel.shmall=2097152
```

In addition these settings can be saved between reboots in `/etc/sysctl.conf`.

Older distributions may not have the `sysctl` program, but equivalent changes can be made by manipulating the `/proc` file system:

```
$ echo 134217728 >/proc/sys/kernel/shmmax
$ echo 2097152 >/proc/sys/kernel/shmall
```

MacOS X

In OS X 10.2 and earlier, edit the file `/System/Library/StartupItems/SystemTuning/SystemTuning` and change the values in the following commands:

```
sysctl -w kern.sysv.shmmax
sysctl -w kern.sysv.shmmin
sysctl -w kern.sysv.shmmni
sysctl -w kern.sysv.shmseg
sysctl -w kern.sysv.shmall
```

In OS X 10.3 and later, these commands have been moved to `/etc/rc` and must be edited there. Note that `/etc/rc` is usually overwritten by OS X updates (such as 10.3.6 to 10.3.7) so you should expect to have to redo your editing after each update.

In OS X 10.3.9 and later, instead of editing `/etc/rc` you may create a file named `/etc/sysctl.conf`, containing variable assignments such as

```
kern.sysv.shmmax=4194304
kern.sysv.shmmin=1
kern.sysv.shmmni=32
kern.sysv.shmseg=8
kern.sysv.shmall=1024
```

This method is better than editing `/etc/rc` because your changes will be preserved across system updates. Note that *all five* shared-memory parameters must be set in `/etc/sysctl.conf`, else the values will be ignored.

Beware that recent releases of OS X ignore attempts to set `SHMMAX` to a value that isn't an exact multiple of 4096.

`SHMALL` is measured in 4 kB pages on this platform.

In all OS X versions, you'll need to reboot to make changes in the shared memory parameters take effect.

SCO OpenServer

In the default configuration, only 512 kB of shared memory per segment is allowed. To increase the setting, first change to the directory `/etc/conf/cf.d`. To display the current value of `SHMMAX`, run

```
./configure -y SHMMAX
```

To set a new value for `SHMMAX`, run

```
./configure SHMMAX=value
```

where *`value`* is the new value you want to use (in bytes). After setting `SHMMAX`, rebuild the kernel:

```
./link_unix
```

and reboot.

AIX

At least as of version 5.1, it should not be necessary to do any special configuration for such parameters as `SHMMAX`, as it appears this is configured to allow all memory to be used as shared memory. That is the sort of configuration commonly used for other databases such as DB/2.

It may, however, be necessary to modify the global `ulimit` information in `/etc/security/limits`, as the default hard limits for file sizes (`fsize`) and numbers of files (`nofiles`) may be too low.

Solaris

At least in version 2.6, the default maximum size of a shared memory segments is too low for PostgreSQL. The relevant settings can be changed in `/etc/system`, for example:

```
set shmsys:shminfo_shmmax=0x2000000
set shmsys:shminfo_shmmin=1
set shmsys:shminfo_shmmni=256
set shmsys:shminfo_shmseg=256

set semsys:seminfo_semmap=256
set semsys:seminfo_semmni=512
set semsys:seminfo_semmns=512
set semsys:seminfo_semmsl=32
```

You need to reboot for the changes to take effect.

See also http://sunsite.uakom.sk/sunworldonline/swol-09-1997/swol-09-insidesolaris.html for information on shared memory under Solaris.

UnixWare

On UnixWare 7, the maximum size for shared memory segments is only 512 kB in the default configuration. To display the current value of `SHMMAX`, run

```
/etc/conf/bin/idtune -g SHMMAX
```

which displays the current, default, minimum, and maximum values. To set a new value for `SHMMAX`, run

```
/etc/conf/bin/idtune SHMMAX value
```

where *`value`* is the new value you want to use (in bytes). After setting `SHMMAX`, rebuild the kernel:

```
/etc/conf/bin/idbuild -B
```

and reboot.

Table 16-2. Configuration parameters affecting PostgreSQL's shared memory usage

Name	Approximate multiplier (bytes per increment)
max_connections	1800 + 270 * max_locks_per_transaction
max_prepared_transactions	700 + 270 * max_locks_per_transaction
shared_buffers	8300 (assuming 8K `BLCKSZ`)
wal_buffers	8200 (assuming 8K `XLOG_BLCKSZ`)
max_fsm_relations	70
max_fsm_pages	6

16.4.2. Resource Limits

Unix-like operating systems enforce various kinds of resource limits that might interfere with the operation of your PostgreSQL server. Of particular importance are limits on the number of processes per user, the number of open files per process, and the amount of memory available to each process. Each of these have a "hard" and a "soft" limit. The soft limit is what actually counts but it can be changed by the user up to the hard limit. The hard limit can only be changed by the root user. The system call `setrlimit` is responsible for setting these parameters. The shell's built-in command `ulimit` (Bourne shells) or `limit` (csh) is used to control the resource limits from the command line. On BSD-derived systems the file `/etc/login.conf` controls the various resource limits set during login. See the operating system documentation for details. The relevant parameters are `maxproc`, `openfiles`, and `datasize`. For example:

```
default:\
...
        :datasize-cur=256M:\
        :maxproc-cur=256:\
        :openfiles-cur=256:\
...
```

(`-cur` is the soft limit. Append `-max` to set the hard limit.)

Kernels can also have system-wide limits on some resources.

- On Linux `/proc/sys/fs/file-max` determines the maximum number of open files that the kernel will support. It can be changed by writing a different number into the file or by adding an assignment in `/etc/sysctl.conf`. The maximum limit of files per process is fixed at the time the kernel is compiled; see `/usr/src/linux/Documentation/proc.txt` for more information.

The PostgreSQL server uses one process per connection so you should provide for at least as many processes as allowed connections, in addition to what you need for the rest of your system. This is usually not a problem but if you run several servers on one machine things might get tight.

The factory default limit on open files is often set to "socially friendly" values that allow many users to coexist on a machine without using an inappropriate fraction of the system resources. If you run many

servers on a machine this is perhaps what you want, but on dedicated servers you may want to raise this limit.

On the other side of the coin, some systems allow individual processes to open large numbers of files; if more than a few processes do so then the system-wide limit can easily be exceeded. If you find this happening, and you do not want to alter the system-wide limit, you can set PostgreSQL's max_files_per_process configuration parameter to limit the consumption of open files.

16.4.3. Linux Memory Overcommit

In Linux 2.4 and later, the default virtual memory behavior is not optimal for PostgreSQL. Because of the way that the kernel implements memory overcommit, the kernel may terminate the PostgreSQL server (the master server process) if the memory demands of another process cause the system to run out of virtual memory.

If this happens, you will see a kernel message that looks like this (consult your system documentation and configuration on where to look for such a message):

```
Out of Memory: Killed process 12345 (postgres).
```

This indicates that the `postgres` process has been terminated due to memory pressure. Although existing database connections will continue to function normally, no new connections will be accepted. To recover, PostgreSQL will need to be restarted.

One way to avoid this problem is to run PostgreSQL on a machine where you can be sure that other processes will not run the machine out of memory.

On Linux 2.6 and later, a better solution is to modify the kernel's behavior so that it will not "overcommit" memory. This is done by selecting strict overcommit mode via `sysctl`:

```
sysctl -w vm.overcommit_memory=2
```

or placing an equivalent entry in `/etc/sysctl.conf`. You may also wish to modify the related setting `vm.overcommit_ratio`. For details see the kernel documentation file `Documentation/vm/overcommit-accounting`.

Some vendors' Linux 2.4 kernels are reported to have early versions of the 2.6 overcommit `sysctl` parameter. However, setting `vm.overcommit_memory` to 2 on a kernel that does not have the relevant code will make things worse not better. It is recommended that you inspect the actual kernel source code (see the function `vm_enough_memory` in the file `mm/mmap.c`) to verify what is supported in your copy before you try this in a 2.4 installation. The presence of the `overcommit-accounting` documentation file should *not* be taken as evidence that the feature is there. If in any doubt, consult a kernel expert or your kernel vendor.

16.5. Shutting Down the Server

There are several ways to shut down the database server. You control the type of shutdown by sending

different signals to the master `postgres` process.

SIGTERM

After receiving SIGTERM, the server disallows new connections, but lets existing sessions end their work normally. It shuts down only after all of the sessions terminate normally. This is the *Smart Shutdown*.

SIGINT

The server disallows new connections and sends all existing server processes SIGTERM, which will cause them to abort their current transactions and exit promptly. It then waits for the server processes to exit and finally shuts down. This is the *Fast Shutdown*.

SIGQUIT

This is the *Immediate Shutdown*, which will cause the master `postgres` process to send a SIGQUIT to all child processes and exit immediately, without properly shutting itself down. The child processes likewise exit immediately upon receiving SIGQUIT. This will lead to recovery (by replaying the WAL log) upon next start-up. This is recommended only in emergencies.

The pg_ctl program provides a convenient interface for sending these signals to shut down the server.

Alternatively, you can send the signal directly using `kill`. The PID of the `postgres` process can be found using the `ps` program, or from the file `postmaster.pid` in the data directory. For example, to do a fast shutdown:

```
$ kill -INT `head -1 /usr/local/pgsql/data/postmaster.pid`
```

Important: It is best not to use SIGKILL to shut down the server. Doing so will prevent the server from releasing shared memory and semaphores, which may then have to be done manually before a new server can be started. Furthermore, SIGKILL kills the `postgres` process without letting it relay the signal to its subprocesses, so it will be necessary to kill the individual subprocesses by hand as well.

16.6. Encryption Options

PostgreSQL offers encryption at several levels, and provides flexibility in protecting data from disclosure due to database server theft, unscrupulous administrators, and insecure networks. Encryption might also be required to secure sensitive data such as medical records or financial transactions.

Password Storage Encryption

By default, database user passwords are stored as MD5 hashes, so the administrator cannot determine the actual password assigned to the user. If MD5 encryption is used for client authentication, the unencrypted password is never even temporarily present on the server because the client MD5 encrypts it before being sent across the network.

Encryption For Specific Columns

The `/contrib` function library `pgcrypto` allows certain fields to be stored encrypted. This is useful if only some of the data is sensitive. The client supplies the decryption key and the data is decrypted on the server and then sent to the client.

The decrypted data and the decryption key are present on the server for a brief time while it is being decrypted and communicated between the client and server. This presents a brief moment where the data and keys can be intercepted by someone with complete access to the database server, such as the system administrator.

Data Partition Encryption

On Linux, encryption can be layered on top of a file system mount using a "loopback device". This allows an entire file system partition be encrypted on disk, and decrypted by the operating system. On FreeBSD, the equivalent facility is called GEOM Based Disk Encryption, or gbde.

This mechanism prevents unencrypted data from being read from the drives if the drives or the entire computer is stolen. This does not protect against attacks while the file system is mounted, because when mounted, the operating system provides an unencrypted view of the data. However, to mount the file system, you need some way for the encryption key to be passed to the operating system, and sometimes the key is stored somewhere on the host that mounts the disk.

Encrypting Passwords Across A Network

The `MD5` authentication method double-encrypts the password on the client before sending it to the server. It first MD5 encrypts it based on the user name, and then encrypts it based on a random salt sent by the server when the database connection was made. It is this double-encrypted value that is sent over the network to the server. Double-encryption not only prevents the password from being discovered, it also prevents another connection from using the same encrypted password to connect to the database server at a later time.

Encrypting Data Across A Network

SSL connections encrypt all data sent across the network: the password, the queries, and the data returned. The `pg_hba.conf` file allows administrators to specify which hosts can use non-encrypted connections (`host`) and which require SSL-encrypted connections (`hostssl`). Also, clients can specify that they connect to servers only via SSL. Stunnel or SSH can also be used to encrypt transmissions.

SSL Host Authentication

It is possible for both the client and server to provide SSL keys or certificates to each other. It takes some extra configuration on each side, but this provides stronger verification of identity than the mere use of passwords. It prevents a computer from pretending to be the server just long enough to read the password send by the client. It also helps prevent "man in the middle" attacks where a computer between the client and server pretends to be the server and reads and passes all data between the client and server.

Client-Side Encryption

If the system administrator cannot be trusted, it is necessary for the client to encrypt the data; this way, unencrypted data never appears on the database server. Data is encrypted on the client before being sent to the server, and database results have to be decrypted on the client before being used.

16.7. Secure TCP/IP Connections with SSL

PostgreSQL has native support for using SSL connections to encrypt client/server communications for increased security. This requires that OpenSSL is installed on both client and server systems and that support in PostgreSQL is enabled at build time (see Chapter 14).

With SSL support compiled in, the PostgreSQL server can be started with SSL enabled by setting the parameter ssl to `on` in `postgresql.conf`. When starting in SSL mode, the server will look for the files `server.key` and `server.crt` in the data directory, which must contain the server private key and certificate, respectively. These files must be set up correctly before an SSL-enabled server can start. If the private key is protected with a passphrase, the server will prompt for the passphrase and will not start until it has been entered.

The server will listen for both standard and SSL connections on the same TCP port, and will negotiate with any connecting client on whether to use SSL. By default, this is at the client's option; see Section 20.1 about how to set up the server to require use of SSL for some or all connections.

For details on how to create your server private key and certificate, refer to the OpenSSL documentation. A self-signed certificate can be used for testing, but a certificate signed by a certificate authority (CA) (either one of the global CAs or a local one) should be used in production so the client can verify the server's identity. To create a quick self-signed certificate, use the following OpenSSL command:

```
openssl req -new -text -out server.req
```

Fill out the information that `openssl` asks for. Make sure that you enter the local host name as "Common Name"; the challenge password can be left blank. The program will generate a key that is passphrase protected; it will not accept a passphrase that is less than four characters long. To remove the passphrase (as you must if you want automatic start-up of the server), run the commands

```
openssl rsa -in privkey.pem -out server.key
rm privkey.pem
```

Enter the old passphrase to unlock the existing key. Now do

```
openssl req -x509 -in server.req -text -key server.key -out server.crt
chmod og-rwx server.key
```

to turn the certificate into a self-signed certificate and to copy the key and certificate to where the server will look for them.

If verification of client certificates is required, place the certificates of the CA(s) you wish to check for in the file `root.crt` in the data directory. When present, a client certificate will be requested from the client during SSL connection startup, and it must have been signed by one of the certificates present in `root.crt`. (See Section 29.16 for a description of how to set up client certificates.) Certificate Revocation List (CRL) entries are also checked if the file `root.crl` exists.

When the `root.crt` file is not present, client certificates will not be requested or checked. In this mode, SSL provides communication security but not authentication.

The files `server.key`, `server.crt`, `root.crt`, and `root.crl` are only examined during server start; so you must restart the server to make changes in them take effect.

16.8. Secure TCP/IP Connections with SSH Tunnels

One can use SSH to encrypt the network connection between clients and a PostgreSQL server. Done properly, this provides an adequately secure network connection, even for non-SSL-capable clients.

First make sure that an SSH server is running properly on the same machine as the PostgreSQL server and that you can log in using `ssh` as some user. Then you can establish a secure tunnel with a command like this from the client machine:

```
ssh -L 3333:foo.com:5432 joe@foo.com
```

The first number in the `-L` argument, 3333, is the port number of your end of the tunnel; it can be chosen freely. The second number, 5432, is the remote end of the tunnel: the port number your server is using. The name or IP address between the port numbers is the host with the database server you are going to connect to. In order to connect to the database server using this tunnel, you connect to port 3333 on the local machine:

```
psql -h localhost -p 3333 postgres
```

To the database server it will then look as though you are really user `joe@foo.com` and it will use whatever authentication procedure was configured for connections from this user and host. Note that the server will not think the connection is SSL-encrypted, since in fact it is not encrypted between the SSH server and the PostgreSQL server. This should not pose any extra security risk as long as they are on the same machine.

In order for the tunnel setup to succeed you must be allowed to connect via `ssh` as `joe@foo.com`, just as if you had attempted to use `ssh` to set up a terminal session.

> **Tip:** Several other applications exist that can provide secure tunnels using a procedure similar in concept to the one just described.

Chapter 17. Server Configuration

There are many configuration parameters that affect the behavior of the database system. In the first section of this chapter, we describe how to set configuration parameters. The subsequent sections discuss each parameter in detail.

17.1. Setting Parameters

All parameter names are case-insensitive. Every parameter takes a value of one of four types: Boolean, integer, floating point, or string. Boolean values may be written as `ON`, `OFF`, `TRUE`, `FALSE`, `YES`, `NO`, `1`, `0` (all case-insensitive) or any unambiguous prefix of these.

Some settings specify a memory or time value. Each of these has an implicit unit, which is either kilobytes, blocks (typically eight kilobytes), milliseconds, seconds, or minutes. Default units can be queried by referencing `pg_settings.unit`. For convenience, a different unit can also be specified explicitly. Valid memory units are `kB` (kilobytes), `MB` (megabytes), and `GB` (gigabytes); valid time units are `ms` (milliseconds), `s` (seconds), `min` (minutes), `h` (hours), and `d` (days). Note that the multiplier for memory units is 1024, not 1000.

One way to set these parameters is to edit the file `postgresql.conf`, which is normally kept in the data directory. (initdb installs a default copy there.) An example of what this file might look like is:

```
# This is a comment
log_connections = yes
log_destination = 'syslog'
search_path = '"$user", public'
shared_buffers = 128MB
```

One parameter is specified per line. The equal sign between name and value is optional. Whitespace is insignificant and blank lines are ignored. Hash marks (`#`) introduce comments anywhere. Parameter values that are not simple identifiers or numbers must be single-quoted. To embed a single quote in a parameter value, write either two quotes (preferred) or backslash-quote.

In addition to parameter settings, the `postgresql.conf` file can contain *include directives*, which specify another file to read and process as if it were inserted into the configuration file at this point. Include directives simply look like

```
include 'filename'
```

If the file name is not an absolute path, it is taken as relative to the directory containing the referencing configuration file. Inclusions can be nested.

The configuration file is reread whenever the main server process receives a SIGHUP signal (which is most easily sent by means of `pg_ctl reload`). The main server process also propagates this signal to all currently running server processes so that existing sessions also get the new value. Alternatively, you can send the signal to a single server process directly. Some parameters can only be set at server start; any changes to their entries in the configuration file will be ignored until the server is restarted.

A second way to set these configuration parameters is to give them as a command-line option to the `postgres` command, such as:

```
postgres -c log_connections=yes -c log_destination='syslog'
```

Command-line options override any conflicting settings in `postgresql.conf`. Note that this means you won't be able to change the value on-the-fly by editing `postgresql.conf`, so while the command-line method may be convenient, it can cost you flexibility later.

Occasionally it is useful to give a command line option to one particular session only. The environment variable `PGOPTIONS` can be used for this purpose on the client side:

```
env PGOPTIONS='-c geqo=off' psql
```

(This works for any libpq-based client application, not just psql.) Note that this won't work for parameters that are fixed when the server is started or that must be specified in `postgresql.conf`.

Furthermore, it is possible to assign a set of parameter settings to a user or a database. Whenever a session is started, the default settings for the user and database involved are loaded. The commands *ALTER USER* and *ALTER DATABASE*, respectively, are used to configure these settings. Per-database settings override anything received from the `postgres` command-line or the configuration file, and in turn are overridden by per-user settings; both are overridden by per-session settings.

Some parameters can be changed in individual SQL sessions with the *SET* command, for example:

```
SET ENABLE_SEQSCAN TO OFF;
```

If `SET` is allowed, it overrides all other sources of values for the parameter. Some parameters cannot be changed via `SET`: for example, if they control behavior that cannot be changed without restarting the entire PostgreSQL server. Also, some parameters can be modified via `SET` or `ALTER` by superusers, but not by ordinary users.

The *SHOW* command allows inspection of the current values of all parameters.

The virtual table `pg_settings` (described in Section 43.44) also allows displaying and updating session run-time parameters. It is equivalent to `SHOW` and `SET`, but can be more convenient to use because it can be joined with other tables, or selected from using any desired selection condition.

17.2. File Locations

In addition to the `postgresql.conf` file already mentioned, PostgreSQL uses two other manually-edited configuration files, which control client authentication (their use is discussed in Chapter 20). By default, all three configuration files are stored in the database cluster's data directory. The parameters described in this section allow the configuration files to be placed elsewhere. (Doing so can ease administration. In particular it is often easier to ensure that the configuration files are properly backed-up when they are kept separate.)

`data_directory` (`string`)

Specifies the directory to use for data storage. This parameter can only be set at server start.

`config_file` (`string`)

Specifies the main server configuration file (customarily called `postgresql.conf`). This parameter can only be set on the `postgres` command line.

`hba_file` (`string`)

Specifies the configuration file for host-based authentication (customarily called `pg_hba.conf`). This parameter can only be set at server start.

`ident_file` (`string`)

Specifies the configuration file for ident authentication (customarily called `pg_ident.conf`). This parameter can only be set at server start.

`external_pid_file` (`string`)

Specifies the name of an additional process-id (PID) file that the server should create for use by server administration programs. This parameter can only be set at server start.

In a default installation, none of the above parameters are set explicitly. Instead, the data directory is specified by the `-D` command-line option or the `PGDATA` environment variable, and the configuration files are all found within the data directory.

If you wish to keep the configuration files elsewhere than the data directory, the `postgres -D` command-line option or `PGDATA` environment variable must point to the directory containing the configuration files, and the `data_directory` parameter must be set in `postgresql.conf` (or on the command line) to show where the data directory is actually located. Notice that `data_directory` overrides `-D` and `PGDATA` for the location of the data directory, but not for the location of the configuration files.

If you wish, you can specify the configuration file names and locations individually using the parameters `config_file`, `hba_file` and/or `ident_file`. `config_file` can only be specified on the `postgres` command line, but the others can be set within the main configuration file. If all three parameters plus `data_directory` are explicitly set, then it is not necessary to specify `-D` or `PGDATA`.

When setting any of these parameters, a relative path will be interpreted with respect to the directory in which `postgres` is started.

17.3. Connections and Authentication

17.3.1. Connection Settings

`listen_addresses` (`string`)

Specifies the TCP/IP address(es) on which the server is to listen for connections from client applications. The value takes the form of a comma-separated list of host names and/or numeric IP addresses. The special entry `*` corresponds to all available IP interfaces. If the list is empty, the server does not listen on any IP interface at all, in which case only Unix-domain sockets can be used to connect to it. The default value is localhost, which allows only local "loopback" connections to be made. This parameter can only be set at server start.

`port` (`integer`)

The TCP port the server listens on; 5432 by default. Note that the same port number is used for all IP addresses the server listens on. This parameter can only be set at server start.

`max_connections` (`integer`)

Determines the maximum number of concurrent connections to the database server. The default is typically 100 connections, but may be less if your kernel settings will not support it (as determined during initdb). This parameter can only be set at server start.

Increasing this parameter may cause PostgreSQL to request more System V shared memory or semaphores than your operating system's default configuration allows. See Section 16.4.1 for information on how to adjust those parameters, if necessary.

`superuser_reserved_connections` (`integer`)

Determines the number of connection "slots" that are reserved for connections by PostgreSQL superusers. At most max_connections connections can ever be active simultaneously. Whenever the number of active concurrent connections is at least `max_connections` minus `superuser_reserved_connections`, new connections will be accepted only for superusers.

The default value is three connections. The value must be less than the value of `max_connections`. This parameter can only be set at server start.

`unix_socket_directory` (`string`)

Specifies the directory of the Unix-domain socket on which the server is to listen for connections from client applications. The default is normally `/tmp`, but can be changed at build time. This parameter can only be set at server start.

`unix_socket_group` (`string`)

Sets the owning group of the Unix-domain socket. (The owning user of the socket is always the user that starts the server.) In combination with the parameter `unix_socket_permissions` this can be used as an additional access control mechanism for Unix-domain connections. By default this is the empty string, which selects the default group for the current user. This parameter can only be set at server start.

`unix_socket_permissions` (`integer`)

Sets the access permissions of the Unix-domain socket. Unix-domain sockets use the usual Unix file system permission set. The parameter value is expected to be a numeric mode specification in the form accepted by the `chmod` and `umask` system calls. (To use the customary octal format the number must start with a `0` (zero).)

The default permissions are `0777`, meaning anyone can connect. Reasonable alternatives are `0770` (only user and group, see also `unix_socket_group`) and `0700` (only user). (Note that for a Unix-domain socket, only write permission matters and so there is no point in setting or revoking read or execute permissions.)

This access control mechanism is independent of the one described in Chapter 20.

This parameter can only be set at server start.

`bonjour_name` (`string`)

Specifies the Bonjour broadcast name. The computer name is used if this parameter is set to the empty string `''` (which is the default). This parameter is ignored if the server was not compiled with Bonjour support. This parameter can only be set at server start.

tcp_keepalives_idle (integer)

On systems that support the TCP_KEEPIDLE socket option, specifies the number of seconds between sending keepalives on an otherwise idle connection. A value of zero uses the system default. If TCP_KEEPIDLE is not supported, this parameter must be zero. This parameter is ignored for connections made via a Unix-domain socket.

tcp_keepalives_interval (integer)

On systems that support the TCP_KEEPINTVL socket option, specifies how long, in seconds, to wait for a response to a keepalive before retransmitting. A value of zero uses the system default. If TCP_KEEPINTVL is not supported, this parameter must be zero. This parameter is ignored for connections made via a Unix-domain socket.

tcp_keepalives_count (integer)

On systems that support the TCP_KEEPCNT socket option, specifies how many keepalives may be lost before the connection is considered dead. A value of zero uses the system default. If TCP_KEEPCNT is not supported, this parameter must be zero. This parameter is ignored for connections made via a Unix-domain socket.

17.3.2. Security and Authentication

authentication_timeout (integer)

Maximum time to complete client authentication, in seconds. If a would-be client has not completed the authentication protocol in this much time, the server breaks the connection. This prevents hung clients from occupying a connection indefinitely. The default is one minute (1m). This parameter can only be set in the postgresql.conf file or on the server command line.

ssl (boolean)

Enables SSL connections. Please read Section 16.7 before using this. The default is off. This parameter can only be set at server start.

password_encryption (boolean)

When a password is specified in *CREATE USER* or *ALTER USER* without writing either ENCRYPTED or UNENCRYPTED, this parameter determines whether the password is to be encrypted. The default is on (encrypt the password).

krb_server_keyfile (string)

Sets the location of the Kerberos server key file. See Section 20.2.3 for details. This parameter can only be set at server start.

krb_srvname (string)

Sets the Kerberos service name. See Section 20.2.3 for details. This parameter can only be set at server start.

krb_server_hostname (string)

Sets the host name part of the service principal. This, combined with krb_srvname, is used to generate the complete service principal, that is krb_srvname/krb_server_hostname@REALM.

If not set, the default is the server host name. See Section 20.2.3 for details. This parameter can only be set at server start.

`krb_caseins_users` (`boolean`)

Sets whether Kerberos user names should be treated case-insensitively. The default is `off` (case sensitive). This parameter can only be set at server start.

`db_user_namespace` (`boolean`)

This parameter enables per-database user names. It is off by default. This parameter can only be set in the `postgresql.conf` file or on the server command line.

If this is on, you should create users as `username@dbname`. When `username` is passed by a connecting client, `@` and the database name are appended to the user name and that database-specific user name is looked up by the server. Note that when you create users with names containing `@` within the SQL environment, you will need to quote the user name.

With this parameter enabled, you can still create ordinary global users. Simply append `@` when specifying the user name in the client. The `@` will be stripped off before the user name is looked up by the server.

> **Note:** This feature is intended as a temporary measure until a complete solution is found. At that time, this option will be removed.

17.4. Resource Consumption

17.4.1. Memory

`shared_buffers` (`integer`)

Sets the amount of memory the database server uses for shared memory buffers. The default is typically 32 megabytes (`32MB`), but may be less if your kernel settings will not support it (as determined during initdb). This setting must be at least 128 kilobytes and at least 16 kilobytes times max_connections. (Non-default values of `BLCKSZ` change the minimum.) However, settings significantly higher than the minimum are usually needed for good performance. Several tens of megabytes are recommended for production installations. This parameter can only be set at server start.

Increasing this parameter may cause PostgreSQL to request more System V shared memory than your operating system's default configuration allows. See Section 16.4.1 for information on how to adjust those parameters, if necessary.

`temp_buffers` (`integer`)

Sets the maximum number of temporary buffers used by each database session. These are session-local buffers used only for access to temporary tables. The default is eight megabytes (`8MB`). The

setting can be changed within individual sessions, but only up until the first use of temporary tables within a session; subsequent attempts to change the value will have no effect on that session.

A session will allocate temporary buffers as needed up to the limit given by `temp_buffers`. The cost of setting a large value in sessions that do not actually need a lot of temporary buffers is only a buffer descriptor, or about 64 bytes, per increment in `temp_buffers`. However if a buffer is actually used an additional 8192 bytes will be consumed for it (or in general, `BLCKSZ` bytes).

`max_prepared_transactions` (`integer`)

Sets the maximum number of transactions that can be in the "prepared" state simultaneously (see *PREPARE TRANSACTION*). Setting this parameter to zero disables the prepared-transaction feature. The default is five transactions. This parameter can only be set at server start.

If you are not using prepared transactions, this parameter may as well be set to zero. If you are using them, you will probably want `max_prepared_transactions` to be at least as large as max_connections, to avoid unwanted failures at the prepare step.

Increasing this parameter may cause PostgreSQL to request more System V shared memory than your operating system's default configuration allows. See Section 16.4.1 for information on how to adjust those parameters, if necessary.

`work_mem` (`integer`)

Specifies the amount of memory to be used by internal sort operations and hash tables before switching to temporary disk files. The value is defaults to one megabyte (`1MB`). Note that for a complex query, several sort or hash operations might be running in parallel; each one will be allowed to use as much memory as this value specifies before it starts to put data into temporary files. Also, several running sessions could be doing such operations concurrently. So the total memory used could be many times the value of `work_mem`; it is necessary to keep this fact in mind when choosing the value. Sort operations are used for `ORDER BY`, `DISTINCT`, and merge joins. Hash tables are used in hash joins, hash-based aggregation, and hash-based processing of `IN` subqueries.

`maintenance_work_mem` (`integer`)

Specifies the maximum amount of memory to be used in maintenance operations, such as `VACUUM`, `CREATE INDEX`, and `ALTER TABLE ADD FOREIGN KEY`. It defaults to 16 megabytes (`16MB`). Since only one of these operations can be executed at a time by a database session, and an installation normally doesn't have many of them running concurrently, it's safe to set this value significantly larger than `work_mem`. Larger settings may improve performance for vacuuming and for restoring database dumps.

`max_stack_depth` (`integer`)

Specifies the maximum safe depth of the server's execution stack. The ideal setting for this parameter is the actual stack size limit enforced by the kernel (as set by `ulimit -s` or local equivalent), less a safety margin of a megabyte or so. The safety margin is needed because the stack depth is not checked in every routine in the server, but only in key potentially-recursive routines such as expression evaluation. The default setting is two megabytes (`2MB`), which is conservatively small and unlikely to risk crashes. However, it may be too small to allow execution of complex functions. Only superusers can change this setting.

Setting `max_stack_depth` higher than the actual kernel limit will mean that a runaway recursive function can crash an individual backend process. On platforms where PostgreSQL can determine

the kernel limit, it will not let you set this variable to an unsafe value. However, not all platforms provide the information, so caution is recommended in selecting a value.

17.4.2. Free Space Map

These parameters control the size of the shared *free space map*, which tracks the locations of unused space in the database. An undersized free space map may cause the database to consume increasing amounts of disk space over time, because free space that is not in the map cannot be re-used; instead PostgreSQL will request more disk space from the operating system when it needs to store new data. The last few lines displayed by a database-wide `VACUUM VERBOSE` command can help in determining if the current settings are adequate. A `NOTICE` message is also printed during such an operation if the current settings are too low.

Increasing these parameters may cause PostgreSQL to request more System V shared memory than your operating system's default configuration allows. See Section 16.4.1 for information on how to adjust those parameters, if necessary.

`max_fsm_pages` (`integer`)

Sets the maximum number of disk pages for which free space will be tracked in the shared free-space map. Six bytes of shared memory are consumed for each page slot. This setting must be at least 16 * `max_fsm_relations`. The default is chosen by initdb depending on the amount of available memory, and can range from 20k to 200k pages. This parameter can only be set at server start.

`max_fsm_relations` (`integer`)

Sets the maximum number of relations (tables and indexes) for which free space will be tracked in the shared free-space map. Roughly seventy bytes of shared memory are consumed for each slot. The default is one thousand relations. This parameter can only be set at server start.

17.4.3. Kernel Resource Usage

`max_files_per_process` (`integer`)

Sets the maximum number of simultaneously open files allowed to each server subprocess. The default is one thousand files. If the kernel is enforcing a safe per-process limit, you don't need to worry about this setting. But on some platforms (notably, most BSD systems), the kernel will allow individual processes to open many more files than the system can really support when a large number of processes all try to open that many files. If you find yourself seeing "Too many open files" failures, try reducing this setting. This parameter can only be set at server start.

`shared_preload_libraries` (`string`)

This variable specifies one or more shared libraries that are to be preloaded at server start. If more than one library is to be loaded, separate their names with commas. For example, `'$libdir/mylib'` would cause `mylib.so` (or on some platforms, `mylib.sl`) to be preloaded from the installation's standard library directory. This parameter can only be set at server start.

PostgreSQL procedural language libraries can be preloaded in this way, typically by using the syntax `'$libdir/plXXX'` where `XXX` is `pgsql`, `perl`, `tcl`, or `python`.

By preloading a shared library, the library startup time is avoided when the library is first used. However, the time to start each new server process may increase slightly, even if that process never uses the library. So this parameter is recommended only for libraries that will be used in most sessions.

> **Note:** On Windows hosts, preloading a library at server start will not reduce the time required to start each new server process; each server process will re-load all preload libraries. However, `shared_preload_libraries` is still useful on Windows hosts because some shared libraries may need to perform certain operations that only take place at postmaster start (for example, a shared library may need to reserve lightweight locks or shared memory and you can't do that after the postmaster has started).

If a specified library is not found, the server will fail to start.

Every PostgreSQL-supported library has a "magic block" that is checked to guarantee compatibility. For this reason, non-PostgreSQL libraries cannot be loaded in this way.

17.4.4. Cost-Based Vacuum Delay

During the execution of *VACUUM* and *ANALYZE* commands, the system maintains an internal counter that keeps track of the estimated cost of the various I/O operations that are performed. When the accumulated cost reaches a limit (specified by `vacuum_cost_limit`), the process performing the operation will sleep for a while (specified by `vacuum_cost_delay`). Then it will reset the counter and continue execution.

The intent of this feature is to allow administrators to reduce the I/O impact of these commands on concurrent database activity. There are many situations in which it is not very important that maintenance commands like `VACUUM` and `ANALYZE` finish quickly; however, it is usually very important that these commands do not significantly interfere with the ability of the system to perform other database operations. Cost-based vacuum delay provides a way for administrators to achieve this.

This feature is disabled by default. To enable it, set the `vacuum_cost_delay` variable to a nonzero value.

`vacuum_cost_delay` (`integer`)

The length of time, in milliseconds, that the process will sleep when the cost limit has been exceeded. The default value is zero, which disables the cost-based vacuum delay feature. Positive values enable cost-based vacuuming. Note that on many systems, the effective resolution of sleep delays is 10 milliseconds; setting `vacuum_cost_delay` to a value that is not a multiple of 10 may have the same results as setting it to the next higher multiple of 10.

`vacuum_cost_page_hit` (`integer`)

The estimated cost for vacuuming a buffer found in the shared buffer cache. It represents the cost to lock the buffer pool, lookup the shared hash table and scan the content of the page. The default value is one.

`vacuum_cost_page_miss` (`integer`)

The estimated cost for vacuuming a buffer that has to be read from disk. This represents the effort to lock the buffer pool, lookup the shared hash table, read the desired block in from the disk and scan its content. The default value is 10.

vacuum_cost_page_dirty (integer)

The estimated cost charged when vacuum modifies a block that was previously clean. It represents the extra I/O required to flush the dirty block out to disk again. The default value is 20.

vacuum_cost_limit (integer)

The accumulated cost that will cause the vacuuming process to sleep. The default value is 200.

Note: There are certain operations that hold critical locks and should therefore complete as quickly as possible. Cost-based vacuum delays do not occur during such operations. Therefore it is possible that the cost accumulates far higher than the specified limit. To avoid uselessly long delays in such cases, the actual delay is calculated as vacuum_cost_delay * accumulated_balance / vacuum_cost_limit with a maximum of vacuum_cost_delay * 4.

17.4.5. Background Writer

Beginning in PostgreSQL 8.0, there is a separate server process called the *background writer*, whose sole function is to issue writes of "dirty" shared buffers. The intent is that server processes handling user queries should seldom or never have to wait for a write to occur, because the background writer will do it. This arrangement also reduces the performance penalty associated with checkpoints. The background writer will continuously trickle out dirty pages to disk, so that only a few pages will need to be forced out when checkpoint time arrives, instead of the storm of dirty-buffer writes that formerly occurred at each checkpoint. However there is a net overall increase in I/O load, because where a repeatedly-dirtied page might before have been written only once per checkpoint interval, the background writer might write it several times in the same interval. In most situations a continuous low load is preferable to periodic spikes, but the parameters discussed in this subsection can be used to tune the behavior for local needs.

bgwriter_delay (integer)

Specifies the delay between activity rounds for the background writer. In each round the writer issues writes for some number of dirty buffers (controllable by the following parameters). It then sleeps for bgwriter_delay milliseconds, and repeats. The default value is 200 milliseconds (200ms). Note that on many systems, the effective resolution of sleep delays is 10 milliseconds; setting bgwriter_delay to a value that is not a multiple of 10 may have the same results as setting it to the next higher multiple of 10. This parameter can only be set in the postgresql.conf file or on the server command line.

bgwriter_lru_percent (floating point)

To reduce the probability that server processes will need to issue their own writes, the background writer tries to write buffers that are likely to be recycled soon. In each round, it examines up to bgwriter_lru_percent of the buffers that are nearest to being recycled, and writes any that are dirty. The default value is 1.0 (1% of the total number of shared buffers). This parameter can only be set in the postgresql.conf file or on the server command line.

bgwriter_lru_maxpages (integer)

In each round, no more than this many buffers will be written as a result of scanning soon-to-be-recycled buffers. The default value is five buffers. This parameter can only be set in the

postgresql.conf file or on the server command line.

bgwriter_all_percent (floating point)

To reduce the amount of work that will be needed at checkpoint time, the background writer also does a circular scan through the entire buffer pool, writing buffers that are found to be dirty. In each round, it examines up to bgwriter_all_percent of the buffers for this purpose. The default value is 0.333 (0.333% of the total number of shared buffers). With the default bgwriter_delay setting, this will allow the entire shared buffer pool to be scanned about once per minute. This parameter can only be set in the postgresql.conf file or on the server command line.

bgwriter_all_maxpages (integer)

In each round, no more than this many buffers will be written as a result of the scan of the entire buffer pool. (If this limit is reached, the scan stops, and resumes at the next buffer during the next round.) The default value is five buffers. This parameter can only be set in the postgresql.conf file or on the server command line.

Smaller values of bgwriter_all_percent and bgwriter_all_maxpages reduce the extra I/O load caused by the background writer, but leave more work to be done at checkpoint time. To reduce load spikes at checkpoints, increase these two values. Similarly, smaller values of bgwriter_lru_percent and bgwriter_lru_maxpages reduce the extra I/O load caused by the background writer, but make it more likely that server processes will have to issue writes for themselves, delaying interactive queries. To disable background writing entirely, set both maxpages values and/or both percent values to zero.

17.5. Write Ahead Log

See also Section 27.3 for details on WAL tuning.

17.5.1. Settings

fsync (boolean)

If this parameter is on, the PostgreSQL server will try to make sure that updates are physically written to disk, by issuing fsync() system calls or various equivalent methods (see wal_sync_method). This ensures that the database cluster can recover to a consistent state after an operating system or hardware crash.

However, using fsync results in a performance penalty: when a transaction is committed, PostgreSQL must wait for the operating system to flush the write-ahead log to disk. When fsync is disabled, the operating system is allowed to do its best in buffering, ordering, and delaying writes. This can result in significantly improved performance. However, if the system crashes, the results of the last few committed transactions may be lost in part or whole. In the worst case, unrecoverable data corruption may occur. (Crashes of the database software itself are *not* a risk factor here. Only an operating-system-level crash creates a risk of corruption.)

Due to the risks involved, there is no universally correct setting for fsync. Some administrators always disable fsync, while others only turn it off during initial bulk data loads, where there is a clear restart point if something goes wrong. Others always leave fsync enabled. The default is to

enable fsync, for maximum reliability. If you trust your operating system, your hardware, and your utility company (or your battery backup), you can consider disabling fsync.

This parameter can only be set in the postgresql.conf file or on the server command line. If you turn this parameter off, also consider turning off full_page_writes.

wal_sync_method (string)

Method used for forcing WAL updates out to disk. If fsync is off then this setting is irrelevant, since updates will not be forced out at all. Possible values are:

- open_datasync (write WAL files with open() option O_DSYNC)
- fdatasync (call fdatasync() at each commit)
- fsync_writethrough (call fsync() at each commit, forcing write-through of any disk write cache)
- fsync (call fsync() at each commit)
- open_sync (write WAL files with open() option O_SYNC)

Not all of these choices are available on all platforms. The default is the first method in the above list that is supported by the platform. The open_* options also use O_DIRECT if available. This parameter can only be set in the postgresql.conf file or on the server command line.

full_page_writes (boolean)

When this parameter is on, the PostgreSQL server writes the entire content of each disk page to WAL during the first modification of that page after a checkpoint. This is needed because a page write that is in process during an operating system crash might be only partially completed, leading to an on-disk page that contains a mix of old and new data. The row-level change data normally stored in WAL will not be enough to completely restore such a page during post-crash recovery. Storing the full page image guarantees that the page can be correctly restored, but at a price in increasing the amount of data that must be written to WAL. (Because WAL replay always starts from a checkpoint, it is sufficient to do this during the first change of each page after a checkpoint. Therefore, one way to reduce the cost of full-page writes is to increase the checkpoint interval parameters.)

Turning this parameter off speeds normal operation, but might lead to a corrupt database after an operating system crash or power failure. The risks are similar to turning off fsync, though smaller. It may be safe to turn off this parameter if you have hardware (such as a battery-backed disk controller) or file-system software that reduces the risk of partial page writes to an acceptably low level (e.g., ReiserFS 4).

Turning off this parameter does not affect use of WAL archiving for point-in-time recovery (PITR) (see Section 23.3).

This parameter can only be set in the postgresql.conf file or on the server command line. The default is on.

wal_buffers (integer)

The amount of memory used in shared memory for WAL data. The default is 64 kilobytes (64kB). The setting need only be large enough to hold the amount of WAL data generated by one typical transaction, since the data is written out to disk at every transaction commit. This parameter can only be set at server start.

Increasing this parameter may cause PostgreSQL to request more System V shared memory than your operating system's default configuration allows. See Section 16.4.1 for information on how to adjust those parameters, if necessary.

`commit_delay` (`integer`)

Time delay between writing a commit record to the WAL buffer and flushing the buffer out to disk, in microseconds. A nonzero delay can allow multiple transactions to be committed with only one `fsync()` system call, if system load is high enough that additional transactions become ready to commit within the given interval. But the delay is just wasted if no other transactions become ready to commit. Therefore, the delay is only performed if at least `commit_siblings` other transactions are active at the instant that a server process has written its commit record. The default is zero (no delay).

`commit_siblings` (`integer`)

Minimum number of concurrent open transactions to require before performing the `commit_delay` delay. A larger value makes it more probable that at least one other transaction will become ready to commit during the delay interval. The default is five transactions.

17.5.2. Checkpoints

`checkpoint_segments` (`integer`)

Maximum distance between automatic WAL checkpoints, in log file segments (each segment is normally 16 megabytes). The default is three segments. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`checkpoint_timeout` (`integer`)

Maximum time between automatic WAL checkpoints, in seconds. The default is five minutes (`5min`). This parameter can only be set in the `postgresql.conf` file or on the server command line.

`checkpoint_warning` (`integer`)

Write a message to the server log if checkpoints caused by the filling of checkpoint segment files happen closer together than this many seconds (which suggests that `checkpoint_segments` ought to be raised). The default is 30 seconds (`30s`). Zero disables the warning. This parameter can only be set in the `postgresql.conf` file or on the server command line.

17.5.3. Archiving

`archive_command` (`string`)

The shell command to execute to archive a completed segment of the WAL file series. If this is an empty string (the default), WAL archiving is disabled. Any `%p` in the string is replaced by the path name of the file to archive, and any `%f` is replaced by the file name only. (The path name is relative to the working directory of the server, i.e., the cluster's data directory.) Use `%%` to embed an actual `%` character in the command. For more information see Section 23.3.1. This parameter can only be set in the `postgresql.conf` file or on the server command line.

It is important for the command to return a zero exit status if and only if it succeeds. Examples:

```
archive_command = 'cp "%p" /mnt/server/archivedir/"%f"'
archive_command = 'copy "%p" /mnt/server/archivedir/"%f"'  # Windows
```

`archive_timeout` (`integer`)

The archive_command is only invoked on completed WAL segments. Hence, if your server generates little WAL traffic (or has slack periods where it does so), there could be a long delay between the completion of a transaction and its safe recording in archive storage. To put a limit on how old unarchived data can be, you can set `archive_timeout` to force the server to switch to a new WAL segment file periodically. When this parameter is greater than zero, the server will switch to a new segment file whenever this many seconds have elapsed since the last segment file switch. Note that archived files that are closed early due to a forced switch are still the same length as completely full files. Therefore, it is unwise to use a very short `archive_timeout` — it will bloat your archive storage. `archive_timeout` settings of a minute or so are usually reasonable. This parameter can only be set in the `postgresql.conf` file or on the server command line.

17.6. Query Planning

17.6.1. Planner Method Configuration

These configuration parameters provide a crude method of influencing the query plans chosen by the query optimizer. If the default plan chosen by the optimizer for a particular query is not optimal, a temporary solution may be found by using one of these configuration parameters to force the optimizer to choose a different plan. Turning one of these settings off permanently is seldom a good idea, however. Better ways to improve the quality of the plans chosen by the optimizer include adjusting the *Planner Cost Constants*, running *ANALYZE* more frequently, increasing the value of the default_statistics_target configuration parameter, and increasing the amount of statistics collected for specific columns using `ALTER TABLE SET STATISTICS`.

`enable_bitmapscan` (`boolean`)

Enables or disables the query planner's use of bitmap-scan plan types. The default is `on`.

`enable_hashagg` (`boolean`)

Enables or disables the query planner's use of hashed aggregation plan types. The default is `on`.

`enable_hashjoin` (`boolean`)

Enables or disables the query planner's use of hash-join plan types. The default is `on`.

`enable_indexscan` (`boolean`)

Enables or disables the query planner's use of index-scan plan types. The default is `on`.

`enable_mergejoin` (`boolean`)

Enables or disables the query planner's use of merge-join plan types. The default is `on`.

`enable_nestloop` (`boolean`)

Enables or disables the query planner's use of nested-loop join plans. It's not possible to suppress nested-loop joins entirely, but turning this variable off discourages the planner from using one if there are other methods available. The default is `on`.

`enable_seqscan` (`boolean`)

Enables or disables the query planner's use of sequential scan plan types. It's not possible to suppress sequential scans entirely, but turning this variable off discourages the planner from using one if there are other methods available. The default is `on`.

`enable_sort` (`boolean`)

Enables or disables the query planner's use of explicit sort steps. It's not possible to suppress explicit sorts entirely, but turning this variable off discourages the planner from using one if there are other methods available. The default is `on`.

`enable_tidscan` (`boolean`)

Enables or disables the query planner's use of TID scan plan types. The default is `on`.

17.6.2. Planner Cost Constants

The *cost* variables described in this section are measured on an arbitrary scale. Only their relative values matter, hence scaling them all up or down by the same factor will result in no change in the planner's choices. Traditionally, these variables have been referenced to sequential page fetches as the unit of cost; that is, `seq_page_cost` is conventionally set to `1.0` and the other cost variables are set with reference to that. But you can use a different scale if you prefer, such as actual execution times in milliseconds on a particular machine.

Note: Unfortunately, there is no well-defined method for determining ideal values for the cost variables. They are best treated as averages over the entire mix of queries that a particular installation will get. This means that changing them on the basis of just a few experiments is very risky.

`seq_page_cost` (`floating point`)

Sets the planner's estimate of the cost of a disk page fetch that is part of a series of sequential fetches. The default is 1.0.

`random_page_cost` (`floating point`)

Sets the planner's estimate of the cost of a non-sequentially-fetched disk page. The default is 4.0. Reducing this value relative to `seq_page_cost` will cause the system to prefer index scans; raising it will make index scans look relatively more expensive. You can raise or lower both values together to change the importance of disk I/O costs relative to CPU costs, which are described by the following parameters.

Tip: Although the system will let you set `random_page_cost` to less than `seq_page_cost`, it is not physically sensible to do so. However, setting them equal makes sense if the database is entirely cached in RAM, since in that case there is no penalty for touching pages out of sequence. Also,

in a heavily-cached database you should lower both values relative to the CPU parameters, since the cost of fetching a page already in RAM is much smaller than it would normally be.

`cpu_tuple_cost` (`floating point`)

Sets the planner's estimate of the cost of processing each row during a query. The default is 0.01.

`cpu_index_tuple_cost` (`floating point`)

Sets the planner's estimate of the cost of processing each index entry during an index scan. The default is 0.005.

`cpu_operator_cost` (`floating point`)

Sets the planner's estimate of the cost of processing each operator or function executed during a query. The default is 0.0025.

`effective_cache_size` (`integer`)

Sets the planner's assumption about the effective size of the disk cache that is available to a single query. This is factored into estimates of the cost of using an index; a higher value makes it more likely index scans will be used, a lower value makes it more likely sequential scans will be used. When setting this parameter you should consider both PostgreSQL's shared buffers and the portion of the kernel's disk cache that will be used for PostgreSQL data files. Also, take into account the expected number of concurrent queries on different tables, since they will have to share the available space. This parameter has no effect on the size of shared memory allocated by PostgreSQL, nor does it reserve kernel disk cache; it is used only for estimation purposes. The default is 128 megabytes (`128MB`).

17.6.3. Genetic Query Optimizer

`geqo` (`boolean`)

Enables or disables genetic query optimization, which is an algorithm that attempts to do query planning without exhaustive searching. This is on by default. The `geqo_threshold` variable provides a more granular way to disable GEQO for certain classes of queries.

`geqo_threshold` (`integer`)

Use genetic query optimization to plan queries with at least this many `FROM` items involved. (Note that a `FULL OUTER JOIN` construct counts as only one `FROM` item.) The default is 12. For simpler queries it is usually best to use the deterministic, exhaustive planner, but for queries with many tables the deterministic planner takes too long.

`geqo_effort` (`integer`)

Controls the trade off between planning time and query plan efficiency in GEQO. This variable must be an integer in the range from 1 to 10. The default value is five. Larger values increase the time spent doing query planning, but also increase the likelihood that an efficient query plan will be chosen.

`geqo_effort` doesn't actually do anything directly; it is only used to compute the default values for the other variables that influence GEQO behavior (described below). If you prefer, you can set the other parameters by hand instead.

`geqo_pool_size` (`integer`)

Controls the pool size used by GEQO. The pool size is the number of individuals in the genetic population. It must be at least two, and useful values are typically 100 to 1000. If it is set to zero (the default setting) then a suitable default is chosen based on `geqo_effort` and the number of tables in the query.

`geqo_generations` (`integer`)

Controls the number of generations used by GEQO. Generations specifies the number of iterations of the algorithm. It must be at least one, and useful values are in the same range as the pool size. If it is set to zero (the default setting) then a suitable default is chosen based on `geqo_pool_size`.

`geqo_selection_bias` (`floating point`)

Controls the selection bias used by GEQO. The selection bias is the selective pressure within the population. Values can be from 1.50 to 2.00; the latter is the default.

17.6.4. Other Planner Options

`default_statistics_target` (`integer`)

Sets the default statistics target for table columns that have not had a column-specific target set via `ALTER TABLE SET STATISTICS`. Larger values increase the time needed to do `ANALYZE`, but may improve the quality of the planner's estimates. The default is 10. For more information on the use of statistics by the PostgreSQL query planner, refer to Section 13.2.

`constraint_exclusion` (`boolean`)

Enables or disables the query planner's use of table constraints to optimize queries. The default is `off`.

When this parameter is `on`, the planner compares query conditions with table `CHECK` constraints, and omits scanning tables for which the conditions contradict the constraints. For example:

```
CREATE TABLE parent(key integer, ...);
CREATE TABLE child1000(check (key between 1000 and 1999)) INHERITS(parent);
CREATE TABLE child2000(check (key between 2000 and 2999)) INHERITS(parent);
...
SELECT * FROM parent WHERE key = 2400;
```

With constraint exclusion enabled, this `SELECT` will not scan `child1000` at all. This can improve performance when inheritance is used to build partitioned tables.

Currently, `constraint_exclusion` is disabled by default because it risks incorrect results if query plans are cached — if a table constraint is changed or dropped, the previously generated plan might now be wrong, and there is no built-in mechanism to force re-planning. (This deficiency will probably be addressed in a future PostgreSQL release.) Another reason for keeping it off is that the constraint checks are relatively expensive, and in many circumstances will yield no savings. It is recommended to turn this on only if you are actually using partitioned tables designed to take advantage of the feature.

Refer to Section 5.9 for more information on using constraint exclusion and partitioning.

from_collapse_limit (integer)

The planner will merge sub-queries into upper queries if the resulting FROM list would have no more than this many items. Smaller values reduce planning time but may yield inferior query plans. The default is eight. It is usually wise to keep this less than geqo_threshold. For more information see Section 13.3.

join_collapse_limit (integer)

The planner will rewrite explicit JOIN constructs (except FULL JOINs) into lists of FROM items whenever a list of no more than this many items would result. Smaller values reduce planning time but may yield inferior query plans.

By default, this variable is set the same as from_collapse_limit, which is appropriate for most uses. Setting it to 1 prevents any reordering of explicit JOINs. Thus, the explicit join order specified in the query will be the actual order in which the relations are joined. The query planner does not always choose the optimal join order; advanced users may elect to temporarily set this variable to 1, and then specify the join order they desire explicitly. For more information see Section 13.3.

17.7. Error Reporting and Logging

17.7.1. Where To Log

log_destination (string)

PostgreSQL supports several methods for logging server messages, including stderr and syslog. On Windows, eventlog is also supported. Set this parameter to a list of desired log destinations separated by commas. The default is to log to stderr only. This parameter can only be set in the postgresql.conf file or on the server command line.

redirect_stderr (boolean)

This parameter allows messages sent to stderr to be captured and redirected into log files. This method, in combination with logging to stderr, is often more useful than logging to syslog, since some types of messages may not appear in syslog output (a common example is dynamic-linker failure messages). This parameter can only be set at server start.

log_directory (string)

When redirect_stderr is enabled, this parameter determines the directory in which log files will be created. It may be specified as an absolute path, or relative to the cluster data directory. This parameter can only be set in the postgresql.conf file or on the server command line.

log_filename (string)

When redirect_stderr is enabled, this parameter sets the file names of the created log files. The value is treated as a strftime pattern, so %-escapes can be used to specify time-varying file names. If no %-escapes are present, PostgreSQL will append the epoch of the new log file's open time. For example, if log_filename were server_log, then the chosen file name would be server_log.1093827753 for a log starting at Sun Aug 29 19:02:33 2004 MST. This parameter can only be set in the postgresql.conf file or on the server command line.

log_rotation_age (integer)

When redirect_stderr is enabled, this parameter determines the maximum lifetime of an individual log file. After this many minutes have elapsed, a new log file will be created. Set to zero to disable time-based creation of new log files. This parameter can only be set in the postgresql.conf file or on the server command line.

log_rotation_size (integer)

When redirect_stderr is enabled, this parameter determines the maximum size of an individual log file. After this many kilobytes have been emitted into a log file, a new log file will be created. Set to zero to disable size-based creation of new log files. This parameter can only be set in the postgresql.conf file or on the server command line.

log_truncate_on_rotation (boolean)

When redirect_stderr is enabled, this parameter will cause PostgreSQL to truncate (overwrite), rather than append to, any existing log file of the same name. However, truncation will occur only when a new file is being opened due to time-based rotation, not during server startup or size-based rotation. When off, pre-existing files will be appended to in all cases. For example, using this setting in combination with a log_filename like postgresql-%H.log would result in generating twenty-four hourly log files and then cyclically overwriting them. This parameter can only be set in the postgresql.conf file or on the server command line.

Example: To keep 7 days of logs, one log file per day named server_log.Mon, server_log.Tue, etc, and automatically overwrite last week's log with this week's log, set log_filename to server_log.%a, log_truncate_on_rotation to on, and log_rotation_age to 1440.

Example: To keep 24 hours of logs, one log file per hour, but also rotate sooner if the log file size exceeds 1GB, set log_filename to server_log.%H%M, log_truncate_on_rotation to on, log_rotation_age to 60, and log_rotation_size to 1000000. Including %M in log_filename allows any size-driven rotations that may occur to select a file name different from the hour's initial file name.

syslog_facility (string)

When logging to syslog is enabled, this parameter determines the syslog "facility" to be used. You may choose from LOCAL0, LOCAL1, LOCAL2, LOCAL3, LOCAL4, LOCAL5, LOCAL6, LOCAL7; the default is LOCAL0. See also the documentation of your system's syslog daemon. This parameter can only be set in the postgresql.conf file or on the server command line.

syslog_ident (string)

When logging to syslog is enabled, this parameter determines the program name used to identify PostgreSQL messages in syslog logs. The default is postgres. This parameter can only be set in the postgresql.conf file or on the server command line.

17.7.2. When To Log

client_min_messages (string)

Controls which message levels are sent to the client. Valid values are DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, LOG, NOTICE, WARNING, ERROR, FATAL, and PANIC. Each level includes all the

levels that follow it. The later the level, the fewer messages are sent. The default is NOTICE. Note that LOG has a different rank here than in log_min_messages.

log_min_messages (string)

Controls which message levels are written to the server log. Valid values are DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, INFO, NOTICE, WARNING, ERROR, LOG, FATAL, and PANIC. Each level includes all the levels that follow it. The later the level, the fewer messages are sent to the log. The default is NOTICE. Note that LOG has a different rank here than in client_min_messages. Only superusers can change this setting.

log_error_verbosity (string)

Controls the amount of detail written in the server log for each message that is logged. Valid values are TERSE, DEFAULT, and VERBOSE, each adding more fields to displayed messages. Only superusers can change this setting.

log_min_error_statement (string)

Controls whether or not the SQL statement that causes an error condition will be recorded in the server log. The current SQL statement is included in the log entry for any message of the specified severity or higher. Valid values are DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, INFO, NOTICE, WARNING, ERROR, FATAL, and PANIC. The default is ERROR, which means statements causing errors, fatal errors, or panics will be logged. To effectively turn off logging of failing statements, set this parameter to PANIC. Only superusers can change this setting.

log_min_duration_statement (integer)

Causes the duration of each completed statement to be logged if the statement ran for at least the specified number of milliseconds. Setting this to zero prints all statement durations. Minus-one (the default) disables logging statement durations. For example, if you set it to 250ms then all SQL statements that run 250ms or longer will be logged. Enabling this parameter can be helpful in tracking down unoptimized queries in your applications. Only superusers can change this setting.

For clients using extended query protocol, durations of the Parse, Bind, and Execute steps are logged independently.

> **Note:** When using this option together with log_statement, the text of statements that are logged because of log_statement will not be repeated in the duration log message. If you are not using syslog, it is recommended that you log the PID or session ID using log_line_prefix so that you can link the statement message to the later duration message using the process ID or session ID.

silent_mode (boolean)

Runs the server silently. If this parameter is set, the server will automatically run in background and any controlling terminals are disassociated. The server's standard output and standard error are redirected to /dev/null, so any messages sent to them will be lost. Unless syslog logging is selected or redirect_stderr is enabled, using this parameter is discouraged because it makes it impossible to see error messages. This parameter can only be set at server start.

Here is a list of the various message severity levels used in these settings:

DEBUG[1-5]

Provides information for use by developers.

INFO

Provides information implicitly requested by the user, e.g., during VACUUM VERBOSE.

NOTICE

Provides information that may be helpful to users, e.g., truncation of long identifiers and the creation of indexes as part of primary keys.

WARNING

Provides warnings to the user, e.g., COMMIT outside a transaction block.

ERROR

Reports an error that caused the current command to abort.

LOG

Reports information of interest to administrators, e.g., checkpoint activity.

FATAL

Reports an error that caused the current session to abort.

PANIC

Reports an error that caused all sessions to abort.

17.7.3. What To Log

debug_print_parse (boolean)
debug_print_rewritten (boolean)
debug_print_plan (boolean)
debug_pretty_print (boolean)

These parameters enable various debugging output to be emitted. For each executed query, they print the resulting parse tree, the query rewriter output, or the execution plan. debug_pretty_print indents these displays to produce a more readable but much longer output format. client_min_messages or log_min_messages must be DEBUG1 or lower to actually send this output to the client or the server log, respectively. These parameters are off by default.

log_connections (boolean)

This outputs a line to the server log detailing each successful connection. This is off by default, although it is probably very useful. Some client programs, like psql, attempt to connect twice while determining if a password is required, so duplicate "connection received" messages do not necessarily indicate a problem. This parameter can only be set in the postgresql.conf file or on the server command line.

log_disconnections (boolean)

This outputs a line in the server log similar to log_connections but at session termination, and includes the duration of the session. This is off by default. This parameter can only be set in the postgresql.conf file or on the server command line.

log_duration (boolean)

Causes the duration of every completed statement to be logged. The default is off. Only superusers can change this setting.

For clients using extended query protocol, durations of the Parse, Bind, and Execute steps are logged independently.

> **Note:** The difference between setting this option and setting log_min_duration_statement to zero is that exceeding log_min_duration_statement forces the text of the query to be logged, but this option doesn't. Thus, if log_duration is on and log_min_duration_statement has a positive value, all durations are logged but the query text is included only for statements exceeding the threshold. This behavior can be useful for gathering statistics in high-load installations.

log_line_prefix (string)

This is a printf-style string that is output at the beginning of each log line. The default is an empty string. Each recognized escape is replaced as outlined below - anything else that looks like an escape is ignored. Other characters are copied straight to the log line. Some escapes are only recognized by session processes, and do not apply to background processes such as the main server process. Syslog produces its own time stamp and process ID information, so you probably do not want to use those escapes if you are using syslog. This parameter can only be set in the postgresql.conf file or on the server command line.

Escape	Effect	Session only
%u	User name	yes
%d	Database name	yes
%r	Remote host name or IP address, and remote port	yes
%h	Remote host name or IP address	yes
%p	Process ID	no
%t	Time stamp (no milliseconds, no timezone on Windows)	no
%m	Time stamp with milliseconds	no
%i	Command tag: This is the command that generated the log line.	yes

Escape	Effect	Session only
%c	Session ID: A unique identifier for each session. It is 2 4-byte hexadecimal numbers (without leading zeros) separated by a dot. The numbers are the session start time and the process ID, so this can also be used as a space saving way of printing these items.	yes
%l	Number of the log line for each process, starting at 1	no
%s	Session start time stamp	yes
%x	Transaction ID	yes
%q	Does not produce any output, but tells non-session processes to stop at this point in the string. Ignored by session processes.	no
%%	Literal %	no

log_statement (string)

Controls which SQL statements are logged. Valid values are none, ddl, mod, and all. ddl logs all data definition statements, such as CREATE, ALTER, and DROP statements. mod logs all ddl statements, plus data-modifying statements such as INSERT, UPDATE, DELETE, TRUNCATE, and COPY FROM. PREPARE, EXECUTE, and EXPLAIN ANALYZE statements are also logged if their contained command is of an appropriate type. For clients using extended query protocol, logging occurs when an Execute message is received, and values of the Bind parameters are included (with any embedded single-quote marks doubled).

The default is none. Only superusers can change this setting.

> **Note:** Statements that contain simple syntax errors are not logged even by the log_statement = all setting, because the log message is emitted only after basic parsing has been done to determine the statement type. In the case of extended query protocol, this setting likewise does not log statements that fail before the Execute phase (i.e., during parse analysis or planning). Set log_min_error_statement to ERROR (or lower) to log such statements.

log_hostname (boolean)

By default, connection log messages only show the IP address of the connecting host. Turning on this parameter causes logging of the host name as well. Note that depending on your host name resolution setup this might impose a non-negligible performance penalty. This parameter can only be set in the postgresql.conf file or on the server command line.

17.8. Run-Time Statistics

17.8.1. Query and Index Statistics Collector

These parameters control a server-wide statistics collection feature. When statistics collection is enabled, the data that is produced can be accessed via the `pg_stat` and `pg_statio` family of system views. Refer to Chapter 25 for more information.

> **Note:** As of PostgreSQL 8.2, `stats_command_string` controls a separate data collection mechanism that can be turned on or off independently of whether the statistics-collection subprocess is running. The subprocess is only needed to support collection of block-level or row-level statistics.

`stats_command_string` (`boolean`)

Enables the collection of information on the currently executing command of each session, along with the time at which that command began execution. This parameter is on by default. Note that even when enabled, this information is not visible to all users, only to superusers and the user owning the session being reported on; so it should not represent a security risk. Only superusers can change this setting.

`update_process_title` (`boolean`)

Enables updating of the process title every time a new SQL command is received by the server. The process title is typically viewed by the `ps` command or in Windows using the Process Explorer. Only superusers can change this setting.

`stats_start_collector` (`boolean`)

Controls whether the server should start the statistics-collection subprocess. This is on by default, but may be turned off if you know you have no interest in collecting statistics or running autovacuum. This parameter can only be set at server start, because the collection subprocess cannot be started or stopped on-the-fly. (However, the extent to which statistics are actually gathered can be changed while the server is running, so long as the subprocess exists.)

`stats_block_level` (`boolean`)

Enables the collection of block-level statistics on database activity. This parameter is off by default. Only superusers can change this setting.

`stats_row_level` (`boolean`)

Enables the collection of row-level statistics on database activity. This parameter is off by default. Only superusers can change this setting.

`stats_reset_on_server_start` (`boolean`)

If on, collected block-level and row-level statistics are zeroed out whenever the server is restarted. If off, statistics are accumulated across server restarts. This parameter is off by default. This parameter can only be set at server start.

17.8.2. Statistics Monitoring

`log_statement_stats` (`boolean`)
`log_parser_stats` (`boolean`)
`log_planner_stats` (`boolean`)
`log_executor_stats` (`boolean`)

For each query, write performance statistics of the respective module to the server log. This is a crude profiling instrument. `log_statement_stats` reports total statement statistics, while the others report per-module statistics. `log_statement_stats` cannot be enabled together with any of the per-module options. All of these options are disabled by default. Only superusers can change these settings.

17.9. Automatic Vacuuming

These settings control the behavior of the *autovacuum* feature. Refer to Section 22.1.4 for more information.

`autovacuum` (`boolean`)

Controls whether the server should run the autovacuum daemon. This is off by default. `stats_start_collector` and `stats_row_level` must also be turned on for autovacuum to work. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`autovacuum_naptime` (`integer`)

Specifies the delay between activity rounds for the autovacuum daemon. In each round the daemon examines one database and issues `VACUUM` and `ANALYZE` commands as needed for tables in that database. The delay is measured in seconds, and the default is one minute (`1m`). This parameter can only be set in the `postgresql.conf` file or on the server command line.

Note that even when this parameter is disabled, the system will periodically launch autovacuum processes in order to prevent transaction ID wraparound. See Section 22.1.3 for more information.

`autovacuum_vacuum_threshold` (`integer`)

Specifies the minimum number of updated or deleted tuples needed to trigger a `VACUUM` in any one table. The default is 500 tuples. This parameter can only be set in the `postgresql.conf` file or on the server command line. This setting can be overridden for individual tables by entries in `pg_autovacuum`.

`autovacuum_analyze_threshold` (`integer`)

Specifies the minimum number of inserted, updated or deleted tuples needed to trigger an `ANALYZE` in any one table. The default is 250 tuples. This parameter can only be set in the `postgresql.conf` file or on the server command line. This setting can be overridden for individual tables by entries in `pg_autovacuum`.

`autovacuum_vacuum_scale_factor` (`floating point`)

Specifies a fraction of the table size to add to `autovacuum_vacuum_threshold` when deciding whether to trigger a `VACUUM`. The default is 0.2 (20% of table size). This parameter can only be set

in the postgresql.conf file or on the server command line. This setting can be overridden for individual tables by entries in pg_autovacuum.

autovacuum_analyze_scale_factor (floating point)

Specifies a fraction of the table size to add to autovacuum_analyze_threshold when deciding whether to trigger an ANALYZE. The default is 0.1 (10% of table size). This parameter can only be set in the postgresql.conf file or on the server command line. This setting can be overridden for individual tables by entries in pg_autovacuum.

autovacuum_freeze_max_age (integer)

Specifies the maximum age (in transactions) that a table's pg_class.relfrozenxid field can attain before a VACUUM operation is forced to prevent transaction ID wraparound within the table. Note that the system will launch autovacuum processes to prevent wraparound even when autovacuum is otherwise disabled. The default is 200 million transactions. This parameter can only be set at server start, but the setting can be reduced for individual tables by entries in pg_autovacuum. For more information see Section 22.1.3.

autovacuum_vacuum_cost_delay (integer)

Specifies the cost delay value that will be used in automatic VACUUM operations. If -1 is specified (which is the default), the regular vacuum_cost_delay value will be used. This parameter can only be set in the postgresql.conf file or on the server command line. This setting can be overridden for individual tables by entries in pg_autovacuum.

autovacuum_vacuum_cost_limit (integer)

Specifies the cost limit value that will be used in automatic VACUUM operations. If -1 is specified (which is the default), the regular vacuum_cost_limit value will be used. This parameter can only be set in the postgresql.conf file or on the server command line. This setting can be overridden for individual tables by entries in pg_autovacuum.

17.10. Client Connection Defaults

17.10.1. Statement Behavior

search_path (string)

This variable specifies the order in which schemas are searched when an object (table, data type, function, etc.) is referenced by a simple name with no schema component. When there are objects of identical names in different schemas, the one found first in the search path is used. An object that is not in any of the schemas in the search path can only be referenced by specifying its containing schema with a qualified (dotted) name.

The value for search_path has to be a comma-separated list of schema names. If one of the list items is the special value $user, then the schema having the name returned by SESSION_USER is substituted, if there is such a schema. (If not, $user is ignored.)

The system catalog schema, pg_catalog, is always searched, whether it is mentioned in the path or not. If it is mentioned in the path then it will be searched in the specified order. If pg_catalog is

not in the path then it will be searched *before* searching any of the path items.

Likewise, the current session's temporary-table schema, `pg_temp_nnn`, is always searched if it exists. It can be explicitly listed in the path by using the alias `pg_temp`. If it is not listed in the path then it is searched first (before even `pg_catalog`). However, the temporary schema is only searched for relation (table, view, sequence, etc) and data type names. It will never be searched for function or operator names.

When objects are created without specifying a particular target schema, they will be placed in the first schema listed in the search path. An error is reported if the search path is empty.

The default value for this parameter is `'"$user", public'` (where the second part will be ignored if there is no schema named `public`). This supports shared use of a database (where no users have private schemas, and all share use of `public`), private per-user schemas, and combinations of these. Other effects can be obtained by altering the default search path setting, either globally or per-user.

The current effective value of the search path can be examined via the SQL function `current_schemas()`. This is not quite the same as examining the value of `search_path`, since `current_schemas()` shows how the requests appearing in `search_path` were resolved.

For more information on schema handling, see Section 5.7.

`default_tablespace` (`string`)

This variable specifies the default tablespace in which to create objects (tables and indexes) when a `CREATE` command does not explicitly specify a tablespace.

The value is either the name of a tablespace, or an empty string to specify using the default tablespace of the current database. If the value does not match the name of any existing tablespace, PostgreSQL will automatically use the default tablespace of the current database.

For more information on tablespaces, see Section 19.6.

`check_function_bodies` (`boolean`)

This parameter is normally on. When set to `off`, it disables validation of the function body string during *CREATE FUNCTION*. Disabling validation is occasionally useful to avoid problems such as forward references when restoring function definitions from a dump.

`default_transaction_isolation` (`string`)

Each SQL transaction has an isolation level, which can be either "read uncommitted", "read committed", "repeatable read", or "serializable". This parameter controls the default isolation level of each new transaction. The default is "read committed".

Consult Chapter 12 and *SET TRANSACTION* for more information.

`default_transaction_read_only` (`boolean`)

A read-only SQL transaction cannot alter non-temporary tables. This parameter controls the default read-only status of each new transaction. The default is `off` (read/write).

Consult *SET TRANSACTION* for more information.

`statement_timeout` (`integer`)

Abort any statement that takes over the specified number of milliseconds, starting from the time the command arrives at the server from the client. If `log_min_error_statement` is set to `ERROR` or

lower, the statement that timed out will also be logged. A value of zero (the default) turns off the limitation.

`vacuum_freeze_min_age` (`integer`)

Specifies the cutoff age (in transactions) that `VACUUM` should use to decide whether to replace transaction IDs with `FrozenXID` while scanning a table. The default is 100 million transactions. Although users can set this value anywhere from zero to one billion, `VACUUM` will silently limit the effective value to half the value of autovacuum_freeze_max_age, so that there is not an unreasonably short time between forced autovacuums. For more information see Section 22.1.3.

17.10.2. Locale and Formatting

`DateStyle` (`string`)

Sets the display format for date and time values, as well as the rules for interpreting ambiguous date input values. For historical reasons, this variable contains two independent components: the output format specification (`ISO`, `Postgres`, `SQL`, or `German`) and the input/output specification for year/month/day ordering (`DMY`, `MDY`, or `YMD`). These can be set separately or together. The keywords `Euro` and `European` are synonyms for `DMY`; the keywords `US`, `NonEuro`, and `NonEuropean` are synonyms for `MDY`. See Section 8.5 for more information. The built-in default is `ISO, MDY`, but initdb will initialize the configuration file with a setting that corresponds to the behavior of the chosen `lc_time` locale.

`timezone` (`string`)

Sets the time zone for displaying and interpreting time stamps. The default is `'unknown'`, which means to use whatever the system environment specifies as the time zone. See Section 8.5 for more information.

`timezone_abbreviations` (`string`)

Sets the collection of time zone abbreviations that will be accepted by the server for datetime input. The default is `'Default'`, which is a collection that works in most of the world; there are also 'Australia' and 'India', and other collections can be defined for a particular installation. See Appendix B for more information.

`extra_float_digits` (`integer`)

This parameter adjusts the number of digits displayed for floating-point values, including `float4`, `float8`, and geometric data types. The parameter value is added to the standard number of digits (`FLT_DIG` or `DBL_DIG` as appropriate). The value can be set as high as 2, to include partially-significant digits; this is especially useful for dumping float data that needs to be restored exactly. Or it can be set negative to suppress unwanted digits.

`client_encoding` (`string`)

Sets the client-side encoding (character set). The default is to use the database encoding.

`lc_messages` (`string`)

Sets the language in which messages are displayed. Acceptable values are system-dependent; see Section 21.1 for more information. If this variable is set to the empty string (which is the default) then the value is inherited from the execution environment of the server in a system-dependent way.

On some systems, this locale category does not exist. Setting this variable will still work, but there will be no effect. Also, there is a chance that no translated messages for the desired language exist. In that case you will continue to see the English messages.

Only superusers can change this setting, because it affects the messages sent to the server log as well as to the client.

`lc_monetary` (`string`)

Sets the locale to use for formatting monetary amounts, for example with the `to_char` family of functions. Acceptable values are system-dependent; see Section 21.1 for more information. If this variable is set to the empty string (which is the default) then the value is inherited from the execution environment of the server in a system-dependent way.

`lc_numeric` (`string`)

Sets the locale to use for formatting numbers, for example with the `to_char` family of functions. Acceptable values are system-dependent; see Section 21.1 for more information. If this variable is set to the empty string (which is the default) then the value is inherited from the execution environment of the server in a system-dependent way.

`lc_time` (`string`)

Sets the locale to use for formatting date and time values. (Currently, this setting does nothing, but it may in the future.) Acceptable values are system-dependent; see Section 21.1 for more information. If this variable is set to the empty string (which is the default) then the value is inherited from the execution environment of the server in a system-dependent way.

17.10.3. Other Defaults

`explain_pretty_print` (`boolean`)

Determines whether `EXPLAIN VERBOSE` uses the indented or non-indented format for displaying detailed query-tree dumps. The default is `on`.

`dynamic_library_path` (`string`)

If a dynamically loadable module needs to be opened and the file name specified in the `CREATE FUNCTION` or `LOAD` command does not have a directory component (i.e. the name does not contain a slash), the system will search this path for the required file.

The value for `dynamic_library_path` has to be a list of absolute directory paths separated by colons (or semi-colons on Windows). If a list element starts with the special string `$libdir`, the compiled-in PostgreSQL package library directory is substituted for `$libdir`. This is where the modules provided by the standard PostgreSQL distribution are installed. (Use `pg_config --pkglibdir` to find out the name of this directory.) For example:

```
dynamic_library_path = '/usr/local/lib/postgresql:/home/my_project/lib:$libdir'
```

or, in a Windows environment:

```
dynamic_library_path = 'C:\tools\postgresql;H:\my_project\lib;$libdir'
```

The default value for this parameter is `'$libdir'`. If the value is set to an empty string, the automatic path search is turned off.

This parameter can be changed at run time by superusers, but a setting done that way will only persist until the end of the client connection, so this method should be reserved for development purposes. The recommended way to set this parameter is in the `postgresql.conf` configuration file.

`gin_fuzzy_search_limit` (`integer`)

Soft upper limit of the size of the set returned by GIN index. For more information see Section 51.4.

`local_preload_libraries` (`string`)

This variable specifies one or more shared libraries that are to be preloaded at connection start. If more than one library is to be loaded, separate their names with commas. This parameter cannot be changed after the start of a particular session.

Because this is not a superuser-only option, the libraries that can be loaded are restricted to those appearing in the `plugins` subdirectory of the installation's standard library directory. (It is the database administrator's responsibility to ensure that only "safe" libraries are installed there.) Entries in `local_preload_libraries` can specify this directory explicitly, for example `$libdir/plugins/mylib`, or just specify the library name — `mylib` would have the same effect as `$libdir/plugins/mylib`.

There is no performance advantage to loading a library at session start rather than when it is first used. Rather, the intent of this feature is to allow debugging or performance-measurement libraries to be loaded into specific sessions without an explicit `LOAD` command being given. For example, debugging could be enabled for all sessions under a given user name by setting this parameter with `ALTER USER SET`.

If a specified library is not found, the connection attempt will fail.

Every PostgreSQL-supported library has a "magic block" that is checked to guarantee compatibility. For this reason, non-PostgreSQL libraries cannot be loaded in this way.

17.11. Lock Management

`deadlock_timeout` (`integer`)

This is the amount of time, in milliseconds, to wait on a lock before checking to see if there is a deadlock condition. The check for deadlock is relatively slow, so the server doesn't run it every time it waits for a lock. We (optimistically?) assume that deadlocks are not common in production applications and just wait on the lock for a while before starting the check for a deadlock. Increasing this value reduces the amount of time wasted in needless deadlock checks, but slows down reporting of real deadlock errors. The default is one second (`1s`), which is probably about the smallest value you would want in practice. On a heavily loaded server you might want to raise it. Ideally the setting should exceed your typical transaction time, so as to improve the odds that a lock will be released before the waiter decides to check for deadlock.

`max_locks_per_transaction` (`integer`)

The shared lock table is created to track locks on `max_locks_per_transaction` * (max_connections + max_prepared_transactions) objects (e.g. tables); hence, no more than this many distinct objects can be locked at any one time. This parameter controls the average number of object locks allocated for each transaction; individual transactions can lock more objects as long as

the locks of all transactions fit in the lock table. This is *not* the number of rows that can be locked; that value is unlimited. The default, 64, has historically proven sufficient, but you might need to raise this value if you have clients that touch many different tables in a single transaction. This parameter can only be set at server start.

Increasing this parameter may cause PostgreSQL to request more System V shared memory than your operating system's default configuration allows. See Section 16.4.1 for information on how to adjust those parameters, if necessary.

17.12. Version and Platform Compatibility

17.12.1. Previous PostgreSQL Versions

`add_missing_from` (`boolean`)

When on, tables that are referenced by a query will be automatically added to the `FROM` clause if not already present. This behavior does not comply with the SQL standard and many people dislike it because it can mask mistakes (such as referencing a table where you should have referenced its alias). The default is `off`. This variable can be enabled for compatibility with releases of PostgreSQL prior to 8.1, where this behavior was allowed by default.

Note that even when this variable is enabled, a warning message will be emitted for each implicit `FROM` entry referenced by a query. Users are encouraged to update their applications to not rely on this behavior, by adding all tables referenced by a query to the query's `FROM` clause (or its `USING` clause in the case of `DELETE`).

`array_nulls` (`boolean`)

This controls whether the array input parser recognizes unquoted `NULL` as specifying a null array element. By default, this is `on`, allowing array values containing null values to be entered. However, PostgreSQL versions before 8.2 did not support null values in arrays, and therefore would treat `NULL` as specifying a normal array element with the string value "NULL". For backwards compatibility with applications that require the old behavior, this variable can be turned `off`.

Note that it is possible to create array values containing null values even when this variable is `off`.

`backslash_quote` (`string`)

This controls whether a quote mark can be represented by `\'` in a string literal. The preferred, SQL-standard way to represent a quote mark is by doubling it (`''`) but PostgreSQL has historically also accepted `\'`. However, use of `\'` creates security risks because in some client character set encodings, there are multibyte characters in which the last byte is numerically equivalent to ASCII `\`. If client-side code does escaping incorrectly then a SQL-injection attack is possible. This risk can be prevented by making the server reject queries in which a quote mark appears to be escaped by a backslash. The allowed values of `backslash_quote` are `on` (allow `\'` always), `off` (reject always), and `safe_encoding` (allow only if client encoding does not allow ASCII `\` within a multibyte character). `safe_encoding` is the default setting.

Note that in a standard-conforming string literal, `\` just means `\` anyway. This parameter affects the handling of non-standard-conforming literals, including escape string syntax (`E'...'`).

default_with_oids (boolean)

This controls whether CREATE TABLE and CREATE TABLE AS include an OID column in newly-created tables, if neither WITH OIDS nor WITHOUT OIDS is specified. It also determines whether OIDs will be included in tables created by SELECT INTO. In PostgreSQL 8.1 default_with_oids is off by default; in prior versions of PostgreSQL, it was on by default.

The use of OIDs in user tables is considered deprecated, so most installations should leave this variable disabled. Applications that require OIDs for a particular table should specify WITH OIDS when creating the table. This variable can be enabled for compatibility with old applications that do not follow this behavior.

escape_string_warning (boolean)

When on, a warning is issued if a backslash (\) appears in an ordinary string literal ('...' syntax) and standard_conforming_strings is off. The default is on.

Applications that wish to use backslash as escape should be modified to use escape string syntax (E'...'), because the default behavior of ordinary strings will change in a future release for SQL compatibility. This variable can be enabled to help detect applications that will break.

regex_flavor (string)

The regular expression "flavor" can be set to advanced, extended, or basic. The default is advanced. The extended setting may be useful for exact backwards compatibility with pre-7.4 releases of PostgreSQL. See Section 9.7.3.1 for details.

sql_inheritance (boolean)

This controls the inheritance semantics. If turned off, subtables are not included by various commands by default; basically an implied ONLY key word. This was added for compatibility with releases prior to 7.1. See Section 5.8 for more information.

standard_conforming_strings (boolean)

This controls whether ordinary string literals ('...') treat backslashes literally, as specified in the SQL standard. The default is currently off, causing PostgreSQL to have its historical behavior of treating backslashes as escape characters. The default will change to on in a future release to improve compatibility with the standard. Applications may check this parameter to determine how string literals will be processed. The presence of this parameter can also be taken as an indication that the escape string syntax (E'...') is supported. Escape string syntax should be used if an application desires backslashes to be treated as escape characters.

17.12.2. Platform and Client Compatibility

transform_null_equals (boolean)

When on, expressions of the form *expr* = NULL (or NULL = *expr*) are treated as *expr* IS NULL, that is, they return true if *expr* evaluates to the null value, and false otherwise. The correct SQL-spec-compliant behavior of *expr* = NULL is to always return null (unknown). Therefore this parameter defaults to off.

However, filtered forms in Microsoft Access generate queries that appear to use *expr* = NULL to test for null values, so if you use that interface to access the database you might want to turn this option

on. Since expressions of the form *expr* = NULL always return the null value (using the correct interpretation) they are not very useful and do not appear often in normal applications, so this option does little harm in practice. But new users are frequently confused about the semantics of expressions involving null values, so this option is not on by default.

Note that this option only affects the exact form = NULL, not other comparison operators or other expressions that are computationally equivalent to some expression involving the equals operator (such as IN). Thus, this option is not a general fix for bad programming.

Refer to Section 9.2 for related information.

17.13. Preset Options

The following "parameters" are read-only, and are determined when PostgreSQL is compiled or when it is installed. As such, they have been excluded from the sample postgresql.conf file. These options report various aspects of PostgreSQL behavior that may be of interest to certain applications, particularly administrative front-ends.

block_size (integer)

Reports the size of a disk block. It is determined by the value of BLCKSZ when building the server. The default value is 8192 bytes. The meaning of some configuration variables (such as shared_buffers) is influenced by block_size. See Section 17.4 for information.

integer_datetimes (boolean)

Reports whether PostgreSQL was built with support for 64-bit-integer dates and times. It is set by configuring with --enable-integer-datetimes when building PostgreSQL. The default value is off.

lc_collate (string)

Reports the locale in which sorting of textual data is done. See Section 21.1 for more information. The value is determined when the database cluster is initialized.

lc_ctype (string)

Reports the locale that determines character classifications. See Section 21.1 for more information. The value is determined when the database cluster is initialized. Ordinarily this will be the same as lc_collate, but for special applications it might be set differently.

max_function_args (integer)

Reports the maximum number of function arguments. It is determined by the value of FUNC_MAX_ARGS when building the server. The default value is 100 arguments.

max_identifier_length (integer)

Reports the maximum identifier length. It is determined as one less than the value of NAMEDATALEN when building the server. The default value of NAMEDATALEN is 64; therefore the default max_identifier_length is 63 bytes.

max_index_keys (integer)

Reports the maximum number of index keys. It is determined by the value of INDEX_MAX_KEYS when building the server. The default value is 32 keys.

server_encoding (string)

Reports the database encoding (character set). It is determined when the database is created. Ordinarily, clients need only be concerned with the value of client_encoding.

server_version (string)

Reports the version number of the server. It is determined by the value of PG_VERSION when building the server.

server_version_num (integer)

Reports the version number of the server as an integer. It is determined by the value of PG_VERSION_NUM when building the server.

17.14. Customized Options

This feature was designed to allow parameters not normally known to PostgreSQL to be added by add-on modules (such as procedural languages). This allows add-on modules to be configured in the standard ways.

custom_variable_classes (string)

This variable specifies one or several class names to be used for custom variables, in the form of a comma-separated list. A custom variable is a variable not normally known to PostgreSQL proper but used by some add-on module. Such variables must have names consisting of a class name, a dot, and a variable name. custom_variable_classes specifies all the class names in use in a particular installation. This parameter can only be set in the postgresql.conf file or on the server command line.

The difficulty with setting custom variables in postgresql.conf is that the file must be read before add-on modules have been loaded, and so custom variables would ordinarily be rejected as unknown. When custom_variable_classes is set, the server will accept definitions of arbitrary variables within each specified class. These variables will be treated as placeholders and will have no function until the module that defines them is loaded. When a module for a specific class is loaded, it will add the proper variable definitions for its class name, convert any placeholder values according to those definitions, and issue warnings for any placeholders of its class that remain (which presumably would be misspelled configuration variables).

Here is an example of what postgresql.conf might contain when using custom variables:

```
custom_variable_classes = 'plr,plperl'
plr.path = '/usr/lib/R'
plperl.use_strict = true
plruby.use_strict = true        # generates error: unknown class name
```

17.15. Developer Options

The following parameters are intended for work on the PostgreSQL source, and in some cases to assist with recovery of severely damaged databases. There should be no reason to use them in a production database setup. As such, they have been excluded from the sample `postgresql.conf` file. Note that many of these parameters require special source compilation flags to work at all.

`allow_system_table_mods` (`boolean`)

Allows modification of the structure of system tables. This is used by `initdb`. This parameter can only be set at server start.

`debug_assertions` (`boolean`)

Turns on various assertion checks. This is a debugging aid. If you are experiencing strange problems or crashes you might want to turn this on, as it might expose programming mistakes. To use this parameter, the macro `USE_ASSERT_CHECKING` must be defined when PostgreSQL is built (accomplished by the `configure` option `--enable-cassert`). Note that `debug_assertions` defaults to `on` if PostgreSQL has been built with assertions enabled.

`ignore_system_indexes` (`boolean`)

Ignore system indexes when reading system tables (but still update the indexes when modifying the tables). This is useful when recovering from damaged system indexes. This parameter cannot be changed after session start.

`post_auth_delay` (`integer`)

If nonzero, a delay of this many seconds occurs when a new server process is started, after it conducts the authentication procedure. This is intended to give an opportunity to attach to the server process with a debugger. This parameter cannot be changed after session start.

`pre_auth_delay` (`integer`)

If nonzero, a delay of this many seconds occurs just after a new server process is forked, before it conducts the authentication procedure. This is intended to give an opportunity to attach to the server process with a debugger to trace down misbehavior in authentication. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`trace_notify` (`boolean`)

Generates a great amount of debugging output for the `LISTEN` and `NOTIFY` commands. client_min_messages or log_min_messages must be `DEBUG1` or lower to send this output to the client or server log, respectively.

`trace_sort` (`boolean`)

If on, emit information about resource usage during sort operations. This parameter is only available if the `TRACE_SORT` macro was defined when PostgreSQL was compiled. (However, `TRACE_SORT` is currently defined by default.)

trace_locks (boolean)
trace_lwlocks (boolean)
trace_userlocks (boolean)
trace_lock_oidmin (boolean)
trace_lock_table (boolean)
debug_deadlocks (boolean)
log_btree_build_stats (boolean)

Various other code tracing and debugging options.

wal_debug (boolean)

If on, emit WAL-related debugging output. This parameter is only available if the WAL_DEBUG macro was defined when PostgreSQL was compiled.

zero_damaged_pages (boolean)

Detection of a damaged page header normally causes PostgreSQL to report an error, aborting the current command. Setting zero_damaged_pages to on causes the system to instead report a warning, zero out the damaged page, and continue processing. This behavior *will destroy data*, namely all the rows on the damaged page. But it allows you to get past the error and retrieve rows from any undamaged pages that may be present in the table. So it is useful for recovering data if corruption has occurred due to hardware or software error. You should generally not set this on until you have given up hope of recovering data from the damaged page(s) of a table. The default setting is off, and it can only be changed by a superuser.

17.16. Short Options

For convenience there are also single letter command-line option switches available for some parameters. They are described in Table 17-1. Some of these options exist for historical reasons, and their presence as a single-letter option does not necessarily indicate an endorsement to use the option heavily.

Table 17-1. Short option key

Short option	Equivalent
-A *x*	debug_assertions = *x*
-B *x*	shared_buffers = *x*
-d *x*	log_min_messages = DEBUG*x*
-e	datestyle = euro
-fb, -fh, -fi, -fm, -fn, -fs, -ft	enable_bitmapscan = off, enable_hashjoin = off, enable_indexscan = off, enable_mergejoin = off, enable_nestloop = off, enable_seqscan = off, enable_tidscan = off
-F	fsync = off
-h *x*	listen_addresses = *x*

Short option	Equivalent
-i	listen_addresses = '*'
-k *x*	unix_socket_directory = *x*
-l	ssl = on
-N *x*	max_connections = *x*
-O	allow_system_table_mods = on
-p *x*	port = *x*
-P	ignore_system_indexes = on
-s	log_statement_stats = on
-S *x*	work_mem = *x*
-tpa, -tpl, -te	log_parser_stats = on, log_planner_stats = on, log_executor_stats = on
-W *x*	post_auth_delay = *x*

Chapter 18. Database Roles and Privileges

PostgreSQL manages database access permissions using the concept of *roles*. A role can be thought of as either a database user, or a group of database users, depending on how the role is set up. Roles can own database objects (for example, tables) and can assign privileges on those objects to other roles to control who has access to which objects. Furthermore, it is possible to grant *membership* in a role to another role, thus allowing the member role use of privileges assigned to the role it is a member of.

The concept of roles subsumes the concepts of "users" and "groups". In PostgreSQL versions before 8.1, users and groups were distinct kinds of entities, but now there are only roles. Any role can act as a user, a group, or both.

This chapter describes how to create and manage roles and introduces the privilege system. More information about the various types of database objects and the effects of privileges can be found in Chapter 5.

18.1. Database Roles

Database roles are conceptually completely separate from operating system users. In practice it might be convenient to maintain a correspondence, but this is not required. Database roles are global across a database cluster installation (and not per individual database). To create a role use the *CREATE ROLE* SQL command:

```
CREATE ROLE name;
```

name follows the rules for SQL identifiers: either unadorned without special characters, or double-quoted. (In practice, you will usually want to add additional options, such as `LOGIN`, to the command. More details appear below.) To remove an existing role, use the analogous *DROP ROLE* command:

```
DROP ROLE name;
```

For convenience, the programs createuser and dropuser are provided as wrappers around these SQL commands that can be called from the shell command line:

```
createuser name
dropuser name
```

To determine the set of existing roles, examine the `pg_roles` system catalog, for example

```
SELECT rolname FROM pg_roles;
```

The psql program's `\du` meta-command is also useful for listing the existing roles.

In order to bootstrap the database system, a freshly initialized system always contains one predefined role. This role is always a "superuser", and by default (unless altered when running `initdb`) it will have the same name as the operating system user that initialized the database cluster. Customarily, this role will be named `postgres`. In order to create more roles you first have to connect as this initial role.

Every connection to the database server is made in the name of some particular role, and this role determines the initial access privileges for commands issued on that connection. The role name to use for a particular database connection is indicated by the client that is initiating the connection request in an application-specific fashion. For example, the `psql` program uses the `-U` command line option to indicate the role to connect as. Many applications assume the name of the current operating system user by default (including `createuser` and `psql`). Therefore it is often convenient to maintain a naming correspondence between roles and operating system users.

The set of database roles a given client connection may connect as is determined by the client authentication setup, as explained in Chapter 20. (Thus, a client is not necessarily limited to connect as the role with the same name as its operating system user, just as a person's login name need not match her real name.) Since the role identity determines the set of privileges available to a connected client, it is important to carefully configure this when setting up a multiuser environment.

18.2. Role Attributes

A database role may have a number of attributes that define its privileges and interact with the client authentication system.

login privilege

Only roles that have the `LOGIN` attribute can be used as the initial role name for a database connection. A role with the `LOGIN` attribute can be considered the same thing as a "database user". To create a role with login privilege, use either

```
CREATE ROLE name LOGIN;
CREATE USER name;
```

(`CREATE USER` is equivalent to `CREATE ROLE` except that `CREATE USER` assumes `LOGIN` by default, while `CREATE ROLE` does not.)

superuser status

A database superuser bypasses all permission checks. This is a dangerous privilege and should not be used carelessly; it is best to do most of your work as a role that is not a superuser. To create a new database superuser, use `CREATE ROLE name SUPERUSER`. You must do this as a role that is already a superuser.

database creation

A role must be explicitly given permission to create databases (except for superusers, since those bypass all permission checks). To create such a role, use `CREATE ROLE name CREATEDB`.

role creation

A role must be explicitly given permission to create more roles (except for superusers, since those bypass all permission checks). To create such a role, use `CREATE ROLE name CREATEROLE`. A role with `CREATEROLE` privilege can alter and drop other roles, too, as well as grant or revoke membership in them. However, to create, alter, drop, or change membership of a superuser role, superuser status is required; `CREATEROLE` is not sufficient for that.

password

A password is only significant if the client authentication method requires the user to supply a password when connecting to the database. The `password`, `md5`, and `crypt` authentication methods make use of passwords. Database passwords are separate from operating system passwords. Specify a password upon role creation with `CREATE ROLE` *`name`* `PASSWORD '`*`string`*`'`.

A role's attributes can be modified after creation with `ALTER ROLE`. See the reference pages for the *CREATE ROLE* and *ALTER ROLE* commands for details.

Tip: It is good practice to create a role that has the `CREATEDB` and `CREATEROLE` privileges, but is not a superuser, and then use this role for all routine management of databases and roles. This approach avoids the dangers of operating as a superuser for tasks that do not really require it.

A role can also have role-specific defaults for many of the run-time configuration settings described in Chapter 17. For example, if for some reason you want to disable index scans (hint: not a good idea) anytime you connect, you can use

```
ALTER ROLE myname SET enable_indexscan TO off;
```

This will save the setting (but not set it immediately). In subsequent connections by this role it will appear as though `SET enable_indexscan TO off;` had been executed just before the session started. You can still alter this setting during the session; it will only be the default. To remove a role-specific default setting, use `ALTER ROLE` *`rolename`* `RESET` *`varname`*`;`. Note that role-specific defaults attached to roles without `LOGIN` privilege are fairly useless, since they will never be invoked.

18.3. Privileges

When an object is created, it is assigned an owner. The owner is normally the role that executed the creation statement. For most kinds of objects, the initial state is that only the owner (or a superuser) can do anything with the object. To allow other roles to use it, *privileges* must be granted. There are several different kinds of privilege: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `REFERENCES`, `TRIGGER`, `CREATE`, `CONNECT`, `TEMPORARY`, `EXECUTE`, and `USAGE`. For more information on the different types of privileges supported by PostgreSQL, see the *GRANT* reference page.

To assign privileges, the `GRANT` command is used. So, if `joe` is an existing role, and `accounts` is an existing table, the privilege to update the table can be granted with

```
GRANT UPDATE ON accounts TO joe;
```

The special name `PUBLIC` can be used to grant a privilege to every role on the system. Writing `ALL` in place of a specific privilege specifies that all privileges that apply to the object will be granted.

To revoke a privilege, use the fittingly named *REVOKE* command:

```
REVOKE ALL ON accounts FROM PUBLIC;
```

The special privileges of an object's owner (i.e., the right to modify or destroy the object) are always implicit in being the owner, and cannot be granted or revoked. But the owner can choose to revoke his own ordinary privileges, for example to make a table read-only for himself as well as others.

An object can be assigned to a new owner with an `ALTER` command of the appropriate kind for the object. Superusers can always do this; ordinary roles can only do it if they are both the current owner of the object (or a member of the owning role) and a member of the new owning role.

18.4. Role Membership

It is frequently convenient to group users together to ease management of privileges: that way, privileges can be granted to, or revoked from, a group as a whole. In PostgreSQL this is done by creating a role that represents the group, and then granting *membership* in the group role to individual user roles.

To set up a group role, first create the role:

```
CREATE ROLE name;
```

Typically a role being used as a group would not have the `LOGIN` attribute, though you can set it if you wish.

Once the group role exists, you can add and remove members using the *GRANT* and *REVOKE* commands:

```
GRANT group_role TO role1, ... ;
REVOKE group_role FROM role1, ... ;
```

You can grant membership to other group roles, too (since there isn't really any distinction between group roles and non-group roles). The database will not let you set up circular membership loops. Also, it is not permitted to grant membership in a role to `PUBLIC`.

The members of a role can use the privileges of the group role in two ways. First, every member of a group can explicitly do *SET ROLE* to temporarily "become" the group role. In this state, the database session has access to the privileges of the group role rather than the original login role, and any database objects created are considered owned by the group role not the login role. Second, member roles that have the `INHERIT` attribute automatically have use of privileges of roles they are members of. As an example, suppose we have done

```
CREATE ROLE joe LOGIN INHERIT;
CREATE ROLE admin NOINHERIT;
CREATE ROLE wheel NOINHERIT;
GRANT admin TO joe;
GRANT wheel TO admin;
```

Immediately after connecting as role `joe`, a database session will have use of privileges granted directly to `joe` plus any privileges granted to `admin`, because `joe` "inherits" `admin`'s privileges. However, privileges granted to `wheel` are not available, because even though `joe` is indirectly a member of `wheel`, the membership is via `admin` which has the `NOINHERIT` attribute. After

```
SET ROLE admin;
```

the session would have use of only those privileges granted to `admin`, and not those granted to `joe`. After

```
SET ROLE wheel;
```

the session would have use of only those privileges granted to `wheel`, and not those granted to either `joe` or `admin`. The original privilege state can be restored with any of

```
SET ROLE joe;
SET ROLE NONE;
RESET ROLE;
```

> **Note:** The `SET ROLE` command always allows selecting any role that the original login role is directly or indirectly a member of. Thus, in the above example, it is not necessary to become `admin` before becoming `wheel`.

> **Note:** In the SQL standard, there is a clear distinction between users and roles, and users do not automatically inherit privileges while roles do. This behavior can be obtained in PostgreSQL by giving roles being used as SQL roles the `INHERIT` attribute, while giving roles being used as SQL users the `NOINHERIT` attribute. However, PostgreSQL defaults to giving all roles the `INHERIT` attribute, for backwards compatibility with pre-8.1 releases in which users always had use of permissions granted to groups they were members of.

The role attributes `LOGIN`, `SUPERUSER`, `CREATEDB`, and `CREATEROLE` can be thought of as special privileges, but they are never inherited as ordinary privileges on database objects are. You must actually `SET ROLE` to a specific role having one of these attributes in order to make use of the attribute. Continuing the above example, we might well choose to grant `CREATEDB` and `CREATEROLE` to the `admin` role. Then a session connecting as role `joe` would not have these privileges immediately, only after doing `SET ROLE admin`.

To destroy a group role, use *DROP ROLE*:

```
DROP ROLE name;
```

Any memberships in the group role are automatically revoked (but the member roles are not otherwise affected). Note however that any objects owned by the group role must first be dropped or reassigned to other owners; and any permissions granted to the group role must be revoked.

18.5. Functions and Triggers

Functions and triggers allow users to insert code into the backend server that other users may execute unintentionally. Hence, both mechanisms permit users to "Trojan horse" others with relative ease. The only real protection is tight control over who can define functions.

Functions run inside the backend server process with the operating system permissions of the database server daemon. If the programming language used for the function allows unchecked memory accesses, it is possible to change the server's internal data structures. Hence, among many other things, such functions

can circumvent any system access controls. Function languages that allow such access are considered "untrusted", and PostgreSQL allows only superusers to create functions written in those languages.

Chapter 19. Managing Databases

Every instance of a running PostgreSQL server manages one or more databases. Databases are therefore the topmost hierarchical level for organizing SQL objects ("database objects"). This chapter describes the properties of databases, and how to create, manage, and destroy them.

19.1. Overview

A database is a named collection of SQL objects ("database objects"). Generally, every database object (tables, functions, etc.) belongs to one and only one database. (But there are a few system catalogs, for example `pg_database`, that belong to a whole cluster and are accessible from each database within the cluster.) More accurately, a database is a collection of schemas and the schemas contain the tables, functions, etc. So the full hierarchy is: server, database, schema, table (or some other kind of object, such as a function).

When connecting to the database server, a client must specify in its connection request the name of the database it wants to connect to. It is not possible to access more than one database per connection. (But an application is not restricted in the number of connections it opens to the same or other databases.) Databases are physically separated and access control is managed at the connection level. If one PostgreSQL server instance is to house projects or users that should be separate and for the most part unaware of each other, it is therefore recommendable to put them into separate databases. If the projects or users are interrelated and should be able to use each other's resources they should be put in the same database, but possibly into separate schemas. Schemas are a purely logical structure and who can access what is managed by the privilege system. More information about managing schemas is in Section 5.7.

Databases are created with the `CREATE DATABASE` command (see Section 19.2) and destroyed with the `DROP DATABASE` command (see Section 19.5). To determine the set of existing databases, examine the `pg_database` system catalog, for example

```
SELECT datname FROM pg_database;
```

The psql program's `\l` meta-command and `-l` command-line option are also useful for listing the existing databases.

Note: The SQL standard calls databases "catalogs", but there is no difference in practice.

19.2. Creating a Database

In order to create a database, the PostgreSQL server must be up and running (see Section 16.3).

Databases are created with the SQL command *CREATE DATABASE*:

```
CREATE DATABASE name;
```

where *name* follows the usual rules for SQL identifiers. The current role automatically becomes the owner of the new database. It is the privilege of the owner of a database to remove it later on (which also removes all the objects in it, even if they have a different owner).

The creation of databases is a restricted operation. See Section 18.2 for how to grant permission.

Since you need to be connected to the database server in order to execute the CREATE DATABASE command, the question remains how the *first* database at any given site can be created. The first database is always created by the initdb command when the data storage area is initialized. (See Section 16.2.) This database is called postgres. So to create the first "ordinary" database you can connect to postgres.

A second database, template1, is also created by initdb. Whenever a new database is created within the cluster, template1 is essentially cloned. This means that any changes you make in template1 are propagated to all subsequently created databases. Therefore it is unwise to use template1 for real work, but when used judiciously this feature can be convenient. More details appear in Section 19.3.

As a convenience, there is a program that you can execute from the shell to create new databases, createdb.

```
createdb dbname
```

createdb does no magic. It connects to the postgres database and issues the CREATE DATABASE command, exactly as described above. The createdb reference page contains the invocation details. Note that createdb without any arguments will create a database with the current user name, which may or may not be what you want.

> **Note:** Chapter 20 contains information about how to restrict who can connect to a given database.

Sometimes you want to create a database for someone else. That role should become the owner of the new database, so he can configure and manage it himself. To achieve that, use one of the following commands:

```
CREATE DATABASE dbname OWNER rolename;
```

from the SQL environment, or

```
createdb -O rolename dbname
```

from the shell. You must be a superuser to be allowed to create a database for someone else (that is, for a role you are not a member of).

19.3. Template Databases

CREATE DATABASE actually works by copying an existing database. By default, it copies the standard system database named template1. Thus that database is the "template" from which new databases are made. If you add objects to template1, these objects will be copied into subsequently created user databases. This behavior allows site-local modifications to the standard set of objects in databases. For example, if you install the procedural language PL/pgSQL in template1, it will automatically be available in user databases without any extra action being taken when those databases are made.

There is a second standard system database named template0. This database contains the same data as the initial contents of template1, that is, only the standard objects predefined by your version of PostgreSQL. template0 should never be changed after initdb. By instructing CREATE DATABASE to copy template0 instead of template1, you can create a "virgin" user database that contains none of the site-local additions in template1. This is particularly handy when restoring a pg_dump dump: the dump script should be restored in a virgin database to ensure that one recreates the correct contents of the dumped database, without any conflicts with additions that may now be present in template1.

To create a database by copying template0, use

```
CREATE DATABASE dbname TEMPLATE template0;
```

from the SQL environment, or

```
createdb -T template0 dbname
```

from the shell.

It is possible to create additional template databases, and indeed one may copy any database in a cluster by specifying its name as the template for CREATE DATABASE. It is important to understand, however, that this is not (yet) intended as a general-purpose "COPY DATABASE" facility. The principal limitation is that no other sessions can be connected to the source database while it is being copied. CREATE DATABASE will fail if any other connection exists when it starts; otherwise, new connections to the source database are locked out until CREATE DATABASE completes.

Two useful flags exist in pg_database for each database: the columns datistemplate and datallowconn. datistemplate may be set to indicate that a database is intended as a template for CREATE DATABASE. If this flag is set, the database may be cloned by any user with CREATEDB privileges; if it is not set, only superusers and the owner of the database may clone it. If datallowconn is false, then no new connections to that database will be allowed (but existing sessions are not killed simply by setting the flag false). The template0 database is normally marked datallowconn = false to prevent modification of it. Both template0 and template1 should always be marked with datistemplate = true.

> **Note:** template1 and template0 do not have any special status beyond the fact that the name template1 is the default source database name for CREATE DATABASE. For example, one could drop template1 and recreate it from template0 without any ill effects. This course of action might be advisable if one has carelessly added a bunch of junk in template1. (To delete template1, it must have datistemplate = false.)
>
> The postgres database is also created when a database cluster is initialized. This database is meant as a default database for users and applications to connect to. It is simply a copy of template1 and may be dropped and recreated if required.

19.4. Database Configuration

Recall from Chapter 17 that the PostgreSQL server provides a large number of run-time configuration variables. You can set database-specific default values for many of these settings.

For example, if for some reason you want to disable the GEQO optimizer for a given database, you'd ordinarily have to either disable it for all databases or make sure that every connecting client is careful to issue `SET geqo TO off;`. To make this setting the default within a particular database, you can execute the command

```
ALTER DATABASE mydb SET geqo TO off;
```

This will save the setting (but not set it immediately). In subsequent connections to this database it will appear as though `SET geqo TO off;` had been executed just before the session started. Note that users can still alter this setting during their sessions; it will only be the default. To undo any such setting, use `ALTER DATABASE` *`dbname`* `RESET` *`varname`*`;`.

19.5. Destroying a Database

Databases are destroyed with the command *DROP DATABASE*:

```
DROP DATABASE name;
```

Only the owner of the database, or a superuser, can drop a database. Dropping a database removes all objects that were contained within the database. The destruction of a database cannot be undone.

You cannot execute the `DROP DATABASE` command while connected to the victim database. You can, however, be connected to any other database, including the `template1` database. `template1` would be the only option for dropping the last user database of a given cluster.

For convenience, there is also a shell program to drop databases, dropdb:

```
dropdb dbname
```

(Unlike `createdb`, it is not the default action to drop the database with the current user name.)

19.6. Tablespaces

Tablespaces in PostgreSQL allow database administrators to define locations in the file system where the files representing database objects can be stored. Once created, a tablespace can be referred to by name when creating database objects.

By using tablespaces, an administrator can control the disk layout of a PostgreSQL installation. This is useful in at least two ways. First, if the partition or volume on which the cluster was initialized runs out of space and cannot be extended, a tablespace can be created on a different partition and used until the system can be reconfigured.

Second, tablespaces allow an administrator to use knowledge of the usage pattern of database objects to optimize performance. For example, an index which is very heavily used can be placed on a very fast, highly available disk, such as an expensive solid state device. At the same time a table storing archived data which is rarely used or not performance critical could be stored on a less expensive, slower disk system.

To define a tablespace, use the *CREATE TABLESPACE* command, for example:

```
CREATE TABLESPACE fastspace LOCATION '/mnt/sda1/postgresql/data';
```

The location must be an existing, empty directory that is owned by the PostgreSQL system user. All objects subsequently created within the tablespace will be stored in files underneath this directory.

> **Note:** There is usually not much point in making more than one tablespace per logical file system, since you cannot control the location of individual files within a logical file system. However, PostgreSQL does not enforce any such limitation, and indeed it is not directly aware of the file system boundaries on your system. It just stores files in the directories you tell it to use.

Creation of the tablespace itself must be done as a database superuser, but after that you can allow ordinary database users to make use of it. To do that, grant them the `CREATE` privilege on it.

Tables, indexes, and entire databases can be assigned to particular tablespaces. To do so, a user with the `CREATE` privilege on a given tablespace must pass the tablespace name as a parameter to the relevant command. For example, the following creates a table in the tablespace `space1`:

```
CREATE TABLE foo(i int) TABLESPACE space1;
```

Alternatively, use the default_tablespace parameter:

```
SET default_tablespace = space1;
CREATE TABLE foo(i int);
```

When `default_tablespace` is set to anything but an empty string, it supplies an implicit `TABLESPACE` clause for `CREATE TABLE` and `CREATE INDEX` commands that do not have an explicit one.

The tablespace associated with a database is used to store the system catalogs of that database, as well as any temporary files created by server processes using that database. Furthermore, it is the default tablespace selected for tables and indexes created within the database, if no `TABLESPACE` clause is given (either explicitly or via `default_tablespace`) when the objects are created. If a database is created without specifying a tablespace for it, it uses the same tablespace as the template database it is copied from.

Two tablespaces are automatically created by `initdb`. The `pg_global` tablespace is used for shared system catalogs. The `pg_default` tablespace is the default tablespace of the `template1` and `template0` databases (and, therefore, will be the default tablespace for other databases as well, unless overridden by a `TABLESPACE` clause in `CREATE DATABASE`).

Once created, a tablespace can be used from any database, provided the requesting user has sufficient privilege. This means that a tablespace cannot be dropped until all objects in all databases using the tablespace have been removed.

To remove an empty tablespace, use the *DROP TABLESPACE* command.

To determine the set of existing tablespaces, examine the `pg_tablespace` system catalog, for example

```
SELECT spcname FROM pg_tablespace;
```

The psql program's `\db` meta-command is also useful for listing the existing tablespaces.

PostgreSQL makes extensive use of symbolic links to simplify the implementation of tablespaces. This means that tablespaces can be used *only* on systems that support symbolic links.

The directory `$PGDATA/pg_tblspc` contains symbolic links that point to each of the non-built-in tablespaces defined in the cluster. Although not recommended, it is possible to adjust the tablespace layout by hand by redefining these links. Two warnings: do not do so while the server is running; and after you restart the server, update the `pg_tablespace` catalog to show the new locations. (If you do not, `pg_dump` will continue to show the old tablespace locations.)

Chapter 20. Client Authentication

When a client application connects to the database server, it specifies which PostgreSQL database user name it wants to connect as, much the same way one logs into a Unix computer as a particular user. Within the SQL environment the active database user name determines access privileges to database objects — see Chapter 18 for more information. Therefore, it is essential to restrict which database users can connect.

> **Note:** As explained in Chapter 18, PostgreSQL actually does privilege management in terms of "roles". In this chapter, we consistently use *database user* to mean "role with the `LOGIN` privilege".

Authentication is the process by which the database server establishes the identity of the client, and by extension determines whether the client application (or the user who runs the client application) is permitted to connect with the database user name that was requested.

PostgreSQL offers a number of different client authentication methods. The method used to authenticate a particular client connection can be selected on the basis of (client) host address, database, and user.

PostgreSQL database user names are logically separate from user names of the operating system in which the server runs. If all the users of a particular server also have accounts on the server's machine, it makes sense to assign database user names that match their operating system user names. However, a server that accepts remote connections may have many database users who have no local operating system account, and in such cases there need be no connection between database user names and OS user names.

20.1. The `pg_hba.conf` file

Client authentication is controlled by a configuration file, which traditionally is named `pg_hba.conf` and is stored in the database cluster's data directory. (HBA stands for host-based authentication.) A default `pg_hba.conf` file is installed when the data directory is initialized by `initdb`. It is possible to place the authentication configuration file elsewhere, however; see the hba_file configuration parameter.

The general format of the `pg_hba.conf` file is a set of records, one per line. Blank lines are ignored, as is any text after the `#` comment character. A record is made up of a number of fields which are separated by spaces and/or tabs. Fields can contain white space if the field value is quoted. Records cannot be continued across lines.

Each record specifies a connection type, a client IP address range (if relevant for the connection type), a database name, a user name, and the authentication method to be used for connections matching these parameters. The first record with a matching connection type, client address, requested database, and user name is used to perform authentication. There is no "fall-through" or "backup": if one record is chosen and the authentication fails, subsequent records are not considered. If no record matches, access is denied.

A record may have one of the seven formats

```
local      database  user  auth-method  [auth-option]
host       database  user  CIDR-address  auth-method  [auth-option]
hostssl    database  user  CIDR-address  auth-method  [auth-option]
hostnossl  database  user  CIDR-address  auth-method  [auth-option]
host       database  user  IP-address  IP-mask  auth-method  [auth-option]
hostssl    database  user  IP-address  IP-mask  auth-method  [auth-option]
```

```
hostnossl  database  user  IP-address  IP-mask  auth-method  [auth-option]
```

The meaning of the fields is as follows:

`local`

This record matches connection attempts using Unix-domain sockets. Without a record of this type, Unix-domain socket connections are disallowed.

`host`

This record matches connection attempts made using TCP/IP. `host` records match either SSL or non-SSL connection attempts.

> **Note:** Remote TCP/IP connections will not be possible unless the server is started with an appropriate value for the listen_addresses configuration parameter, since the default behavior is to listen for TCP/IP connections only on the local loopback address `localhost`.

`hostssl`

This record matches connection attempts made using TCP/IP, but only when the connection is made with SSL encryption.

To make use of this option the server must be built with SSL support. Furthermore, SSL must be enabled at server start time by setting the ssl configuration parameter (see Section 16.7 for more information).

`hostnossl`

This record type has the opposite logic to `hostssl`: it only matches connection attempts made over TCP/IP that do not use SSL.

`database`

Specifies which database names this record matches. The value `all` specifies that it matches all databases. The value `sameuser` specifies that the record matches if the requested database has the same name as the requested user. The value `samerole` specifies that the requested user must be a member of the role with the same name as the requested database. (`samegroup` is an obsolete but still accepted spelling of `samerole`.) Otherwise, this is the name of a specific PostgreSQL database. Multiple database names can be supplied by separating them with commas. A separate file containing database names can be specified by preceding the file name with `@`.

`user`

Specifies which database user names this record matches. The value `all` specifies that it matches all users. Otherwise, this is either the name of a specific database user, or a group name preceded by +. (Recall that there is no real distinction between users and groups in PostgreSQL; a + mark really means “match any of the roles that are directly or indirectly members of this role”, while a name without a + mark matches only that specific role.) Multiple user names can be supplied by separating them with commas. A separate file containing user names can be specified by preceding the file name with `@`.

CIDR-address

Specifies the client machine IP address range that this record matches. It contains an IP address in standard dotted decimal notation and a CIDR mask length. (IP addresses can only be specified numerically, not as domain or host names.) The mask length indicates the number of high-order bits of the client IP address that must match. Bits to the right of this must be zero in the given IP address. There must not be any white space between the IP address, the `/`, and the CIDR mask length.

Typical examples of a *CIDR-address* are `172.20.143.89/32` for a single host, or `172.20.143.0/24` for a small network, or `10.6.0.0/16` for a larger one. To specify a single host, use a CIDR mask of 32 for IPv4 or 128 for IPv6. In a network address, do not omit trailing zeroes.

An IP address given in IPv4 format will match IPv6 connections that have the corresponding address, for example `127.0.0.1` will match the IPv6 address `::ffff:127.0.0.1`. An entry given in IPv6 format will match only IPv6 connections, even if the represented address is in the IPv4-in-IPv6 range. Note that entries in IPv6 format will be rejected if the system's C library does not have support for IPv6 addresses.

This field only applies to `host`, `hostssl`, and `hostnossl` records.

IP-address
IP-mask

These fields may be used as an alternative to the *CIDR-address* notation. Instead of specifying the mask length, the actual mask is specified in a separate column. For example, `255.0.0.0` represents an IPv4 CIDR mask length of 8, and `255.255.255.255` represents a CIDR mask length of 32.

These fields only apply to `host`, `hostssl`, and `hostnossl` records.

auth-method

Specifies the authentication method to use when connecting via this record. The possible choices are summarized here; details are in Section 20.2.

`trust`

Allow the connection unconditionally. This method allows anyone that can connect to the PostgreSQL database server to login as any PostgreSQL user they like, without the need for a password. See Section 20.2.1 for details.

`reject`

Reject the connection unconditionally. This is useful for "filtering out" certain hosts from a group.

`md5`

Require the client to supply an MD5-encrypted password for authentication. See Section 20.2.2 for details.

`crypt`

Note: This option is recommended only for communicating with pre-7.2 clients.

Require the client to supply a `crypt()`-encrypted password for authentication. `md5` is now recommended over `crypt`. See Section 20.2.2 for details.

`password`

Require the client to supply an unencrypted password for authentication. Since the password is sent in clear text over the network, this should not be used on untrusted networks. It also does not usually work with threaded client applications. See Section 20.2.2 for details.

`krb5`

Use Kerberos V5 to authenticate the user. This is only available for TCP/IP connections. See Section 20.2.3 for details.

`ident`

Obtain the operating system user name of the client (for TCP/IP connections by contacting the ident server on the client, for local connections by getting it from the operating system) and check if the user is allowed to connect as the requested database user by consulting the map specified after the `ident` key word. See Section 20.2.4 for details.

`ldap`

Authenticate using LDAP to a central server. See Section 20.2.5 for details.

`pam`

Authenticate using the Pluggable Authentication Modules (PAM) service provided by the operating system. See Section 20.2.6 for details.

`auth-option`

The meaning of this optional field depends on the chosen authentication method. Details appear below.

Files included by `@` constructs are read as lists of names, which can be separated by either whitespace or commas. Comments are introduced by `#`, just as in `pg_hba.conf`, and nested `@` constructs are allowed. Unless the file name following `@` is an absolute path, it is taken to be relative to the directory containing the referencing file.

Since the `pg_hba.conf` records are examined sequentially for each connection attempt, the order of the records is significant. Typically, earlier records will have tight connection match parameters and weaker authentication methods, while later records will have looser match parameters and stronger authentication methods. For example, one might wish to use `trust` authentication for local TCP/IP connections but require a password for remote TCP/IP connections. In this case a record specifying `trust` authentication for connections from 127.0.0.1 would appear before a record specifying password authentication for a wider range of allowed client IP addresses.

The `pg_hba.conf` file is read on start-up and when the main server process receives a SIGHUP signal. If you edit the file on an active system, you will need to signal the server (using `pg_ctl reload` or `kill -HUP`) to make it re-read the file.

Tip: To connect to a particular database, a user must not only pass the `pg_hba.conf` checks, but must have the `CONNECT` privilege for the database. If you wish to restrict which users can connect to which databases, it's usually easier to control this by granting/revoking `CONNECT` privilege than to put the rules into `pg_hba.conf` entries.

Some examples of `pg_hba.conf` entries are shown in Example 20-1. See the next section for details on the different authentication methods.

Example 20-1. Example `pg_hba.conf` entries

```
# Allow any user on the local system to connect to any database under
# any database user name using Unix-domain sockets (the default for local
# connections).
#
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD
local   all         all                               trust

# The same using local loopback TCP/IP connections.
#
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD
host    all         all         127.0.0.1/32          trust

# The same as the last line but using a separate netmask column
#
# TYPE  DATABASE    USER        IP-ADDRESS    IP-MASK             METHOD
host    all         all         127.0.0.1     255.255.255.255     trust

# Allow any user from any host with IP address 192.168.93.x to connect
# to database "postgres" as the same user name that ident reports for
# the connection (typically the Unix user name).
#
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD
host    postgres    all         192.168.93.0/24       ident sameuser

# Allow a user from host 192.168.12.10 to connect to database
# "postgres" if the user's password is correctly supplied.
#
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD
host    postgres    all         192.168.12.10/32      md5

# In the absence of preceding "host" lines, these two lines will
# reject all connection from 192.168.54.1 (since that entry will be
# matched first), but allow Kerberos 5 connections from anywhere else
# on the Internet.  The zero mask means that no bits of the host IP
# address are considered so it matches any host.
#
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD
host    all         all         192.168.54.1/32       reject
host    all         all         0.0.0.0/0             krb5

# Allow users from 192.168.x.x hosts to connect to any database, if
# they pass the ident check.  If, for example, ident says the user is
# "bryanh" and he requests to connect as PostgreSQL user "guest1", the
# connection is allowed if there is an entry in pg_ident.conf for map
# "omicron" that says "bryanh" is allowed to connect as "guest1".
#
```

```
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD
host    all         all         192.168.0.0/16        ident omicron

# If these are the only three lines for local connections, they will
# allow local users to connect only to their own databases (databases
# with the same name as their database user name) except for administrators
# and members of role "support", who may connect to all databases.  The file
# $PGDATA/admins contains a list of names of administrators.  Passwords
# are required in all cases.
#
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD
local   sameuser    all                               md5
local   all         @admins                           md5
local   all         +support                          md5

# The last two lines above can be combined into a single line:
local   all         @admins,+support                  md5

# The database column can also use lists and file names:
local   db1,db2,@demodbs  all                         md5
```

20.2. Authentication methods

The following subsections describe the authentication methods in more detail.

20.2.1. Trust authentication

When `trust` authentication is specified, PostgreSQL assumes that anyone who can connect to the server is authorized to access the database with whatever database user name they specify (including superusers). Of course, restrictions made in the `database` and `user` columns still apply. This method should only be used when there is adequate operating-system-level protection on connections to the server.

`trust` authentication is appropriate and very convenient for local connections on a single-user workstation. It is usually *not* appropriate by itself on a multiuser machine. However, you may be able to use `trust` even on a multiuser machine, if you restrict access to the server's Unix-domain socket file using file-system permissions. To do this, set the `unix_socket_permissions` (and possibly `unix_socket_group`) configuration parameters as described in Section 17.3. Or you could set the `unix_socket_directory` configuration parameter to place the socket file in a suitably restricted directory.

Setting file-system permissions only helps for Unix-socket connections. Local TCP/IP connections are not restricted by it; therefore, if you want to use file-system permissions for local security, remove the `host ... 127.0.0.1 ...` line from `pg_hba.conf`, or change it to a non-`trust` authentication method.

`trust` authentication is only suitable for TCP/IP connections if you trust every user on every machine that is allowed to connect to the server by the `pg_hba.conf` lines that specify `trust`. It is seldom reasonable to use `trust` for any TCP/IP connections other than those from localhost (127.0.0.1).

20.2.2. Password authentication

The password-based authentication methods are `md5`, `crypt`, and `password`. These methods operate similarly except for the way that the password is sent across the connection: respectively, MD5-hashed, crypt-encrypted, and clear-text. A limitation is that the `crypt` method does not work with passwords that have been encrypted in `pg_authid`.

If you are at all concerned about password "sniffing" attacks then `md5` is preferred, with `crypt` to be used only if you must support pre-7.2 clients. Plain `password` should be avoided especially for connections over the open Internet (unless you use SSL, SSH, or another communications security wrapper around the connection).

PostgreSQL database passwords are separate from operating system user passwords. The password for each database user is stored in the `pg_authid` system catalog. Passwords can be managed with the SQL commands *CREATE USER* and *ALTER USER*, e.g., **`CREATE USER foo WITH PASSWORD 'secret';`**. By default, that is, if no password has been set up, the stored password is null and password authentication will always fail for that user.

20.2.3. Kerberos authentication

Kerberos is an industry-standard secure authentication system suitable for distributed computing over a public network. A description of the Kerberos system is far beyond the scope of this document; in full generality it can be quite complex (yet powerful). The Kerberos FAQ[1] or MIT Kerberos page[2] can be good starting points for exploration. Several sources for Kerberos distributions exist. Kerberos provides secure authentication but does not encrypt queries or data passed over the network; for that use SSL.

PostgreSQL supports Kerberos version 5. Kerberos support has to be enabled when PostgreSQL is built; see Chapter 14 for more information.

PostgreSQL operates like a normal Kerberos service. The name of the service principal is *`servicename`*`/`*`hostname`*`@`*`realm`*.

`servicename` can be set on the server side using the krb_srvname configuration parameter, and on the client side using the `krbsrvname` connection parameter. (See also Section 29.1.) The installation default can be changed from the default `postgres` at build time using `./configure --with-krb-srvnam=whatever`. In most environments, this parameter never needs to be changed. However, to support multiple PostgreSQL installations on the same host it is necessary. Some Kerberos implementations may also require a different service name, such as Microsoft Active Directory which requires the service name to be in uppercase (`POSTGRES`).

`hostname` is the fully qualified host name of the server machine. The service principal's realm is the preferred realm of the server machine.

Client principals must have their PostgreSQL database user name as their first component, for example `pgusername/otherstuff@realm`. At present the realm of the client is not checked by PostgreSQL; so if you have cross-realm authentication enabled, then any principal in any realm that can communicate with yours will be accepted.

Make sure that your server keytab file is readable (and preferably only readable) by the PostgreSQL server account. (See also Section 16.1.) The location of the key file is specified by the krb_server_keyfile config-

1. http://www.nrl.navy.mil/CCS/people/kenh/kerberos-faq.html
2. http://web.mit.edu/kerberos/www/

uration parameter. The default is `/usr/local/pgsql/etc/krb5.keytab` (or whichever directory was specified as `sysconfdir` at build time).

The keytab file is generated by the Kerberos software; see the Kerberos documentation for details. The following example is for MIT-compatible Kerberos 5 implementations:

```
kadmin% ank -randkey postgres/server.my.domain.org
kadmin% ktadd -k krb5.keytab postgres/server.my.domain.org
```

When connecting to the database make sure you have a ticket for a principal matching the requested database user name. For example, for database user name `fred`, both principal `fred@EXAMPLE.COM` and `fred/users.example.com@EXAMPLE.COM` could be used to authenticate to the database server.

If you use mod_auth_kerb[3] and mod_perl on your Apache web server, you can use `AuthType KerberosV5SaveCredentials` with a mod_perl script. This gives secure database access over the web, no extra passwords required.

20.2.4. Ident-based authentication

The ident authentication method works by obtaining the client's operating system user name, then determining the allowed database user names using a map file that lists the permitted corresponding pairs of names. The determination of the client's user name is the security-critical point, and it works differently depending on the connection type.

20.2.4.1. Ident Authentication over TCP/IP

The "Identification Protocol" is described in *RFC 1413*. Virtually every Unix-like operating system ships with an ident server that listens on TCP port 113 by default. The basic functionality of an ident server is to answer questions like "What user initiated the connection that goes out of your port *X* and connects to my port *Y*?". Since PostgreSQL knows both *X* and *Y* when a physical connection is established, it can interrogate the ident server on the host of the connecting client and could theoretically determine the operating system user for any given connection this way.

The drawback of this procedure is that it depends on the integrity of the client: if the client machine is untrusted or compromised an attacker could run just about any program on port 113 and return any user name he chooses. This authentication method is therefore only appropriate for closed networks where each client machine is under tight control and where the database and system administrators operate in close contact. In other words, you must trust the machine running the ident server. Heed the warning:

> The Identification Protocol is not intended as an authorization or access control protocol.
>
> —RFC 1413

Some ident servers have a nonstandard option that causes the returned user name to be encrypted, using a key that only the originating machine's administrator knows. This option *must not* be used when using the ident server with PostgreSQL, since PostgreSQL does not have any way to decrypt the returned string to determine the actual user name.

3. http://modauthkerb.sf.net

20.2.4.2. Ident Authentication over Local Sockets

On systems supporting `SO_PEERCRED` requests for Unix-domain sockets (currently Linux, FreeBSD, NetBSD, OpenBSD, and BSD/OS), ident authentication can also be applied to local connections. In this case, no security risk is added by using ident authentication; indeed it is a preferable choice for local connections on such systems.

On systems without `SO_PEERCRED` requests, ident authentication is only available for TCP/IP connections. As a work-around, it is possible to specify the localhost address 127.0.0.1 and make connections to this address. This method is trustworthy to the extent that you trust the local ident server.

20.2.4.3. Ident Maps

When using ident-based authentication, after having determined the name of the operating system user that initiated the connection, PostgreSQL checks whether that user is allowed to connect as the database user he is requesting to connect as. This is controlled by the ident map argument that follows the `ident` key word in the `pg_hba.conf` file. There is a predefined ident map `sameuser`, which allows any operating system user to connect as the database user of the same name (if the latter exists). Other maps must be created manually.

Ident maps other than `sameuser` are defined in the ident map file, which by default is named `pg_ident.conf` and is stored in the cluster's data directory. (It is possible to place the map file elsewhere, however; see the ident_file configuration parameter.) The ident map file contains lines of the general form:

```
map-name ident-username database-username
```

Comments and whitespace are handled in the same way as in `pg_hba.conf`. The *`map-name`* is an arbitrary name that will be used to refer to this mapping in `pg_hba.conf`. The other two fields specify which operating system user is allowed to connect as which database user. The same *`map-name`* can be used repeatedly to specify more user-mappings within a single map. There is no restriction regarding how many database users a given operating system user may correspond to, nor vice versa.

The `pg_ident.conf` file is read on start-up and when the main server process receives a SIGHUP signal. If you edit the file on an active system, you will need to signal the server (using `pg_ctl reload` or `kill -HUP`) to make it re-read the file.

A `pg_ident.conf` file that could be used in conjunction with the `pg_hba.conf` file in Example 20-1 is shown in Example 20-2. In this example setup, anyone logged in to a machine on the 192.168 network that does not have the Unix user name `bryanh`, `ann`, or `robert` would not be granted access. Unix user `robert` would only be allowed access when he tries to connect as PostgreSQL user `bob`, not as `robert` or anyone else. `ann` would only be allowed to connect as `ann`. User `bryanh` would be allowed to connect as either `bryanh` himself or as `guest1`.

Example 20-2. An example `pg_ident.conf` file

```
# MAPNAME     IDENT-USERNAME    PG-USERNAME

omicron       bryanh            bryanh
omicron       ann               ann
# bob has user name robert on these machines
```

```
omicron       robert            bob
# bryanh can also connect as guest1
omicron       bryanh            guest1
```

20.2.5. LDAP authentication

This authentication method operates similarly to `password` except that it uses LDAP as the authentication method. LDAP is used only to validate the user name/password pairs. Therefore the user must already exist in the database before LDAP can be used for authentication. The server and parameters used are specified after the `ldap` key word in the file `pg_hba.conf`. The format of this parameter is:

```
ldap[s]://servername[:port]/base dn[;prefix[;suffix]]
```

Commas are used to specify multiple items in an `ldap` component. However, because unquoted commas are treated as item separators in `pg_hba.conf`, it is wise to double-quote the `ldap` URL to preserve any commas present, e.g.:

```
"ldap://ldap.example.net/dc=example,dc=net;EXAMPLE\"
```

If `ldaps` is specified instead of `ldap`, TLS encryption will be enabled for the connection. Note that this will encrypt only the connection between the PostgreSQL server and the LDAP server. The connection between the client and the PostgreSQL server is not affected by this setting. To make use of TLS encryption, you may need to configure the LDAP library prior to configuring PostgreSQL. Note that encrypted LDAP is available only if the platform's LDAP library supports it.

If no port is specified, the default port as configured in the LDAP library will be used.

The server will bind to the distinguished name specified as *`base dn`* using the user name supplied by the client. If *`prefix`* and *`suffix`* is specified, it will be prepended and appended to the user name before the bind. Typically, the prefix parameter is used to specify *`cn=`*, or *`DOMAIN\`* in an Active Directory environment.

20.2.6. PAM authentication

This authentication method operates similarly to `password` except that it uses PAM (Pluggable Authentication Modules) as the authentication mechanism. The default PAM service name is `postgresql`. You can optionally supply your own service name after the `pam` key word in the file `pg_hba.conf`. PAM is used only to validate user name/password pairs. Therefore the user must already exist in the database before PAM can be used for authentication. For more information about PAM, please read the Linux-PAM Page[4] and the Solaris PAM Page[5].

4. http://www.kernel.org/pub/linux/libs/pam/
5. http://www.sun.com/software/solaris/pam/

20.3. Authentication problems

Genuine authentication failures and related problems generally manifest themselves through error messages like the following.

```
FATAL:  no pg_hba.conf entry for host "123.123.123.123", user "andym", database "testdb"
```

This is what you are most likely to get if you succeed in contacting the server, but it does not want to talk to you. As the message suggests, the server refused the connection request because it found no matching entry in its `pg_hba.conf` configuration file.

```
FATAL:  Password authentication failed for user "andym"
```

Messages like this indicate that you contacted the server, and it is willing to talk to you, but not until you pass the authorization method specified in the `pg_hba.conf` file. Check the password you are providing, or check your Kerberos or ident software if the complaint mentions one of those authentication types.

```
FATAL:  user "andym" does not exist
```

The indicated user name was not found.

```
FATAL:  database "testdb" does not exist
```

The database you are trying to connect to does not exist. Note that if you do not specify a database name, it defaults to the database user name, which may or may not be the right thing.

> **Tip:** The server log may contain more information about an authentication failure than is reported to the client. If you are confused about the reason for a failure, check the log.

Chapter 21. Localization

This chapter describes the available localization features from the point of view of the administrator. PostgreSQL supports localization with two approaches:

- Using the locale features of the operating system to provide locale-specific collation order, number formatting, translated messages, and other aspects.
- Providing a number of different character sets defined in the PostgreSQL server, including multiple-byte character sets, to support storing text in all kinds of languages, and providing character set translation between client and server.

21.1. Locale Support

Locale support refers to an application respecting cultural preferences regarding alphabets, sorting, number formatting, etc. PostgreSQL uses the standard ISO C and POSIX locale facilities provided by the server operating system. For additional information refer to the documentation of your system.

21.1.1. Overview

Locale support is automatically initialized when a database cluster is created using `initdb`. `initdb` will initialize the database cluster with the locale setting of its execution environment by default, so if your system is already set to use the locale that you want in your database cluster then there is nothing else you need to do. If you want to use a different locale (or you are not sure which locale your system is set to), you can instruct `initdb` exactly which locale to use by specifying the `--locale` option. For example:

```
initdb --locale=sv_SE
```

This example sets the locale to Swedish (`sv`) as spoken in Sweden (`SE`). Other possibilities might be `en_US` (U.S. English) and `fr_CA` (French Canadian). If more than one character set can be useful for a locale then the specifications look like this: `cs_CZ.ISO8859-2`. What locales are available under what names on your system depends on what was provided by the operating system vendor and what was installed. (On most systems, the command `locale -a` will provide a list of available locales.)

Occasionally it is useful to mix rules from several locales, e.g., use English collation rules but Spanish messages. To support that, a set of locale subcategories exist that control only a certain aspect of the localization rules:

`LC_COLLATE`	String sort order
`LC_CTYPE`	Character classification (What is a letter? Its upper-case equivalent?)
`LC_MESSAGES`	Language of messages

LC_MONETARY	Formatting of currency amounts
LC_NUMERIC	Formatting of numbers
LC_TIME	Formatting of dates and times

The category names translate into names of initdb options to override the locale choice for a specific category. For instance, to set the locale to French Canadian, but use U.S. rules for formatting currency, use initdb --locale=fr_CA --lc-monetary=en_US.

If you want the system to behave as if it had no locale support, use the special locale C or POSIX.

The nature of some locale categories is that their value has to be fixed for the lifetime of a database cluster. That is, once initdb has run, you cannot change them anymore. LC_COLLATE and LC_CTYPE are those categories. They affect the sort order of indexes, so they must be kept fixed, or indexes on text columns will become corrupt. PostgreSQL enforces this by recording the values of LC_COLLATE and LC_CTYPE that are seen by initdb. The server automatically adopts those two values when it is started.

The other locale categories can be changed as desired whenever the server is running by setting the run-time configuration variables that have the same name as the locale categories (see Section 17.10.2 for details). The defaults that are chosen by initdb are actually only written into the configuration file postgresql.conf to serve as defaults when the server is started. If you delete these assignments from postgresql.conf then the server will inherit the settings from its execution environment.

Note that the locale behavior of the server is determined by the environment variables seen by the server, not by the environment of any client. Therefore, be careful to configure the correct locale settings before starting the server. A consequence of this is that if client and server are set up in different locales, messages may appear in different languages depending on where they originated.

> **Note:** When we speak of inheriting the locale from the execution environment, this means the following on most operating systems: For a given locale category, say the collation, the following environment variables are consulted in this order until one is found to be set: LC_ALL, LC_COLLATE (the variable corresponding to the respective category), LANG. If none of these environment variables are set then the locale defaults to C.
>
> Some message localization libraries also look at the environment variable LANGUAGE which overrides all other locale settings for the purpose of setting the language of messages. If in doubt, please refer to the documentation of your operating system, in particular the documentation about gettext, for more information.

To enable messages to be translated to the user's preferred language, NLS must have been enabled at build time. This choice is independent of the other locale support.

21.1.2. Behavior

The locale settings influence the following SQL features:

- Sort order in queries using ORDER BY on textual data
- The ability to use indexes with LIKE clauses

- The `upper`, `lower`, and `initcap` functions
- The `to_char` family of functions

The drawback of using locales other than `C` or `POSIX` in PostgreSQL is its performance impact. It slows character handling and prevents ordinary indexes from being used by `LIKE`. For this reason use locales only if you actually need them.

As a workaround to allow PostgreSQL to use indexes with `LIKE` clauses under a non-C locale, several custom operator classes exist. These allow the creation of an index that performs a strict character-by-character comparison, ignoring locale comparison rules. Refer to Section 11.8 for more information.

21.1.3. Problems

If locale support doesn't work in spite of the explanation above, check that the locale support in your operating system is correctly configured. To check what locales are installed on your system, you may use the command `locale -a` if your operating system provides it.

Check that PostgreSQL is actually using the locale that you think it is. `LC_COLLATE` and `LC_CTYPE` settings are determined at `initdb` time and cannot be changed without repeating `initdb`. Other locale settings including `LC_MESSAGES` and `LC_MONETARY` are initially determined by the environment the server is started in, but can be changed on-the-fly. You can check the active locale settings using the `SHOW` command.

The directory `src/test/locale` in the source distribution contains a test suite for PostgreSQL's locale support.

Client applications that handle server-side errors by parsing the text of the error message will obviously have problems when the server's messages are in a different language. Authors of such applications are advised to make use of the error code scheme instead.

Maintaining catalogs of message translations requires the on-going efforts of many volunteers that want to see PostgreSQL speak their preferred language well. If messages in your language are currently not available or not fully translated, your assistance would be appreciated. If you want to help, refer to Chapter 46 or write to the developers' mailing list.

21.2. Character Set Support

The character set support in PostgreSQL allows you to store text in a variety of character sets, including single-byte character sets such as the ISO 8859 series and multiple-byte character sets such as EUC (Extended Unix Code), UTF-8, and Mule internal code. All supported character sets can be used transparently by clients, but a few are not supported for use within the server (that is, as a server-side encoding). The default character set is selected while initializing your PostgreSQL database cluster using `initdb`. It can be overridden when you create a database, so you can have multiple databases each with a different character set.

21.2.1. Supported Character Sets

Table 21-1 shows the character sets available for use in PostgreSQL.

Table 21-1. PostgreSQL Character Sets

Name	Description	Language	Server?	Bytes/Char	Aliases
BIG5	Big Five	Traditional Chinese	No	1-2	WIN950, Windows950
EUC_CN	Extended UNIX Code-CN	Simplified Chinese	Yes	1-3	
EUC_JP	Extended UNIX Code-JP	Japanese	Yes	1-3	
EUC_KR	Extended UNIX Code-KR	Korean	Yes	1-3	
EUC_TW	Extended UNIX Code-TW	Traditional Chinese, Taiwanese	Yes	1-3	
GB18030	National Standard	Chinese	No	1-2	
GBK	Extended National Standard	Simplified Chinese	No	1-2	WIN936, Windows936
ISO_8859_5	ISO 8859-5, ECMA 113	Latin/Cyrillic	Yes	1	
ISO_8859_6	ISO 8859-6, ECMA 114	Latin/Arabic	Yes	1	
ISO_8859_7	ISO 8859-7, ECMA 118	Latin/Greek	Yes	1	
ISO_8859_8	ISO 8859-8, ECMA 121	Latin/Hebrew	Yes	1	
JOHAB	JOHAB	Korean (Hangul)	Yes	1-3	
KOI8	KOI8-R(U)	Cyrillic	Yes	1	KOI8R
LATIN1	ISO 8859-1, ECMA 94	Western European	Yes	1	ISO88591
LATIN2	ISO 8859-2, ECMA 94	Central European	Yes	1	ISO88592
LATIN3	ISO 8859-3, ECMA 94	South European	Yes	1	ISO88593
LATIN4	ISO 8859-4, ECMA 94	North European	Yes	1	ISO88594

Name	Description	Language	Server?	Bytes/Char	Aliases
LATIN5	ISO 8859-9, ECMA 128	Turkish	Yes	1	ISO88599
LATIN6	ISO 8859-10, ECMA 144	Nordic	Yes	1	ISO885910
LATIN7	ISO 8859-13	Baltic	Yes	1	ISO885913
LATIN8	ISO 8859-14	Celtic	Yes	1	ISO885914
LATIN9	ISO 8859-15	LATIN1 with Euro and accents	Yes	1	ISO885915
LATIN10	ISO 8859-16, ASRO SR 14111	Romanian	Yes	1	ISO885916
MULE_INTERNAL	Mule internal code	Multilingual Emacs	Yes	1-4	
SJIS	Shift JIS	Japanese	No	1-2	Mskanji, ShiftJIS, WIN932, Windows932
SQL_ASCII	unspecified (see text)	*any*	Yes	1	
UHC	Unified Hangul Code	Korean	No	1-2	WIN949, Windows949
UTF8	Unicode, 8-bit	*all*	Yes	1-4	Unicode
WIN866	Windows CP866	Cyrillic	Yes	1	ALT
WIN874	Windows CP874	Thai	Yes	1	
WIN1250	Windows CP1250	Central European	Yes	1	
WIN1251	Windows CP1251	Cyrillic	Yes	1	WIN
WIN1252	Windows CP1252	Western European	Yes	1	
WIN1253	Windows CP1253	Greek	Yes	1	
WIN1254	Windows CP1254	Turkish	Yes	1	
WIN1255	Windows CP1255	Hebrew	Yes	1	
WIN1256	Windows CP1256	Arabic	Yes	1	

Name	Description	Language	Server?	Bytes/Char	Aliases
WIN1257	Windows CP1257	Baltic	Yes	1	
WIN1258	Windows CP1258	Vietnamese	Yes	1	ABC, TCVN, TCVN5712, VSCII

Not all APIs support all the listed character sets. For example, the PostgreSQL JDBC driver does not support MULE_INTERNAL, LATIN6, LATIN8, and LATIN10.

The SQL_ASCII setting behaves considerably differently from the other settings. When the server character set is SQL_ASCII, the server interprets byte values 0-127 according to the ASCII standard, while byte values 128-255 are taken as uninterpreted characters. No encoding conversion will be done when the setting is SQL_ASCII. Thus, this setting is not so much a declaration that a specific encoding is in use, as a declaration of ignorance about the encoding. In most cases, if you are working with any non-ASCII data, it is unwise to use the SQL_ASCII setting, because PostgreSQL will be unable to help you by converting or validating non-ASCII characters.

21.2.2. Setting the Character Set

initdb defines the default character set for a PostgreSQL cluster. For example,

```
initdb -E EUC_JP
```

sets the default character set (encoding) to EUC_JP (Extended Unix Code for Japanese). You can use --encoding instead of -E if you prefer to type longer option strings. If no -E or --encoding option is given, initdb attempts to determine the appropriate encoding to use based on the specified or default locale.

You can create a database with a different character set:

```
createdb -E EUC_KR korean
```

This will create a database named korean that uses the character set EUC_KR. Another way to accomplish this is to use this SQL command:

```
CREATE DATABASE korean WITH ENCODING 'EUC_KR';
```

The encoding for a database is stored in the system catalog pg_database. You can see that by using the -l option or the \l command of psql.

```
$ psql -l
            List of databases
   Database    |  Owner  |   Encoding
---------------+---------+---------------
 euc_cn        | t-ishii | EUC_CN
 euc_jp        | t-ishii | EUC_JP
 euc_kr        | t-ishii | EUC_KR
 euc_tw        | t-ishii | EUC_TW
 mule_internal | t-ishii | MULE_INTERNAL
 postgres      | t-ishii | EUC_JP
```

```
 regression    | t-ishii | SQL_ASCII
 template1     | t-ishii | EUC_JP
 test          | t-ishii | EUC_JP
 utf8          | t-ishii | UTF8
(9 rows)
```

Important: Although you can specify any encoding you want for a database, it is unwise to choose an encoding that is not what is expected by the locale you have selected. The `LC_COLLATE` and `LC_CTYPE` settings imply a particular encoding, and locale-dependent operations (such as sorting) are likely to misinterpret data that is in an incompatible encoding.

Since these locale settings are frozen by `initdb`, the apparent flexibility to use different encodings in different databases of a cluster is more theoretical than real. It is likely that these mechanisms will be revisited in future versions of PostgreSQL.

One way to use multiple encodings safely is to set the locale to `C` or `POSIX` during `initdb`, thus disabling any real locale awareness.

21.2.3. Automatic Character Set Conversion Between Server and Client

PostgreSQL supports automatic character set conversion between server and client for certain character set combinations. The conversion information is stored in the `pg_conversion` system catalog. PostgreSQL comes with some predefined conversions, as shown in Table 21-2. You can create a new conversion using the SQL command `CREATE CONVERSION`.

Table 21-2. Client/Server Character Set Conversions

Server Character Set	Available Client Character Sets
BIG5	*not supported as a server encoding*
EUC_CN	*EUC_CN*, MULE_INTERNAL, UTF8
EUC_JP	*EUC_JP*, MULE_INTERNAL, SJIS, UTF8
EUC_KR	*EUC_KR*, MULE_INTERNAL, UTF8
EUC_TW	*EUC_TW*, BIG5, MULE_INTERNAL, UTF8
GB18030	*not supported as a server encoding*
GBK	*not supported as a server encoding*
ISO_8859_5	*ISO_8859_5*, KOI8, MULE_INTERNAL, UTF8, WIN866, WIN1251
ISO_8859_6	*ISO_8859_6*, UTF8
ISO_8859_7	*ISO_8859_7*, UTF8
ISO_8859_8	*ISO_8859_8*, UTF8
JOHAB	*JOHAB*, UTF8

Server Character Set	Available Client Character Sets
KOI8	*KOI8*, ISO_8859_5, MULE_INTERNAL, UTF8, WIN866, WIN1251
LATIN1	*LATIN1*, MULE_INTERNAL, UTF8
LATIN2	*LATIN2*, MULE_INTERNAL, UTF8, WIN1250
LATIN3	*LATIN3*, MULE_INTERNAL, UTF8
LATIN4	*LATIN4*, MULE_INTERNAL, UTF8
LATIN5	*LATIN5*, UTF8
LATIN6	*LATIN6*, UTF8
LATIN7	*LATIN7*, UTF8
LATIN8	*LATIN8*, UTF8
LATIN9	*LATIN9*, UTF8
LATIN10	*LATIN10*, UTF8
MULE_INTERNAL	*MULE_INTERNAL*, BIG5, EUC_CN, EUC_JP, EUC_KR, EUC_TW, ISO_8859_5, KOI8, LATIN1 to LATIN4, SJIS, WIN866, WIN1250, WIN1251
SJIS	*not supported as a server encoding*
SQL_ASCII	*any (no conversion will be performed)*
UHC	*not supported as a server encoding*
UTF8	*all supported encodings*
WIN866	*WIN866*, ISO_8859_5, KOI8, MULE_INTERNAL, UTF8, WIN1251
WIN874	*WIN874*, UTF8
WIN1250	*WIN1250*, LATIN2, MULE_INTERNAL, UTF8
WIN1251	*WIN1251*, ISO_8859_5, KOI8, MULE_INTERNAL, UTF8, WIN866
WIN1252	*WIN1252*, UTF8
WIN1253	*WIN1253*, UTF8
WIN1254	*WIN1254*, UTF8
WIN1255	*WIN1255*, UTF8
WIN1256	*WIN1256*, UTF8
WIN1257	*WIN1257*, UTF8
WIN1258	*WIN1258*, UTF8

To enable automatic character set conversion, you have to tell PostgreSQL the character set (encoding) you would like to use in the client. There are several ways to accomplish this:

- Using the \encoding command in psql. \encoding allows you to change client encoding on the fly. For example, to change the encoding to SJIS, type:

```
\encoding SJIS
```

- Using libpq functions. `\encoding` actually calls `PQsetClientEncoding()` for its purpose.

  ```
  int PQsetClientEncoding(PGconn *conn, const char *encoding);
  ```
  where *conn* is a connection to the server, and *encoding* is the encoding you want to use. If the function successfully sets the encoding, it returns 0, otherwise -1. The current encoding for this connection can be determined by using:

  ```
  int PQclientEncoding(const PGconn *conn);
  ```
  Note that it returns the encoding ID, not a symbolic string such as `EUC_JP`. To convert an encoding ID to an encoding name, you can use:

  ```
  char *pg_encoding_to_char(int encoding_id);
  ```

- Using `SET client_encoding TO`. Setting the client encoding can be done with this SQL command:

  ```
  SET CLIENT_ENCODING TO 'value';
  ```
  Also you can use the standard SQL syntax `SET NAMES` for this purpose:

  ```
  SET NAMES 'value';
  ```
  To query the current client encoding:

  ```
  SHOW client_encoding;
  ```
  To return to the default encoding:

  ```
  RESET client_encoding;
  ```

- Using `PGCLIENTENCODING`. If the environment variable `PGCLIENTENCODING` is defined in the client's environment, that client encoding is automatically selected when a connection to the server is made. (This can subsequently be overridden using any of the other methods mentioned above.)

- Using the configuration variable client_encoding. If the `client_encoding` variable is set, that client encoding is automatically selected when a connection to the server is made. (This can subsequently be overridden using any of the other methods mentioned above.)

If the conversion of a particular character is not possible — suppose you chose `EUC_JP` for the server and `LATIN1` for the client, then some Japanese characters do not have a representation in `LATIN1` — then an error is reported.

If the client character set is defined as `SQL_ASCII`, encoding conversion is disabled, regardless of the server's character set. Just as for the server, use of `SQL_ASCII` is unwise unless you are working with all-ASCII data.

21.2.4. Further Reading

These are good sources to start learning about various kinds of encoding systems.

http://www.i18ngurus.com/docs/984813247.html

An extensive collection of documents about character sets, encodings, and code pages.

ftp://ftp.ora.com/pub/examples/nutshell/ujip/doc/cjk.inf

Detailed explanations of `EUC_JP`, `EUC_CN`, `EUC_KR`, `EUC_TW` appear in section 3.2.

http://www.unicode.org/

The web site of the Unicode Consortium

RFC 2044

UTF-8 is defined here.

Chapter 22. Routine Database Maintenance Tasks

PostgreSQL, like any database software, requires that certain tasks be performed regularly to achieve optimum performance. The tasks discussed here are *required*, but they are repetitive in nature and can easily be automated using standard Unix tools such as cron scripts or Windows' Task Scheduler. But it is the database administrator's responsibility to set up appropriate scripts, and to check that they execute successfully.

One obvious maintenance task is creation of backup copies of the data on a regular schedule. Without a recent backup, you have no chance of recovery after a catastrophe (disk failure, fire, mistakenly dropping a critical table, etc.). The backup and recovery mechanisms available in PostgreSQL are discussed at length in Chapter 23.

The other main category of maintenance task is periodic "vacuuming" of the database. This activity is discussed in Section 22.1. Closely related to this is updating the statistics that will be used by the query planner, as discussed in Section 22.1.2.

Another task that might need periodic attention is log file management. This is discussed in Section 22.3.

PostgreSQL is low-maintenance compared to some other database management systems. Nonetheless, appropriate attention to these tasks will go far towards ensuring a pleasant and productive experience with the system.

22.1. Routine Vacuuming

PostgreSQL's `VACUUM` command *must* be run on a regular basis for several reasons:

1. To recover or reuse disk space occupied by updated or deleted rows.
2. To update data statistics used by the PostgreSQL query planner.
3. To protect against loss of very old data due to *transaction ID wraparound*.

The frequency and scope of the `VACUUM` operations performed for each of these reasons will vary depending on the needs of each site. Therefore, database administrators must understand these issues and develop an appropriate maintenance strategy. This section concentrates on explaining the high-level issues; for details about command syntax and so on, see the *VACUUM* reference page.

The standard form of `VACUUM` can run in parallel with production database operations. Commands such as `SELECT`, `INSERT`, `UPDATE`, and `DELETE` will continue to function as normal, though you will not be able to modify the definition of a table with commands such as `ALTER TABLE ADD COLUMN` while it is being vacuumed. Also, `VACUUM` requires a substantial amount of I/O traffic, which can cause poor performance for other active sessions. There are configuration parameters that can be adjusted to reduce the performance impact of background vacuuming — see Section 17.4.4.

An automated mechanism for performing the necessary `VACUUM` operations has been added in PostgreSQL 8.1. See Section 22.1.4.

22.1.1. Recovering disk space

In normal PostgreSQL operation, an UPDATE or DELETE of a row does not immediately remove the old version of the row. This approach is necessary to gain the benefits of multiversion concurrency control (see Chapter 12): the row version must not be deleted while it is still potentially visible to other transactions. But eventually, an outdated or deleted row version is no longer of interest to any transaction. The space it occupies must be reclaimed for reuse by new rows, to avoid infinite growth of disk space requirements. This is done by running VACUUM.

Clearly, a table that receives frequent updates or deletes will need to be vacuumed more often than tables that are seldom updated. It may be useful to set up periodic cron tasks that VACUUM only selected tables, skipping tables that are known not to change often. This is only likely to be helpful if you have both large heavily-updated tables and large seldom-updated tables — the extra cost of vacuuming a small table isn't enough to be worth worrying about.

There are two variants of the VACUUM command. The first form, known as "lazy vacuum" or just VACUUM, marks expired data in tables and indexes for future reuse; it does *not* attempt to reclaim the space used by this expired data unless the space is at the end of the table and an exclusive table lock can be easily obtained. Unused space at the start or middle of the file does not result in the file being shortened and space returned to the operating system. This variant of VACUUM can be run concurrently with normal database operations.

The second form is the VACUUM FULL command. This uses a more aggressive algorithm for reclaiming the space consumed by expired row versions. Any space that is freed by VACUUM FULL is immediately returned to the operating system. Unfortunately, this variant of the VACUUM command acquires an exclusive lock on each table while VACUUM FULL is processing it. Therefore, frequently using VACUUM FULL can have an extremely negative effect on the performance of concurrent database queries.

The standard form of VACUUM is best used with the goal of maintaining a fairly level steady-state usage of disk space. If you need to return disk space to the operating system you can use VACUUM FULL — but what's the point of releasing disk space that will only have to be allocated again soon? Moderately frequent standard VACUUM runs are a better approach than infrequent VACUUM FULL runs for maintaining heavily-updated tables.

Recommended practice for most sites is to schedule a database-wide VACUUM once a day at a low-usage time of day, supplemented by more frequent vacuuming of heavily-updated tables if necessary. (Some installations with extremely high update rates vacuum their busiest tables as often as once every few minutes.) If you have multiple databases in a cluster, don't forget to VACUUM each one; the program *vacuumdb* may be helpful.

VACUUM FULL is recommended for cases where you know you have deleted the majority of rows in a table, so that the steady-state size of the table can be shrunk substantially with VACUUM FULL's more aggressive approach. Use plain VACUUM, not VACUUM FULL, for routine vacuuming for space recovery.

If you have a table whose entire contents are deleted on a periodic basis, consider doing it with TRUNCATE rather than using DELETE followed by VACUUM. TRUNCATE removes the entire content of the table immediately, without requiring a subsequent VACUUM or VACUUM FULL to reclaim the now-unused disk space.

22.1.2. Updating planner statistics

The PostgreSQL query planner relies on statistical information about the contents of tables in order to

generate good plans for queries. These statistics are gathered by the ANALYZE command, which can be invoked by itself or as an optional step in VACUUM. It is important to have reasonably accurate statistics, otherwise poor choices of plans may degrade database performance.

As with vacuuming for space recovery, frequent updates of statistics are more useful for heavily-updated tables than for seldom-updated ones. But even for a heavily-updated table, there may be no need for statistics updates if the statistical distribution of the data is not changing much. A simple rule of thumb is to think about how much the minimum and maximum values of the columns in the table change. For example, a timestamp column that contains the time of row update will have a constantly-increasing maximum value as rows are added and updated; such a column will probably need more frequent statistics updates than, say, a column containing URLs for pages accessed on a website. The URL column may receive changes just as often, but the statistical distribution of its values probably changes relatively slowly.

It is possible to run ANALYZE on specific tables and even just specific columns of a table, so the flexibility exists to update some statistics more frequently than others if your application requires it. In practice, however, it is usually best to just analyze the entire database because it is a fast operation. It uses a statistical random sampling of the rows of a table rather than reading every single row.

> **Tip:** Although per-column tweaking of ANALYZE frequency may not be very productive, you may well find it worthwhile to do per-column adjustment of the level of detail of the statistics collected by ANALYZE. Columns that are heavily used in WHERE clauses and have highly irregular data distributions may require a finer-grain data histogram than other columns. See ALTER TABLE SET STATISTICS.

Recommended practice for most sites is to schedule a database-wide ANALYZE once a day at a low-usage time of day; this can usefully be combined with a nightly VACUUM. However, sites with relatively slowly changing table statistics may find that this is overkill, and that less-frequent ANALYZE runs are sufficient.

22.1.3. Preventing transaction ID wraparound failures

PostgreSQL's MVCC transaction semantics depend on being able to compare transaction ID (XID) numbers: a row version with an insertion XID greater than the current transaction's XID is "in the future" and should not be visible to the current transaction. But since transaction IDs have limited size (32 bits at this writing) a cluster that runs for a long time (more than 4 billion transactions) would suffer *transaction ID wraparound*: the XID counter wraps around to zero, and all of a sudden transactions that were in the past appear to be in the future — which means their outputs become invisible. In short, catastrophic data loss. (Actually the data is still there, but that's cold comfort if you can't get at it.) To avoid this, it is necessary to vacuum every table in every database at least once every two billion transactions.

The reason that periodic vacuuming solves the problem is that PostgreSQL distinguishes a special XID FrozenXID. This XID is always considered older than every normal XID. Normal XIDs are compared using modulo-2^{31} arithmetic. This means that for every normal XID, there are two billion XIDs that are "older" and two billion that are "newer"; another way to say it is that the normal XID space is circular with no endpoint. Therefore, once a row version has been created with a particular normal XID, the row version will appear to be "in the past" for the next two billion transactions, no matter which normal XID we are talking about. If the row version still exists after more than two billion transactions, it will suddenly appear to be in the future. To prevent data loss, old row versions must be reassigned the XID FrozenXID sometime before they reach the two-billion-transactions-old mark. Once they are assigned this special XID, they will appear to be "in the past" to all normal transactions regardless of wraparound issues, and

so such row versions will be good until deleted, no matter how long that is. This reassignment of old XIDs is handled by VACUUM.

VACUUM's behavior is controlled by the configuration parameter vacuum_freeze_min_age: any XID older than vacuum_freeze_min_age transactions is replaced by FrozenXID. Larger values of vacuum_freeze_min_age preserve transactional information longer, while smaller values increase the number of transactions that can elapse before the table must be vacuumed again.

The maximum time that a table can go unvacuumed is two billion transactions minus the vacuum_freeze_min_age that was used when it was last vacuumed. If it were to go unvacuumed for longer than that, data loss could result. To ensure that this does not happen, the *autovacuum* facility described in Section 22.1.4 is invoked on any table that might contain XIDs older than the age specified by the configuration parameter autovacuum_freeze_max_age. (This will happen even if autovacuum is otherwise disabled.)

This implies that if a table is not otherwise vacuumed, autovacuum will be invoked on it approximately once every autovacuum_freeze_max_age minus vacuum_freeze_min_age transactions. For tables that are regularly vacuumed for space reclamation purposes, this is of little importance. However, for static tables (including tables that receive inserts, but no updates or deletes), there is no need for vacuuming for space reclamation, and so it can be useful to try to maximize the interval between forced autovacuums on very large static tables. Obviously one can do this either by increasing autovacuum_freeze_max_age or by decreasing vacuum_freeze_min_age.

The sole disadvantage of increasing autovacuum_freeze_max_age is that the pg_clog subdirectory of the database cluster will take more space, because it must store the commit status for all transactions back to the autovacuum_freeze_max_age horizon. The commit status uses two bits per transaction, so if autovacuum_freeze_max_age has its maximum allowed value of a little less than two billion, pg_clog can be expected to grow to about half a gigabyte. If this is trivial compared to your total database size, setting autovacuum_freeze_max_age to its maximum allowed value is recommended. Otherwise, set it depending on what you are willing to allow for pg_clog storage. (The default, 200 million transactions, translates to about 50MB of pg_clog storage.)

One disadvantage of decreasing vacuum_freeze_min_age is that it may cause VACUUM to do useless work: changing a table row's XID to FrozenXID is a waste of time if the row is modified soon thereafter (causing it to acquire a new XID). So the setting should be large enough that rows are not frozen until they are unlikely to change any more. Another disadvantage of decreasing this setting is that details about exactly which transaction inserted or modified a row will be lost sooner. This information sometimes comes in handy, particularly when trying to analyze what went wrong after a database failure. For these two reasons, decreasing this setting is not recommended except for completely static tables.

To track the age of the oldest XIDs in a database, VACUUM stores XID statistics in the system tables pg_class and pg_database. In particular, the relfrozenxid column of a table's pg_class row contains the freeze cutoff XID that was used by the last VACUUM for that table. All normal XIDs older than this cutoff XID are guaranteed to have been replaced by FrozenXID within the table. Similarly, the datfrozenxid column of a database's pg_database row is a lower bound on the normal XIDs appearing in that database — it is just the minimum of the per-table relfrozenxid values within the database. A convenient way to examine this information is to execute queries such as

```
SELECT relname, age(relfrozenxid) FROM pg_class WHERE relkind = 'r';
SELECT datname, age(datfrozenxid) FROM pg_database;
```

The age column measures the number of transactions from the cutoff XID to the current transaction's

XID. Immediately after a VACUUM, age(relfrozenxid) should be a little more than the vacuum_freeze_min_age setting that was used (more by the number of transactions started since the VACUUM started). If age(relfrozenxid) exceeds autovacuum_freeze_max_age, an autovacuum will soon be forced for the table.

If for some reason autovacuum fails to clear old XIDs from a table, the system will begin to emit warning messages like this when the database's oldest XIDs reach ten million transactions from the wraparound point:

```
WARNING:  database "mydb" must be vacuumed within 177009986 transactions
HINT:  To avoid a database shutdown, execute a full-database VACUUM in "mydb".
```

If these warnings are ignored, the system will shut down and refuse to execute any new transactions once there are fewer than 1 million transactions left until wraparound:

```
ERROR:  database is shut down to avoid wraparound data loss in database "mydb"
HINT:  Stop the postmaster and use a standalone backend to VACUUM in "mydb".
```

The 1-million-transaction safety margin exists to let the administrator recover without data loss, by manually executing the required VACUUM commands. However, since the system will not execute commands once it has gone into the safety shutdown mode, the only way to do this is to stop the server and use a single-user backend to execute VACUUM. The shutdown mode is not enforced by a single-user backend. See the postgres reference page for details about using a single-user backend.

22.1.4. The auto-vacuum daemon

Beginning in PostgreSQL 8.1, there is a separate optional server process called the *autovacuum daemon*, whose purpose is to automate the execution of VACUUM and ANALYZE commands. When enabled, the autovacuum daemon runs periodically and checks for tables that have had a large number of inserted, updated or deleted tuples. These checks use the row-level statistics collection facility; therefore, the autovacuum daemon cannot be used unless stats_start_collector and stats_row_level are set to true. Also, it's important to allow a slot for the autovacuum process when choosing the value of superuser_reserved_connections.

The autovacuum daemon, when enabled, runs every autovacuum_naptime seconds. On each run, it selects one database to process and checks each table within that database. VACUUM or ANALYZE commands are issued as needed.

Tables whose relfrozenxid value is more than autovacuum_freeze_max_age transactions old are always vacuumed. Otherwise, two conditions are used to determine which operation(s) to apply. If the number of obsolete tuples since the last VACUUM exceeds the "vacuum threshold", the table is vacuumed. The vacuum threshold is defined as:

```
vacuum threshold = vacuum base threshold + vacuum scale factor * number of tuples
```

where the vacuum base threshold is autovacuum_vacuum_threshold, the vacuum scale factor is autovacuum_vacuum_scale_factor, and the number of tuples is pg_class.reltuples. The number of obsolete tuples is obtained from the statistics collector; it is a semi-accurate count updated by each UPDATE and DELETE operation. (It is only semi-accurate because some information may be lost under heavy load.) For analyze, a similar condition is used: the threshold, defined as

```
analyze threshold = analyze base threshold + analyze scale factor * number of tuples
```

is compared to the total number of tuples inserted, updated, or deleted since the last ANALYZE.

The default thresholds and scale factors are taken from postgresql.conf, but it is possible to override them on a table-by-table basis by making entries in the system catalog pg_autovacuum. If a pg_autovacuum row exists for a particular table, the settings it specifies are applied; otherwise the global settings are used. See Section 17.9 for more details on the global settings.

Besides the base threshold values and scale factors, there are five more parameters that can be set for each table in pg_autovacuum. The first, pg_autovacuum.enabled, can be set to false to instruct the autovacuum daemon to skip that particular table entirely. In this case autovacuum will only touch the table if it must do so to prevent transaction ID wraparound. The next two parameters, the vacuum cost delay (pg_autovacuum.vac_cost_delay) and the vacuum cost limit (pg_autovacuum.vac_cost_limit), are used to set table-specific values for the *Cost-Based Vacuum Delay* feature. The last two parameters, (pg_autovacuum.freeze_min_age) and (pg_autovacuum.freeze_max_age), are used to set table-specific values for vacuum_freeze_min_age and autovacuum_freeze_max_age respectively.

If any of the values in pg_autovacuum are set to a negative number, or if a row is not present at all in pg_autovacuum for any particular table, the corresponding values from postgresql.conf are used.

There is not currently any support for making pg_autovacuum entries, except by doing manual INSERTs into the catalog. This feature will be improved in future releases, and it is likely that the catalog definition will change.

Caution

The contents of the pg_autovacuum system catalog are currently not saved in database dumps created by the tools pg_dump and pg_dumpall. If you want to preserve them across a dump/reload cycle, make sure you dump the catalog manually.

22.2. Routine Reindexing

In some situations it is worthwhile to rebuild indexes periodically with the *REINDEX* command.

In PostgreSQL releases before 7.4, periodic reindexing was frequently necessary to avoid “index bloat”, due to lack of internal space reclamation in B-tree indexes. Any situation in which the range of index keys changed over time — for example, an index on timestamps in a table where old entries are eventually deleted — would result in bloat, because index pages for no-longer-needed portions of the key range were not reclaimed for re-use. Over time, the index size could become indefinitely much larger than the amount of useful data in it.

In PostgreSQL 7.4 and later, index pages that have become completely empty are reclaimed for re-use. There is still a possibility for inefficient use of space: if all but a few index keys on a page have been deleted, the page remains allocated. So a usage pattern in which all but a few keys in each range are eventually deleted will see poor use of space. For such usage patterns, periodic reindexing is recommended.

The potential for bloat in non-B-tree indexes has not been well characterized. It is a good idea to keep an eye on the index’s physical size when using any non-B-tree index type.

Also, for B-tree indexes a freshly-constructed index is somewhat faster to access than one that has been updated many times, because logically adjacent pages are usually also physically adjacent in a newly built index. (This consideration does not currently apply to non-B-tree indexes.) It might be worthwhile to reindex periodically just to improve access speed.

22.3. Log File Maintenance

It is a good idea to save the database server's log output somewhere, rather than just routing it to `/dev/null`. The log output is invaluable when it comes time to diagnose problems. However, the log output tends to be voluminous (especially at higher debug levels) and you won't want to save it indefinitely. You need to "rotate" the log files so that new log files are started and old ones removed after a reasonable period of time.

If you simply direct the stderr of `postgres` into a file, you will have log output, but the only way to truncate the log file is to stop and restart the server. This may be OK if you are using PostgreSQL in a development environment, but few production servers would find this behavior acceptable.

A better approach is to send the server's stderr output to some type of log rotation program. There is a built-in log rotation program, which you can use by setting the configuration parameter `redirect_stderr` to `true` in `postgresql.conf`. The control parameters for this program are described in Section 17.7.1.

Alternatively, you might prefer to use an external log rotation program, if you have one that you are already using with other server software. For example, the rotatelogs tool included in the Apache distribution can be used with PostgreSQL. To do this, just pipe the server's stderr output to the desired program. If you start the server with `pg_ctl`, then stderr is already redirected to stdout, so you just need a pipe command, for example:

```
pg_ctl start | rotatelogs /var/log/pgsql_log 86400
```

Another production-grade approach to managing log output is to send it all to syslog and let syslog deal with file rotation. To do this, set the configuration parameter `log_destination` to `syslog` (to log to syslog only) in `postgresql.conf`. Then you can send a `SIGHUP` signal to the syslog daemon whenever you want to force it to start writing a new log file. If you want to automate log rotation, the logrotate program can be configured to work with log files from syslog.

On many systems, however, syslog is not very reliable, particularly with large log messages; it may truncate or drop messages just when you need them the most. Also, on Linux, syslog will sync each message to disk, yielding poor performance. (You can use a - at the start of the file name in the syslog configuration file to disable this behavior.)

Note that all the solutions described above take care of starting new log files at configurable intervals, but they do not handle deletion of old, no-longer-interesting log files. You will probably want to set up a batch job to periodically delete old log files. Another possibility is to configure the rotation program so that old log files are overwritten cyclically.

Chapter 23. Backup and Restore

As with everything that contains valuable data, PostgreSQL databases should be backed up regularly. While the procedure is essentially simple, it is important to have a basic understanding of the underlying techniques and assumptions.

There are three fundamentally different approaches to backing up PostgreSQL data:

- SQL dump
- File system level backup
- Continuous archiving

Each has its own strengths and weaknesses.

23.1. SQL Dump

The idea behind this dump method is to generate a text file with SQL commands that, when fed back to the server, will recreate the database in the same state as it was at the time of the dump. PostgreSQL provides the utility program pg_dump for this purpose. The basic usage of this command is:

```
pg_dump dbname > outfile
```

As you see, pg_dump writes its results to the standard output. We will see below how this can be useful.

pg_dump is a regular PostgreSQL client application (albeit a particularly clever one). This means that you can do this backup procedure from any remote host that has access to the database. But remember that pg_dump does not operate with special permissions. In particular, it must have read access to all tables that you want to back up, so in practice you almost always have to run it as a database superuser.

To specify which database server pg_dump should contact, use the command line options `-h host` and `-p port`. The default host is the local host or whatever your `PGHOST` environment variable specifies. Similarly, the default port is indicated by the `PGPORT` environment variable or, failing that, by the compiled-in default. (Conveniently, the server will normally have the same compiled-in default.)

As any other PostgreSQL client application, pg_dump will by default connect with the database user name that is equal to the current operating system user name. To override this, either specify the `-U` option or set the environment variable `PGUSER`. Remember that pg_dump connections are subject to the normal client authentication mechanisms (which are described in Chapter 20).

Dumps created by pg_dump are internally consistent, that is, updates to the database while pg_dump is running will not be in the dump. pg_dump does not block other operations on the database while it is working. (Exceptions are those operations that need to operate with an exclusive lock, such as `VACUUM FULL`.)

> **Important:** If your database schema relies on OIDs (for instance as foreign keys) you must instruct pg_dump to dump the OIDs as well. To do this, use the `-o` command line option.

23.1.1. Restoring the dump

The text files created by pg_dump are intended to be read in by the psql program. The general command form to restore a dump is

```
psql dbname < infile
```

where *infile* is what you used as *outfile* for the pg_dump command. The database *dbname* will not be created by this command, so you must create it yourself from `template0` before executing psql (e.g., with `createdb -T template0 dbname`). psql supports similar options to pg_dump for specifying the database server to connect to and the user name to use. See the psql reference page for more information.

Before restoring a SQL dump, all the users who own objects or were granted permissions on objects in the dumped database must already exist. If they do not, then the restore will fail to recreate the objects with the original ownership and/or permissions. (Sometimes this is what you want, but usually it is not.)

By default, the psql script will continue to execute after an SQL error is encountered. You may wish to use the following command at the top of the script to alter that behaviour and have psql exit with an exit status of 3 if an SQL error occurs:

```
\set ON_ERROR_STOP
```

Either way, you will only have a partially restored dump. Alternatively, you can specify that the whole dump should be restored as a single transaction, so the restore is either fully completed or fully rolled back. This mode can be specified by passing the `-1` or `--single-transaction` command-line options to psql. When using this mode, be aware that even the smallest of errors can rollback a restore that has already run for many hours. However, that may still be preferable to manually cleaning up a complex database after a partially restored dump.

The ability of pg_dump and psql to write to or read from pipes makes it possible to dump a database directly from one server to another; for example:

```
pg_dump -h host1 dbname | psql -h host2 dbname
```

> **Important:** The dumps produced by pg_dump are relative to `template0`. This means that any languages, procedures, etc. added to `template1` will also be dumped by pg_dump. As a result, when restoring, if you are using a customized `template1`, you must create the empty database from `template0`, as in the example above.

After restoring a backup, it is wise to run *ANALYZE* on each database so the query optimizer has useful statistics. An easy way to do this is to run `vacuumdb -a -z`; this is equivalent to running `VACUUM ANALYZE` on each database manually. For more advice on how to load large amounts of data into PostgreSQL efficiently, refer to Section 13.4.

23.1.2. Using pg_dumpall

pg_dump dumps only a single database at a time, and it does not dump information about roles or tablespaces (because those are cluster-wide rather than per-database). To support convenient dumping of

the entire contents of a database cluster, the pg_dumpall program is provided. pg_dumpall backs up each database in a given cluster, and also preserves cluster-wide data such as role and tablespace definitions. The basic usage of this command is:

```
pg_dumpall > outfile
```

The resulting dump can be restored with psql:

```
psql -f infile postgres
```

(Actually, you can specify any existing database name to start from, but if you are reloading in an empty cluster then `postgres` should generally be used.) It is always necessary to have database superuser access when restoring a pg_dumpall dump, as that is required to restore the role and tablespace information. If you use tablespaces, be careful that the tablespace paths in the dump are appropriate for the new installation.

23.1.3. Handling large databases

Since PostgreSQL allows tables larger than the maximum file size on your system, it can be problematic to dump such a table to a file, since the resulting file will likely be larger than the maximum size allowed by your system. Since pg_dump can write to the standard output, you can use standard Unix tools to work around this possible problem.

Use compressed dumps. You can use your favorite compression program, for example gzip.

```
pg_dump dbname | gzip > filename.gz
```

Reload with

```
createdb dbname
gunzip -c filename.gz | psql dbname
```

or

```
cat filename.gz | gunzip | psql dbname
```

Use `split`. The `split` command allows you to split the output into pieces that are acceptable in size to the underlying file system. For example, to make chunks of 1 megabyte:

```
pg_dump dbname | split -b 1m - filename
```

Reload with

```
createdb dbname
cat filename* | psql dbname
```

Use the custom dump format. If PostgreSQL was built on a system with the zlib compression library installed, the custom dump format will compress data as it writes it to the output file. This will produce dump file sizes similar to using `gzip`, but it has the added advantage that tables can be restored selectively. The following command dumps a database using the custom dump format:

```
pg_dump -Fc dbname > filename
```

A custom-format dump is not a script for psql, but instead must be restored with pg_restore. See the pg_dump and pg_restore reference pages for details.

23.2. File System Level Backup

An alternative backup strategy is to directly copy the files that PostgreSQL uses to store the data in the database. In Section 16.2 it is explained where these files are located, but you have probably found them already if you are interested in this method. You can use whatever method you prefer for doing usual file system backups, for example

```
tar -cf backup.tar /usr/local/pgsql/data
```

There are two restrictions, however, which make this method impractical, or at least inferior to the pg_dump method:

1. The database server *must* be shut down in order to get a usable backup. Half-way measures such as disallowing all connections will *not* work (mainly because `tar` and similar tools do not take an atomic snapshot of the state of the file system at a point in time). Information about stopping the server can be found in Section 16.5. Needless to say that you also need to shut down the server before restoring the data.
2. If you have dug into the details of the file system layout of the database, you may be tempted to try to back up or restore only certain individual tables or databases from their respective files or directories. This will *not* work because the information contained in these files contains only half the truth. The other half is in the commit log files `pg_clog/*`, which contain the commit status of all transactions. A table file is only usable with this information. Of course it is also impossible to restore only a table and the associated `pg_clog` data because that would render all other tables in the database cluster useless. So file system backups only work for complete restoration of an entire database cluster.

An alternative file-system backup approach is to make a "consistent snapshot" of the data directory, if the file system supports that functionality (and you are willing to trust that it is implemented correctly). The typical procedure is to make a "frozen snapshot" of the volume containing the database, then copy the whole data directory (not just parts, see above) from the snapshot to a backup device, then release the frozen snapshot. This will work even while the database server is running. However, a backup created in this way saves the database files in a state where the database server was not properly shut down; therefore, when you start the database server on the backed-up data, it will think the server had crashed and replay the WAL log. This is not a problem, just be aware of it (and be sure to include the WAL files in your backup).

If your database is spread across multiple file systems, there may not be any way to obtain exactly-simultaneous frozen snapshots of all the volumes. For example, if your data files and WAL log are on different disks, or if tablespaces are on different file systems, it might not be possible to use snapshot backup because the snapshots must be simultaneous. Read your file system documentation very carefully

before trusting to the consistent-snapshot technique in such situations. The safest approach is to shut down the database server for long enough to establish all the frozen snapshots.

Another option is to use rsync to perform a file system backup. This is done by first running rsync while the database server is running, then shutting down the database server just long enough to do a second rsync. The second rsync will be much quicker than the first, because it has relatively little data to transfer, and the end result will be consistent because the server was down. This method allows a file system backup to be performed with minimal downtime.

Note that a file system backup will not necessarily be smaller than an SQL dump. On the contrary, it will most likely be larger. (pg_dump does not need to dump the contents of indexes for example, just the commands to recreate them.)

23.3. Continuous Archiving and Point-In-Time Recovery (PITR)

At all times, PostgreSQL maintains a *write ahead log* (WAL) in the `pg_xlog/` subdirectory of the cluster's data directory. The log describes every change made to the database's data files. This log exists primarily for crash-safety purposes: if the system crashes, the database can be restored to consistency by "replaying" the log entries made since the last checkpoint. However, the existence of the log makes it possible to use a third strategy for backing up databases: we can combine a file-system-level backup with backup of the WAL files. If recovery is needed, we restore the backup and then replay from the backed-up WAL files to bring the backup up to current time. This approach is more complex to administer than either of the previous approaches, but it has some significant benefits:

- We do not need a perfectly consistent backup as the starting point. Any internal inconsistency in the backup will be corrected by log replay (this is not significantly different from what happens during crash recovery). So we don't need file system snapshot capability, just tar or a similar archiving tool.
- Since we can string together an indefinitely long sequence of WAL files for replay, continuous backup can be achieved simply by continuing to archive the WAL files. This is particularly valuable for large databases, where it may not be convenient to take a full backup frequently.
- There is nothing that says we have to replay the WAL entries all the way to the end. We could stop the replay at any point and have a consistent snapshot of the database as it was at that time. Thus, this technique supports *point-in-time recovery*: it is possible to restore the database to its state at any time since your base backup was taken.
- If we continuously feed the series of WAL files to another machine that has been loaded with the same base backup file, we have a *warm standby* system: at any point we can bring up the second machine and it will have a nearly-current copy of the database.

As with the plain file-system-backup technique, this method can only support restoration of an entire database cluster, not a subset. Also, it requires a lot of archival storage: the base backup may be bulky, and a busy system will generate many megabytes of WAL traffic that have to be archived. Still, it is the preferred backup technique in many situations where high reliability is needed.

To recover successfully using continuous archiving (also called "online backup" by many database vendors), you need a continuous sequence of archived WAL files that extends back at least as far as the start time of your backup. So to get started, you should setup and test your procedure for archiving WAL files *before* you take your first base backup. Accordingly, we first discuss the mechanics of archiving WAL files.

23.3.1. Setting up WAL archiving

In an abstract sense, a running PostgreSQL system produces an indefinitely long sequence of WAL records. The system physically divides this sequence into WAL *segment files*, which are normally 16MB apiece (although the size can be altered when building PostgreSQL). The segment files are given numeric names that reflect their position in the abstract WAL sequence. When not using WAL archiving, the system normally creates just a few segment files and then "recycles" them by renaming no-longer-needed segment files to higher segment numbers. It's assumed that a segment file whose contents precede the checkpoint-before-last is no longer of interest and can be recycled.

When archiving WAL data, we want to capture the contents of each segment file once it is filled, and save that data somewhere before the segment file is recycled for reuse. Depending on the application and the available hardware, there could be many different ways of "saving the data somewhere": we could copy the segment files to an NFS-mounted directory on another machine, write them onto a tape drive (ensuring that you have a way of identifying the original name of each file), or batch them together and burn them onto CDs, or something else entirely. To provide the database administrator with as much flexibility as possible, PostgreSQL tries not to make any assumptions about how the archiving will be done. Instead, PostgreSQL lets the administrator specify a shell command to be executed to copy a completed segment file to wherever it needs to go. The command could be as simple as a `cp`, or it could invoke a complex shell script — it's all up to you.

The shell command to use is specified by the archive_command configuration parameter, which in practice will always be placed in the `postgresql.conf` file. In this string, any `%p` is replaced by the path name of the file to archive, while any `%f` is replaced by the file name only. (The path name is relative to the working directory of the server, i.e., the cluster's data directory.) Write `%%` if you need to embed an actual `%` character in the command. The simplest useful command is something like

```
archive_command = 'cp -i %p /mnt/server/archivedir/%f </dev/null'
```

which will copy archivable WAL segments to the directory `/mnt/server/archivedir`. (This is an example, not a recommendation, and may not work on all platforms.)

The archive command will be executed under the ownership of the same user that the PostgreSQL server is running as. Since the series of WAL files being archived contains effectively everything in your database, you will want to be sure that the archived data is protected from prying eyes; for example, archive into a directory that does not have group or world read access.

It is important that the archive command return zero exit status if and only if it succeeded. Upon getting a zero result, PostgreSQL will assume that the WAL segment file has been successfully archived, and will remove or recycle it. However, a nonzero status tells PostgreSQL that the file was not archived; it will try again periodically until it succeeds.

The archive command should generally be designed to refuse to overwrite any pre-existing archive file. This is an important safety feature to preserve the integrity of your archive in case of administrator error (such as sending the output of two different servers to the same archive directory). It is advisable to test

your proposed archive command to ensure that it indeed does not overwrite an existing file, *and that it returns nonzero status in this case*. We have found that `cp -i` does this correctly on some platforms but not others. If the chosen command does not itself handle this case correctly, you should add a command to test for pre-existence of the archive file. For example, something like

```
archive_command = 'test ! -f .../%f && cp %p .../%f'
```

works correctly on most Unix variants.

While designing your archiving setup, consider what will happen if the archive command fails repeatedly because some aspect requires operator intervention or the archive runs out of space. For example, this could occur if you write to tape without an autochanger; when the tape fills, nothing further can be archived until the tape is swapped. You should ensure that any error condition or request to a human operator is reported appropriately so that the situation can be resolved relatively quickly. The `pg_xlog/` directory will continue to fill with WAL segment files until the situation is resolved.

The speed of the archiving command is not important, so long as it can keep up with the average rate at which your server generates WAL data. Normal operation continues even if the archiving process falls a little behind. If archiving falls significantly behind, this will increase the amount of data that would be lost in the event of a disaster. It will also mean that the `pg_xlog/` directory will contain large numbers of not-yet-archived segment files, which could eventually exceed available disk space. You are advised to monitor the archiving process to ensure that it is working as you intend.

In writing your archive command, you should assume that the file names to be archived may be up to 64 characters long and may contain any combination of ASCII letters, digits, and dots. It is not necessary to remember the original relative path (`%p`) but it is necessary to remember the file name (`%f`).

Note that although WAL archiving will allow you to restore any modifications made to the data in your PostgreSQL database, it will not restore changes made to configuration files (that is, `postgresql.conf`, `pg_hba.conf` and `pg_ident.conf`), since those are edited manually rather than through SQL operations. You may wish to keep the configuration files in a location that will be backed up by your regular file system backup procedures. See Section 17.2 for how to relocate the configuration files.

The archive command is only invoked on completed WAL segments. Hence, if your server generates only little WAL traffic (or has slack periods where it does so), there could be a long delay between the completion of a transaction and its safe recording in archive storage. To put a limit on how old unarchived data can be, you can set archive_timeout to force the server to switch to a new WAL segment file at least that often. Note that archived files that are ended early due to a forced switch are still the same length as completely full files. It is therefore unwise to set a very short `archive_timeout` — it will bloat your archive storage. `archive_timeout` settings of a minute or so are usually reasonable.

Also, you can force a segment switch manually with `pg_switch_xlog`, if you want to ensure that a just-finished transaction is archived immediately. Other utility functions related to WAL management are listed in Table 9-47.

23.3.2. Making a Base Backup

The procedure for making a base backup is relatively simple:

1. Ensure that WAL archiving is enabled and working.

2. Connect to the database as a superuser, and issue the command

   ```
   SELECT pg_start_backup('label');
   ```
   where `label` is any string you want to use to uniquely identify this backup operation. (One good practice is to use the full path where you intend to put the backup dump file.) `pg_start_backup` creates a *backup label* file, called `backup_label`, in the cluster directory with information about your backup.

   It does not matter which database within the cluster you connect to to issue this command. You can ignore the result returned by the function; but if it reports an error, deal with that before proceeding.

3. Perform the backup, using any convenient file-system-backup tool such as tar or cpio. It is neither necessary nor desirable to stop normal operation of the database while you do this.

4. Again connect to the database as a superuser, and issue the command

   ```
   SELECT pg_stop_backup();
   ```
   This terminates the backup mode and performs an automatic switch to the next WAL segment. The reason for the switch is to arrange that the last WAL segment file written during the backup interval is immediately ready to archive.

5. Once the WAL segment files used during the backup are archived, you are done. The file identified by `pg_stop_backup`'s result is the last segment that needs to be archived to complete the backup. Archival of these files will happen automatically, since you have already configured `archive_command`. In many cases, this happens fairly quickly, but you are advised to monitor your archival system to ensure this has taken place so that you can be certain you have a complete backup.

Some backup tools that you might wish to use emit warnings or errors if the files they are trying to copy change while the copy proceeds. This situation is normal, and not an error, when taking a base backup of an active database; so you need to ensure that you can distinguish complaints of this sort from real errors. For example, some versions of rsync return a separate exit code for "vanished source files", and you can write a driver script to accept this exit code as a non-error case. Also, some versions of GNU tar return an error code indistinguishable from a fatal error if a file was truncated while tar was copying it. Fortunately, GNU tar versions 1.16 and later exits with `1` if a file was changed during the backup, and `2` for other errors.

It is not necessary to be very concerned about the amount of time elapsed between `pg_start_backup` and the start of the actual backup, nor between the end of the backup and `pg_stop_backup`; a few minutes' delay won't hurt anything. (However, if you normally run the server with `full_page_writes` disabled, you may notice a drop in performance between `pg_start_backup` and `pg_stop_backup`, since `full_page_writes` is effectively forced on during backup mode.) You must ensure that these steps are carried out in sequence without any possible overlap, or you will invalidate the backup.

Be certain that your backup dump includes all of the files underneath the database cluster directory (e.g., `/usr/local/pgsql/data`). If you are using tablespaces that do not reside underneath this directory, be careful to include them as well (and be sure that your backup dump archives symbolic links as links, otherwise the restore will mess up your tablespaces).

You may, however, omit from the backup dump the files within the `pg_xlog/` subdirectory of the cluster directory. This slight complication is worthwhile because it reduces the risk of mistakes when restoring. This is easy to arrange if `pg_xlog/` is a symbolic link pointing to someplace outside the cluster directory, which is a common setup anyway for performance reasons.

To make use of the backup, you will need to keep around all the WAL segment files generated during and after the file system backup. To aid you in doing this, the pg_stop_backup function creates a *backup history file* that is immediately stored into the WAL archive area. This file is named after the first WAL segment file that you need to have to make use of the backup. For example, if the starting WAL file is 000000010000123400000055CD the backup history file will be named something like 000000010000123400000055CD.007C9330.backup. (The second number in the file name stands for an exact position within the WAL file, and can ordinarily be ignored.) Once you have safely archived the file system backup and the WAL segment files used during the backup (as specified in the backup history file), all archived WAL segments with names numerically less are no longer needed to recover the file system backup and may be deleted. However, you should consider keeping several backup sets to be absolutely certain that you can recover your data.

The backup history file is just a small text file. It contains the label string you gave to pg_start_backup, as well as the starting and ending times and WAL segments of the backup. If you used the label to identify where the associated dump file is kept, then the archived history file is enough to tell you which dump file to restore, should you need to do so.

Since you have to keep around all the archived WAL files back to your last base backup, the interval between base backups should usually be chosen based on how much storage you want to expend on archived WAL files. You should also consider how long you are prepared to spend recovering, if recovery should be necessary — the system will have to replay all those WAL segments, and that could take awhile if it has been a long time since the last base backup.

It's also worth noting that the pg_start_backup function makes a file named backup_label in the database cluster directory, which is then removed again by pg_stop_backup. This file will of course be archived as a part of your backup dump file. The backup label file includes the label string you gave to pg_start_backup, as well as the time at which pg_start_backup was run, and the name of the starting WAL file. In case of confusion it will therefore be possible to look inside a backup dump file and determine exactly which backup session the dump file came from.

It is also possible to make a backup dump while the server is stopped. In this case, you obviously cannot use pg_start_backup or pg_stop_backup, and you will therefore be left to your own devices to keep track of which backup dump is which and how far back the associated WAL files go. It is generally better to follow the continuous archiving procedure above.

23.3.3. Recovering using a Continuous Archive Backup

Okay, the worst has happened and you need to recover from your backup. Here is the procedure:

1. Stop the server, if it's running.
2. If you have the space to do so, copy the whole cluster data directory and any tablespaces to a temporary location in case you need them later. Note that this precaution will require that you have enough free space on your system to hold two copies of your existing database. If you do not have enough space, you need at the least to copy the contents of the pg_xlog subdirectory of the cluster data directory, as it may contain logs which were not archived before the system went down.
3. Clean out all existing files and subdirectories under the cluster data directory and under the root directories of any tablespaces you are using.

4. Restore the database files from your backup dump. Be careful that they are restored with the right ownership (the database system user, not root!) and with the right permissions. If you are using tablespaces, you should verify that the symbolic links in pg_tblspc/ were correctly restored.
5. Remove any files present in pg_xlog/; these came from the backup dump and are therefore probably obsolete rather than current. If you didn't archive pg_xlog/ at all, then recreate it, and be sure to recreate the subdirectory pg_xlog/archive_status/ as well.
6. If you had unarchived WAL segment files that you saved in step 2, copy them into pg_xlog/. (It is best to copy them, not move them, so that you still have the unmodified files if a problem occurs and you have to start over.)
7. Create a recovery command file recovery.conf in the cluster data directory (see Recovery Settings). You may also want to temporarily modify pg_hba.conf to prevent ordinary users from connecting until you are sure the recovery has worked.
8. Start the server. The server will go into recovery mode and proceed to read through the archived WAL files it needs. Should the recovery be terminated because of an external error, the server can simply be restarted and it will continue recovery. Upon completion of the recovery process, the server will rename recovery.conf to recovery.done (to prevent accidentally re-entering recovery mode in case of a crash later) and then commence normal database operations.
9. Inspect the contents of the database to ensure you have recovered to where you want to be. If not, return to step 1. If all is well, let in your users by restoring pg_hba.conf to normal.

The key part of all this is to setup a recovery command file that describes how you want to recover and how far the recovery should run. You can use recovery.conf.sample (normally installed in the installation share/ directory) as a prototype. The one thing that you absolutely must specify in recovery.conf is the restore_command, which tells PostgreSQL how to get back archived WAL file segments. Like the archive_command, this is a shell command string. It may contain %f, which is replaced by the name of the desired log file, and %p, which is replaced by the path name to copy the log file to. (The path name is relative to the working directory of the server, i.e., the cluster's data directory.) Write %% if you need to embed an actual % character in the command. The simplest useful command is something like

```
restore_command = 'cp /mnt/server/archivedir/%f %p'
```

which will copy previously archived WAL segments from the directory /mnt/server/archivedir. You could of course use something much more complicated, perhaps even a shell script that requests the operator to mount an appropriate tape.

It is important that the command return nonzero exit status on failure. The command *will* be asked for log files that are not present in the archive; it must return nonzero when so asked. This is not an error condition. Be aware also that the base name of the %p path will be different from %f; do not expect them to be interchangeable.

WAL segments that cannot be found in the archive will be sought in pg_xlog/; this allows use of recent un-archived segments. However segments that are available from the archive will be used in preference to files in pg_xlog/. The system will not overwrite the existing contents of pg_xlog/ when retrieving archived files.

Normally, recovery will proceed through all available WAL segments, thereby restoring the database to the current point in time (or as close as we can get given the available WAL segments). But if you want to

recover to some previous point in time (say, right before the junior DBA dropped your main transaction table), just specify the required stopping point in `recovery.conf`. You can specify the stop point, known as the "recovery target", either by date/time or by completion of a specific transaction ID. As of this writing only the date/time option is very usable, since there are no tools to help you identify with any accuracy which transaction ID to use.

> **Note:** The stop point must be after the ending time of the base backup (the time of `pg_stop_backup`). You cannot use a base backup to recover to a time when that backup was still going on. (To recover to such a time, you must go back to your previous base backup and roll forward from there.)

If recovery finds a corruption in the WAL data then recovery will complete at that point and the server will not start. In such a case the recovery process could be re-run from the beginning, specifying a "recovery target" before the point of corruption so that recovery can complete normally. If recovery fails for an external reason, such as a system crash or if the WAL archive has become inaccessible, then the recovery can simply be restarted and it will restart almost from where it failed. Recovery restart works much like checkpointing in normal operation: the server periodically forces all its state to disk, and then updates the `pg_control` file to indicate that the already-processed WAL data need not be scanned again.

23.3.3.1. Recovery Settings

These settings can only be made in the `recovery.conf` file, and apply only for the duration of the recovery. They must be reset for any subsequent recovery you wish to perform. They cannot be changed once recovery has begun.

`restore_command` (`string`)

The shell command to execute to retrieve an archived segment of the WAL file series. This parameter is required. Any `%f` in the string is replaced by the name of the file to retrieve from the archive, and any `%p` is replaced by the path name to copy it to on the server. (The path name is relative to the working directory of the server, i.e., the cluster's data directory.) Write `%%` to embed an actual `%` character in the command.

It is important for the command to return a zero exit status if and only if it succeeds. The command *will* be asked for file names that are not present in the archive; it must return nonzero when so asked. Examples:

```
restore_command = 'cp /mnt/server/archivedir/%f "%p"'
restore_command = 'copy /mnt/server/archivedir/%f "%p"'  # Windows
```

`recovery_target_time` (`timestamp`)

This parameter specifies the time stamp up to which recovery will proceed. At most one of `recovery_target_time` and recovery_target_xid can be specified. The default is to recover to the end of the WAL log. The precise stopping point is also influenced by recovery_target_inclusive.

`recovery_target_xid` (`string`)

This parameter specifies the transaction ID up to which recovery will proceed. Keep in mind that while transaction IDs are assigned sequentially at transaction start, transactions can complete in a different numeric order. The transactions that will be recovered are those that committed before (and optionally including) the specified one. At most one of `recovery_target_xid` and recov-

ery_target_time can be specified. The default is to recover to the end of the WAL log. The precise stopping point is also influenced by recovery_target_inclusive.

`recovery_target_inclusive` (`boolean`)

Specifies whether we stop just after the specified recovery target (`true`), or just before the recovery target (`false`). Applies to both recovery_target_time and recovery_target_xid, whichever one is specified for this recovery. This indicates whether transactions having exactly the target commit time or ID, respectively, will be included in the recovery. Default is `true`.

`recovery_target_timeline` (`string`)

Specifies recovering into a particular timeline. The default is to recover along the same timeline that was current when the base backup was taken. You would only need to set this parameter in complex re-recovery situations, where you need to return to a state that itself was reached after a point-in-time recovery. See Section 23.3.4 for discussion.

23.3.4. Timelines

The ability to restore the database to a previous point in time creates some complexities that are akin to science-fiction stories about time travel and parallel universes. In the original history of the database, perhaps you dropped a critical table at 5:15PM on Tuesday evening. Unfazed, you get out your backup, restore to the point-in-time 5:14PM Tuesday evening, and are up and running. In *this* history of the database universe, you never dropped the table at all. But suppose you later realize this wasn't such a great idea after all, and would like to return to some later point in the original history. You won't be able to if, while your database was up-and-running, it overwrote some of the sequence of WAL segment files that led up to the time you now wish you could get back to. So you really want to distinguish the series of WAL records generated after you've done a point-in-time recovery from those that were generated in the original database history.

To deal with these problems, PostgreSQL has a notion of *timelines*. Whenever an archive recovery is completed, a new timeline is created to identify the series of WAL records generated after that recovery. The timeline ID number is part of WAL segment file names, and so a new timeline does not overwrite the WAL data generated by previous timelines. It is in fact possible to archive many different timelines. While that might seem like a useless feature, it's often a lifesaver. Consider the situation where you aren't quite sure what point-in-time to recover to, and so have to do several point-in-time recoveries by trial and error until you find the best place to branch off from the old history. Without timelines this process would soon generate an unmanageable mess. With timelines, you can recover to *any* prior state, including states in timeline branches that you later abandoned.

Each time a new timeline is created, PostgreSQL creates a "timeline history" file that shows which timeline it branched off from and when. These history files are necessary to allow the system to pick the right WAL segment files when recovering from an archive that contains multiple timelines. Therefore, they are archived into the WAL archive area just like WAL segment files. The history files are just small text files, so it's cheap and appropriate to keep them around indefinitely (unlike the segment files which are large). You can, if you like, add comments to a history file to make your own notes about how and why this particular timeline came to be. Such comments will be especially valuable when you have a thicket of different timelines as a result of experimentation.

The default behavior of recovery is to recover along the same timeline that was current when the base backup was taken. If you want to recover into some child timeline (that is, you want to return to some state that was itself generated after a recovery attempt), you need to specify the target timeline ID in `recovery.conf`. You cannot recover into timelines that branched off earlier than the base backup.

23.3.5. Caveats

At this writing, there are several limitations of the continuous archiving technique. These will probably be fixed in future releases:

- Operations on hash indexes are not presently WAL-logged, so replay will not update these indexes. The recommended workaround is to manually *REINDEX* each such index after completing a recovery operation.
- If a *CREATE DATABASE* command is executed while a base backup is being taken, and then the template database that the `CREATE DATABASE` copied is modified while the base backup is still in progress, it is possible that recovery will cause those modifications to be propagated into the created database as well. This is of course undesirable. To avoid this risk, it is best not to modify any template databases while taking a base backup.
- *CREATE TABLESPACE* commands are WAL-logged with the literal absolute path, and will therefore be replayed as tablespace creations with the same absolute path. This might be undesirable if the log is being replayed on a different machine. It can be dangerous even if the log is being replayed on the same machine, but into a new data directory: the replay will still overwrite the contents of the original tablespace. To avoid potential gotchas of this sort, the best practice is to take a new base backup after creating or dropping tablespaces.

It should also be noted that the default WAL format is fairly bulky since it includes many disk page snapshots. These page snapshots are designed to support crash recovery, since we may need to fix partially-written disk pages. Depending on your system hardware and software, the risk of partial writes may be small enough to ignore, in which case you can significantly reduce the total volume of archived logs by turning off page snapshots using the full_page_writes parameter. (Read the notes and warnings in Chapter 27 before you do so.) Turning off page snapshots does not prevent use of the logs for PITR operations. An area for future development is to compress archived WAL data by removing unnecessary page copies even when `full_page_writes` is on. In the meantime, administrators may wish to reduce the number of page snapshots included in WAL by increasing the checkpoint interval parameters as much as feasible.

23.4. Warm Standby Servers for High Availability

Continuous archiving can be used to create a *high availability* (HA) cluster configuration with one or more *standby servers* ready to take over operations if the primary server fails. This capability is widely referred to as *warm standby* or *log shipping*.

The primary and standby server work together to provide this capability, though the servers are only loosely coupled. The primary server operates in continuous archiving mode, while each standby server

operates in continuous recovery mode, reading the WAL files from the primary. No changes to the database tables are required to enable this capability, so it offers low administration overhead in comparison with some other replication approaches. This configuration also has relatively low performance impact on the primary server.

Directly moving WAL or "log" records from one database server to another is typically described as log shipping. PostgreSQL implements file-based log shipping, which means that WAL records are transferred one file (WAL segment) at a time. WAL files can be shipped easily and cheaply over any distance, whether it be to an adjacent system, another system on the same site or another system on the far side of the globe. The bandwidth required for this technique varies according to the transaction rate of the primary server. Record-based log shipping is also possible with custom-developed procedures, as discussed in Section 23.4.4.

It should be noted that the log shipping is asynchronous, i.e. the WAL records are shipped after transaction commit. As a result there is a window for data loss should the primary server suffer a catastrophic failure: transactions not yet shipped will be lost. The length of the window of data loss can be limited by use of the `archive_timeout` parameter, which can be set as low as a few seconds if required. However such low settings will substantially increase the bandwidth requirements for file shipping. If you need a window of less than a minute or so, it's probably better to look into record-based log shipping.

The standby server is not available for access, since it is continually performing recovery processing. Recovery performance is sufficiently good that the standby will typically be only moments away from full availability once it has been activated. As a result, we refer to this capability as a warm standby configuration that offers high availability. Restoring a server from an archived base backup and rollforward will take considerably longer, so that technique only really offers a solution for disaster recovery, not HA.

23.4.1. Planning

It is usually wise to create the primary and standby servers so that they are as similar as possible, at least from the perspective of the database server. In particular, the path names associated with tablespaces will be passed across as-is, so both primary and standby servers must have the same mount paths for tablespaces if that feature is used. Keep in mind that if *CREATE TABLESPACE* is executed on the primary, any new mount point needed for it must be created on both the primary and all standby servers before the command is executed. Hardware need not be exactly the same, but experience shows that maintaining two identical systems is easier than maintaining two dissimilar ones over the lifetime of the application and system. In any case the hardware architecture must be the same — shipping from, say, a 32-bit to a 64-bit system will not work.

In general, log shipping between servers running different major release levels will not be possible. It is the policy of the PostgreSQL Global Development Group not to make changes to disk formats during minor release upgrades, so it is likely that running different minor release levels on primary and standby servers will work successfully. However, no formal support for that is offered and you are advised to keep primary and standby servers at the same release level as much as possible. When updating to a new minor release, the safest policy is to update the standby servers first — a new minor release is more likely to be able to read WAL files from a previous minor release than vice versa.

There is no special mode required to enable a standby server. The operations that occur on both primary and standby servers are entirely normal continuous archiving and recovery tasks. The only point of contact between the two database servers is the archive of WAL files that both share: primary writing to the

archive, standby reading from the archive. Care must be taken to ensure that WAL archives for separate primary servers do not become mixed together or confused.

The magic that makes the two loosely coupled servers work together is simply a `restore_command` used on the standby that waits for the next WAL file to become available from the primary. The `restore_command` is specified in the `recovery.conf` file on the standby server. Normal recovery processing would request a file from the WAL archive, reporting failure if the file was unavailable. For standby processing it is normal for the next file to be unavailable, so we must be patient and wait for it to appear. A waiting `restore_command` can be written as a custom script that loops after polling for the existence of the next WAL file. There must also be some way to trigger failover, which should interrupt the `restore_command`, break the loop and return a file-not-found error to the standby server. This ends recovery and the standby will then come up as a normal server.

Pseudocode for a suitable `restore_command` is:

```
triggered = false;
while (!NextWALFileReady() && !triggered)
{
    sleep(100000L);         /* wait for ~0.1 sec */
    if (CheckForExternalTrigger())
        triggered = true;
}
if (!triggered)
        CopyWALFileForRecovery();
```

PostgreSQL does not provide the system software required to identify a failure on the primary and notify the standby system and then the standby database server. Many such tools exist and are well integrated with other aspects required for successful failover, such as IP address migration.

The means for triggering failover is an important part of planning and design. The `restore_command` is executed in full once for each WAL file. The process running the `restore_command` is therefore created and dies for each file, so there is no daemon or server process and so we cannot use signals and a signal handler. A more permanent notification is required to trigger the failover. It is possible to use a simple timeout facility, especially if used in conjunction with a known `archive_timeout` setting on the primary. This is somewhat error prone since a network problem or busy primary server might be sufficient to initiate failover. A notification mechanism such as the explicit creation of a trigger file is less error prone, if this can be arranged.

23.4.2. Implementation

The short procedure for configuring a standby server is as follows. For full details of each step, refer to previous sections as noted.

1. Set up primary and standby systems as near identically as possible, including two identical copies of PostgreSQL at the same release level.
2. Set up continuous archiving from the primary to a WAL archive located in a directory on the standby server. Ensure that archive_command and archive_timeout are set appropriately on the primary (see Section 23.3.1).

3. Make a base backup of the primary server (see Section 23.3.2), and load this data onto the standby.
4. Begin recovery on the standby server from the local WAL archive, using a `recovery.conf` that specifies a `restore_command` that waits as described previously (see Section 23.3.3).

Recovery treats the WAL archive as read-only, so once a WAL file has been copied to the standby system it can be copied to tape at the same time as it is being read by the standby database server. Thus, running a standby server for high availability can be performed at the same time as files are stored for longer term disaster recovery purposes.

For testing purposes, it is possible to run both primary and standby servers on the same system. This does not provide any worthwhile improvement in server robustness, nor would it be described as HA.

23.4.3. Failover

If the primary server fails then the standby server should begin failover procedures.

If the standby server fails then no failover need take place. If the standby server can be restarted, even some time later, then the recovery process can also be immediately restarted, taking advantage of restartable recovery. If the standby server cannot be restarted, then a full new standby server should be created.

If the primary server fails and then immediately restarts, you must have a mechanism for informing it that it is no longer the primary. This is sometimes known as STONITH (Shoot the Other Node In The Head), which is necessary to avoid situations where both systems think they are the primary, which can lead to confusion and ultimately data loss.

Many failover systems use just two systems, the primary and the standby, connected by some kind of heartbeat mechanism to continually verify the connectivity between the two and the viability of the primary. It is also possible to use a third system (called a witness server) to avoid some problems of inappropriate failover, but the additional complexity may not be worthwhile unless it is set-up with sufficient care and rigorous testing.

Once failover to the standby occurs, we have only a single server in operation. This is known as a degenerate state. The former standby is now the primary, but the former primary is down and may stay down. To return to normal operation we must fully recreate a standby server, either on the former primary system when it comes up, or on a third, possibly new, system. Once complete the primary and standby can be considered to have switched roles. Some people choose to use a third server to provide backup to the new primary until the new standby server is recreated, though clearly this complicates the system configuration and operational processes.

So, switching from primary to standby server can be fast but requires some time to re-prepare the failover cluster. Regular switching from primary to standby is encouraged, since it allows regular downtime on each system for maintenance. This also acts as a test of the failover mechanism to ensure that it will really work when you need it. Written administration procedures are advised.

23.4.4. Record-based Log Shipping

PostgreSQL directly supports file-based log shipping as described above. It is also possible to implement record-based log shipping, though this requires custom development.

An external program can call the `pg_xlogfile_name_offset()` function (see Section 9.20) to find out the file name and the exact byte offset within it of the current end of WAL. It can then access the WAL file directly and copy the data from the last known end of WAL through the current end over to the standby server(s). With this approach, the window for data loss is the polling cycle time of the copying program, which can be very small, but there is no wasted bandwidth from forcing partially-used segment files to be archived. Note that the standby servers' `restore_command` scripts still deal in whole WAL files, so the incrementally copied data is not ordinarily made available to the standby servers. It is of use only when the primary dies — then the last partial WAL file is fed to the standby before allowing it to come up. So correct implementation of this process requires cooperation of the `restore_command` script with the data copying program.

23.4.5. Incrementally Updated Backups

In a warm standby configuration, it is possible to offload the expense of taking periodic base backups from the primary server; instead base backups can be made by backing up a standby server's files. This concept is generally known as incrementally updated backups, log change accumulation or more simply, change accumulation.

If we take a backup of the standby server's files while it is following logs shipped from the primary, we will be able to reload that data and restart the standby's recovery process from the last restart point. We no longer need to keep WAL files from before the restart point. If we need to recover, it will be faster to recover from the incrementally updated backup than from the original base backup.

Since the standby server is not "live", it is not possible to use `pg_start_backup()` and `pg_stop_backup()` to manage the backup process; it will be up to you to determine how far back you need to keep WAL segment files to have a recoverable backup. You can do this by running pg_controldata on the standby server to inspect the control file and determine the current checkpoint WAL location.

23.5. Migration Between Releases

This section discusses how to migrate your database data from one PostgreSQL release to a newer one. The software installation procedure *per se* is not the subject of this section; those details are in Chapter 14.

As a general rule, the internal data storage format is subject to change between major releases of PostgreSQL (where the number after the first dot changes). This does not apply to different minor releases under the same major release (where the number after the second dot changes); these always have compatible storage formats. For example, releases 7.2.1, 7.3.2, and 7.4 are not compatible, whereas 7.2.1 and 7.2.2 are. When you update between compatible versions, you can simply replace the executables and reuse the data directory on disk. Otherwise you need to back up your data and restore it on the new server. This has to be done using pg_dump; file system level backup methods obviously won't work. There are checks in place that prevent you from using a data directory with an incompatible version of PostgreSQL, so no great harm can be done by trying to start the wrong server version on a data directory.

It is recommended that you use the pg_dump and pg_dumpall programs from the newer version of PostgreSQL, to take advantage of any enhancements that may have been made in these programs. Current releases of the dump programs can read data from any server version back to 7.0.

The least downtime can be achieved by installing the new server in a different directory and running both the old and the new servers in parallel, on different ports. Then you can use something like

```
pg_dumpall -p 5432 | psql -d postgres -p 6543
```

to transfer your data. Or use an intermediate file if you want. Then you can shut down the old server and start the new server at the port the old one was running at. You should make sure that the old database is not updated after you run pg_dumpall, otherwise you will obviously lose that data. See Chapter 20 for information on how to prohibit access.

In practice you probably want to test your client applications on the new setup before switching over completely. This is another reason for setting up concurrent installations of old and new versions.

If you cannot or do not want to run two servers in parallel you can do the backup step before installing the new version, bring down the server, move the old version out of the way, install the new version, start the new server, restore the data. For example:

```
pg_dumpall > backup
pg_ctl stop
mv /usr/local/pgsql /usr/local/pgsql.old
cd ~/postgresql-8.2.6
gmake install
initdb -D /usr/local/pgsql/data
postgres -D /usr/local/pgsql/data
psql -f backup postgres
```

See Chapter 16 about ways to start and stop the server and other details. The installation instructions will advise you of strategic places to perform these steps.

> **Note:** When you “move the old installation out of the way” it may no longer be perfectly usable. Some of the executable programs contain absolute paths to various installed programs and data files. This is usually not a big problem but if you plan on using two installations in parallel for a while you should assign them different installation directories at build time. (This problem is rectified in PostgreSQL 8.0 and later, but you need to be wary of moving older installations.)

Chapter 24. High Availability and Load Balancing

Database servers can work together to allow a second server to take over quickly if the primary server fails (high availability), or to allow several computers to serve the same data (load balancing). Ideally, database servers could work together seamlessly. Web servers serving static web pages can be combined quite easily by merely load-balancing web requests to multiple machines. In fact, read-only database servers can be combined relatively easily too. Unfortunately, most database servers have a read/write mix of requests, and read/write servers are much harder to combine. This is because though read-only data needs to be placed on each server only once, a write to any server has to be propagated to all servers so that future read requests to those servers return consistent results.

This synchronization problem is the fundamental difficulty for servers working together. Because there is no single solution that eliminates the impact of the sync problem for all use cases, there are multiple solutions. Each solution addresses this problem in a different way, and minimizes its impact for a specific workload.

Some solutions deal with synchronization by allowing only one server to modify the data. Servers that can modify data are called read/write or "master" servers. Servers that can reply to read-only queries are called "slave" servers. Servers that cannot be accessed until they are changed to master servers are called "standby" servers.

Some failover and load balancing solutions are synchronous, meaning that a data-modifying transaction is not considered committed until all servers have committed the transaction. This guarantees that a failover will not lose any data and that all load-balanced servers will return consistent results no matter which server is queried. In contrast, asynchronous solutions allow some delay between the time of a commit and its propagation to the other servers, opening the possibility that some transactions might be lost in the switch to a backup server, and that load balanced servers might return slightly stale results. Asynchronous communication is used when synchronous would be too slow.

Solutions can also be categorized by their granularity. Some solutions can deal only with an entire database server, while others allow control at the per-table or per-database level.

Performance must be considered in any failover or load balancing choice. There is usually a tradeoff between functionality and performance. For example, a full synchronous solution over a slow network might cut performance by more than half, while an asynchronous one might have a minimal performance impact.

The remainder of this section outlines various failover, replication, and load balancing solutions.

Shared Disk Failover

> Shared disk failover avoids synchronization overhead by having only one copy of the database. It uses a single disk array that is shared by multiple servers. If the main database server fails, the standby server is able to mount and start the database as though it was recovering from a database crash. This allows rapid failover with no data loss.
>
> Shared hardware functionality is common in network storage devices. Using a network file system is also possible, though care must be taken that the file system has full POSIX behavior. One significant limitation of this method is that if the shared disk array fails or becomes corrupt, the primary and

standby servers are both nonfunctional. Another issue is that the standby server should never access the shared storage while the primary server is running.

A modified version of shared hardware functionality is file system replication, where all changes to a file system are mirrored to a file system residing on another computer. The only restriction is that the mirroring must be done in a way that ensures the standby server has a consistent copy of the file system — specifically, writes to the standby must be done in the same order as those on the master. DRBD is a popular file system replication solution for Linux.

Warm Standby Using Point-In-Time Recovery

A warm standby server (see Section 23.4) can be kept current by reading a stream of write-ahead log (WAL) records. If the main server fails, the warm standby contains almost all of the data of the main server, and can be quickly made the new master database server. This is asynchronous and can only be done for the entire database server.

Master-Slave Replication

A master-slave replication setup sends all data modification queries to the master server. The master server asynchronously sends data changes to the slave server. The slave can answer read-only queries while the master server is running. The slave server is ideal for data warehouse queries.

Slony-I is an example of this type of replication, with per-table granularity, and support for multiple slaves. Because it updates the slave server asynchronously (in batches), there is possible data loss during fail over.

Statement-Based Replication Middleware

With statement-based replication middleware, a program intercepts every SQL query and sends it to one or all servers. Each server operates independently. Read-write queries are sent to all servers, while read-only queries can be sent to just one server, allowing the read workload to be distributed.

If queries are simply broadcast unmodified, functions like `random()`, `CURRENT_TIMESTAMP`, and sequences would have different values on different servers. This is because each server operates independently, and because SQL queries are broadcast (and not actual modified rows). If this is unacceptable, either the middleware or the application must query such values from a single server and then use those values in write queries. Also, care must be taken that all transactions either commit or abort on all servers, perhaps using two-phase commit (*PREPARE TRANSACTION* and *COMMIT PREPARED*. Pgpool and Sequoia are an example of this type of replication.

Synchronous Multi-Master Replication

In synchronous multi-master replication, each server can accept write requests, and modified data is transmitted from the original server to every other server before each transaction commits. Heavy write activity can cause excessive locking, leading to poor performance. In fact, write performance is often worse than that of a single server. Read requests can be sent to any server. Some implementations use shared disk to reduce the communication overhead. Synchronous multi-master replication is best for mostly read workloads, though its big advantage is that any server can accept write requests — there is no need to partition workloads between master and slave servers, and because the data changes are sent from one server to another, there is no problem with non-deterministic functions like `random()`.

PostgreSQL does not offer this type of replication, though PostgreSQL two-phase commit (*PREPARE TRANSACTION* and *COMMIT PREPARED*) can be used to implement this in application code or middleware.

Asynchronous Multi-Master Replication

For servers that are not regularly connected, like laptops or remote servers, keeping data consistent among servers is a challenge. Using asynchronous multi-master replication, each server works independently, and periodically communicates with the other servers to identify conflicting transactions. The conflicts can be resolved by users or conflict resolution rules.

Data Partitioning

Data partitioning splits tables into data sets. Each set can be modified by only one server. For example, data can be partitioned by offices, e.g. London and Paris, with a server in each office. If queries combining London and Paris data are necessary, an application can query both servers, or master/slave replication can be used to keep a read-only copy of the other office's data on each server.

Multi-Server Parallel Query Execution

Many of the above solutions allow multiple servers to handle multiple queries, but none allow a single query to use multiple servers to complete faster. This solution allows multiple servers to work concurrently on a single query. This is usually accomplished by splitting the data among servers and having each server execute its part of the query and return results to a central server where they are combined and returned to the user. Pgpool-II has this capability.

Commercial Solutions

Because PostgreSQL is open source and easily extended, a number of companies have taken PostgreSQL and created commercial closed-source solutions with unique failover, replication, and load balancing capabilities.

Chapter 25. Monitoring Database Activity

A database administrator frequently wonders, "What is the system doing right now?" This chapter discusses how to find that out.

Several tools are available for monitoring database activity and analyzing performance. Most of this chapter is devoted to describing PostgreSQL's statistics collector, but one should not neglect regular Unix monitoring programs such as `ps`, `top`, `iostat`, and `vmstat`. Also, once one has identified a poorly-performing query, further investigation may be needed using PostgreSQL's *EXPLAIN* command. Section 13.1 discusses `EXPLAIN` and other methods for understanding the behavior of an individual query.

25.1. Standard Unix Tools

On most platforms, PostgreSQL modifies its command title as reported by `ps`, so that individual server processes can readily be identified. A sample display is

```
$ ps auxww | grep ^postgres
postgres   960  0.0  1.1  6104 1480 pts/1    SN   13:17   0:00 postgres -i
postgres   963  0.0  1.1  7084 1472 pts/1    SN   13:17   0:00 postgres: writer process
postgres   965  0.0  1.1  6152 1512 pts/1    SN   13:17   0:00 postgres: stats collector pr
postgres   998  0.0  2.3  6532 2992 pts/1    SN   13:18   0:00 postgres: tgl runbug 127.0.0
postgres  1003  0.0  2.4  6532 3128 pts/1    SN   13:19   0:00 postgres: tgl regression [lo
postgres  1016  0.1  2.4  6532 3080 pts/1    SN   13:19   0:00 postgres: tgl regression [lo
```

(The appropriate invocation of `ps` varies across different platforms, as do the details of what is shown. This example is from a recent Linux system.) The first process listed here is the master server process. The command arguments shown for it are the same ones given when it was launched. The next two processes are background worker processes automatically launched by the master process. (The "stats collector" process will not be present if you have set the system not to start the statistics collector.) Each of the remaining processes is a server process handling one client connection. Each such process sets its command line display in the form

```
postgres: user database host activity
```

The user, database, and connection source host items remain the same for the life of the client connection, but the activity indicator changes. The activity may be `idle` (i.e., waiting for a client command), `idle in transaction` (waiting for client inside a `BEGIN` block), or a command type name such as `SELECT`. Also, `waiting` is attached if the server process is presently waiting on a lock held by another server process. In the above example we can infer that process 1003 is waiting for process 1016 to complete its transaction and thereby release some lock or other.

If you have turned off update_process_title then the activity indicator is not updated; the process title is set only once when a new process is launched. On some platforms this saves a useful amount of per-command overhead, on others it's insignificant.

> **Tip:** Solaris requires special handling. You must use `/usr/ucb/ps`, rather than `/bin/ps`. You also must use two `w` flags, not just one. In addition, your original invocation of the `postgres` command must have a shorter `ps` status display than that provided by each server process. If you fail to do all three things, the `ps` output for each server process will be the original `postgres` command line.

25.2. The Statistics Collector

PostgreSQL's *statistics collector* is a subsystem that supports collection and reporting of information about server activity. Presently, the collector can count accesses to tables and indexes in both disk-block and individual-row terms.

PostgreSQL also supports determining the exact command currently being executed by other server processes. This is an independent facility that can be enabled or disabled whether or not block-level and row-level statistics are being collected.

25.2.1. Statistics Collection Configuration

Since collection of statistics adds some overhead to query execution, the system can be configured to collect or not collect information. This is controlled by configuration parameters that are normally set in `postgresql.conf`. (See Chapter 17 for details about setting configuration parameters.)

The parameter stats_start_collector must be set to `true` for the statistics collector to be launched at all. This is the default and recommended setting, but it may be turned off if you have no interest in statistics and want to squeeze out every last drop of overhead. (The savings is likely to be small, however.) Note that this option cannot be changed while the server is running.

The parameters stats_block_level and stats_row_level control how much information is actually sent to the collector and thus determine how much run-time overhead occurs. These respectively determine whether a server process tracks disk-block-level access statistics and row-level access statistics and sends these to the collector. Additionally, per-database transaction commit and abort statistics are collected if either of these parameters are set.

The parameter stats_command_string enables monitoring of the current command being executed by any server process. The statistics collector subprocess need not be running to enable this feature.

Normally these parameters are set in `postgresql.conf` so that they apply to all server processes, but it is possible to turn them on or off in individual sessions using the *SET* command. (To prevent ordinary users from hiding their activity from the administrator, only superusers are allowed to change these parameters with `SET`.)

> **Note:** Since the parameters `stats_block_level`, and `stats_row_level` default to `false`, very few statistics are collected in the default configuration. Enabling either of these configuration variables will significantly increase the amount of useful data produced by the statistics facilities, at the expense of additional run-time overhead.

25.2.2. Viewing Collected Statistics

Several predefined views, listed in Table 25-1, are available to show the results of statistics collection. Alternatively, one can build custom views using the underlying statistics functions.

When using the statistics to monitor current activity, it is important to realize that the information does not update instantaneously. Each individual server process transmits new block and row access counts to the collector just before going idle; so a query or transaction still in progress does not affect the displayed totals. Also, the collector itself emits a new report at most once per `PGSTAT_STAT_INTERVAL` milliseconds (500 unless altered while building the server). So the displayed information lags behind actual activity. However, current-query information collected by `stats_command_string` is always up-to-date.

Another important point is that when a server process is asked to display any of these statistics, it first fetches the most recent report emitted by the collector process and then continues to use this snapshot for all statistical views and functions until the end of its current transaction. So the statistics will appear not to change as long as you continue the current transaction. Similarly, information about the current queries of all processes is collected when any such information is first requested within a transaction, and the same information will be displayed throughout the transaction. This is a feature, not a bug, because it allows you to perform several queries on the statistics and correlate the results without worrying that the numbers are changing underneath you. But if you want to see new results with each query, be sure to do the queries outside any transaction block.

Table 25-1. Standard Statistics Views

View Name	Description
`pg_stat_activity`	One row per server process, showing database OID, database name, process ID, user OID, user name, current query, query's waiting status, time at which the current query began execution, time at which the process was started, and client's address and port number. The columns that report data on the current query are available unless the parameter `stats_command_string` has been turned off. Furthermore, these columns are only visible if the user examining the view is a superuser or the same as the user owning the process being reported on.
`pg_stat_database`	One row per database, showing database OID, database name, number of active server processes connected to that database, number of transactions committed and rolled back in that database, total disk blocks read, and total buffer hits (i.e., block read requests avoided by finding the block already in buffer cache).

View Name	Description
pg_stat_all_tables	For each table in the current database (including TOAST tables), the table OID, schema and table name, number of sequential scans initiated, number of live rows fetched by sequential scans, number of index scans initiated (over all indexes belonging to the table), number of live rows fetched by index scans, numbers of row insertions, updates, and deletions, the last time the table was vacuumed manually, the last time it was vacuumed by the autovacuum daemon, the last time it was analyzed manually, and the last time it was analyzed by the autovacuum daemon.
pg_stat_sys_tables	Same as pg_stat_all_tables, except that only system tables are shown.
pg_stat_user_tables	Same as pg_stat_all_tables, except that only user tables are shown.
pg_stat_all_indexes	For each index in the current database, the table and index OID, schema, table and index name, number of index scans initiated on that index, number of index entries returned by index scans, and number of live table rows fetched by simple index scans using that index.
pg_stat_sys_indexes	Same as pg_stat_all_indexes, except that only indexes on system tables are shown.
pg_stat_user_indexes	Same as pg_stat_all_indexes, except that only indexes on user tables are shown.
pg_statio_all_tables	For each table in the current database (including TOAST tables), the table OID, schema and table name, number of disk blocks read from that table, number of buffer hits, numbers of disk blocks read and buffer hits in all indexes of that table, numbers of disk blocks read and buffer hits from that table's auxiliary TOAST table (if any), and numbers of disk blocks read and buffer hits for the TOAST table's index.
pg_statio_sys_tables	Same as pg_statio_all_tables, except that only system tables are shown.
pg_statio_user_tables	Same as pg_statio_all_tables, except that only user tables are shown.
pg_statio_all_indexes	For each index in the current database, the table and index OID, schema, table and index name, numbers of disk blocks read and buffer hits in that index.

View Name	Description
pg_statio_sys_indexes	Same as pg_statio_all_indexes, except that only indexes on system tables are shown.
pg_statio_user_indexes	Same as pg_statio_all_indexes, except that only indexes on user tables are shown.
pg_statio_all_sequences	For each sequence object in the current database, the sequence OID, schema and sequence name, numbers of disk blocks read and buffer hits in that sequence.
pg_statio_sys_sequences	Same as pg_statio_all_sequences, except that only system sequences are shown. (Presently, no system sequences are defined, so this view is always empty.)
pg_statio_user_sequences	Same as pg_statio_all_sequences, except that only user sequences are shown.

The per-index statistics are particularly useful to determine which indexes are being used and how effective they are.

Beginning in PostgreSQL 8.1, indexes can be used either directly or via "bitmap scans". In a bitmap scan the output of several indexes can be combined via AND or OR rules; so it is difficult to associate individual heap row fetches with specific indexes when a bitmap scan is used. Therefore, a bitmap scan increments the pg_stat_all_indexes.idx_tup_read count(s) for the index(es) it uses, and it increments the pg_stat_all_tables.idx_tup_fetch count for the table, but it does not affect pg_stat_all_indexes.idx_tup_fetch.

> **Note:** Before PostgreSQL 8.1, the idx_tup_read and idx_tup_fetch counts were essentially always equal. Now they can be different even without considering bitmap scans, because idx_tup_read counts index entries retrieved from the index while idx_tup_fetch counts live rows fetched from the table; the latter will be less if any dead or not-yet-committed rows are fetched using the index.

The pg_statio_ views are primarily useful to determine the effectiveness of the buffer cache. When the number of actual disk reads is much smaller than the number of buffer hits, then the cache is satisfying most read requests without invoking a kernel call. However, these statistics do not give the entire story: due to the way in which PostgreSQL handles disk I/O, data that is not in the PostgreSQL buffer cache may still reside in the kernel's I/O cache, and may therefore still be fetched without requiring a physical read. Users interested in obtaining more detailed information on PostgreSQL I/O behavior are advised to use the PostgreSQL statistics collector in combination with operating system utilities that allow insight into the kernel's handling of I/O.

Other ways of looking at the statistics can be set up by writing queries that use the same underlying statistics access functions as these standard views do. These functions are listed in Table 25-2. The per-database access functions take a database OID as argument to identify which database to report on. The per-table and per-index functions take a table or index OID. (Note that only tables and indexes in the current database can be seen with these functions.) The per-server-process access functions take a server process number, which ranges from one to the number of currently active server processes.

Table 25-2. Statistics Access Functions

Function	Return Type	Description
pg_stat_get_db_numbackends(oid)	integer	Number of active server processes for database
pg_stat_get_db_xact_commit(oid)	bigint	Transactions committed in database
pg_stat_get_db_xact_rollback(oid)	bigint	Transactions rolled back in database
pg_stat_get_db_blocks_fetched(oid)	bigint	Number of disk block fetch requests for database
pg_stat_get_db_blocks_hit(oid)	bigint	Number of disk block fetch requests found in cache for database
pg_stat_get_numscans(oid)	bigint	Number of sequential scans done when argument is a table, or number of index scans done when argument is an index
pg_stat_get_tuples_returned(oid)	bigint	Number of rows read by sequential scans when argument is a table, or number of index entries returned when argument is an index
pg_stat_get_tuples_fetched(oid)	bigint	Number of table rows fetched by bitmap scans when argument is a table, or table rows fetched by simple index scans using the index when argument is an index
pg_stat_get_tuples_inserted(oid)	bigint	Number of rows inserted into table
pg_stat_get_tuples_updated(oid)	bigint	Number of rows updated in table
pg_stat_get_tuples_deleted(oid)	bigint	Number of rows deleted from table
pg_stat_get_blocks_fetched(oid)	bigint	Number of disk block fetch requests for table or index
pg_stat_get_blocks_hit(oid)	bigint	Number of disk block requests found in cache for table or index
pg_stat_get_last_vacuum_time(oid)	timestamptz	Time of the last vacuum initiated by the user on this table
pg_stat_get_last_autovacuum_time(oid)	timestamptz	Time of the last vacuum initiated by the autovacuum daemon on this table

Function	Return Type	Description
pg_stat_get_last_analyze_time(oid)	timestamptz	Time of the last analyze initiated by the user on this table
pg_stat_get_last_autoanalyze_time(oid)	timestamptz	Time of the last analyze initiated by the autovacuum daemon on this table
pg_stat_get_backend_idset()	setof integer	Set of currently active server process numbers (from 1 to the number of active server processes). See usage example in the text
pg_backend_pid()	integer	Process ID of the server process attached to the current session
pg_stat_get_backend_pid(integer)	integer	Process ID of the given server process
pg_stat_get_backend_dbid(integer)	oid	Database ID of the given server process
pg_stat_get_backend_userid(integer)	oid	User ID of the given server process
pg_stat_get_backend_activity(integer)	text	Active command of the given server process, but only if the current user is a superuser or the same user as that of the session being queried (and stats_command_string is on)
pg_stat_get_backend_waiting(integer)	boolean	True if the given server process is waiting for a lock, but only if the current user is a superuser or the same user as that of the session being queried (and stats_command_string is on)
pg_stat_get_backend_activity_start(integer)	timestamp with time zone	The time at which the given server process' currently executing query was started, but only if the current user is a superuser or the same user as that of the session being queried (and stats_command_string is on)

Function	Return Type	Description
pg_stat_get_backend_start(integer)	timestamp with time zone	The time at which the given server process was started, or null if the current user is not a superuser nor the same user as that of the session being queried
pg_stat_get_backend_client_addr(integer)	inet	The IP address of the client connected to the given server process. Null if the connection is over a Unix domain socket. Also null if the current user is not a superuser nor the same user as that of the session being queried
pg_stat_get_backend_client_port(integer)	integer	The IP port number of the client connected to the given server process. -1 if the connection is over a Unix domain socket. Null if the current user is not a superuser nor the same user as that of the session being queried
pg_stat_reset()	boolean	Reset all block-level and row-level statistics to zero

Note: `blocks_fetched` minus `blocks_hit` gives the number of kernel `read()` calls issued for the table, index, or database; but the actual number of physical reads is usually lower due to kernel-level buffering.

The function `pg_stat_get_backend_idset` provides a convenient way to generate one row for each active server process. For example, to show the PIDs and current queries of all server processes:

```
SELECT pg_stat_get_backend_pid(s.backendid) AS procpid,
       pg_stat_get_backend_activity(s.backendid) AS current_query
    FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

25.3. Viewing Locks

Another useful tool for monitoring database activity is the `pg_locks` system table. It allows the database administrator to view information about the outstanding locks in the lock manager. For example, this capability can be used to:

- View all the locks currently outstanding, all the locks on relations in a particular database, all the locks

on a particular relation, or all the locks held by a particular PostgreSQL session.

- Determine the relation in the current database with the most ungranted locks (which might be a source of contention among database clients).
- Determine the effect of lock contention on overall database performance, as well as the extent to which contention varies with overall database traffic.

Details of the `pg_locks` view appear in Section 43.39. For more information on locking and managing concurrency with PostgreSQL, refer to Chapter 12.

25.4. Dynamic Tracing

PostgreSQL provides facilities to support dynamic tracing of the database server. This allows an external utility to be called at specific points in the code and thereby trace execution. Currently, this facility is primarily intended for use by database developers, as it requires substantial familiarity with the code.

A number of trace points, often called probes, are already inserted into the source code. By default these probes are disabled, and the user needs to explicitly tell the configure script to make the probes available in PostgreSQL.

Currently, only the DTrace utility is supported, which is only available on Solaris Express and Solaris 10+. It is expected that DTrace will be available in the future on FreeBSD and Mac OS X. Supporting other dynamic tracing utilities is theoretically possible by changing the definitions for the `PG_TRACE` macros in `src/include/pg_trace.h`.

25.4.1. Compiling for Dynamic Tracing

By default, trace points are disabled, so you will need to explicitly tell the configure script to make the probes available in PostgreSQL. To include DTrace support specify `--enable-dtrace` to configure. See Section 14.5 for further information.

25.4.2. Built-in Trace Points

A few standard trace points are provided in the source code (of course, more can be added as needed for a particular problem). These are shown in Table 25-3.

Table 25-3. Built-in Trace Points

Name	Parameters	Overview
transaction__start	(int transactionId)	The start of a new transaction.
transaction__commit	(int transactionId)	The successful completion of a transaction.
transaction__abort	(int transactionId)	The unsuccessful completion of a transaction.
lwlock__acquire	(int lockid, int mode)	An LWLock has been acquired.
lwlock__release	(int lockid, int mode)	An LWLock has been released.

Name	Parameters	Overview
lwlock__startwait	(int lockid, int mode)	An LWLock was not immediately available and a backend has begun to wait for the lock to become available.
lwlock__endwait	(int lockid, int mode)	A backend has been released from its wait for an LWLock.
lwlock__condacquire	(int lockid, int mode)	An LWLock was successfully acquired when the caller specified no waiting.
lwlock__condacquire__fail	(int lockid, int mode)	An LWLock was not successfully acquired when the caller specified no waiting.
lock__startwait	(int locktag_field2, int lockmode)	A request for a heavyweight lock (lmgr lock) has begun to wait because the lock is not available.
lock__endwait	(int locktag_field2, int lockmode)	A request for a heavyweight lock (lmgr lock) has finished waiting (i.e., has acquired the lock).

25.4.3. Using Trace Points

The example below shows a DTrace script for analyzing transaction counts on the system, as an alternative to snapshotting `pg_stat_database` before and after a performance test.

```
#!/usr/sbin/dtrace -qs

postgresql$1:::transaction-start
{
      @start["Start"] = count();
      self->ts  = timestamp;
}

postgresql$1:::transaction-abort
{
      @abort["Abort"] = count();
}

postgresql$1:::transaction-commit
/self->ts/
{
      @commit["Commit"] = count();
      @time["Total time (ns)"] = sum(timestamp - self->ts);
      self->ts=0;
}
```

Note how the double underline in trace point names needs to be replaced by a hyphen when using D script. When executed, the example D script gives output such as:

```
# ./txn_count.d `pgrep -n postgres`
^C

Start                                          71
Commit                                         70
Total time (ns)                        2312105013
```

You should remember that trace programs need to be carefully written and debugged prior to their use, otherwise the trace information collected may be meaningless. In most cases where problems are found it is the instrumentation that is at fault, not the underlying system. When discussing information found using dynamic tracing, be sure to enclose the script used to allow that too to be checked and discussed.

25.4.4. Defining Trace Points

New trace points can be defined within the code wherever the developer desires, though this will require a recompilation.

A trace point can be inserted by using one of the trace macros. These are chosen according to how many variables will be made available for inspection at that trace point. Tracing the occurrence of an event can be achieved with a single line, using just the trace point name, e.g.

```
PG_TRACE (my__new__trace__point);
```

More complex trace points can be provided with one or more variables for inspection by the dynamic tracing utility by using the `PG_TRACE`*n* macro that corresponds to the number of parameters after the trace point name:

```
PG_TRACE3 (my__complex__event, varX, varY, varZ);
```

The definition of the transaction__start trace point is shown below:

```
static void
StartTransaction(void)
{
    ...

    /*
     * generate a new transaction id
     */
    s->transactionId = GetNewTransactionId(false);

    XactLockTableInsert(s->transactionId);

    PG_TRACE1(transaction__start, s->transactionId);

    ...
}
```

Note how the transaction ID is made available to the dynamic tracing utility.

The dynamic tracing utility may require you to further define these trace points. For example, DTrace requires you to add new probes to the file `src/backend/utils/probes.d` as shown here:

```
provider postgresql {
      ...
      probe transaction__start(int);
      ...
 };
```

You should take care that the data types specified for the probe arguments match the datatypes of the variables used in the `PG_TRACE` macro. This is not checked at compile time. You can check that your newly added trace point is available by recompiling, then running the new binary, and as root, executing a DTrace command such as:

```
dtrace -l -n transaction-start
```

Chapter 26. Monitoring Disk Usage

This chapter discusses how to monitor the disk usage of a PostgreSQL database system.

26.1. Determining Disk Usage

Each table has a primary heap disk file where most of the data is stored. If the table has any columns with potentially-wide values, there is also a TOAST file associated with the table, which is used to store values too wide to fit comfortably in the main table (see Section 52.2). There will be one index on the TOAST table, if present. There may also be indexes associated with the base table. Each table and index is stored in a separate disk file — possibly more than one file, if the file would exceed one gigabyte. Naming conventions for these files are described in Section 52.1.

You can monitor disk space from three ways: using SQL functions listed in Table 9-48, using `VACUUM` information, and from the command line using the tools in `contrib/oid2name`. The SQL functions are the easiest to use and report information about tables, tables with indexes and long value storage (TOAST), databases, and tablespaces.

Using psql on a recently vacuumed or analyzed database, you can issue queries to see the disk usage of any table:

```
SELECT relfilenode, relpages FROM pg_class WHERE relname = 'customer';

 relfilenode | relpages
-------------+----------
       16806 |       60
(1 row)
```

Each page is typically 8 kilobytes. (Remember, `relpages` is only updated by `VACUUM`, `ANALYZE`, and a few DDL commands such as `CREATE INDEX`.) The `relfilenode` value is of interest if you want to examine the table's disk file directly.

To show the space used by TOAST tables, use a query like the following:

```
SELECT relname, relpages
    FROM pg_class,
         (SELECT reltoastrelid FROM pg_class
          WHERE relname = 'customer') ss
    WHERE oid = ss.reltoastrelid
       OR oid = (SELECT reltoastidxid FROM pg_class
                 WHERE oid = ss.reltoastrelid)
    ORDER BY relname;

       relname        | relpages
----------------------+----------
 pg_toast_16806       |        0
 pg_toast_16806_index |        1
```

You can easily display index sizes, too:

```
SELECT c2.relname, c2.relpages
    FROM pg_class c, pg_class c2, pg_index i
    WHERE c.relname = 'customer'
        AND c.oid = i.indrelid
        AND c2.oid = i.indexrelid
    ORDER BY c2.relname;

       relname        | relpages
----------------------+----------
 customer_id_indexdex |       26
```

It is easy to find your largest tables and indexes using this information:

```
SELECT relname, relpages FROM pg_class ORDER BY relpages DESC;

       relname        | relpages
----------------------+----------
 bigtable             |     3290
 customer             |     3144
```

You can also use `contrib/oid2name` to show disk usage. See `README.oid2name` in that directory for examples. It includes a script that shows disk usage for each database.

26.2. Disk Full Failure

The most important disk monitoring task of a database administrator is to make sure the disk doesn't grow full. A filled data disk will not result in data corruption, but it may well prevent useful activity from occurring. If the disk holding the WAL files grows full, database server panic and consequent shutdown may occur.

If you cannot free up additional space on the disk by deleting other things, you can move some of the database files to other file systems by making use of tablespaces. See Section 19.6 for more information about that.

> **Tip:** Some file systems perform badly when they are almost full, so do not wait until the disk is completely full to take action.

If your system supports per-user disk quotas, then the database will naturally be subject to whatever quota is placed on the user the server runs as. Exceeding the quota will have the same bad effects as running out of space entirely.

Chapter 27. Reliability and the Write-Ahead Log

This chapter explain how the Write-Ahead Log is used to obtain efficient, reliable operation.

27.1. Reliability

Reliability is an important property of any serious database system, and PostgreSQL does everything possible to guarantee reliable operation. One aspect of reliable operation is that all data recorded by a committed transaction should be stored in a nonvolatile area that is safe from power loss, operating system failure, and hardware failure (except failure of the nonvolatile area itself, of course). Successfully writing the data to the computer's permanent storage (disk drive or equivalent) ordinarily meets this requirement. In fact, even if a computer is fatally damaged, if the disk drives survive they can be moved to another computer with similar hardware and all committed transactions will remain intact.

While forcing data periodically to the disk platters might seem like a simple operation, it is not. Because disk drives are dramatically slower than main memory and CPUs, several layers of caching exist between the computer's main memory and the disk platters. First, there is the operating system's buffer cache, which caches frequently requested disk blocks and combines disk writes. Fortunately, all operating systems give applications a way to force writes from the buffer cache to disk, and PostgreSQL uses those features. (See the wal_sync_method parameter to adjust how this is done.)

Next, there may be a cache in the disk drive controller; this is particularly common on RAID controller cards. Some of these caches are *write-through*, meaning writes are passed along to the drive as soon as they arrive. Others are *write-back*, meaning data is passed on to the drive at some later time. Such caches can be a reliability hazard because the memory in the disk controller cache is volatile, and will lose its contents in a power failure. Better controller cards have *battery-backed* caches, meaning the card has a battery that maintains power to the cache in case of system power loss. After power is restored the data will be written to the disk drives.

And finally, most disk drives have caches. Some are write-through while some are write-back, and the same concerns about data loss exist for write-back drive caches as exist for disk controller caches. Consumer-grade IDE drives are particularly likely to contain write-back caches that will not survive a power failure.

When the operating system sends a write request to the disk hardware, there is little it can do to make sure the data has arrived at a truly non-volatile storage area. Rather, it is the administrator's responsibility to be sure that all storage components ensure data integrity. Avoid disk controllers that have non-battery-backed write caches. At the drive level, disable write-back caching if the drive cannot guarantee the data will be written before shutdown.

Another risk of data loss is posed by the disk platter write operations themselves. Disk platters are divided into sectors, commonly 512 bytes each. Every physical read or write operation processes a whole sector. When a write request arrives at the drive, it might be for 512 bytes, 1024 bytes, or 8192 bytes, and the process of writing could fail due to power loss at any time, meaning some of the 512-byte sectors were written, and others were not. To guard against such failures, PostgreSQL periodically writes full page images to permanent storage *before* modifying the actual page on disk. By doing this, during crash recovery PostgreSQL can restore partially-written pages. If you have a battery-backed disk controller or file-system software that prevents partial page writes (e.g., ReiserFS 4), you can turn off this page imaging by using the full_page_writes parameter.

27.2. Write-Ahead Logging (WAL)

Write-Ahead Logging (WAL) is a standard approach to transaction logging. Its detailed description may be found in most (if not all) books about transaction processing. Briefly, WAL's central concept is that changes to data files (where tables and indexes reside) must be written only after those changes have been logged, that is, when log records describing the changes have been flushed to permanent storage. If we follow this procedure, we do not need to flush data pages to disk on every transaction commit, because we know that in the event of a crash we will be able to recover the database using the log: any changes that have not been applied to the data pages can be redone from the log records. (This is roll-forward recovery, also known as REDO.)

A major benefit of using WAL is a significantly reduced number of disk writes, because only the log file needs to be flushed to disk at the time of transaction commit, rather than every data file changed by the transaction. In multiuser environments, commits of many transactions may be accomplished with a single `fsync` of the log file. Furthermore, the log file is written sequentially, and so the cost of syncing the log is much less than the cost of flushing the data pages. This is especially true for servers handling many small transactions touching different parts of the data store.

WAL also makes it possible to support on-line backup and point-in-time recovery, as described in Section 23.3. By archiving the WAL data we can support reverting to any time instant covered by the available WAL data: we simply install a prior physical backup of the database, and replay the WAL log just as far as the desired time. What's more, the physical backup doesn't have to be an instantaneous snapshot of the database state — if it is made over some period of time, then replaying the WAL log for that period will fix any internal inconsistencies.

27.3. WAL Configuration

There are several WAL-related configuration parameters that affect database performance. This section explains their use. Consult Chapter 17 for general information about setting server configuration parameters.

Checkpoints are points in the sequence of transactions at which it is guaranteed that the data files have been updated with all information written before the checkpoint. At checkpoint time, all dirty data pages are flushed to disk and a special checkpoint record is written to the log file. In the event of a crash, the crash recovery procedure looks at the latest checkpoint record to determine the point in the log (known as the redo record) from which it should start the REDO operation. Any changes made to data files before that point are known to be already on disk. Hence, after a checkpoint has been made, any log segments preceding the one containing the redo record are no longer needed and can be recycled or removed. (When WAL archiving is being done, the log segments must be archived before being recycled or removed.)

The server's background writer process will automatically perform a checkpoint every so often. A checkpoint is created every checkpoint_segments log segments, or every checkpoint_timeout seconds, whichever comes first. The default settings are 3 segments and 300 seconds respectively. It is also possible to force a checkpoint by using the SQL command `CHECKPOINT`.

Reducing `checkpoint_segments` and/or `checkpoint_timeout` causes checkpoints to be done more often. This allows faster after-crash recovery (since less work will need to be redone). However, one must balance this against the increased cost of flushing dirty data pages more often. If full_page_writes is set (as is the default), there is another factor to consider. To ensure data page consistency, the first modification

of a data page after each checkpoint results in logging the entire page content. In that case, a smaller checkpoint interval increases the volume of output to the WAL log, partially negating the goal of using a smaller interval, and in any case causing more disk I/O.

Checkpoints are fairly expensive, first because they require writing out all currently dirty buffers, and second because they result in extra subsequent WAL traffic as discussed above. It is therefore wise to set the checkpointing parameters high enough that checkpoints don't happen too often. As a simple sanity check on your checkpointing parameters, you can set the checkpoint_warning parameter. If checkpoints happen closer together than `checkpoint_warning` seconds, a message will be output to the server log recommending increasing `checkpoint_segments`. Occasional appearance of such a message is not cause for alarm, but if it appears often then the checkpoint control parameters should be increased. Bulk operations such as large `COPY` transfers may cause a number of such warnings to appear if you have not set `checkpoint_segments` high enough.

There will be at least one WAL segment file, and will normally not be more than 2 * `checkpoint_segments` + 1 files. Each segment file is normally 16 MB (though this size can be altered when building the server). You can use this to estimate space requirements for WAL. Ordinarily, when old log segment files are no longer needed, they are recycled (renamed to become the next segments in the numbered sequence). If, due to a short-term peak of log output rate, there are more than 2 * `checkpoint_segments` + 1 segment files, the unneeded segment files will be deleted instead of recycled until the system gets back under this limit.

There are two commonly used internal WAL functions: `LogInsert` and `LogFlush`. `LogInsert` is used to place a new record into the WAL buffers in shared memory. If there is no space for the new record, `LogInsert` will have to write (move to kernel cache) a few filled WAL buffers. This is undesirable because `LogInsert` is used on every database low level modification (for example, row insertion) at a time when an exclusive lock is held on affected data pages, so the operation needs to be as fast as possible. What is worse, writing WAL buffers may also force the creation of a new log segment, which takes even more time. Normally, WAL buffers should be written and flushed by a `LogFlush` request, which is made, for the most part, at transaction commit time to ensure that transaction records are flushed to permanent storage. On systems with high log output, `LogFlush` requests may not occur often enough to prevent `LogInsert` from having to do writes. On such systems one should increase the number of WAL buffers by modifying the configuration parameter wal_buffers. The default number of WAL buffers is 8. Increasing this value will correspondingly increase shared memory usage. When full_page_writes is set and the system is very busy, setting this value higher will help smooth response times during the period immediately following each checkpoint.

The commit_delay parameter defines for how many microseconds the server process will sleep after writing a commit record to the log with `LogInsert` but before performing a `LogFlush`. This delay allows other server processes to add their commit records to the log so as to have all of them flushed with a single log sync. No sleep will occur if fsync is not enabled, nor if fewer than commit_siblings other sessions are currently in active transactions; this avoids sleeping when it's unlikely that any other session will commit soon. Note that on most platforms, the resolution of a sleep request is ten milliseconds, so that any nonzero `commit_delay` setting between 1 and 10000 microseconds would have the same effect. Good values for these parameters are not yet clear; experimentation is encouraged.

The wal_sync_method parameter determines how PostgreSQL will ask the kernel to force WAL updates out to disk. All the options should be the same as far as reliability goes, but it's quite platform-specific which one will be the fastest. Note that this parameter is irrelevant if `fsync` has been turned off.

Enabling the wal_debug configuration parameter (provided that PostgreSQL has been compiled with sup-

port for it) will result in each `LogInsert` and `LogFlush` WAL call being logged to the server log. This option may be replaced by a more general mechanism in the future.

27.4. WAL Internals

WAL is automatically enabled; no action is required from the administrator except ensuring that the disk-space requirements for the WAL logs are met, and that any necessary tuning is done (see Section 27.3).

WAL logs are stored in the directory `pg_xlog` under the data directory, as a set of segment files, normally each 16 MB in size. Each segment is divided into pages, normally 8 kB each. The log record headers are described in `access/xlog.h`; the record content is dependent on the type of event that is being logged. Segment files are given ever-increasing numbers as names, starting at `000000010000000000000000`. The numbers do not wrap, at present, but it should take a very very long time to exhaust the available stock of numbers.

It is of advantage if the log is located on another disk than the main database files. This may be achieved by moving the directory `pg_xlog` to another location (while the server is shut down, of course) and creating a symbolic link from the original location in the main data directory to the new location.

The aim of WAL, to ensure that the log is written before database records are altered, may be subverted by disk drives that falsely report a successful write to the kernel, when in fact they have only cached the data and not yet stored it on the disk. A power failure in such a situation may still lead to irrecoverable data corruption. Administrators should try to ensure that disks holding PostgreSQL's WAL log files do not make such false reports.

After a checkpoint has been made and the log flushed, the checkpoint's position is saved in the file `pg_control`. Therefore, when recovery is to be done, the server first reads `pg_control` and then the checkpoint record; then it performs the REDO operation by scanning forward from the log position indicated in the checkpoint record. Because the entire content of data pages is saved in the log on the first page modification after a checkpoint, all pages changed since the checkpoint will be restored to a consistent state.

To deal with the case where `pg_control` is corrupted, we should support the possibility of scanning existing log segments in reverse order — newest to oldest — in order to find the latest checkpoint. This has not been implemented yet. `pg_control` is small enough (less than one disk page) that it is not subject to partial-write problems, and as of this writing there have been no reports of database failures due solely to inability to read `pg_control` itself. So while it is theoretically a weak spot, `pg_control` does not seem to be a problem in practice.

Chapter 28. Regression Tests

The regression tests are a comprehensive set of tests for the SQL implementation in PostgreSQL. They test standard SQL operations as well as the extended capabilities of PostgreSQL.

28.1. Running the Tests

The regression tests can be run against an already installed and running server, or using a temporary installation within the build tree. Furthermore, there is a "parallel" and a "sequential" mode for running the tests. The sequential method runs each test script in turn, whereas the parallel method starts up multiple server processes to run groups of tests in parallel. Parallel testing gives confidence that interprocess communication and locking are working correctly. For historical reasons, the sequential test is usually run against an existing installation and the parallel method against a temporary installation, but there are no technical reasons for this.

To run the regression tests after building but before installation, type

```
gmake check
```

in the top-level directory. (Or you can change to `src/test/regress` and run the command there.) This will first build several auxiliary files, such as some sample user-defined trigger functions, and then run the test driver script. At the end you should see something like

```
=======================
 All 100 tests passed.
=======================
```

or otherwise a note about which tests failed. See Section 28.2 below before assuming that a "failure" represents a serious problem.

Because this test method runs a temporary server, it will not work when you are the root user (since the server will not start as root). If you already did the build as root, you do not have to start all over. Instead, make the regression test directory writable by some other user, log in as that user, and restart the tests. For example

```
root# chmod -R a+w src/test/regress
root# chmod -R a+w contrib/spi
root# su - joeuser
joeuser$ cd top-level build directory
joeuser$ gmake check
```

(The only possible "security risk" here is that other users might be able to alter the regression test results behind your back. Use common sense when managing user permissions.)

Alternatively, run the tests after installation.

If you have configured PostgreSQL to install into a location where an older PostgreSQL installation already exists, and you perform `gmake check` before installing the new version, you may find that the tests fail because the new programs try to use the already-installed shared libraries. (Typical symptoms are complaints about undefined symbols.) If you wish to run the tests before overwriting the old installation,

you'll need to build with `configure --disable-rpath`. It is not recommended that you use this option for the final installation, however.

The parallel regression test starts quite a few processes under your user ID. Presently, the maximum concurrency is twenty parallel test scripts, which means forty processes: there's a server process and a psql process for each test script. So if your system enforces a per-user limit on the number of processes, make sure this limit is at least fifty or so, else you may get random-seeming failures in the parallel test. If you are not in a position to raise the limit, you can cut down the degree of parallelism by setting the `MAX_CONNECTIONS` parameter. For example,

```
gmake MAX_CONNECTIONS=10 check
```

runs no more than ten tests concurrently.

To run the tests after installation (see Chapter 14), initialize a data area and start the server, as explained in Chapter 16, then type

```
gmake installcheck
```

or for a parallel test

```
gmake installcheck-parallel
```

The tests will expect to contact the server at the local host and the default port number, unless directed otherwise by `PGHOST` and `PGPORT` environment variables.

The source distribution also contains regression tests for the optional procedural languages and for some of the `contrib` modules. At present, these tests can be used only against an already-installed server. To run the tests for all procedural languages that have been built and installed, change to the `src/pl` directory of the build tree and type

```
gmake installcheck
```

You can also do this in any of the subdirectories of `src/pl` to run tests for just one procedural language. To run the tests for all `contrib` modules that have them, change to the `contrib` directory of the build tree and type

```
gmake installcheck
```

The `contrib` modules must have been built and installed first. You can also do this in a subdirectory of `contrib` to run the tests for just one module.

28.2. Test Evaluation

Some properly installed and fully functional PostgreSQL installations can "fail" some of these regression tests due to platform-specific artifacts such as varying floating-point representation and message wording. The tests are currently evaluated using a simple `diff` comparison against the outputs generated on a reference system, so the results are sensitive to small system differences. When a test is reported as "failed", always examine the differences between expected and actual results; you may well find that the differences are not significant. Nonetheless, we still strive to maintain accurate reference files across all supported platforms, so it can be expected that all tests pass.

The actual outputs of the regression tests are in files in the `src/test/regress/results` directory. The test script uses `diff` to compare each output file against the reference outputs stored in the `src/test/regress/expected` directory. Any differences are saved for your inspection in `src/test/regress/regression.diffs`. (Or you can run `diff` yourself, if you prefer.)

If for some reason a particular platform generates a "failure" for a given test, but inspection of the output convinces you that the result is valid, you can add a new comparison file to silence the failure report in future test runs. See Section 28.3 for details.

28.2.1. Error message differences

Some of the regression tests involve intentional invalid input values. Error messages can come from either the PostgreSQL code or from the host platform system routines. In the latter case, the messages may vary between platforms, but should reflect similar information. These differences in messages will result in a "failed" regression test that can be validated by inspection.

28.2.2. Locale differences

If you run the tests against an already-installed server that was initialized with a collation-order locale other than C, then there may be differences due to sort order and follow-up failures. The regression test suite is set up to handle this problem by providing alternative result files that together are known to handle a large number of locales.

28.2.3. Date and time differences

Most of the date and time results are dependent on the time zone environment. The reference files are generated for time zone `PST8PDT` (Berkeley, California), and there will be apparent failures if the tests are not run with that time zone setting. The regression test driver sets environment variable `PGTZ` to `PST8PDT`, which normally ensures proper results.

28.2.4. Floating-point differences

Some of the tests involve computing 64-bit floating-point numbers (`double precision`) from table columns. Differences in results involving mathematical functions of `double precision` columns have been observed. The `float8` and `geometry` tests are particularly prone to small differences across platforms, or even with different compiler optimization options. Human eyeball comparison is needed to determine the real significance of these differences which are usually 10 places to the right of the decimal point.

Some systems display minus zero as `-0`, while others just show `0`.

Some systems signal errors from `pow()` and `exp()` differently from the mechanism expected by the current PostgreSQL code.

28.2.5. Row ordering differences

You might see differences in which the same rows are output in a different order than what appears in the expected file. In most cases this is not, strictly speaking, a bug. Most of the regression test scripts are not so pedantic as to use an `ORDER BY` for every single `SELECT`, and so their result row orderings are not well-defined according to the letter of the SQL specification. In practice, since we are looking at the same queries being executed on the same data by the same software, we usually get the same result ordering on all platforms, and so the lack of `ORDER BY` isn't a problem. Some queries do exhibit cross-platform ordering differences, however. When testing against an already-installed server, ordering differences can also be caused by non-C locale settings or non-default parameter settings, such as custom values of `work_mem` or the planner cost parameters.

Therefore, if you see an ordering difference, it's not something to worry about, unless the query does have an `ORDER BY` that your result is violating. But please report it anyway, so that we can add an `ORDER BY` to that particular query and thereby eliminate the bogus "failure" in future releases.

You might wonder why we don't order all the regression test queries explicitly to get rid of this issue once and for all. The reason is that that would make the regression tests less useful, not more, since they'd tend to exercise query plan types that produce ordered results to the exclusion of those that don't.

28.2.6. Insufficient stack depth

If the `errors` test results in a server crash at the `select infinite_recurse()` command, it means that the platform's limit on process stack size is smaller than the max_stack_depth parameter indicates. This can be fixed by running the server under a higher stack size limit (4MB is recommended with the default value of `max_stack_depth`). If you are unable to do that, an alternative is to reduce the value of `max_stack_depth`.

28.2.7. The "random" test

The `random` test script is intended to produce random results. In rare cases, this causes the random regression test to fail. Typing

```
diff results/random.out expected/random.out
```

should produce only one or a few lines of differences. You need not worry unless the random test fails repeatedly.

28.3. Variant Comparison Files

Since some of the tests inherently produce environment-dependent results, we have provided ways to specify alternative "expected" result files. Each regression test can have several comparison files showing possible results on different platforms. There are two independent mechanisms for determining which comparison file is used for each test.

The first mechanism allows comparison files to be selected for specific platforms. There is a mapping file, `src/test/regress/resultmap`, that defines which comparison file to use for each platform. To eliminate bogus test "failures" for a particular platform, you first choose or make a variant result file, and then add a line to the `resultmap` file.

Each line in the mapping file is of the form

```
testname/platformpattern=comparisonfilename
```

The test name is just the name of the particular regression test module. The platform pattern is a pattern in the style of the Unix tool `expr` (that is, a regular expression with an implicit `^` anchor at the start). It is matched against the platform name as printed by `config.guess`. The comparison file name is the base name of the substitute result comparison file.

For example: some systems interpret very small floating-point values as zero, rather than reporting an underflow error. This causes a few differences in the `float8` regression test. Therefore, we provide a variant comparison file, `float8-small-is-zero.out`, which includes the results to be expected on these systems. To silence the bogus "failure" message on OpenBSD platforms, `resultmap` includes

```
float8/i.86-.*-openbsd=float8-small-is-zero
```

which will trigger on any machine for which the output of `config.guess` matches `i.86-.*-openbsd`. Other lines in `resultmap` select the variant comparison file for other platforms where it's appropriate.

The second selection mechanism for variant comparison files is much more automatic: it simply uses the "best match" among several supplied comparison files. The regression test driver script considers both the standard comparison file for a test, *`testname`*`.out`, and variant files named *`testname_digit`*`.out` (where the *`digit`* is any single digit `0-9`). If any such file is an exact match, the test is considered to pass; otherwise, the one that generates the shortest diff is used to create the failure report. (If `resultmap` includes an entry for the particular test, then the base *`testname`* is the substitute name given in `resultmap`.)

For example, for the `char` test, the comparison file `char.out` contains results that are expected in the `C` and `POSIX` locales, while the file `char_1.out` contains results sorted as they appear in many other locales.

The best-match mechanism was devised to cope with locale-dependent results, but it can be used in any situation where the test results cannot be predicted easily from the platform name alone. A limitation of this mechanism is that the test driver cannot tell which variant is actually "correct" for the current environment; it will just pick the variant that seems to work best. Therefore it is safest to use this mechanism only for variant results that you are willing to consider equally valid in all contexts.

IV. Client Interfaces

This part describes the client programming interfaces distributed with PostgreSQL. Each of these chapters can be read independently. Note that there are many other programming interfaces for client programs that are distributed separately and contain their own documentation (Appendix H lists some of the more popular ones). Readers of this part should be familiar with using SQL commands to manipulate and query the database (see Part II) and of course with the programming language that the interface uses.

Chapter 29. libpq - C Library

libpq is the C application programmer's interface to PostgreSQL. libpq is a set of library functions that allow client programs to pass queries to the PostgreSQL backend server and to receive the results of these queries.

libpq is also the underlying engine for several other PostgreSQL application interfaces, including those written for C++, Perl, Python, Tcl and ECPG. So some aspects of libpq's behavior will be important to you if you use one of those packages. In particular, Section 29.12, Section 29.13 and Section 29.16 describe behavior that is visible to the user of any application that uses libpq.

Some short programs are included at the end of this chapter (Section 29.19) to show how to write programs that use libpq. There are also several complete examples of libpq applications in the directory `src/test/examples` in the source code distribution.

Client programs that use libpq must include the header file `libpq-fe.h` and must link with the libpq library.

29.1. Database Connection Control Functions

The following functions deal with making a connection to a PostgreSQL backend server. An application program can have several backend connections open at one time. (One reason to do that is to access more than one database.) Each connection is represented by a `PGconn` object, which is obtained from the function `PQconnectdb` or `PQsetdbLogin`. Note that these functions will always return a non-null object pointer, unless perhaps there is too little memory even to allocate the `PGconn` object. The `PQstatus` function should be called to check whether a connection was successfully made before queries are sent via the connection object.

`PQconnectdb`

Makes a new connection to the database server.

```
PGconn *PQconnectdb(const char *conninfo);
```

This function opens a new database connection using the parameters taken from the string `conninfo`. Unlike `PQsetdbLogin` below, the parameter set can be extended without changing the function signature, so use of this function (or its nonblocking analogues `PQconnectStart` and `PQconnectPoll`) is preferred for new application programming.

The passed string can be empty to use all default parameters, or it can contain one or more parameter settings separated by whitespace. Each parameter setting is in the form `keyword = value`. Spaces around the equal sign are optional. To write an empty value or a value containing spaces, surround it with single quotes, e.g., `keyword = 'a value'`. Single quotes and backslashes within the value must be escaped with a backslash, i.e., `\'` and `\\`.

The currently recognized parameter key words are:

`host`

Name of host to connect to. If this begins with a slash, it specifies Unix-domain communication rather than TCP/IP communication; the value is the name of the directory in which the socket

file is stored. The default behavior when `host` is not specified is to connect to a Unix-domain socket in `/tmp` (or whatever socket directory was specified when PostgreSQL was built). On machines without Unix-domain sockets, the default is to connect to `localhost`.

`hostaddr`

Numeric IP address of host to connect to. This should be in the standard IPv4 address format, e.g., `172.28.40.9`. If your machine supports IPv6, you can also use those addresses. TCP/IP communication is always used when a nonempty string is specified for this parameter.

Using `hostaddr` instead of `host` allows the application to avoid a host name look-up, which may be important in applications with time constraints. However, Kerberos authentication requires the host name. The following therefore applies: If `host` is specified without `hostaddr`, a host name lookup occurs. If `hostaddr` is specified without `host`, the value for `hostaddr` gives the remote address. When Kerberos is used, a reverse name query occurs to obtain the host name for Kerberos. If both `host` and `hostaddr` are specified, the value for `hostaddr` gives the remote address; the value for `host` is ignored, unless Kerberos is used, in which case that value is used for Kerberos authentication. (Note that authentication is likely to fail if libpq is passed a host name that is not the name of the machine at `hostaddr`.) Also, `host` rather than `hostaddr` is used to identify the connection in `~/.pgpass` (see Section 29.13).

Without either a host name or host address, libpq will connect using a local Unix-domain socket; or on machines without Unix-domain sockets, it will attempt to connect to `localhost`.

`port`

Port number to connect to at the server host, or socket file name extension for Unix-domain connections.

`dbname`

The database name. Defaults to be the same as the user name.

`user`

PostgreSQL user name to connect as. Defaults to be the same as the operating system name of the user running the application.

`password`

Password to be used if the server demands password authentication.

`connect_timeout`

Maximum wait for connection, in seconds (write as a decimal integer string). Zero or not specified means wait indefinitely. It is not recommended to use a timeout of less than 2 seconds.

`options`

Command-line options to be sent to the server.

`tty`

Ignored (formerly, this specified where to send server debug output).

`sslmode`

This option determines whether or with what priority an SSL connection will be negotiated with the server. There are four modes: `disable` will attempt only an unencrypted SSL connection;

`allow` will negotiate, trying first a non-SSL connection, then if that fails, trying an SSL connection; `prefer` (the default) will negotiate, trying first an SSL connection, then if that fails, trying a regular non-SSL connection; `require` will try only an SSL connection.

If PostgreSQL is compiled without SSL support, using option `require` will cause an error, while options `allow` and `prefer` will be accepted but libpq will not in fact attempt an SSL connection.

`requiressl`

This option is deprecated in favor of the `sslmode` setting.

If set to 1, an SSL connection to the server is required (this is equivalent to `sslmode require`). libpq will then refuse to connect if the server does not accept an SSL connection. If set to 0 (default), libpq will negotiate the connection type with the server (equivalent to `sslmode prefer`). This option is only available if PostgreSQL is compiled with SSL support.

`krbsrvname`

Kerberos service name to use when authenticating with Kerberos 5. This must match the service name specified in the server configuration for Kerberos authentication to succeed. (See also Section 20.2.3.)

`service`

Service name to use for additional parameters. It specifies a service name in `pg_service.conf` that holds additional connection parameters. This allows applications to specify only a service name so connection parameters can be centrally maintained. See Section 29.14.

If any parameter is unspecified, then the corresponding environment variable (see Section 29.12) is checked. If the environment variable is not set either, then the indicated built-in defaults are used.

`PQsetdbLogin`

Makes a new connection to the database server.

```
PGconn *PQsetdbLogin(const char *pghost,
                     const char *pgport,
                     const char *pgoptions,
                     const char *pgtty,
                     const char *dbName,
                     const char *login,
                     const char *pwd);
```

This is the predecessor of `PQconnectdb` with a fixed set of parameters. It has the same functionality except that the missing parameters will always take on default values. Write `NULL` or an empty string for any one of the fixed parameters that is to be defaulted.

`PQsetdb`

Makes a new connection to the database server.

```
PGconn *PQsetdb(char *pghost,
                char *pgport,
                char *pgoptions,
                char *pgtty,
                char *dbName);
```

This is a macro that calls PQsetdbLogin with null pointers for the login and pwd parameters. It is provided for backward compatibility with very old programs.

PQconnectStart
PQconnectPoll

Make a connection to the database server in a nonblocking manner.

```
PGconn *PQconnectStart(const char *conninfo);
```

```
PostgresPollingStatusType PQconnectPoll(PGconn *conn);
```

These two functions are used to open a connection to a database server such that your application's thread of execution is not blocked on remote I/O whilst doing so. The point of this approach is that the waits for I/O to complete can occur in the application's main loop, rather than down inside PQconnectdb, and so the application can manage this operation in parallel with other activities.

The database connection is made using the parameters taken from the string conninfo, passed to PQconnectStart. This string is in the same format as described above for PQconnectdb.

Neither PQconnectStart nor PQconnectPoll will block, so long as a number of restrictions are met:

- The hostaddr and host parameters are used appropriately to ensure that name and reverse name queries are not made. See the documentation of these parameters under PQconnectdb above for details.
- If you call PQtrace, ensure that the stream object into which you trace will not block.
- You ensure that the socket is in the appropriate state before calling PQconnectPoll, as described below.

To begin a nonblocking connection request, call conn = PQconnectStart("*connection_info_string*"). If conn is null, then libpq has been unable to allocate a new PGconn structure. Otherwise, a valid PGconn pointer is returned (though not yet representing a valid connection to the database). On return from PQconnectStart, call status = PQstatus(conn). If status equals CONNECTION_BAD, PQconnectStart has failed.

If PQconnectStart succeeds, the next stage is to poll libpq so that it may proceed with the connection sequence. Use PQsocket(conn) to obtain the descriptor of the socket underlying the database connection. Loop thus: If PQconnectPoll(conn) last returned PGRES_POLLING_READING, wait until the socket is ready to read (as indicated by select(), poll(), or similar system function). Then call PQconnectPoll(conn) again. Conversely, if PQconnectPoll(conn) last returned PGRES_POLLING_WRITING, wait until the socket is ready to write, then call PQconnectPoll(conn) again. If you have yet to call PQconnectPoll, i.e., just after the call to PQconnectStart, behave as if it last returned PGRES_POLLING_WRITING. Continue this loop until PQconnectPoll(conn) returns PGRES_POLLING_FAILED, indicating the connection procedure has failed, or PGRES_POLLING_OK, indicating the connection has been successfully made.

At any time during connection, the status of the connection may be checked by calling PQstatus. If this gives CONNECTION_BAD, then the connection procedure has failed; if it gives CONNECTION_OK, then the connection is ready. Both of these states are equally detectable from the return value of

PQconnectPoll, described above. Other states may also occur during (and only during) an asynchronous connection procedure. These indicate the current stage of the connection procedure and may be useful to provide feedback to the user for example. These statuses are:

CONNECTION_STARTED

Waiting for connection to be made.

CONNECTION_MADE

Connection OK; waiting to send.

CONNECTION_AWAITING_RESPONSE

Waiting for a response from the server.

CONNECTION_AUTH_OK

Received authentication; waiting for backend start-up to finish.

CONNECTION_SSL_STARTUP

Negotiating SSL encryption.

CONNECTION_SETENV

Negotiating environment-driven parameter settings.

Note that, although these constants will remain (in order to maintain compatibility), an application should never rely upon these occurring in a particular order, or at all, or on the status always being one of these documented values. An application might do something like this:

```
switch(PQstatus(conn))
{
    case CONNECTION_STARTED:
        feedback = "Connecting...";
        break;

    case CONNECTION_MADE:
        feedback = "Connected to server...";
        break;
.
.
.
    default:
        feedback = "Connecting...";
}
```

The connect_timeout connection parameter is ignored when using PQconnectPoll; it is the application's responsibility to decide whether an excessive amount of time has elapsed. Otherwise, PQconnectStart followed by a PQconnectPoll loop is equivalent to PQconnectdb.

Note that if PQconnectStart returns a non-null pointer, you must call PQfinish when you are finished with it, in order to dispose of the structure and any associated memory blocks. This must be done even if the connection attempt fails or is abandoned.

`PQconndefaults`

Returns the default connection options.

```
PQconninfoOption *PQconndefaults(void);

typedef struct
{
    char   *keyword;   /* The keyword of the option */
    char   *envvar;    /* Fallback environment variable name */
    char   *compiled;  /* Fallback compiled in default value */
    char   *val;       /* Option's current value, or NULL */
    char   *label;     /* Label for field in connect dialog */
    char   *dispchar;  /* Character to display for this field
                          in a connect dialog. Values are:
                          ""        Display entered value as is
                          "*"       Password field - hide value
                          "D"       Debug option - don't show by default */
    int     dispsize;  /* Field size in characters for dialog */
} PQconninfoOption;
```

Returns a connection options array. This may be used to determine all possible `PQconnectdb` options and their current default values. The return value points to an array of `PQconninfoOption` structures, which ends with an entry having a null `keyword` pointer. The null pointer is returned if memory could not be allocated. Note that the current default values (`val` fields) will depend on environment variables and other context. Callers must treat the connection options data as read-only.

After processing the options array, free it by passing it to `PQconninfoFree`. If this is not done, a small amount of memory is leaked for each call to `PQconndefaults`.

`PQfinish`

Closes the connection to the server. Also frees memory used by the `PGconn` object.

```
void PQfinish(PGconn *conn);
```

Note that even if the server connection attempt fails (as indicated by `PQstatus`), the application should call `PQfinish` to free the memory used by the `PGconn` object. The `PGconn` pointer must not be used again after `PQfinish` has been called.

`PQreset`

Resets the communication channel to the server.

```
void PQreset(PGconn *conn);
```

This function will close the connection to the server and attempt to reestablish a new connection to the same server, using all the same parameters previously used. This may be useful for error recovery if a working connection is lost.

`PQresetStart`
`PQresetPoll`

Reset the communication channel to the server, in a nonblocking manner.

```
int PQresetStart(PGconn *conn);

PostgresPollingStatusType PQresetPoll(PGconn *conn);
```

These functions will close the connection to the server and attempt to reestablish a new connection to the same server, using all the same parameters previously used. This may be useful for error recovery if a working connection is lost. They differ from `PQreset` (above) in that they act in a nonblocking manner. These functions suffer from the same restrictions as `PQconnectStart` and `PQconnectPoll`.

To initiate a connection reset, call `PQresetStart`. If it returns 0, the reset has failed. If it returns 1, poll the reset using `PQresetPoll` in exactly the same way as you would create the connection using `PQconnectPoll`.

29.2. Connection Status Functions

These functions may be used to interrogate the status of an existing database connection object.

> **Tip:** libpq application programmers should be careful to maintain the `PGconn` abstraction. Use the accessor functions described below to get at the contents of `PGconn`. Reference to internal `PGconn` fields using `libpq-int.h` is not recommended because they are subject to change in the future.

The following functions return parameter values established at connection. These values are fixed for the life of the `PGconn` object.

`PQdb`

Returns the database name of the connection.

```
char *PQdb(const PGconn *conn);
```

`PQuser`

Returns the user name of the connection.

```
char *PQuser(const PGconn *conn);
```

`PQpass`

Returns the password of the connection.

```
char *PQpass(const PGconn *conn);
```

`PQhost`

Returns the server host name of the connection.

```
char *PQhost(const PGconn *conn);
```

`PQport`

Returns the port of the connection.

```
char *PQport(const PGconn *conn);
```

PQtty

Returns the debug TTY of the connection. (This is obsolete, since the server no longer pays attention to the TTY setting, but the function remains for backwards compatibility.)

```
char *PQtty(const PGconn *conn);
```

PQoptions

Returns the command-line options passed in the connection request.

```
char *PQoptions(const PGconn *conn);
```

The following functions return status data that can change as operations are executed on the PGconn object.

PQstatus

Returns the status of the connection.

```
ConnStatusType PQstatus(const PGconn *conn);
```

The status can be one of a number of values. However, only two of these are seen outside of an asynchronous connection procedure: CONNECTION_OK and CONNECTION_BAD. A good connection to the database has the status CONNECTION_OK. A failed connection attempt is signaled by status CONNECTION_BAD. Ordinarily, an OK status will remain so until PQfinish, but a communications failure might result in the status changing to CONNECTION_BAD prematurely. In that case the application could try to recover by calling PQreset.

See the entry for PQconnectStart and PQconnectPoll with regards to other status codes that might be seen.

PQtransactionStatus

Returns the current in-transaction status of the server.

```
PGTransactionStatusType PQtransactionStatus(const PGconn *conn);
```

The status can be PQTRANS_IDLE (currently idle), PQTRANS_ACTIVE (a command is in progress), PQTRANS_INTRANS (idle, in a valid transaction block), or PQTRANS_INERROR (idle, in a failed transaction block). PQTRANS_UNKNOWN is reported if the connection is bad. PQTRANS_ACTIVE is reported only when a query has been sent to the server and not yet completed.

Caution

PQtransactionStatus will give incorrect results when using a PostgreSQL 7.3 server that has the parameter autocommit set to off. The server-side autocommit feature has been deprecated and does not exist in later server versions.

PQparameterStatus

Looks up a current parameter setting of the server.

```
const char *PQparameterStatus(const PGconn *conn, const char *paramName);
```

Certain parameter values are reported by the server automatically at connection startup or whenever their values change. PQparameterStatus can be used to interrogate these settings. It returns the current value of a parameter if known, or NULL if the parameter is not known.

Parameters reported as of the current release include server_version, server_encoding, client_encoding, is_superuser, session_authorization, DateStyle, TimeZone, integer_datetimes, and standard_conforming_strings. (server_encoding, TimeZone, and integer_datetimes were not reported by releases before 8.0; standard_conforming_strings was not reported by releases before 8.1.) Note that server_version, server_encoding and integer_datetimes cannot change after startup.

Pre-3.0-protocol servers do not report parameter settings, but libpq includes logic to obtain values for server_version and client_encoding anyway. Applications are encouraged to use PQparameterStatus rather than *ad hoc* code to determine these values. (Beware however that on a pre-3.0 connection, changing client_encoding via SET after connection startup will not be reflected by PQparameterStatus.) For server_version, see also PQserverVersion, which returns the information in a numeric form that is much easier to compare against.

If no value for standard_conforming_strings is reported, applications may assume it is off, that is, backslashes are treated as escapes in string literals. Also, the presence of this parameter may be taken as an indication that the escape string syntax (E'...') is accepted.

Although the returned pointer is declared const, it in fact points to mutable storage associated with the PGconn structure. It is unwise to assume the pointer will remain valid across queries.

PQprotocolVersion

Interrogates the frontend/backend protocol being used.

```
int PQprotocolVersion(const PGconn *conn);
```

Applications may wish to use this to determine whether certain features are supported. Currently, the possible values are 2 (2.0 protocol), 3 (3.0 protocol), or zero (connection bad). This will not change after connection startup is complete, but it could theoretically change during a connection reset. The 3.0 protocol will normally be used when communicating with PostgreSQL 7.4 or later servers; pre-7.4 servers support only protocol 2.0. (Protocol 1.0 is obsolete and not supported by libpq.)

PQserverVersion

Returns an integer representing the backend version.

```
int PQserverVersion(const PGconn *conn);
```

Applications may use this to determine the version of the database server they are connected to. The number is formed by converting the major, minor, and revision numbers into two-decimal-digit numbers and appending them together. For example, version 8.1.5 will be returned as 80105, and version 8.2 will be returned as 80200 (leading zeroes are not shown). Zero is returned if the connection is bad.

PQerrorMessage

Returns the error message most recently generated by an operation on the connection.

```
char *PQerrorMessage(const PGconn *conn);
```

Nearly all libpq functions will set a message for PQerrorMessage if they fail. Note that by libpq convention, a nonempty PQerrorMessage result will include a trailing newline. The caller should not free the result directly. It will be freed when the associated PGconn handle is passed to PQfinish. The result string should not be expected to remain the same across operations on the PGconn structure.

`PQsocket`

Obtains the file descriptor number of the connection socket to the server. A valid descriptor will be greater than or equal to 0; a result of -1 indicates that no server connection is currently open. (This will not change during normal operation, but could change during connection setup or reset.)

```
int PQsocket(const PGconn *conn);
```

`PQbackendPID`

Returns the process ID (PID) of the backend server process handling this connection.

```
int PQbackendPID(const PGconn *conn);
```

The backend PID is useful for debugging purposes and for comparison to `NOTIFY` messages (which include the PID of the notifying backend process). Note that the PID belongs to a process executing on the database server host, not the local host!

`PQgetssl`

Returns the SSL structure used in the connection, or null if SSL is not in use.

```
SSL *PQgetssl(const PGconn *conn);
```

This structure can be used to verify encryption levels, check server certificates, and more. Refer to the OpenSSL documentation for information about this structure.

You must define `USE_SSL` in order to get the correct prototype for this function. Doing this will also automatically include `ssl.h` from OpenSSL.

29.3. Command Execution Functions

Once a connection to a database server has been successfully established, the functions described here are used to perform SQL queries and commands.

29.3.1. Main Functions

`PQexec`

Submits a command to the server and waits for the result.

```
PGresult *PQexec(PGconn *conn, const char *command);
```

Returns a `PGresult` pointer or possibly a null pointer. A non-null pointer will generally be returned except in out-of-memory conditions or serious errors such as inability to send the command to the server. If a null pointer is returned, it should be treated like a `PGRES_FATAL_ERROR` result. Use `PQerrorMessage` to get more information about such errors.

It is allowed to include multiple SQL commands (separated by semicolons) in the command string. Multiple queries sent in a single `PQexec` call are processed in a single transaction, unless there are explicit `BEGIN`/`COMMIT` commands included in the query string to divide it into multiple transactions. Note however that the returned `PGresult` structure describes only the result of the last command executed from the

string. Should one of the commands fail, processing of the string stops with it and the returned PGresult describes the error condition.

PQexecParams

Submits a command to the server and waits for the result, with the ability to pass parameters separately from the SQL command text.

```
PGresult *PQexecParams(PGconn *conn,
                       const char *command,
                       int nParams,
                       const Oid *paramTypes,
                       const char * const *paramValues,
                       const int *paramLengths,
                       const int *paramFormats,
                       int resultFormat);
```

PQexecParams is like PQexec, but offers additional functionality: parameter values can be specified separately from the command string proper, and query results can be requested in either text or binary format. PQexecParams is supported only in protocol 3.0 and later connections; it will fail when using protocol 2.0.

The function arguments are:

conn

The connection object to send the command through.

command

The SQL command string to be executed. If parameters are used, they are referred to in the command string as $1, $2, etc.

nParams

The number of parameters supplied; it is the length of the arrays paramTypes[], paramValues[], paramLengths[], and paramFormats[]. (The array pointers may be NULL when nParams is zero.)

paramTypes[]

Specifies, by OID, the data types to be assigned to the parameter symbols. If paramTypes is NULL, or any particular element in the array is zero, the server infers a data type for the parameter symbol in the same way it would do for an untyped literal string.

paramValues[]

Specifies the actual values of the parameters. A null pointer in this array means the corresponding parameter is null; otherwise the pointer points to a zero-terminated text string (for text format) or binary data in the format expected by the server (for binary format).

paramLengths[]

Specifies the actual data lengths of binary-format parameters. It is ignored for null parameters and text-format parameters. The array pointer may be null when there are no binary parameters.

`paramFormats[]`

Specifies whether parameters are text (put a zero in the array entry for the corresponding parameter) or binary (put a one in the array entry for the corresponding parameter). If the array pointer is null then all parameters are presumed to be text strings.

`resultFormat`

Specify zero to obtain results in text format, or one to obtain results in binary format. (There is not currently a provision to obtain different result columns in different formats, although that is possible in the underlying protocol.)

The primary advantage of `PQexecParams` over `PQexec` is that parameter values may be separated from the command string, thus avoiding the need for tedious and error-prone quoting and escaping.

Unlike `PQexec`, `PQexecParams` allows at most one SQL command in the given string. (There can be semicolons in it, but not more than one nonempty command.) This is a limitation of the underlying protocol, but has some usefulness as an extra defense against SQL-injection attacks.

Tip: Specifying parameter types via OIDs is tedious, particularly if you prefer not to hard-wire particular OID values into your program. However, you can avoid doing so even in cases where the server by itself cannot determine the type of the parameter, or chooses a different type than you want. In the SQL command text, attach an explicit cast to the parameter symbol to show what data type you will send. For example,

```
select * from mytable where x = $1::bigint;
```

This forces parameter `$1` to be treated as `bigint`, whereas by default it would be assigned the same type as `x`. Forcing the parameter type decision, either this way or by specifying a numeric type OID, is strongly recommended when sending parameter values in binary format, because binary format has less redundancy than text format and so there is less chance that the server will detect a type mismatch mistake for you.

`PQprepare`

Submits a request to create a prepared statement with the given parameters, and waits for completion.

```
PGresult *PQprepare(PGconn *conn,
                    const char *stmtName,
                    const char *query,
                    int nParams,
                    const Oid *paramTypes);
```

`PQprepare` creates a prepared statement for later execution with `PQexecPrepared`. This feature allows commands that will be used repeatedly to be parsed and planned just once, rather than each time they are executed. `PQprepare` is supported only in protocol 3.0 and later connections; it will fail when using protocol 2.0.

The function creates a prepared statement named `stmtName` from the `query` string, which must contain a single SQL command. `stmtName` may be `""` to create an unnamed statement, in which case

any pre-existing unnamed statement is automatically replaced; otherwise it is an error if the statement name is already defined in the current session. If any parameters are used, they are referred to in the query as `$1`, `$2`, etc. `nParams` is the number of parameters for which types are pre-specified in the array `paramTypes[]`. (The array pointer may be `NULL` when `nParams` is zero.) `paramTypes[]` specifies, by OID, the data types to be assigned to the parameter symbols. If `paramTypes` is `NULL`, or any particular element in the array is zero, the server assigns a data type to the parameter symbol in the same way it would do for an untyped literal string. Also, the query may use parameter symbols with numbers higher than `nParams`; data types will be inferred for these symbols as well. (See `PQdescribePrepared` for a means to find out what data types were inferred.)

As with `PQexec`, the result is normally a `PGresult` object whose contents indicate server-side success or failure. A null result indicates out-of-memory or inability to send the command at all. Use `PQerrorMessage` to get more information about such errors.

Prepared statements for use with `PQexecPrepared` can also be created by executing SQL *PREPARE* statements. (But `PQprepare` is more flexible since it does not require parameter types to be pre-specified.) Also, although there is no libpq function for deleting a prepared statement, the SQL *DEALLOCATE* statement can be used for that purpose.

`PQexecPrepared`

Sends a request to execute a prepared statement with given parameters, and waits for the result.

```
PGresult *PQexecPrepared(PGconn *conn,
                         const char *stmtName,
                         int nParams,
                         const char * const *paramValues,
                         const int *paramLengths,
                         const int *paramFormats,
                         int resultFormat);
```

`PQexecPrepared` is like `PQexecParams`, but the command to be executed is specified by naming a previously-prepared statement, instead of giving a query string. This feature allows commands that will be used repeatedly to be parsed and planned just once, rather than each time they are executed. The statement must have been prepared previously in the current session. `PQexecPrepared` is supported only in protocol 3.0 and later connections; it will fail when using protocol 2.0.

The parameters are identical to `PQexecParams`, except that the name of a prepared statement is given instead of a query string, and the `paramTypes[]` parameter is not present (it is not needed since the prepared statement's parameter types were determined when it was created).

`PQdescribePrepared`

Submits a request to obtain information about the specified prepared statement, and waits for completion.

```
PGresult *PQdescribePrepared(PGconn *conn, const char *stmtName);
```

`PQdescribePrepared` allows an application to obtain information about a previously prepared statement. `PQdescribePrepared` is supported only in protocol 3.0 and later connections; it will fail when using protocol 2.0.

`stmtName` may be `""` or NULL to reference the unnamed statement, otherwise it must be the name of an existing prepared statement. On success, a `PGresult` with status `PGRES_COMMAND_OK` is returned. The functions `PQnparams` and `PQparamtype` may be applied to this `PGresult` to obtain in-

formation about the parameters of the prepared statement, and the functions PQnfields, PQfname, PQftype, etc provide information about the result columns (if any) of the statement.

PQdescribePortal

Submits a request to obtain information about the specified portal, and waits for completion.

```
PGresult *PQdescribePortal(PGconn *conn, const char *portalName);
```

PQdescribePortal allows an application to obtain information about a previously created portal. (libpq does not provide any direct access to portals, but you can use this function to inspect the properties of a cursor created with a DECLARE CURSOR SQL command.) PQdescribePortal is supported only in protocol 3.0 and later connections; it will fail when using protocol 2.0.

portalName may be "" or NULL to reference the unnamed portal, otherwise it must be the name of an existing portal. On success, a PGresult with status PGRES_COMMAND_OK is returned. The functions PQnfields, PQfname, PQftype, etc may be applied to the PGresult to obtain information about the result columns (if any) of the portal.

The PGresult structure encapsulates the result returned by the server. libpq application programmers should be careful to maintain the PGresult abstraction. Use the accessor functions below to get at the contents of PGresult. Avoid directly referencing the fields of the PGresult structure because they are subject to change in the future.

PQresultStatus

Returns the result status of the command.

```
ExecStatusType PQresultStatus(const PGresult *res);
```

PQresultStatus can return one of the following values:

PGRES_EMPTY_QUERY

The string sent to the server was empty.

PGRES_COMMAND_OK

Successful completion of a command returning no data.

PGRES_TUPLES_OK

Successful completion of a command returning data (such as a SELECT or SHOW).

PGRES_COPY_OUT

Copy Out (from server) data transfer started.

PGRES_COPY_IN

Copy In (to server) data transfer started.

PGRES_BAD_RESPONSE

The server's response was not understood.

PGRES_NONFATAL_ERROR

A nonfatal error (a notice or warning) occurred.

PGRES_FATAL_ERROR

A fatal error occurred.

If the result status is PGRES_TUPLES_OK, then the functions described below can be used to retrieve the rows returned by the query. Note that a SELECT command that happens to retrieve zero rows still shows PGRES_TUPLES_OK. PGRES_COMMAND_OK is for commands that can never return rows (INSERT, UPDATE, etc.). A response of PGRES_EMPTY_QUERY may indicate a bug in the client software.

A result of status PGRES_NONFATAL_ERROR will never be returned directly by PQexec or other query execution functions; results of this kind are instead passed to the notice processor (see Section 29.11).

PQresStatus

Converts the enumerated type returned by PQresultStatus into a string constant describing the status code. The caller should not free the result.

```
char *PQresStatus(ExecStatusType status);
```

PQresultErrorMessage

Returns the error message associated with the command, or an empty string if there was no error.

```
char *PQresultErrorMessage(const PGresult *res);
```

If there was an error, the returned string will include a trailing newline. The caller should not free the result directly. It will be freed when the associated PGresult handle is passed to PQclear.

Immediately following a PQexec or PQgetResult call, PQerrorMessage (on the connection) will return the same string as PQresultErrorMessage (on the result). However, a PGresult will retain its error message until destroyed, whereas the connection's error message will change when subsequent operations are done. Use PQresultErrorMessage when you want to know the status associated with a particular PGresult; use PQerrorMessage when you want to know the status from the latest operation on the connection.

PQresultErrorField

Returns an individual field of an error report.

```
char *PQresultErrorField(const PGresult *res, int fieldcode);
```

fieldcode is an error field identifier; see the symbols listed below. NULL is returned if the PGresult is not an error or warning result, or does not include the specified field. Field values will normally not include a trailing newline. The caller should not free the result directly. It will be freed when the associated PGresult handle is passed to PQclear.

The following field codes are available:

PG_DIAG_SEVERITY

The severity; the field contents are ERROR, FATAL, or PANIC (in an error message), or WARNING, NOTICE, DEBUG, INFO, or LOG (in a notice message), or a localized translation of one of these. Always present.

PG_DIAG_SQLSTATE

The SQLSTATE code for the error. The SQLSTATE code identifies the type of error that has occurred; it can be used by front-end applications to perform specific operations (such as error

handling) in response to a particular database error. For a list of the possible SQLSTATE codes, see Appendix A. This field is not localizable, and is always present.

`PG_DIAG_MESSAGE_PRIMARY`

The primary human-readable error message (typically one line). Always present.

`PG_DIAG_MESSAGE_DETAIL`

Detail: an optional secondary error message carrying more detail about the problem. May run to multiple lines.

`PG_DIAG_MESSAGE_HINT`

Hint: an optional suggestion what to do about the problem. This is intended to differ from detail in that it offers advice (potentially inappropriate) rather than hard facts. May run to multiple lines.

`PG_DIAG_STATEMENT_POSITION`

A string containing a decimal integer indicating an error cursor position as an index into the original statement string. The first character has index 1, and positions are measured in characters not bytes.

`PG_DIAG_INTERNAL_POSITION`

This is defined the same as the `PG_DIAG_STATEMENT_POSITION` field, but it is used when the cursor position refers to an internally generated command rather than the one submitted by the client. The `PG_DIAG_INTERNAL_QUERY` field will always appear when this field appears.

`PG_DIAG_INTERNAL_QUERY`

The text of a failed internally-generated command. This could be, for example, a SQL query issued by a PL/pgSQL function.

`PG_DIAG_CONTEXT`

An indication of the context in which the error occurred. Presently this includes a call stack traceback of active procedural language functions and internally-generated queries. The trace is one entry per line, most recent first.

`PG_DIAG_SOURCE_FILE`

The file name of the source-code location where the error was reported.

`PG_DIAG_SOURCE_LINE`

The line number of the source-code location where the error was reported.

`PG_DIAG_SOURCE_FUNCTION`

The name of the source-code function reporting the error.

The client is responsible for formatting displayed information to meet its needs; in particular it should break long lines as needed. Newline characters appearing in the error message fields should be treated as paragraph breaks, not line breaks.

Errors generated internally by libpq will have severity and primary message, but typically no other fields. Errors returned by a pre-3.0-protocol server will include severity and primary message, and sometimes a detail message, but no other fields.

Note that error fields are only available from PGresult objects, not PGconn objects; there is no PQerrorField function.

PQclear

Frees the storage associated with a PGresult. Every command result should be freed via PQclear when it is no longer needed.

```
void PQclear(PGresult *res);
```

You can keep a PGresult object around for as long as you need it; it does not go away when you issue a new command, nor even if you close the connection. To get rid of it, you must call PQclear. Failure to do this will result in memory leaks in your application.

PQmakeEmptyPGresult

Constructs an empty PGresult object with the given status.

```
PGresult *PQmakeEmptyPGresult(PGconn *conn, ExecStatusType status);
```

This is libpq's internal function to allocate and initialize an empty PGresult object. This function returns NULL if memory could not be allocated. It is exported because some applications find it useful to generate result objects (particularly objects with error status) themselves. If conn is not null and status indicates an error, the current error message of the specified connection is copied into the PGresult. Note that PQclear should eventually be called on the object, just as with a PGresult returned by libpq itself.

29.3.2. Retrieving Query Result Information

These functions are used to extract information from a PGresult object that represents a successful query result (that is, one that has status PGRES_TUPLES_OK). They can also be used to extract information from a successful Describe operation: a Describe's result has all the same column information that actual execution of the query would provide, but it has zero rows. For objects with other status values, these functions will act as though the result has zero rows and zero columns.

PQntuples

Returns the number of rows (tuples) in the query result.

```
int PQntuples(const PGresult *res);
```

PQnfields

Returns the number of columns (fields) in each row of the query result.

```
int PQnfields(const PGresult *res);
```

PQfname

Returns the column name associated with the given column number. Column numbers start at 0. The caller should not free the result directly. It will be freed when the associated PGresult handle is passed to PQclear.

```
char *PQfname(const PGresult *res,
              int column_number);
```

NULL is returned if the column number is out of range.

PQfnumber

Returns the column number associated with the given column name.

```
int PQfnumber(const PGresult *res,
              const char *column_name);
```

-1 is returned if the given name does not match any column.

The given name is treated like an identifier in an SQL command, that is, it is downcased unless double-quoted. For example, given a query result generated from the SQL command

```
select 1 as FOO, 2 as "BAR";
```

we would have the results:

```
PQfname(res, 0)              foo
PQfname(res, 1)              BAR
PQfnumber(res, "FOO")        0
PQfnumber(res, "foo")        0
PQfnumber(res, "BAR")        -1
PQfnumber(res, "\"BAR\"")    1
```

PQftable

Returns the OID of the table from which the given column was fetched. Column numbers start at 0.

```
Oid PQftable(const PGresult *res,
             int column_number);
```

InvalidOid is returned if the column number is out of range, or if the specified column is not a simple reference to a table column, or when using pre-3.0 protocol. You can query the system table pg_class to determine exactly which table is referenced.

The type Oid and the constant InvalidOid will be defined when you include the libpq header file. They will both be some integer type.

PQftablecol

Returns the column number (within its table) of the column making up the specified query result column. Query-result column numbers start at 0, but table columns have nonzero numbers.

```
int PQftablecol(const PGresult *res,
                int column_number);
```

Zero is returned if the column number is out of range, or if the specified column is not a simple reference to a table column, or when using pre-3.0 protocol.

PQfformat

Returns the format code indicating the format of the given column. Column numbers start at 0.

```
int PQfformat(const PGresult *res,
              int column_number);
```

Format code zero indicates textual data representation, while format code one indicates binary representation. (Other codes are reserved for future definition.)

PQftype

Returns the data type associated with the given column number. The integer returned is the internal OID number of the type. Column numbers start at 0.

```
Oid PQftype(const PGresult *res,
```

```
                int column_number);
```

You can query the system table pg_type to obtain the names and properties of the various data types. The OIDs of the built-in data types are defined in the file src/include/catalog/pg_type.h in the source tree.

PQfmod

Returns the type modifier of the column associated with the given column number. Column numbers start at 0.

```
int PQfmod(const PGresult *res,
           int column_number);
```

The interpretation of modifier values is type-specific; they typically indicate precision or size limits. The value -1 is used to indicate "no information available". Most data types do not use modifiers, in which case the value is always -1.

PQfsize

Returns the size in bytes of the column associated with the given column number. Column numbers start at 0.

```
int PQfsize(const PGresult *res,
            int column_number);
```

PQfsize returns the space allocated for this column in a database row, in other words the size of the server's internal representation of the data type. (Accordingly, it is not really very useful to clients.) A negative value indicates the data type is variable-length.

PQbinaryTuples

Returns 1 if the PGresult contains binary data and 0 if it contains text data.

```
int PQbinaryTuples(const PGresult *res);
```

This function is deprecated (except for its use in connection with COPY), because it is possible for a single PGresult to contain text data in some columns and binary data in others. PQfformat is preferred. PQbinaryTuples returns 1 only if all columns of the result are binary (format 1).

PQgetvalue

Returns a single field value of one row of a PGresult. Row and column numbers start at 0. The caller should not free the result directly. It will be freed when the associated PGresult handle is passed to PQclear.

```
char *PQgetvalue(const PGresult *res,
                 int row_number,
                 int column_number);
```

For data in text format, the value returned by PQgetvalue is a null-terminated character string representation of the field value. For data in binary format, the value is in the binary representation determined by the data type's typsend and typreceive functions. (The value is actually followed by a zero byte in this case too, but that is not ordinarily useful, since the value is likely to contain embedded nulls.)

An empty string is returned if the field value is null. See PQgetisnull to distinguish null values from empty-string values.

The pointer returned by PQgetvalue points to storage that is part of the PGresult structure. One should not modify the data it points to, and one must explicitly copy the data into other storage if it is to be used past the lifetime of the PGresult structure itself.

PQgetisnull

Tests a field for a null value. Row and column numbers start at 0.

```
int PQgetisnull(const PGresult *res,
                int row_number,
                int column_number);
```

This function returns 1 if the field is null and 0 if it contains a non-null value. (Note that PQgetvalue will return an empty string, not a null pointer, for a null field.)

PQgetlength

Returns the actual length of a field value in bytes. Row and column numbers start at 0.

```
int PQgetlength(const PGresult *res,
                int row_number,
                int column_number);
```

This is the actual data length for the particular data value, that is, the size of the object pointed to by PQgetvalue. For text data format this is the same as strlen(). For binary format this is essential information. Note that one should *not* rely on PQfsize to obtain the actual data length.

PQnparams

Returns the number of parameters of a prepared statement.

```
int PQnparams(const PGresult *res);
```

This function is only useful when inspecting the result of PQdescribePrepared. For other types of queries it will return zero.

PQparamtype

Returns the data type of the indicated statement parameter. Parameter numbers start at 0.

```
Oid PQparamtype(const PGresult *res, int param_number);
```

This function is only useful when inspecting the result of PQdescribePrepared. For other types of queries it will return zero.

PQprint

Prints out all the rows and, optionally, the column names to the specified output stream.

```
void PQprint(FILE *fout,      /* output stream */
             const PGresult *res,
             const PQprintOpt *po);

typedef struct {
    pqbool  header;      /* print output field headings and row count */
    pqbool  align;       /* fill align the fields */
    pqbool  standard;    /* old brain dead format */
    pqbool  html3;       /* output HTML tables */
    pqbool  expanded;    /* expand tables */
    pqbool  pager;       /* use pager for output if needed */
    char    *fieldSep;   /* field separator */
    char    *tableOpt;   /* attributes for HTML table element */
```

```
    char    *caption;    /* HTML table caption */
    char    **fieldName; /* null-terminated array of replacement field names */
} PQprintOpt;
```

This function was formerly used by psql to print query results, but this is no longer the case. Note that it assumes all the data is in text format.

29.3.3. Retrieving Result Information for Other Commands

These functions are used to extract information from `PGresult` objects that are not `SELECT` results.

`PQcmdStatus`

Returns the command status tag from the SQL command that generated the `PGresult`.

```
char *PQcmdStatus(PGresult *res);
```

Commonly this is just the name of the command, but it may include additional data such as the number of rows processed. The caller should not free the result directly. It will be freed when the associated `PGresult` handle is passed to `PQclear`.

`PQcmdTuples`

Returns the number of rows affected by the SQL command.

```
char *PQcmdTuples(PGresult *res);
```

This function returns a string containing the number of rows affected by the SQL statement that generated the `PGresult`. This function can only be used following the execution of an `INSERT`, `UPDATE`, `DELETE`, `MOVE`, `FETCH`, or `COPY` statement, or an `EXECUTE` of a prepared query that contains an `INSERT`, `UPDATE`, or `DELETE` statement. If the command that generated the `PGresult` was anything else, `PQcmdTuples` returns an empty string. The caller should not free the return value directly. It will be freed when the associated `PGresult` handle is passed to `PQclear`.

`PQoidValue`

Returns the OID of the inserted row, if the SQL command was an `INSERT` that inserted exactly one row into a table that has OIDs, or a `EXECUTE` of a prepared query containing a suitable `INSERT` statement. Otherwise, this function returns `InvalidOid`. This function will also return `InvalidOid` if the table affected by the `INSERT` statement does not contain OIDs.

```
Oid PQoidValue(const PGresult *res);
```

`PQoidStatus`

Returns a string with the OID of the inserted row, if the SQL command was an `INSERT` that inserted exactly one row, or a `EXECUTE` of a prepared statement consisting of a suitable `INSERT`. (The string will be `0` if the `INSERT` did not insert exactly one row, or if the target table does not have OIDs.) If the command was not an `INSERT`, returns an empty string.

```
char *PQoidStatus(const PGresult *res);
```

This function is deprecated in favor of `PQoidValue`. It is not thread-safe.

29.3.4. Escaping Strings for Inclusion in SQL Commands

`PQescapeStringConn` escapes a string for use within an SQL command. This is useful when inserting data values as literal constants in SQL commands. Certain characters (such as quotes and backslashes) must be escaped to prevent them from being interpreted specially by the SQL parser. `PQescapeStringConn` performs this operation.

> **Tip:** It is especially important to do proper escaping when handling strings that were received from an untrustworthy source. Otherwise there is a security risk: you are vulnerable to "SQL injection" attacks wherein unwanted SQL commands are fed to your database.

Note that it is not necessary nor correct to do escaping when a data value is passed as a separate parameter in `PQexecParams` or its sibling routines.

```
size_t PQescapeStringConn (PGconn *conn,
                           char *to, const char *from, size_t length,
                           int *error);
```

`PQescapeStringConn` writes an escaped version of the `from` string to the `to` buffer, escaping special characters so that they cannot cause any harm, and adding a terminating zero byte. The single quotes that must surround PostgreSQL string literals are not included in the result string; they should be provided in the SQL command that the result is inserted into. The parameter `from` points to the first character of the string that is to be escaped, and the `length` parameter gives the number of bytes in this string. A terminating zero byte is not required, and should not be counted in `length`. (If a terminating zero byte is found before `length` bytes are processed, `PQescapeStringConn` stops at the zero; the behavior is thus rather like `strncpy`.) `to` shall point to a buffer that is able to hold at least one more byte than twice the value of `length`, otherwise the behavior is undefined. Behavior is likewise undefined if the `to` and `from` strings overlap.

If the `error` parameter is not NULL, then `*error` is set to zero on success, nonzero on error. Presently the only possible error conditions involve invalid multibyte encoding in the source string. The output string is still generated on error, but it can be expected that the server will reject it as malformed. On error, a suitable message is stored in the `conn` object, whether or not `error` is NULL.

`PQescapeStringConn` returns the number of bytes written to `to`, not including the terminating zero byte.

```
size_t PQescapeString (char *to, const char *from, size_t length);
```

`PQescapeString` is an older, deprecated version of `PQescapeStringConn`; the difference is that it does not take `conn` or `error` parameters. Because of this, it cannot adjust its behavior depending on the connection properties (such as character encoding) and therefore *it may give the wrong results*. Also, it has no way to report error conditions.

`PQescapeString` can be used safely in single-threaded client programs that work with only one PostgreSQL connection at a time (in this case it can find out what it needs to know "behind the scenes"). In other contexts it is a security hazard and should be avoided in favor of `PQescapeStringConn`.

29.3.5. Escaping Binary Strings for Inclusion in SQL Commands

`PQescapeByteaConn`

Escapes binary data for use within an SQL command with the type `bytea`. As with `PQescapeStringConn`, this is only used when inserting data directly into an SQL command string.

```
unsigned char *PQescapeByteaConn(PGconn *conn,
                                 const unsigned char *from,
                                 size_t from_length,
                                 size_t *to_length);
```

Certain byte values *must* be escaped (but all byte values *can* be escaped) when used as part of a `bytea` literal in an SQL statement. In general, to escape a byte, it is converted into the three digit octal number equal to the octet value, and preceded by usually two backslashes. The single quote (`'`) and backslash (`\`) characters have special alternative escape sequences. See Section 8.4 for more information. `PQescapeByteaConn` performs this operation, escaping only the minimally required bytes.

The `from` parameter points to the first byte of the string that is to be escaped, and the `from_length` parameter gives the number of bytes in this binary string. (A terminating zero byte is neither necessary nor counted.) The `to_length` parameter points to a variable that will hold the resultant escaped string length. This result string length includes the terminating zero byte of the result.

`PQescapeByteaConn` returns an escaped version of the `from` parameter binary string in memory allocated with `malloc()`. This memory must be freed using `PQfreemem()` when the result is no longer needed. The return string has all special characters replaced so that they can be properly processed by the PostgreSQL string literal parser, and the `bytea` input function. A terminating zero byte is also added. The single quotes that must surround PostgreSQL string literals are not part of the result string.

On error, a NULL pointer is returned, and a suitable error message is stored in the `conn` object. Currently, the only possible error is insufficient memory for the result string.

`PQescapeBytea`

`PQescapeBytea` is an older, deprecated version of `PQescapeByteaConn`.

```
unsigned char *PQescapeBytea(const unsigned char *from,
                             size_t from_length,
                             size_t *to_length);
```

The only difference from `PQescapeByteaConn` is that `PQescapeBytea` does not take a `PGconn` parameter. Because of this, it cannot adjust its behavior depending on the connection properties (in particular, whether standard-conforming strings are enabled) and therefore *it may give the wrong results*. Also, it has no way to return an error message on failure.

`PQescapeBytea` can be used safely in single-threaded client programs that work with only one PostgreSQL connection at a time (in this case it can find out what it needs to know "behind the scenes"). In other contexts it is a security hazard and should be avoided in favor of `PQescapeByteaConn`.

PQunescapeBytea

Converts a string representation of binary data into binary data — the reverse of PQescapeBytea. This is needed when retrieving bytea data in text format, but not when retrieving it in binary format.

```
unsigned char *PQunescapeBytea(const unsigned char *from, size_t *to_length);
```

The from parameter points to a string such as might be returned by PQgetvalue when applied to a bytea column. PQunescapeBytea converts this string representation into its binary representation. It returns a pointer to a buffer allocated with malloc(), or null on error, and puts the size of the buffer in to_length. The result must be freed using PQfreemem when it is no longer needed.

This conversion is not exactly the inverse of PQescapeBytea, because the string is not expected to be "escaped" when received from PQgetvalue. In particular this means there is no need for string quoting considerations, and so no need for a PGconn parameter.

PQfreemem

Frees memory allocated by libpq.

```
void PQfreemem(void *ptr);
```

Frees memory allocated by libpq, particularly PQescapeByteaConn, PQescapeBytea, PQunescapeBytea, and PQnotifies. It is particularly important that this function, rather than free(), be used on Microsoft Windows. This is because allocating memory in a DLL and releasing it in the application works only if multithreaded/single-threaded, release/debug, and static/dynamic flags are the same for the DLL and the application. On non-Microsoft Windows platforms, this function is the same as the standard library function free().

29.4. Asynchronous Command Processing

The PQexec function is adequate for submitting commands in normal, synchronous applications. It has a couple of deficiencies, however, that can be of importance to some users:

- PQexec waits for the command to be completed. The application may have other work to do (such as maintaining a user interface), in which case it won't want to block waiting for the response.
- Since the execution of the client application is suspended while it waits for the result, it is hard for the application to decide that it would like to try to cancel the ongoing command. (It can be done from a signal handler, but not otherwise.)
- PQexec can return only one PGresult structure. If the submitted command string contains multiple SQL commands, all but the last PGresult are discarded by PQexec.

Applications that do not like these limitations can instead use the underlying functions that PQexec is built from: PQsendQuery and PQgetResult. There are also PQsendQueryParams, PQsendPrepare, PQsendQueryPrepared, PQsendDescribePrepared, and PQsendDescribePortal, which can be used with PQgetResult to duplicate the functionality of PQexecParams, PQprepare, PQexecPrepared, PQdescribePrepared, and PQdescribePortal respectively.

PQsendQuery

Submits a command to the server without waiting for the result(s). 1 is returned if the command was successfully dispatched and 0 if not (in which case, use PQerrorMessage to get more information about the failure).

```
int PQsendQuery(PGconn *conn, const char *command);
```

After successfully calling PQsendQuery, call PQgetResult one or more times to obtain the results. PQsendQuery may not be called again (on the same connection) until PQgetResult has returned a null pointer, indicating that the command is done.

PQsendQueryParams

Submits a command and separate parameters to the server without waiting for the result(s).

```
int PQsendQueryParams(PGconn *conn,
                      const char *command,
                      int nParams,
                      const Oid *paramTypes,
                      const char * const *paramValues,
                      const int *paramLengths,
                      const int *paramFormats,
                      int resultFormat);
```

This is equivalent to PQsendQuery except that query parameters can be specified separately from the query string. The function's parameters are handled identically to PQexecParams. Like PQexecParams, it will not work on 2.0-protocol connections, and it allows only one command in the query string.

PQsendPrepare

Sends a request to create a prepared statement with the given parameters, without waiting for completion.

```
int PQsendPrepare(PGconn *conn,
                  const char *stmtName,
                  const char *query,
                  int nParams,
                  const Oid *paramTypes);
```

This is an asynchronous version of PQprepare: it returns 1 if it was able to dispatch the request, and 0 if not. After a successful call, call PQgetResult to determine whether the server successfully created the prepared statement. The function's parameters are handled identically to PQprepare. Like PQprepare, it will not work on 2.0-protocol connections.

PQsendQueryPrepared

Sends a request to execute a prepared statement with given parameters, without waiting for the result(s).

```
int PQsendQueryPrepared(PGconn *conn,
                        const char *stmtName,
                        int nParams,
                        const char * const *paramValues,
                        const int *paramLengths,
                        const int *paramFormats,
                        int resultFormat);
```

This is similar to PQsendQueryParams, but the command to be executed is specified by naming a previously-prepared statement, instead of giving a query string. The function's parameters are handled identically to PQexecPrepared. Like PQexecPrepared, it will not work on 2.0-protocol connections.

PQsendDescribePrepared

Submits a request to obtain information about the specified prepared statement, without waiting for completion.

```
int PQsendDescribePrepared(PGconn *conn, const char *stmtName);
```

This is an asynchronous version of PQdescribePrepared: it returns 1 if it was able to dispatch the request, and 0 if not. After a successful call, call PQgetResult to obtain the results. The function's parameters are handled identically to PQdescribePrepared. Like PQdescribePrepared, it will not work on 2.0-protocol connections.

PQsendDescribePortal

Submits a request to obtain information about the specified portal, without waiting for completion.

```
int PQsendDescribePortal(PGconn *conn, const char *portalName);
```

This is an asynchronous version of PQdescribePortal: it returns 1 if it was able to dispatch the request, and 0 if not. After a successful call, call PQgetResult to obtain the results. The function's parameters are handled identically to PQdescribePortal. Like PQdescribePortal, it will not work on 2.0-protocol connections.

PQgetResult

Waits for the next result from a prior PQsendQuery, PQsendQueryParams, PQsendPrepare, or PQsendQueryPrepared call, and returns it. A null pointer is returned when the command is complete and there will be no more results.

```
PGresult *PQgetResult(PGconn *conn);
```

PQgetResult must be called repeatedly until it returns a null pointer, indicating that the command is done. (If called when no command is active, PQgetResult will just return a null pointer at once.) Each non-null result from PQgetResult should be processed using the same PGresult accessor functions previously described. Don't forget to free each result object with PQclear when done with it. Note that PQgetResult will block only if a command is active and the necessary response data has not yet been read by PQconsumeInput.

Using PQsendQuery and PQgetResult solves one of PQexec's problems: If a command string contains multiple SQL commands, the results of those commands can be obtained individually. (This allows a simple form of overlapped processing, by the way: the client can be handling the results of one command while the server is still working on later queries in the same command string.) However, calling PQgetResult will still cause the client to block until the server completes the next SQL command. This can be avoided by proper use of two more functions:

PQconsumeInput

If input is available from the server, consume it.

```
int PQconsumeInput(PGconn *conn);
```

PQconsumeInput normally returns 1 indicating "no error", but returns 0 if there was some kind of trouble (in which case PQerrorMessage can be consulted). Note that the result does not say

whether any input data was actually collected. After calling PQconsumeInput, the application may check PQisBusy and/or PQnotifies to see if their state has changed.

PQconsumeInput may be called even if the application is not prepared to deal with a result or notification just yet. The function will read available data and save it in a buffer, thereby causing a select() read-ready indication to go away. The application can thus use PQconsumeInput to clear the select() condition immediately, and then examine the results at leisure.

PQisBusy

Returns 1 if a command is busy, that is, PQgetResult would block waiting for input. A 0 return indicates that PQgetResult can be called with assurance of not blocking.

```
int PQisBusy(PGconn *conn);
```

PQisBusy will not itself attempt to read data from the server; therefore PQconsumeInput must be invoked first, or the busy state will never end.

A typical application using these functions will have a main loop that uses select() or poll() to wait for all the conditions that it must respond to. One of the conditions will be input available from the server, which in terms of select() means readable data on the file descriptor identified by PQsocket. When the main loop detects input ready, it should call PQconsumeInput to read the input. It can then call PQisBusy, followed by PQgetResult if PQisBusy returns false (0). It can also call PQnotifies to detect NOTIFY messages (see Section 29.7).

A client that uses PQsendQuery/PQgetResult can also attempt to cancel a command that is still being processed by the server; see Section 29.5. But regardless of the return value of PQcancel, the application must continue with the normal result-reading sequence using PQgetResult. A successful cancellation will simply cause the command to terminate sooner than it would have otherwise.

By using the functions described above, it is possible to avoid blocking while waiting for input from the database server. However, it is still possible that the application will block waiting to send output to the server. This is relatively uncommon but can happen if very long SQL commands or data values are sent. (It is much more probable if the application sends data via COPY IN, however.) To prevent this possibility and achieve completely nonblocking database operation, the following additional functions may be used.

PQsetnonblocking

Sets the nonblocking status of the connection.

```
int PQsetnonblocking(PGconn *conn, int arg);
```

Sets the state of the connection to nonblocking if arg is 1, or blocking if arg is 0. Returns 0 if OK, -1 if error.

In the nonblocking state, calls to PQsendQuery, PQputline, PQputnbytes, and PQendcopy will not block but instead return an error if they need to be called again.

Note that PQexec does not honor nonblocking mode; if it is called, it will act in blocking fashion anyway.

PQisnonblocking

Returns the blocking status of the database connection.

```
int PQisnonblocking(const PGconn *conn);
```

Returns 1 if the connection is set to nonblocking mode and 0 if blocking.

`PQflush`

Attempts to flush any queued output data to the server. Returns 0 if successful (or if the send queue is empty), -1 if it failed for some reason, or 1 if it was unable to send all the data in the send queue yet (this case can only occur if the connection is nonblocking).

```
int PQflush(PGconn *conn);
```

After sending any command or data on a nonblocking connection, call `PQflush`. If it returns 1, wait for the socket to be write-ready and call it again; repeat until it returns 0. Once `PQflush` returns 0, wait for the socket to be read-ready and then read the response as described above.

29.5. Cancelling Queries in Progress

A client application can request cancellation of a command that is still being processed by the server, using the functions described in this section.

`PQgetCancel`

Creates a data structure containing the information needed to cancel a command issued through a particular database connection.

```
PGcancel *PQgetCancel(PGconn *conn);
```

`PQgetCancel` creates a `PGcancel` object given a `PGconn` connection object. It will return NULL if the given `conn` is NULL or an invalid connection. The `PGcancel` object is an opaque structure that is not meant to be accessed directly by the application; it can only be passed to `PQcancel` or `PQfreeCancel`.

`PQfreeCancel`

Frees a data structure created by `PQgetCancel`.

```
void PQfreeCancel(PGcancel *cancel);
```

`PQfreeCancel` frees a data object previously created by `PQgetCancel`.

`PQcancel`

Requests that the server abandon processing of the current command.

```
int PQcancel(PGcancel *cancel, char *errbuf, int errbufsize);
```

The return value is 1 if the cancel request was successfully dispatched and 0 if not. If not, `errbuf` is filled with an error message explaining why not. `errbuf` must be a char array of size `errbufsize` (the recommended size is 256 bytes).

Successful dispatch is no guarantee that the request will have any effect, however. If the cancellation is effective, the current command will terminate early and return an error result. If the cancellation fails (say, because the server was already done processing the command), then there will be no visible result at all.

PQcancel can safely be invoked from a signal handler, if the errbuf is a local variable in the signal handler. The PGcancel object is read-only as far as PQcancel is concerned, so it can also be invoked from a thread that is separate from the one manipulating the PGconn object.

PQrequestCancel

Requests that the server abandon processing of the current command.

```
int PQrequestCancel(PGconn *conn);
```

PQrequestCancel is a deprecated variant of PQcancel. It operates directly on the PGconn object, and in case of failure stores the error message in the PGconn object (whence it can be retrieved by PQerrorMessage). Although the functionality is the same, this approach creates hazards for multiple-thread programs and signal handlers, since it is possible that overwriting the PGconn's error message will mess up the operation currently in progress on the connection.

29.6. The Fast-Path Interface

PostgreSQL provides a fast-path interface to send simple function calls to the server.

Tip: This interface is somewhat obsolete, as one may achieve similar performance and greater functionality by setting up a prepared statement to define the function call. Then, executing the statement with binary transmission of parameters and results substitutes for a fast-path function call.

The function PQfn requests execution of a server function via the fast-path interface:

```
PGresult *PQfn(PGconn *conn,
               int fnid,
               int *result_buf,
               int *result_len,
               int result_is_int,
               const PQArgBlock *args,
               int nargs);

typedef struct {
    int len;
    int isint;
    union {
        int *ptr;
        int integer;
    } u;
} PQArgBlock;
```

The fnid argument is the OID of the function to be executed. args and nargs define the parameters to be passed to the function; they must match the declared function argument list. When the isint field of a parameter structure is true, the u.integer value is sent to the server as an integer of the indicated length (this must be 1, 2, or 4 bytes); proper byte-swapping occurs. When isint is false, the indicated

number of bytes at *u.ptr are sent with no processing; the data must be in the format expected by the server for binary transmission of the function's argument data type. result_buf is the buffer in which to place the return value. The caller must have allocated sufficient space to store the return value. (There is no check!) The actual result length will be returned in the integer pointed to by result_len. If a 1, 2, or 4-byte integer result is expected, set result_is_int to 1, otherwise set it to 0. Setting result_is_int to 1 causes libpq to byte-swap the value if necessary, so that it is delivered as a proper int value for the client machine. When result_is_int is 0, the binary-format byte string sent by the server is returned unmodified.

PQfn always returns a valid PGresult pointer. The result status should be checked before the result is used. The caller is responsible for freeing the PGresult with PQclear when it is no longer needed.

Note that it is not possible to handle null arguments, null results, nor set-valued results when using this interface.

29.7. Asynchronous Notification

PostgreSQL offers asynchronous notification via the LISTEN and NOTIFY commands. A client session registers its interest in a particular notification condition with the LISTEN command (and can stop listening with the UNLISTEN command). All sessions listening on a particular condition will be notified asynchronously when a NOTIFY command with that condition name is executed by any session. No additional information is passed from the notifier to the listener. Thus, typically, any actual data that needs to be communicated is transferred through a database table. Commonly, the condition name is the same as the associated table, but it is not necessary for there to be any associated table.

libpq applications submit LISTEN and UNLISTEN commands as ordinary SQL commands. The arrival of NOTIFY messages can subsequently be detected by calling PQnotifies.

The function PQnotifies returns the next notification from a list of unhandled notification messages received from the server. It returns a null pointer if there are no pending notifications. Once a notification is returned from PQnotifies, it is considered handled and will be removed from the list of notifications.

```
PGnotify *PQnotifies(PGconn *conn);

typedef struct pgNotify {
    char *relname;              /* notification condition name */
    int  be_pid;                /* process ID of notifying server process */
    char *extra;                /* notification parameter */
} PGnotify;
```

After processing a PGnotify object returned by PQnotifies, be sure to free it with PQfreemem. It is sufficient to free the PGnotify pointer; the relname and extra fields do not represent separate allocations. (At present, the extra field is unused and will always point to an empty string.)

Example 29-2 gives a sample program that illustrates the use of asynchronous notification.

PQnotifies does not actually read data from the server; it just returns messages previously absorbed by another libpq function. In prior releases of libpq, the only way to ensure timely receipt of NOTIFY messages was to constantly submit commands, even empty ones, and then check PQnotifies after each PQexec. While this still works, it is deprecated as a waste of processing power.

A better way to check for NOTIFY messages when you have no useful commands to execute is to call PQconsumeInput, then check PQnotifies. You can use select() to wait for data to arrive from the server, thereby using no CPU power unless there is something to do. (See PQsocket to obtain the file descriptor number to use with select().) Note that this will work OK whether you submit commands with PQsendQuery/PQgetResult or simply use PQexec. You should, however, remember to check PQnotifies after each PQgetResult or PQexec, to see if any notifications came in during the processing of the command.

29.8. Functions Associated with the COPY Command

The COPY command in PostgreSQL has options to read from or write to the network connection used by libpq. The functions described in this section allow applications to take advantage of this capability by supplying or consuming copied data.

The overall process is that the application first issues the SQL COPY command via PQexec or one of the equivalent functions. The response to this (if there is no error in the command) will be a PGresult object bearing a status code of PGRES_COPY_OUT or PGRES_COPY_IN (depending on the specified copy direction). The application should then use the functions of this section to receive or transmit data rows. When the data transfer is complete, another PGresult object is returned to indicate success or failure of the transfer. Its status will be PGRES_COMMAND_OK for success or PGRES_FATAL_ERROR if some problem was encountered. At this point further SQL commands may be issued via PQexec. (It is not possible to execute other SQL commands using the same connection while the COPY operation is in progress.)

If a COPY command is issued via PQexec in a string that could contain additional commands, the application must continue fetching results via PQgetResult after completing the COPY sequence. Only when PQgetResult returns NULL is it certain that the PQexec command string is done and it is safe to issue more commands.

The functions of this section should be executed only after obtaining a result status of PGRES_COPY_OUT or PGRES_COPY_IN from PQexec or PQgetResult.

A PGresult object bearing one of these status values carries some additional data about the COPY operation that is starting. This additional data is available using functions that are also used in connection with query results:

PQnfields

Returns the number of columns (fields) to be copied.

PQbinaryTuples

0 indicates the overall copy format is textual (rows separated by newlines, columns separated by separator characters, etc). 1 indicates the overall copy format is binary. See *COPY* for more information.

PQfformat

Returns the format code (0 for text, 1 for binary) associated with each column of the copy operation. The per-column format codes will always be zero when the overall copy format is textual, but the binary format can support both text and binary columns. (However, as of the current implementation of COPY, only binary columns appear in a binary copy; so the per-column formats always match the overall format at present.)

Note: These additional data values are only available when using protocol 3.0. When using protocol 2.0, all these functions will return 0.

29.8.1. Functions for Sending COPY Data

These functions are used to send data during COPY FROM STDIN. They will fail if called when the connection is not in COPY_IN state.

PQputCopyData

Sends data to the server during COPY_IN state.

```
int PQputCopyData(PGconn *conn,
                  const char *buffer,
                  int nbytes);
```

Transmits the COPY data in the specified buffer, of length nbytes, to the server. The result is 1 if the data was sent, zero if it was not sent because the attempt would block (this case is only possible if the connection is in nonblocking mode), or -1 if an error occurred. (Use PQerrorMessage to retrieve details if the return value is -1. If the value is zero, wait for write-ready and try again.)

The application may divide the COPY data stream into buffer loads of any convenient size. Buffer-load boundaries have no semantic significance when sending. The contents of the data stream must match the data format expected by the COPY command; see *COPY* for details.

PQputCopyEnd

Sends end-of-data indication to the server during COPY_IN state.

```
int PQputCopyEnd(PGconn *conn,
                 const char *errormsg);
```

Ends the COPY_IN operation successfully if errormsg is NULL. If errormsg is not NULL then the COPY is forced to fail, with the string pointed to by errormsg used as the error message. (One should not assume that this exact error message will come back from the server, however, as the server might have already failed the COPY for its own reasons. Also note that the option to force failure does not work when using pre-3.0-protocol connections.)

The result is 1 if the termination data was sent, zero if it was not sent because the attempt would block (this case is only possible if the connection is in nonblocking mode), or -1 if an error occurred. (Use PQerrorMessage to retrieve details if the return value is -1. If the value is zero, wait for write-ready and try again.)

After successfully calling PQputCopyEnd, call PQgetResult to obtain the final result status of the COPY command. One may wait for this result to be available in the usual way. Then return to normal operation.

29.8.2. Functions for Receiving COPY Data

These functions are used to receive data during COPY TO STDOUT. They will fail if called when the connection is not in COPY_OUT state.

PQgetCopyData

Receives data from the server during COPY_OUT state.

```
int PQgetCopyData(PGconn *conn,
                  char **buffer,
                  int async);
```

Attempts to obtain another row of data from the server during a COPY. Data is always returned one data row at a time; if only a partial row is available, it is not returned. Successful return of a data row involves allocating a chunk of memory to hold the data. The buffer parameter must be non-NULL. *buffer is set to point to the allocated memory, or to NULL in cases where no buffer is returned. A non-NULL result buffer must be freed using PQfreemem when no longer needed.

When a row is successfully returned, the return value is the number of data bytes in the row (this will always be greater than zero). The returned string is always null-terminated, though this is probably only useful for textual COPY. A result of zero indicates that the COPY is still in progress, but no row is yet available (this is only possible when async is true). A result of -1 indicates that the COPY is done. A result of -2 indicates that an error occurred (consult PQerrorMessage for the reason).

When async is true (not zero), PQgetCopyData will not block waiting for input; it will return zero if the COPY is still in progress but no complete row is available. (In this case wait for read-ready and then call PQconsumeInput before calling PQgetCopyData again.) When async is false (zero), PQgetCopyData will block until data is available or the operation completes.

After PQgetCopyData returns -1, call PQgetResult to obtain the final result status of the COPY command. One may wait for this result to be available in the usual way. Then return to normal operation.

29.8.3. Obsolete Functions for COPY

These functions represent older methods of handling COPY. Although they still work, they are deprecated due to poor error handling, inconvenient methods of detecting end-of-data, and lack of support for binary or nonblocking transfers.

PQgetline

Reads a newline-terminated line of characters (transmitted by the server) into a buffer string of size length.

```
int PQgetline(PGconn *conn,
              char *buffer,
              int length);
```

This function copies up to length-1 characters into the buffer and converts the terminating newline into a zero byte. PQgetline returns EOF at the end of input, 0 if the entire line has been read, and 1 if the buffer is full but the terminating newline has not yet been read.

Note that the application must check to see if a new line consists of the two characters \., which indicates that the server has finished sending the results of the COPY command. If the application might receive lines that are more than length-1 characters long, care is needed to be sure it recognizes the \. line correctly (and does not, for example, mistake the end of a long data line for a terminator line).

PQgetlineAsync

Reads a row of COPY data (transmitted by the server) into a buffer without blocking.

```
int PQgetlineAsync(PGconn *conn,
                   char *buffer,
                   int bufsize);
```

This function is similar to PQgetline, but it can be used by applications that must read COPY data asynchronously, that is, without blocking. Having issued the COPY command and gotten a PGRES_COPY_OUT response, the application should call PQconsumeInput and PQgetlineAsync until the end-of-data signal is detected.

Unlike PQgetline, this function takes responsibility for detecting end-of-data.

On each call, PQgetlineAsync will return data if a complete data row is available in libpq's input buffer. Otherwise, no data is returned until the rest of the row arrives. The function returns -1 if the end-of-copy-data marker has been recognized, or 0 if no data is available, or a positive number giving the number of bytes of data returned. If -1 is returned, the caller must next call PQendcopy, and then return to normal processing.

The data returned will not extend beyond a data-row boundary. If possible a whole row will be returned at one time. But if the buffer offered by the caller is too small to hold a row sent by the server, then a partial data row will be returned. With textual data this can be detected by testing whether the last returned byte is \n or not. (In a binary COPY, actual parsing of the COPY data format will be needed to make the equivalent determination.) The returned string is not null-terminated. (If you want to add a terminating null, be sure to pass a bufsize one smaller than the room actually available.)

PQputline

Sends a null-terminated string to the server. Returns 0 if OK and EOF if unable to send the string.

```
int PQputline(PGconn *conn,
              const char *string);
```

The COPY data stream sent by a series of calls to PQputline has the same format as that returned by PQgetlineAsync, except that applications are not obliged to send exactly one data row per PQputline call; it is okay to send a partial line or multiple lines per call.

> **Note:** Before PostgreSQL protocol 3.0, it was necessary for the application to explicitly send the two characters \. as a final line to indicate to the server that it had finished sending COPY data. While this still works, it is deprecated and the special meaning of \. can be expected to be removed in a future release. It is sufficient to call PQendcopy after having sent the actual data.

PQputnbytes

Sends a non-null-terminated string to the server. Returns 0 if OK and EOF if unable to send the string.

```
int PQputnbytes(PGconn *conn,
                const char *buffer,
                int nbytes);
```

This is exactly like PQputline, except that the data buffer need not be null-terminated since the number of bytes to send is specified directly. Use this procedure when sending binary data.

PQendcopy

Synchronizes with the server.

```
int PQendcopy(PGconn *conn);
```

This function waits until the server has finished the copying. It should either be issued when the last string has been sent to the server using PQputline or when the last string has been received from the server using PGgetline. It must be issued or the server will get "out of sync" with the client. Upon return from this function, the server is ready to receive the next SQL command. The return value is 0 on successful completion, nonzero otherwise. (Use PQerrorMessage to retrieve details if the return value is nonzero.)

When using PQgetResult, the application should respond to a PGRES_COPY_OUT result by executing PQgetline repeatedly, followed by PQendcopy after the terminator line is seen. It should then return to the PQgetResult loop until PQgetResult returns a null pointer. Similarly a PGRES_COPY_IN result is processed by a series of PQputline calls followed by PQendcopy, then return to the PQgetResult loop. This arrangement will ensure that a COPY command embedded in a series of SQL commands will be executed correctly.

Older applications are likely to submit a COPY via PQexec and assume that the transaction is done after PQendcopy. This will work correctly only if the COPY is the only SQL command in the command string.

29.9. Control Functions

These functions control miscellaneous details of libpq's behavior.

PQsetErrorVerbosity

Determines the verbosity of messages returned by PQerrorMessage and PQresultErrorMessage.

```
typedef enum {
    PQERRORS_TERSE,
    PQERRORS_DEFAULT,
    PQERRORS_VERBOSE
} PGVerbosity;

PGVerbosity PQsetErrorVerbosity(PGconn *conn, PGVerbosity verbosity);
```

PQsetErrorVerbosity sets the verbosity mode, returning the connection's previous setting. In *TERSE* mode, returned messages include severity, primary text, and position only; this will normally fit on a single line. The default mode produces messages that include the above plus any detail, hint, or context fields (these may span multiple lines). The *VERBOSE* mode includes all available fields. Changing the verbosity does not affect the messages available from already-existing PGresult objects, only subsequently-created ones.

PQtrace

Enables tracing of the client/server communication to a debugging file stream.

```
void PQtrace(PGconn *conn, FILE *stream);
```

Note: On Windows, if the libpq library and an application are compiled with different flags, this function call will crash the application because the internal representation of the FILE pointers differ. Specifically, multithreaded/single-threaded, release/debug, and static/dynamic flags should be the same for the library and all applications using that library.

PQuntrace

Disables tracing started by PQtrace.

```
void PQuntrace(PGconn *conn);
```

29.10. Miscellaneous Functions

As always, there are some functions that just don't fit anywhere.

PQencryptPassword

Prepares the encrypted form of a PostgreSQL password.

```
char * PQencryptPassword(const char *passwd, const char *user);
```

This function is intended to be used by client applications that wish to send commands like ALTER USER joe PASSWORD 'pwd'. It is good practice not to send the original cleartext password in such a command, because it might be exposed in command logs, activity displays, and so on. Instead, use this function to convert the password to encrypted form before it is sent. The arguments are the cleartext password, and the SQL name of the user it is for. The return value is a string allocated by malloc, or NULL if out of memory. The caller may assume the string doesn't contain any special characters that would require escaping. Use PQfreemem to free the result when done with it.

29.11. Notice Processing

Notice and warning messages generated by the server are not returned by the query execution functions, since they do not imply failure of the query. Instead they are passed to a notice handling function, and execution continues normally after the handler returns. The default notice handling function prints the message on stderr, but the application can override this behavior by supplying its own handling function.

For historical reasons, there are two levels of notice handling, called the notice receiver and notice processor. The default behavior is for the notice receiver to format the notice and pass a string to the notice processor for printing. However, an application that chooses to provide its own notice receiver will typically ignore the notice processor layer and just do all the work in the notice receiver.

The function PQsetNoticeReceiver sets or examines the current notice receiver for a connection object. Similarly, PQsetNoticeProcessor sets or examines the current notice processor.

```
typedef void (*PQnoticeReceiver) (void *arg, const PGresult *res);

PQnoticeReceiver
PQsetNoticeReceiver(PGconn *conn,
                    PQnoticeReceiver proc,
                    void *arg);
```

```
typedef void (*PQnoticeProcessor) (void *arg, const char *message);

PQnoticeProcessor
PQsetNoticeProcessor(PGconn *conn,
                     PQnoticeProcessor proc,
                     void *arg);
```

Each of these functions returns the previous notice receiver or processor function pointer, and sets the new value. If you supply a null function pointer, no action is taken, but the current pointer is returned.

When a notice or warning message is received from the server, or generated internally by libpq, the notice receiver function is called. It is passed the message in the form of a `PGRES_NONFATAL_ERROR` `PGresult`. (This allows the receiver to extract individual fields using `PQresultErrorField`, or the complete preformatted message using `PQresultErrorMessage`.) The same void pointer passed to `PQsetNoticeReceiver` is also passed. (This pointer can be used to access application-specific state if needed.)

The default notice receiver simply extracts the message (using `PQresultErrorMessage`) and passes it to the notice processor.

The notice processor is responsible for handling a notice or warning message given in text form. It is passed the string text of the message (including a trailing newline), plus a void pointer that is the same one passed to `PQsetNoticeProcessor`. (This pointer can be used to access application-specific state if needed.)

The default notice processor is simply

```
static void
defaultNoticeProcessor(void *arg, const char *message)
{
    fprintf(stderr, "%s", message);
}
```

Once you have set a notice receiver or processor, you should expect that that function could be called as long as either the `PGconn` object or `PGresult` objects made from it exist. At creation of a `PGresult`, the `PGconn`'s current notice handling pointers are copied into the `PGresult` for possible use by functions like `PQgetvalue`.

29.12. Environment Variables

The following environment variables can be used to select default connection parameter values, which will be used by `PQconnectdb`, `PQsetdbLogin` and `PQsetdb` if no value is directly specified by the calling code. These are useful to avoid hard-coding database connection information into simple client applications, for example.

- `PGHOST` sets the database server name. If this begins with a slash, it specifies Unix-domain communication rather than TCP/IP communication; the value is then the name of the directory in which the

socket file is stored (in a default installation setup this would be `/tmp`).

- `PGHOSTADDR` specifies the numeric IP address of the database server. This can be set instead of or in addition to `PGHOST` to avoid DNS lookup overhead. See the documentation of these parameters, under `PQconnectdb` above, for details on their interaction.

  When neither `PGHOST` nor `PGHOSTADDR` is set, the default behavior is to connect using a local Unix-domain socket; or on machines without Unix-domain sockets, libpq will attempt to connect to `localhost`.

- `PGPORT` sets the TCP port number or Unix-domain socket file extension for communicating with the PostgreSQL server.

- `PGDATABASE` sets the PostgreSQL database name.

- `PGUSER` sets the user name used to connect to the database.

- `PGPASSWORD` sets the password used if the server demands password authentication. Use of this environment variable is not recommended for security reasons (some operating systems allow non-root users to see process environment variables via ps); instead consider using the `~/.pgpass` file (see Section 29.13).

- `PGPASSFILE` specifies the name of the password file to use for lookups. If not set, it defaults to `~/.pgpass` (see Section 29.13).

- `PGSERVICE` sets the service name to be looked up in `pg_service.conf`. This offers a shorthand way of setting all the parameters.

- `PGREALM` sets the Kerberos realm to use with PostgreSQL, if it is different from the local realm. If `PGREALM` is set, libpq applications will attempt authentication with servers for this realm and use separate ticket files to avoid conflicts with local ticket files. This environment variable is only used if Kerberos authentication is selected by the server.

- `PGOPTIONS` sets additional run-time options for the PostgreSQL server.

- `PGSSLMODE` determines whether and with what priority an SSL connection will be negotiated with the server. There are four modes: `disable` will attempt only an unencrypted SSL connection; `allow` will negotiate, trying first a non-SSL connection, then if that fails, trying an SSL connection; `prefer` (the default) will negotiate, trying first an SSL connection, then if that fails, trying a regular non-SSL connection; `require` will try only an SSL connection. If PostgreSQL is compiled without SSL support, using option `require` will cause an error, while options `allow` and `prefer` will be accepted but libpq will not in fact attempt an SSL connection.

- `PGREQUIRESSL` sets whether or not the connection must be made over SSL. If set to "1", libpq will refuse to connect if the server does not accept an SSL connection (equivalent to `sslmode prefer`). This option is deprecated in favor of the `sslmode` setting, and is only available if PostgreSQL is compiled with SSL support.

- `PGKRBSRVNAME` sets the Kerberos service name to use when authenticating with Kerberos 5.

- `PGCONNECT_TIMEOUT` sets the maximum number of seconds that libpq will wait when attempting to connect to the PostgreSQL server. If unset or set to zero, libpq will wait indefinitely. It is not recommended to set the timeout to less than 2 seconds.

The following environment variables can be used to specify default behavior for each PostgreSQL session. (See also the *ALTER USER* and *ALTER DATABASE* commands for ways to set default behavior on a per-user or per-database basis.)

- `PGDATESTYLE` sets the default style of date/time representation. (Equivalent to `SET datestyle TO ...`.)
- `PGTZ` sets the default time zone. (Equivalent to `SET timezone TO ...`.)
- `PGCLIENTENCODING` sets the default client character set encoding. (Equivalent to `SET client_encoding TO ...`.)
- `PGGEQO` sets the default mode for the genetic query optimizer. (Equivalent to `SET geqo TO ...`.)

Refer to the SQL command *SET* for information on correct values for these environment variables.

The following environment variables determine internal behavior of libpq; they override compiled-in defaults.

- `PGSYSCONFDIR` sets the directory containing the `pg_service.conf` file.
- `PGLOCALEDIR` sets the directory containing the `locale` files for message internationalization.

29.13. The Password File

The file `.pgpass` in a user's home directory or the file referenced by `PGPASSFILE` can contain passwords to be used if the connection requires a password (and no password has been specified otherwise). On Microsoft Windows the file is named `%APPDATA%\postgresql\pgpass.conf` (where `%APPDATA%` refers to the Application Data subdirectory in the user's profile).

This file should contain lines of the following format:

hostname`:`*port*`:`*database*`:`*username*`:`*password*

Each of the first four fields may be a literal value, or `*`, which matches anything. The password field from the first line that matches the current connection parameters will be used. (Therefore, put more-specific entries first when you are using wildcards.) If an entry needs to contain `:` or `\`, escape this character with `\`. A host name of `localhost` matches both TCP (hostname `localhost`) and Unix domain socket (`pghost` empty or the default socket directory) connections coming from the local machine.

The permissions on `.pgpass` must disallow any access to world or group; achieve this by the command `chmod 0600 ~/.pgpass`. If the permissions are less strict than this, the file will be ignored. (The file permissions are not currently checked on Microsoft Windows, however.)

29.14. The Connection Service File

The connection service file allows libpq connection parameters to be associated with a single service name. That service name can then be specified by a libpq connection, and the associated settings will

be used. This allows connection parameters to be modified without requiring a recompile of the libpq application. The service name can also be specified using the `PGSERVICE` environment variable.

To use this feature, copy `share/pg_service.conf.sample` to `etc/pg_service.conf` and edit the file to add service names and parameters. This file can be used for client-only installs too. The file's location can also be specified by the `PGSYSCONFDIR` environment variable.

29.15. LDAP Lookup of Connection Parameters

If libpq has been compiled with LDAP support (option `--with-ldap` for `configure`) it is possible to retrieve connection options like `host` or `dbname` via LDAP from a central server. The advantage is that if the connection parameters for a database change, the connection information doesn't have to be updated on all client machines.

LDAP connection parameter lookup uses the connection service file `pg_service.conf` (see Section 29.14). A line in a `pg_service.conf` stanza that starts with `ldap://` will be recognized as an LDAP URL and an LDAP query will be performed. The result must be a list of `keyword = value` pairs which will be used to set connection options. The URL must conform to RFC 1959 and be of the form

ldap://[*hostname*[:*port*]]/*search_base*?*attribute*?*search_scope*?*filter*

where *hostname* defaults to `localhost` and *port* defaults to 389.

Processing of `pg_service.conf` is terminated after a successful LDAP lookup, but is continued if the LDAP server cannot be contacted. This is to provide a fallback with further LDAP URL lines that point to different LDAP servers, classical `keyword = value` pairs, or default connection options. If you would rather get an error message in this case, add a syntactically incorrect line after the LDAP URL.

A sample LDAP entry that has been created with the LDIF file

```
version:1
dn:cn=mydatabase,dc=mycompany,dc=com
changetype:add
objectclass:top
objectclass:groupOfUniqueNames
cn:mydatabase
uniqueMember:host=dbserver.mycompany.com
uniqueMember:port=5439
uniqueMember:dbname=mydb
uniqueMember:user=mydb_user
uniqueMember:sslmode=require
```

might be queried with the following LDAP URL:

```
ldap://ldap.mycompany.com/dc=mycompany,dc=com?uniqueMember?one?(cn=mydatabase)
```

29.16. SSL Support

PostgreSQL has native support for using SSL connections to encrypt client/server communications for increased security. See Section 16.7 for details about the server-side SSL functionality.

If the server demands a client certificate, libpq will send the certificate stored in file `~/.postgresql/postgresql.crt` within the user's home directory. A matching private key file `~/.postgresql/postgresql.key` must also be present, and must not be world-readable. (On Microsoft Windows these files are named `%APPDATA%\postgresql\postgresql.crt` and `%APPDATA%\postgresql\postgresql.key`.)

If the file `~/.postgresql/root.crt` is present in the user's home directory, libpq will use the certificate list stored therein to verify the server's certificate. (On Microsoft Windows the file is named `%APPDATA%\postgresql\root.crt`.) The SSL connection will fail if the server does not present a certificate; therefore, to use this feature the server must have a `server.crt` file. Certificate Revocation List (CRL) entries are also checked if the file `~/.postgresql/root.crl` exists (`%APPDATA%\postgresql\root.crl` on Microsoft Windows).

If you are using SSL inside your application (in addition to inside libpq), you can use `PQinitSSL(int)` to tell libpq that the SSL library has already been initialized by your application.

29.17. Behavior in Threaded Programs

libpq is reentrant and thread-safe if the `configure` command-line option `--enable-thread-safety` was used when the PostgreSQL distribution was built. In addition, you might need to use additional compiler command-line options when you compile your application code. Refer to your system's documentation for information about how to build thread-enabled applications, or look in `src/Makefile.global` for `PTHREAD_CFLAGS` and `PTHREAD_LIBS`. This function allows the querying of libpq's thread-safe status:

`PQisthreadsafe`

Returns the thread safety status of the libpq library.

```
int PQisthreadsafe();
```

Returns 1 if the libpq is thread-safe and 0 if it is not.

One thread restriction is that no two threads attempt to manipulate the same `PGconn` object at the same time. In particular, you cannot issue concurrent commands from different threads through the same connection object. (If you need to run concurrent commands, use multiple connections.)

`PGresult` objects are read-only after creation, and so can be passed around freely between threads.

The deprecated functions `PQrequestCancel` and `PQoidStatus` are not thread-safe and should not be used in multithread programs. `PQrequestCancel` can be replaced by `PQcancel`. `PQoidStatus` can be replaced by `PQoidValue`.

If you are using Kerberos inside your application (in addition to inside libpq), you will need to do locking around Kerberos calls because Kerberos functions are not thread-safe. See function `PQregisterThreadLock` in the libpq source code for a way to do cooperative locking between libpq and your application.

If you experience problems with threaded applications, run the program in `src/tools/thread` to see if your platform has thread-unsafe functions. This program is run by `configure`, but for binary distributions your library might not match the library used to build the binaries.

29.18. Building libpq Programs

To build (i.e., compile and link) a program using libpq you need to do all of the following things:

- Include the `libpq-fe.h` header file:

  ```
  #include <libpq-fe.h>
  ```

  If you failed to do that then you will normally get error messages from your compiler similar to

  ```
  foo.c: In function 'main':
  foo.c:34: 'PGconn' undeclared (first use in this function)
  foo.c:35: 'PGresult' undeclared (first use in this function)
  foo.c:54: 'CONNECTION_BAD' undeclared (first use in this function)
  foo.c:68: 'PGRES_COMMAND_OK' undeclared (first use in this function)
  foo.c:95: 'PGRES_TUPLES_OK' undeclared (first use in this function)
  ```

- Point your compiler to the directory where the PostgreSQL header files were installed, by supplying the `-I`*`directory`* option to your compiler. (In some cases the compiler will look into the directory in question by default, so you can omit this option.) For instance, your compile command line could look like:

  ```
  cc -c -I/usr/local/pgsql/include testprog.c
  ```

  If you are using makefiles then add the option to the `CPPFLAGS` variable:

  ```
  CPPFLAGS += -I/usr/local/pgsql/include
  ```

  If there is any chance that your program might be compiled by other users then you should not hardcode the directory location like that. Instead, you can run the utility `pg_config` to find out where the header files are on the local system:

  ```
  $ pg_config --includedir
  /usr/local/include
  ```

  Failure to specify the correct option to the compiler will result in an error message such as

  ```
  testlibpq.c:8:22: libpq-fe.h: No such file or directory
  ```

- When linking the final program, specify the option `-lpq` so that the libpq library gets pulled in, as well as the option `-L`*`directory`* to point the compiler to the directory where the libpq library resides. (Again, the compiler will search some directories by default.) For maximum portability, put the `-L` option before the `-lpq` option. For example:

  ```
  cc -o testprog testprog1.o testprog2.o -L/usr/local/pgsql/lib -lpq
  ```

  You can find out the library directory using `pg_config` as well:

  ```
  $ pg_config --libdir
  /usr/local/pgsql/lib
  ```

  Error messages that point to problems in this area could look like the following.

  ```
  testlibpq.o: In function 'main':
  ```

```
testlibpq.o(.text+0x60): undefined reference to 'PQsetdbLogin'
testlibpq.o(.text+0x71): undefined reference to 'PQstatus'
testlibpq.o(.text+0xa4): undefined reference to 'PQerrorMessage'
```

This means you forgot `-lpq`.

```
/usr/bin/ld: cannot find -lpq
```

This means you forgot the `-L` option or did not specify the right directory.

29.19. Example Programs

These examples and others can be found in the directory `src/test/examples` in the source code distribution.

Example 29-1. libpq Example Program 1

```
/*
 * testlibpq.c
 *
 *      Test the C version of libpq, the PostgreSQL frontend library.
 */
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    int         nFields;
    int         i,
                j;

    /*
     * If the user supplies a parameter on the command line, use it as the
     * conninfo string; otherwise default to setting dbname=postgres and using
     * environment variables or defaults for all other connection parameters.
     */
    if (argc > 1)
        conninfo = argv[1];
    else
```

```
    conninfo = "dbname = postgres";

/* Make a connection to the database */
conn = PQconnectdb(conninfo);

/* Check to see that the backend connection was successfully made */
if (PQstatus(conn) != CONNECTION_OK)
{
    fprintf(stderr, "Connection to database failed: %s",
            PQerrorMessage(conn));
    exit_nicely(conn);
}

/*
 * Our test case here involves using a cursor, for which we must be inside
 * a transaction block.  We could do the whole thing with a single
 * PQexec() of "select * from pg_database", but that's too trivial to make
 * a good example.
 */

/* Start a transaction block */
res = PQexec(conn, "BEGIN");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "BEGIN command failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

/*
 * Should PQclear PGresult whenever it is no longer needed to avoid memory
 * leaks
 */
PQclear(res);

/*
 * Fetch rows from pg_database, the system catalog of databases
 */
res = PQexec(conn, "DECLARE myportal CURSOR FOR select * from pg_database");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "DECLARE CURSOR failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}
PQclear(res);

res = PQexec(conn, "FETCH ALL in myportal");
if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "FETCH ALL failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
```

```
    }

    /* first, print out the attribute names */
    nFields = PQnfields(res);
    for (i = 0; i < nFields; i++)
        printf("%-15s", PQfname(res, i));
    printf("\n\n");

    /* next, print out the rows */
    for (i = 0; i < PQntuples(res); i++)
    {
        for (j = 0; j < nFields; j++)
            printf("%-15s", PQgetvalue(res, i, j));
        printf("\n");
    }

    PQclear(res);

    /* close the portal ... we don't bother to check for errors ... */
    res = PQexec(conn, "CLOSE myportal");
    PQclear(res);

    /* end the transaction */
    res = PQexec(conn, "END");
    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

Example 29-2. libpq Example Program 2

```
/*
 * testlibpq2.c
 *      Test of the asynchronous notification interface
 *
 * Start this program, then from psql in another window do
 *   NOTIFY TBL2;
 * Repeat four times to get this program to exit.
 *
 * Or, if you want to get fancy, try this:
 * populate a database with the following commands
 * (provided in src/test/examples/testlibpq2.sql):
 *
 *   CREATE TABLE TBL1 (i int4);
 *
 *   CREATE TABLE TBL2 (i int4);
 *
 *   CREATE RULE r1 AS ON INSERT TO TBL1 DO
 *     (INSERT INTO TBL2 VALUES (new.i); NOTIFY TBL2);
```

```
 *
 * and do this four times:
 *
 *   INSERT INTO TBL1 VALUES (10);
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/time.h>
#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    PGnotify   *notify;
    int         nnotifies;

    /*
     * If the user supplies a parameter on the command line, use it as the
     * conninfo string; otherwise default to setting dbname=postgres and using
     * environment variables or defaults for all other connection parameters.
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* Make a connection to the database */
    conn = PQconnectdb(conninfo);

    /* Check to see that the backend connection was successfully made */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /*
     * Issue LISTEN command to enable notifications from the rule's NOTIFY.
     */
    res = PQexec(conn, "LISTEN TBL2");
```

```
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "LISTEN command failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

/*
 * should PQclear PGresult whenever it is no longer needed to avoid memory
 * leaks
 */
PQclear(res);

/* Quit after four notifies are received. */
nnotifies = 0;
while (nnotifies < 4)
{
    /*
     * Sleep until something happens on the connection.  We use select(2)
     * to wait for input, but you could also use poll() or similar
     * facilities.
     */
    int         sock;
    fd_set      input_mask;

    sock = PQsocket(conn);

    if (sock < 0)
        break;              /* shouldn't happen */

    FD_ZERO(&input_mask);
    FD_SET(sock, &input_mask);

    if (select(sock + 1, &input_mask, NULL, NULL, NULL) < 0)
    {
        fprintf(stderr, "select() failed: %s\n", strerror(errno));
        exit_nicely(conn);
    }

    /* Now check for input */
    PQconsumeInput(conn);
    while ((notify = PQnotifies(conn)) != NULL)
    {
        fprintf(stderr,
                "ASYNC NOTIFY of '%s' received from backend pid %d\n",
                notify->relname, notify->be_pid);
        PQfreemem(notify);
        nnotifies++;
    }
}

fprintf(stderr, "Done.\n");
```

```
    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

Example 29-3. libpq Example Program 3

```
/*
 * testlibpq3.c
 *      Test out-of-line parameters and binary I/O.
 *
 * Before running this, populate a database with the following commands
 * (provided in src/test/examples/testlibpq3.sql):
 *
 * CREATE TABLE test1 (i int4, t text, b bytea);
 *
 * INSERT INTO test1 values (1, 'joe"s place', '\\000\\001\\002\\003\\004');
 * INSERT INTO test1 values (2, 'ho there', '\\004\\003\\002\\001\\000');
 *
 * The expected output is:
 *
 * tuple 0: got
 *  i = (4 bytes) 1
 *  t = (11 bytes) 'joe's place'
 *  b = (5 bytes) \000\001\002\003\004
 *
 * tuple 0: got
 *  i = (4 bytes) 2
 *  t = (8 bytes) 'ho there'
 *  b = (5 bytes) \004\003\002\001\000
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include "libpq-fe.h"

/* for ntohl/htonl */
#include <netinet/in.h>
#include <arpa/inet.h>


static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

/*
 * This function prints a query result that is a binary-format fetch from
```

```
 * a table defined as in the comment above.  We split it out because the
 * main() function uses it twice.
 */
static void
show_binary_results(PGresult *res)
{
    int         i,
                j;
    int         i_fnum,
                t_fnum,
                b_fnum;

    /* Use PQfnumber to avoid assumptions about field order in result */
    i_fnum = PQfnumber(res, "i");
    t_fnum = PQfnumber(res, "t");
    b_fnum = PQfnumber(res, "b");

    for (i = 0; i < PQntuples(res); i++)
    {
        char       *iptr;
        char       *tptr;
        char       *bptr;
        int         blen;
        int         ival;

        /* Get the field values (we ignore possibility they are null!) */
        iptr = PQgetvalue(res, i, i_fnum);
        tptr = PQgetvalue(res, i, t_fnum);
        bptr = PQgetvalue(res, i, b_fnum);

        /*
         * The binary representation of INT4 is in network byte order, which
         * we'd better coerce to the local byte order.
         */
        ival = ntohl(*((uint32_t *) iptr));

        /*
         * The binary representation of TEXT is, well, text, and since libpq
         * was nice enough to append a zero byte to it, it'll work just fine
         * as a C string.
         *
         * The binary representation of BYTEA is a bunch of bytes, which could
         * include embedded nulls so we have to pay attention to field length.
         */
        blen = PQgetlength(res, i, b_fnum);

        printf("tuple %d: got\n", i);
        printf(" i = (%d bytes) %d\n",
               PQgetlength(res, i, i_fnum), ival);
        printf(" t = (%d bytes) '%s'\n",
               PQgetlength(res, i, t_fnum), tptr);
        printf(" b = (%d bytes) ", blen);
        for (j = 0; j < blen; j++)
```

```
            printf("\\%03o", bptr[j]);
        printf("\n\n");
    }
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    const char *paramValues[1];
    int         paramLengths[1];
    int         paramFormats[1];
    uint32_t    binaryIntVal;

    /*
     * If the user supplies a parameter on the command line, use it as the
     * conninfo string; otherwise default to setting dbname=postgres and using
     * environment variables or defaults for all other connection parameters.
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* Make a connection to the database */
    conn = PQconnectdb(conninfo);

    /* Check to see that the backend connection was successfully made */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /*
     * The point of this program is to illustrate use of PQexecParams() with
     * out-of-line parameters, as well as binary transmission of data.
     *
     * This first example transmits the parameters as text, but receives the
     * results in binary format.  By using out-of-line parameters we can
     * avoid a lot of tedious mucking about with quoting and escaping, even
     * though the data is text.  Notice how we don't have to do anything
     * special with the quote mark in the parameter value.
     */

    /* Here is our out-of-line parameter value */
    paramValues[0] = "joe's place";

    res = PQexecParams(conn,
                       "SELECT * FROM test1 WHERE t = $1",
```

```
                    1,       /* one param */
                    NULL,    /* let the backend deduce param type */
                    paramValues,
                    NULL,    /* don't need param lengths since text */
                    NULL,    /* default to all text params */
                    1);      /* ask for binary results */

if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

show_binary_results(res);

PQclear(res);

/*
 * In this second example we transmit an integer parameter in binary
 * form, and again retrieve the results in binary form.
 *
 * Although we tell PQexecParams we are letting the backend deduce
 * parameter type, we really force the decision by casting the parameter
 * symbol in the query text.  This is a good safety measure when sending
 * binary parameters.
 */

/* Convert integer value "2" to network byte order */
binaryIntVal = htonl((uint32_t) 2);

/* Set up parameter arrays for PQexecParams */
paramValues[0] = (char *) &binaryIntVal;
paramLengths[0] = sizeof(binaryIntVal);
paramFormats[0] = 1;        /* binary */

res = PQexecParams(conn,
                    "SELECT * FROM test1 WHERE i = $1::int4",
                    1,       /* one param */
                    NULL,    /* let the backend deduce param type */
                    paramValues,
                    paramLengths,
                    paramFormats,
                    1);      /* ask for binary results */

if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

show_binary_results(res);
```

```
    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

Chapter 30. Large Objects

PostgreSQL has a *large object* facility, which provides stream-style access to user data that is stored in a special large-object structure. Streaming access is useful when working with data values that are too large to manipulate conveniently as a whole.

This chapter describes the implementation and the programming and query language interfaces to PostgreSQL large object data. We use the libpq C library for the examples in this chapter, but most programming interfaces native to PostgreSQL support equivalent functionality. Other interfaces may use the large object interface internally to provide generic support for large values. This is not described here.

30.1. Introduction

All large objects are placed in a single system table called `pg_largeobject`. PostgreSQL also supports a storage system called "TOAST" that automatically stores values larger than a single database page into a secondary storage area per table. This makes the large object facility partially obsolete. One remaining advantage of the large object facility is that it allows values up to 2 GB in size, whereas TOASTed fields can be at most 1 GB. Also, large objects can be randomly modified using a read/write API that is more efficient than performing such operations using TOAST.

30.2. Implementation Features

The large object implementation breaks large objects up into "chunks" and stores the chunks in rows in the database. A B-tree index guarantees fast searches for the correct chunk number when doing random access reads and writes.

30.3. Client Interfaces

This section describes the facilities that PostgreSQL client interface libraries provide for accessing large objects. All large object manipulation using these functions *must* take place within an SQL transaction block. The PostgreSQL large object interface is modeled after the Unix file-system interface, with analogues of `open`, `read`, `write`, `lseek`, etc.

Client applications which use the large object interface in libpq should include the header file `libpq/libpq-fs.h` and link with the libpq library.

30.3.1. Creating a Large Object

The function

```
Oid lo_creat(PGconn *conn, int mode);
```

creates a new large object. The return value is the OID that was assigned to the new large object, or `InvalidOid` (zero) on failure. *`mode`* is unused and ignored as of PostgreSQL 8.1; however, for back-

wards compatibility with earlier releases it is best to set it to INV_READ, INV_WRITE, or INV_READ | INV_WRITE. (These symbolic constants are defined in the header file libpq/libpq-fs.h.)

An example:

```
inv_oid = lo_creat(conn, INV_READ|INV_WRITE);
```

The function

```
Oid lo_create(PGconn *conn, Oid lobjId);
```

also creates a new large object. The OID to be assigned can be specified by *lobjId*; if so, failure occurs if that OID is already in use for some large object. If *lobjId* is InvalidOid (zero) then lo_create assigns an unused OID (this is the same behavior as lo_creat). The return value is the OID that was assigned to the new large object, or InvalidOid (zero) on failure.

lo_create is new as of PostgreSQL 8.1; if this function is run against an older server version, it will fail and return InvalidOid.

An example:

```
inv_oid = lo_create(conn, desired_oid);
```

30.3.2. Importing a Large Object

To import an operating system file as a large object, call

```
Oid lo_import(PGconn *conn, const char *filename);
```

filename specifies the operating system name of the file to be imported as a large object. The return value is the OID that was assigned to the new large object, or InvalidOid (zero) on failure. Note that the file is read by the client interface library, not by the server; so it must exist in the client file system and be readable by the client application.

30.3.3. Exporting a Large Object

To export a large object into an operating system file, call

```
int lo_export(PGconn *conn, Oid lobjId, const char *filename);
```

The lobjId argument specifies the OID of the large object to export and the filename argument specifies the operating system name of the file. Note that the file is written by the client interface library, not by the server. Returns 1 on success, -1 on failure.

30.3.4. Opening an Existing Large Object

To open an existing large object for reading or writing, call

```
int lo_open(PGconn *conn, Oid lobjId, int mode);
```

The `lobjId` argument specifies the OID of the large object to open. The `mode` bits control whether the object is opened for reading (`INV_READ`), writing (`INV_WRITE`), or both. (These symbolic constants are defined in the header file `libpq/libpq-fs.h`.) A large object cannot be opened before it is created. `lo_open` returns a (non-negative) large object descriptor for later use in `lo_read`, `lo_write`, `lo_lseek`, `lo_tell`, and `lo_close`. The descriptor is only valid for the duration of the current transaction. On failure, -1 is returned.

The server currently does not distinguish between modes `INV_WRITE` and `INV_READ | INV_WRITE`: you are allowed to read from the descriptor in either case. However there is a significant difference between these modes and `INV_READ` alone: with `INV_READ` you cannot write on the descriptor, and the data read from it will reflect the contents of the large object at the time of the transaction snapshot that was active when `lo_open` was executed, regardless of later writes by this or other transactions. Reading from a descriptor opened with `INV_WRITE` returns data that reflects all writes of other committed transactions as well as writes of the current transaction. This is similar to the behavior of `SERIALIZABLE` versus `READ COMMITTED` transaction modes for ordinary SQL `SELECT` commands.

An example:

```
inv_fd = lo_open(conn, inv_oid, INV_READ|INV_WRITE);
```

30.3.5. Writing Data to a Large Object

The function

```
int lo_write(PGconn *conn, int fd, const char *buf, size_t len);
```

writes `len` bytes from `buf` to large object descriptor `fd`. The `fd` argument must have been returned by a previous `lo_open`. The number of bytes actually written is returned. In the event of an error, the return value is negative.

30.3.6. Reading Data from a Large Object

The function

```
int lo_read(PGconn *conn, int fd, char *buf, size_t len);
```

reads `len` bytes from large object descriptor `fd` into `buf`. The `fd` argument must have been returned by a previous `lo_open`. The number of bytes actually read is returned. In the event of an error, the return value is negative.

30.3.7. Seeking in a Large Object

To change the current read or write location associated with a large object descriptor, call

```
int lo_lseek(PGconn *conn, int fd, int offset, int whence);
```

This function moves the current location pointer for the large object descriptor identified by `fd` to the new location specified by `offset`. The valid values for `whence` are `SEEK_SET` (seek from object start), `SEEK_CUR` (seek from current position), and `SEEK_END` (seek from object end). The return value is the new location pointer, or -1 on error.

30.3.8. Obtaining the Seek Position of a Large Object

To obtain the current read or write location of a large object descriptor, call

```
int lo_tell(PGconn *conn, int fd);
```

If there is an error, the return value is negative.

30.3.9. Closing a Large Object Descriptor

A large object descriptor may be closed by calling

```
int lo_close(PGconn *conn, int fd);
```

where `fd` is a large object descriptor returned by `lo_open`. On success, `lo_close` returns zero. On error, the return value is negative.

Any large object descriptors that remain open at the end of a transaction will be closed automatically.

30.3.10. Removing a Large Object

To remove a large object from the database, call

```
int lo_unlink(PGconn *conn, Oid lobjId);
```

The `lobjId` argument specifies the OID of the large object to remove. Returns 1 if successful, -1 on failure.

30.4. Server-Side Functions

There are server-side functions callable from SQL that correspond to each of the client-side functions described above; indeed, for the most part the client-side functions are simply interfaces to the equiva-

lent server-side functions. The ones that are actually useful to call via SQL commands are `lo_creat`, `lo_create`, `lo_unlink`, `lo_import`, and `lo_export`. Here are examples of their use:

```
CREATE TABLE image (
    name            text,
    raster          oid
);

SELECT lo_creat(-1);       -- returns OID of new, empty large object

SELECT lo_create(43213);   -- attempts to create large object with OID 43213

SELECT lo_unlink(173454);  -- deletes large object with OID 173454

INSERT INTO image (name, raster)
    VALUES ('beautiful image', lo_import('/etc/motd'));

SELECT lo_export(image.raster, '/tmp/motd') FROM image
    WHERE name = 'beautiful image';
```

The server-side `lo_import` and `lo_export` functions behave considerably differently from their client-side analogs. These two functions read and write files in the server's file system, using the permissions of the database's owning user. Therefore, their use is restricted to superusers. In contrast, the client-side import and export functions read and write files in the client's file system, using the permissions of the client program. The client-side functions can be used by any PostgreSQL user.

30.5. Example Program

Example 30-1 is a sample program which shows how the large object interface in libpq can be used. Parts of the program are commented out but are left in the source for the reader's benefit. This program can also be found in `src/test/examples/testlo.c` in the source distribution.

Example 30-1. Large Objects with libpq Example Program

```
/*--------------------------------------------------------------
 *
 * testlo.c--
 *    test using large objects with libpq
 *
 * Copyright (c) 1994, Regents of the University of California
 *
 *--------------------------------------------------------------
 */
#include <stdio.h>
#include "libpq-fe.h"
#include "libpq/libpq-fs.h"

#define BUFSIZE         1024
```

```
/*
 * importFile
 *    import file "in_filename" into database as large object "lobjOid"
 *
 */
Oid
importFile(PGconn *conn, char *filename)
{
    Oid         lobjId;
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * open the file to be read in
     */
    fd = open(filename, O_RDONLY, 0666);
    if (fd < 0)
    {                           /* error */
        fprintf(stderr, "can't open unix file %s\n", filename);
    }

    /*
     * create the large object
     */
    lobjId = lo_creat(conn, INV_READ | INV_WRITE);
    if (lobjId == 0)
        fprintf(stderr, "can't create large object\n");

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);

    /*
     * read in from the Unix file and write to the inversion file
     */
    while ((nbytes = read(fd, buf, BUFSIZE)) > 0)
    {
        tmp = lo_write(conn, lobj_fd, buf, nbytes);
        if (tmp < nbytes)
            fprintf(stderr, "error while reading large object\n");
    }

    (void) close(fd);
    (void) lo_close(conn, lobj_fd);

    return lobjId;
}

void
pickout(PGconn *conn, Oid lobjId, int start, int len)
{
```

```
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nread;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "can't open large object %d\n",
                lobjId);
    }

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    nread = 0;
    while (len - nread > 0)
    {
        nbytes = lo_read(conn, lobj_fd, buf, len - nread);
        buf[nbytes] = ' ';
        fprintf(stderr, ">>> %s", buf);
        nread += nbytes;
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

void
overwrite(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nwritten;
    int         i;

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "can't open large object %d\n",
                lobjId);
    }

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    for (i = 0; i < len; i++)
        buf[i] = 'X';
    buf[i] = ' ';

    nwritten = 0;
    while (len - nwritten > 0)
```

```
    {
        nbytes = lo_write(conn, lobj_fd, buf + nwritten, len - nwritten);
        nwritten += nbytes;
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

/*
 * exportFile
 *    export large object "lobjOid" to file "out_filename"
 *
 */
void
exportFile(PGconn *conn, Oid lobjId, char *filename)
{
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * open the large object
     */
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "can't open large object %d\n",
                lobjId);
    }

    /*
     * open the file to be written to
     */
    fd = open(filename, O_CREAT | O_WRONLY, 0666);
    if (fd < 0)
    {                           /* error */
        fprintf(stderr, "can't open unix file %s\n",
                filename);
    }

    /*
     * read in from the inversion file and write to the Unix file
     */
    while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) > 0)
    {
        tmp = write(fd, buf, nbytes);
        if (tmp < nbytes)
        {
            fprintf(stderr, "error while writing %s\n",
                    filename);
```

```
        }
    }

    (void) lo_close(conn, lobj_fd);
    (void) close(fd);

    return;
}

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    char       *in_filename,
               *out_filename;
    char       *database;
    Oid         lobjOid;
    PGconn     *conn;
    PGresult   *res;

    if (argc != 4)
    {
        fprintf(stderr, "Usage: %s database_name in_filename out_filename\n",
                argv[0]);
        exit(1);
    }

    database = argv[1];
    in_filename = argv[2];
    out_filename = argv[3];

    /*
     * set up the connection
     */
    conn = PQsetdb(NULL, NULL, NULL, NULL, database);

    /* check to see that the backend connection was successfully made */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database '%s' failed.\n", database);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    res = PQexec(conn, "begin");
    PQclear(res);
```

```
    printf("importing file %s\n", in_filename);
/*  lobjOid = importFile(conn, in_filename); */
    lobjOid = lo_import(conn, in_filename);
/*
    printf("as large object %d.\n", lobjOid);

    printf("picking out bytes 1000-2000 of the large object\n");
    pickout(conn, lobjOid, 1000, 1000);

    printf("overwriting bytes 1000-2000 of the large object with X's\n");
    overwrite(conn, lobjOid, 1000, 1000);
*/

    printf("exporting large object to file %s\n", out_filename);
/*    exportFile(conn, lobjOid, out_filename); */
    lo_export(conn, lobjOid, out_filename);

    res = PQexec(conn, "end");
    PQclear(res);
    PQfinish(conn);
    exit(0);
}
```

Chapter 31. ECPG - Embedded SQL in C

This chapter describes the embedded SQL package for PostgreSQL. It was written by Linus Tolke (<linus@epact.se>) and Michael Meskes (<meskes@postgresql.org>). Originally it was written to work with C. It also works with C++, but it does not recognize all C++ constructs yet.

This documentation is quite incomplete. But since this interface is standardized, additional information can be found in many resources about SQL.

31.1. The Concept

An embedded SQL program consists of code written in an ordinary programming language, in this case C, mixed with SQL commands in specially marked sections. To build the program, the source code is first passed through the embedded SQL preprocessor, which converts it to an ordinary C program, and afterwards it can be processed by a C compiler.

Embedded SQL has advantages over other methods for handling SQL commands from C code. First, it takes care of the tedious passing of information to and from variables in your C program. Second, the SQL code in the program is checked at build time for syntactical correctness. Third, embedded SQL in C is specified in the SQL standard and supported by many other SQL database systems. The PostgreSQL implementation is designed to match this standard as much as possible, and it is usually possible to port embedded SQL programs written for other SQL databases to PostgreSQL with relative ease.

As already stated, programs written for the embedded SQL interface are normal C programs with special code inserted to perform database-related actions. This special code always has the form

```
EXEC SQL ...;
```

These statements syntactically take the place of a C statement. Depending on the particular statement, they may appear at the global level or within a function. Embedded SQL statements follow the case-sensitivity rules of normal SQL code, and not those of C.

The following sections explain all the embedded SQL statements.

31.2. Connecting to the Database Server

One connects to a database using the following statement:

```
EXEC SQL CONNECT TO target [AS connection-name] [USER user-name];
```

The *target* can be specified in the following ways:

- *dbname*[@*hostname*][:*port*]
- tcp:postgresql://*hostname*[:*port*][/*dbname*][?*options*]
- unix:postgresql://*hostname*[:*port*][/*dbname*][?*options*]

- an SQL string literal containing one of the above forms
- a reference to a character variable containing one of the above forms (see examples)
- DEFAULT

If you specify the connection target literally (that is, not through a variable reference) and you don't quote the value, then the case-insensitivity rules of normal SQL are applied. In that case you can also double-quote the individual parameters separately as needed. In practice, it is probably less error-prone to use a (single-quoted) string literal or a variable reference. The connection target DEFAULT initiates a connection to the default database under the default user name. No separate user name or connection name may be specified in that case.

There are also different ways to specify the user name:

- *username*
- *username*/*password*
- *username* IDENTIFIED BY *password*
- *username* USING *password*

As above, the parameters *username* and *password* may be an SQL identifier, an SQL string literal, or a reference to a character variable.

The *connection-name* is used to handle multiple connections in one program. It can be omitted if a program uses only one connection. The most recently opened connection becomes the current connection, which is used by default when an SQL statement is to be executed (see later in this chapter).

Here are some examples of CONNECT statements:

```
EXEC SQL CONNECT TO mydb@sql.mydomain.com;

EXEC SQL CONNECT TO unix:postgresql://sql.mydomain.com/mydb AS myconnection USER john;

EXEC SQL BEGIN DECLARE SECTION;
const char *target = "mydb@sql.mydomain.com";
const char *user = "john";
EXEC SQL END DECLARE SECTION;
 ...
EXEC SQL CONNECT TO :target USER :user;
```

The last form makes use of the variant referred to above as character variable reference. You will see in later sections how C variables can be used in SQL statements when you prefix them with a colon.

Be advised that the format of the connection target is not specified in the SQL standard. So if you want to develop portable applications, you might want to use something based on the last example above to encapsulate the connection target string somewhere.

31.3. Closing a Connection

To close a connection, use the following statement:

```
EXEC SQL DISCONNECT [connection];
```

The *connection* can be specified in the following ways:

- *connection-name*
- DEFAULT
- CURRENT
- ALL

If no connection name is specified, the current connection is closed.

It is good style that an application always explicitly disconnect from every connection it opened.

31.4. Running SQL Commands

Any SQL command can be run from within an embedded SQL application. Below are some examples of how to do that.

Creating a table:

```
EXEC SQL CREATE TABLE foo (number integer, ascii char(16));
EXEC SQL CREATE UNIQUE INDEX num1 ON foo(number);
EXEC SQL COMMIT;
```

Inserting rows:

```
EXEC SQL INSERT INTO foo (number, ascii) VALUES (9999, 'doodad');
EXEC SQL COMMIT;
```

Deleting rows:

```
EXEC SQL DELETE FROM foo WHERE number = 9999;
EXEC SQL COMMIT;
```

Single-row select:

```
EXEC SQL SELECT foo INTO :FooBar FROM table1 WHERE ascii = 'doodad';
```

Select using cursors:

```
EXEC SQL DECLARE foo_bar CURSOR FOR
    SELECT number, ascii FROM foo
    ORDER BY ascii;
EXEC SQL OPEN foo_bar;
EXEC SQL FETCH foo_bar INTO :FooBar, DooDad;
...
```

```
EXEC SQL CLOSE foo_bar;
EXEC SQL COMMIT;
```

Updates:

```
EXEC SQL UPDATE foo
    SET ascii = 'foobar'
    WHERE number = 9999;
EXEC SQL COMMIT;
```

The tokens of the form `:something` are *host variables*, that is, they refer to variables in the C program. They are explained in Section 31.6.

In the default mode, statements are committed only when `EXEC SQL COMMIT` is issued. The embedded SQL interface also supports autocommit of transactions (similar to libpq behavior) via the `-t` command-line option to `ecpg` (see below) or via the `EXEC SQL SET AUTOCOMMIT TO ON` statement. In autocommit mode, each command is automatically committed unless it is inside an explicit transaction block. This mode can be explicitly turned off using `EXEC SQL SET AUTOCOMMIT TO OFF`.

31.5. Choosing a Connection

The SQL statements shown in the previous section are executed on the current connection, that is, the most recently opened one. If an application needs to manage multiple connections, then there are two ways to handle this.

The first option is to explicitly choose a connection for each SQL statement, for example

```
EXEC SQL AT connection-name SELECT ...;
```

This option is particularly suitable if the application needs to use several connections in mixed order.

If your application uses multiple threads of execution, they cannot share a connection concurrently. You must either explicitly control access to the connection (using mutexes) or use a connection for each thread. If each thread uses its own connection, you will need to use the AT clause to specify which connection the thread will use.

The second option is to execute a statement to switch the current connection. That statement is:

```
EXEC SQL SET CONNECTION connection-name;
```

This option is particularly convenient if many statements are to be executed on the same connection. It is not thread-aware.

31.6. Using Host Variables

In Section 31.4 you saw how you can execute SQL statements from an embedded SQL program. Some of those statements only used fixed values and did not provide a way to insert user-supplied values into

statements or have the program process the values returned by the query. Those kinds of statements are not really useful in real applications. This section explains in detail how you can pass data between your C program and the embedded SQL statements using a simple mechanism called *host variables*. In an embedded SQL program we consider the SQL statements to be *guests* in the C program code which is the *host language*. Therefore the variables of the C program are called *host variables*.

31.6.1. Overview

Passing data between the C program and the SQL statements is particularly simple in embedded SQL. Instead of having the program paste the data into the statement, which entails various complications, such as properly quoting the value, you can simply write the name of a C variable into the SQL statement, prefixed by a colon. For example:

```
EXEC SQL INSERT INTO sometable VALUES (:v1, 'foo', :v2);
```

This statements refers to two C variables named `v1` and `v2` and also uses a regular SQL string literal, to illustrate that you are not restricted to use one kind of data or the other.

This style of inserting C variables in SQL statements works anywhere a value expression is expected in an SQL statement.

31.6.2. Declare Sections

To pass data from the program to the database, for example as parameters in a query, or to pass data from the database back to the program, the C variables that are intended to contain this data need to be declared in specially marked sections, so the embedded SQL preprocessor is made aware of them.

This section starts with

```
EXEC SQL BEGIN DECLARE SECTION;
```

and ends with

```
EXEC SQL END DECLARE SECTION;
```

Between those lines, there must be normal C variable declarations, such as

```
int   x = 4;
char  foo[16], bar[16];
```

As you can see, you can optionally assign an initial value to the variable. The variable's scope is determined by the location of its declaring section within the program. You can also declare variables with the following syntax which implicitly creates a declare section:

```
EXEC SQL int i = 4;
```

You can have as many declare sections in a program as you like.

The declarations are also echoed to the output file as normal C variables, so there's no need to declare them again. Variables that are not intended to be used in SQL commands can be declared normally outside these special sections.

The definition of a structure or union also must be listed inside a `DECLARE` section. Otherwise the preprocessor cannot handle these types since it does not know the definition.

31.6.3. Different types of host variables

As a host variable you can also use arrays, typedefs, structs and pointers. Moreover there are special types of host variables that exist only in ECPG.

A few examples on host variables:

Arrays

One of the most common uses of an array declaration is probably the allocation of a char array as in

```
EXEC SQL BEGIN DECLARE SECTION;
    char str[50];
EXEC SQL END DECLARE SECTION;
```

Note that you have to take care of the length for yourself. If you use this host variable as the target variable of a query which returns a string with more than 49 characters, a buffer overflow occurs.

Typedefs

Use the `typedef` keyword to map new types to already existing types.

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef char mychartype[40];
    typedef long serial_t;
EXEC SQL END DECLARE SECTION;
```

Note that you could also use

```
EXEC SQL TYPE serial_t IS long;
```

This declaration does not need to be part of a declare section.

Pointers

You can declare pointers to the most common types. Note however that you can not use pointers as target variables of queries without auto-allocation. See Section 31.10 for more information on auto-allocation.

```
EXEC SQL BEGIN DECLARE SECTION;
    int   *intp;
    char **charp;
EXEC SQL END DECLARE SECTION;
```

Special types of variables

ECPG contains some special types that help you to interact easily with data from the SQL server. For example it has implemented support for the `varchar`, `numeric`, `date`, `timestamp`, and `interval` types. Section 31.8 contains basic functions to deal with those types, such that you do not need to send a query to the SQL server just for adding an interval to a timestamp for example.

The special type `VARCHAR` is converted into a named `struct` for every variable. A declaration like

```
VARCHAR var[180];
```

is converted into

```
struct varchar_var { int len; char arr[180]; } var;
```

This structure is suitable for interfacing with SQL datums of type `varchar`.

31.6.4. `SELECT INTO` and `FETCH INTO`

Now you should be able to pass data generated by your program into an SQL command. But how do you retrieve the results of a query? For that purpose, embedded SQL provides special variants of the usual commands `SELECT` and `FETCH`. These commands have a special `INTO` clause that specifies which host variables the retrieved values are to be stored in.

Here is an example:

```
/*
 * assume this table:
 * CREATE TABLE test1 (a int, b varchar(50));
 */

EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

 ...

EXEC SQL SELECT a, b INTO :v1, :v2 FROM test;
```

So the `INTO` clause appears between the select list and the `FROM` clause. The number of elements in the select list and the list after `INTO` (also called the target list) must be equal.

Here is an example using the command `FETCH`:

```
EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

 ...

EXEC SQL DECLARE foo CURSOR FOR SELECT a, b FROM test;

 ...

do {
    ...
    EXEC SQL FETCH NEXT FROM foo INTO :v1, :v2;
    ...
} while (...);
```

Here the `INTO` clause appears after all the normal clauses.

Both of these methods only allow retrieving one row at a time. If you need to process result sets that potentially contain more than one row, you need to use a cursor, as shown in the second example.

31.6.5. Indicators

The examples above do not handle null values. In fact, the retrieval examples will raise an error if they fetch a null value from the database. To be able to pass null values to the database or retrieve null values from the database, you need to append a second host variable specification to each host variable that contains data. This second host variable is called the *indicator* and contains a flag that tells whether the datum is null, in which case the value of the real host variable is ignored. Here is an example that handles the retrieval of null values correctly:

```
EXEC SQL BEGIN DECLARE SECTION;
VARCHAR val;
int val_ind;
EXEC SQL END DECLARE SECTION:

 ...

EXEC SQL SELECT b INTO :val :val_ind FROM test1;
```

The indicator variable `val_ind` will be zero if the value was not null, and it will be negative if the value was null.

The indicator has another function: if the indicator value is positive, it means that the value is not null, but it was truncated when it was stored in the host variable.

31.7. Dynamic SQL

In many cases, the particular SQL statements that an application has to execute are known at the time the application is written. In some cases, however, the SQL statements are composed at run time or provided by an external source. In these cases you cannot embed the SQL statements directly into the C source code, but there is a facility that allows you to call arbitrary SQL statements that you provide in a string variable.

The simplest way to execute an arbitrary SQL statement is to use the command `EXECUTE IMMEDIATE`. For example:

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "CREATE TABLE test1 (...);";
EXEC SQL END DECLARE SECTION;

EXEC SQL EXECUTE IMMEDIATE :stmt;
```

You may not execute statements that retrieve data (e.g., `SELECT`) this way.

A more powerful way to execute arbitrary SQL statements is to prepare them once and execute the prepared statement as often as you like. It is also possible to prepare a generalized version of a statement and then execute specific versions of it by substituting parameters. When preparing the statement, write question marks where you want to substitute parameters later. For example:

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "INSERT INTO test1 VALUES(?, ?);";
```

```
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
 ...
EXEC SQL EXECUTE mystmt USING 42, 'foobar';
```

If the statement you are executing returns values, then add an `INTO` clause:

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "SELECT a, b, c FROM test1 WHERE a > ?";
int v1, v2;
VARCHAR v3;
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
 ...
EXEC SQL EXECUTE mystmt INTO v1, v2, v3 USING 37;
```

An `EXECUTE` command may have an `INTO` clause, a `USING` clause, both, or neither.

When you don't need the prepared statement anymore, you should deallocate it:

```
EXEC SQL DEALLOCATE PREPARE name;
```

31.8. pgtypes library

The pgtypes library maps PostgreSQL database types to C equivalents that can be used in C programs. It also offers functions to do basic calculations with those types within C, i.e. without the help of the PostgreSQL server. See the following example:

```
EXEC SQL BEGIN DECLARE SECTION;
   date date1;
   timestamp ts1, tsout;
   interval iv1;
   char *out;
EXEC SQL END DECLARE SECTION;

PGTYPESdate_today(&date1);
EXEC SQL SELECT started, duration INTO :ts1, :iv1 FROM datetbl WHERE d=:date1;
PGTYPEStimestamp_add_interval(&ts1, &iv1, &tsout);
out = PGTYPEStimestamp_to_asc(&tsout);
printf("Started + duration: %s\n", out);
free(out);
```

31.8.1. The numeric type

The numeric type offers to do calculations with arbitrary precision. See Section 8.1 for the equivalent type in the PostgreSQL server. Because of the arbitrary precision this variable needs to be able to expand and shrink dynamically. That's why you can only create variables on the heap by means of the `PGTYPESnumeric_new` and `PGTYPESnumeric_free` functions. The decimal type, which is similar but limited in the precision, can be created on the stack as well as on the heap.

The following functions can be used to work with the numeric type:

`PGTYPESnumeric_new`

Request a pointer to a newly allocated numeric variable.

```
numeric *PGTYPESnumeric_new(void);
```

`PGTYPESnumeric_free`

Free a numeric type, release all of its memory.

```
void PGTYPESnumeric_free(numeric *var);
```

`PGTYPESnumeric_from_asc`

Parse a numeric type from its string notation.

```
numeric *PGTYPESnumeric_from_asc(char *str, char **endptr);
```

Valid formats are for example: `-2`, `.794`, `+3.44`, `592.49E07` or `-32.84e-4`. If the value could be parsed successfully, a valid pointer is returned, else the NULL pointer. At the moment ecpg always parses the complete string and so it currently does not support to store the address of the first invalid character in `*endptr`. You can safely set `endptr` to NULL.

`PGTYPESnumeric_to_asc`

Returns a pointer to a string allocated by `malloc` that contains the string representation of the numeric type `num`.

```
char *PGTYPESnumeric_to_asc(numeric *num, int dscale);
```

The numeric value will be printed with `dscale` decimal digits, with rounding applied if necessary.

`PGTYPESnumeric_add`

Add two numeric variables into a third one.

```
int PGTYPESnumeric_add(numeric *var1, numeric *var2, numeric *result);
```

The function adds the variables `var1` and `var2` into the result variable `result`. The function returns 0 on success and -1 in case of error.

`PGTYPESnumeric_sub`

Subtract two numeric variables and return the result in a third one.

```
int PGTYPESnumeric_sub(numeric *var1, numeric *var2, numeric *result);
```

The function subtracts the variable `var2` from the variable `var1`. The result of the operation is stored in the variable `result`. The function returns 0 on success and -1 in case of error.

`PGTYPESnumeric_mul`

Multiply two numeric variables and return the result in a third one.

```
int PGTYPESnumeric_mul(numeric *var1, numeric *var2, numeric *result);
```

The function multiplies the variables `var1` and `var2`. The result of the operation is stored in the variable `result`. The function returns 0 on success and -1 in case of error.

`PGTYPESnumeric_div`

Divide two numeric variables and return the result in a third one.

```
int PGTYPESnumeric_div(numeric *var1, numeric *var2, numeric *result);
```

The function divides the variables `var1` by `var2`. The result of the operation is stored in the variable `result`. The function returns 0 on success and -1 in case of error.

`PGTYPESnumeric_cmp`

Compare two numeric variables.

```
int PGTYPESnumeric_cmp(numeric *var1, numeric *var2)
```

This function compares two numeric variables. In case of error, `INT_MAX` is returned. On success, the function returns one of three possible results:

- 1, if `var1` is bigger than `var2`
- -1, if `var1` is smaller than `var2`
- 0, if `var1` and `var2` are equal

`PGTYPESnumeric_from_int`

Convert an int variable to a numeric variable.

```
int PGTYPESnumeric_from_int(signed int int_val, numeric *var);
```

This function accepts a variable of type signed int and stores it in the numeric variable `var`. Upon success, 0 is returned and -1 in case of a failure.

`PGTYPESnumeric_from_long`

Convert a long int variable to a numeric variable.

```
int PGTYPESnumeric_from_long(signed long int long_val, numeric *var);
```

This function accepts a variable of type signed long int and stores it in the numeric variable `var`. Upon success, 0 is returned and -1 in case of a failure.

`PGTYPESnumeric_copy`

Copy over one numeric variable into another one.

```
int PGTYPESnumeric_copy(numeric *src, numeric *dst);
```

This function copies over the value of the variable that `src` points to into the variable that `dst` points to. It returns 0 on success and -1 if an error occurs.

`PGTYPESnumeric_from_double`

Convert a variable of type double to a numeric.

```
int  PGTYPESnumeric_from_double(double d, numeric *dst);
```

This function accepts a variable of type double and stores the result in the variable that `dst` points to. It returns 0 on success and -1 if an error occurs.

`PGTYPESnumeric_to_double`

Convert a variable of type numeric to double.

```
int PGTYPESnumeric_to_double(numeric *nv, double *dp)
```

The function converts the numeric value from the variable that `nv` points to into the double variable that `dp` points to. It returns 0 on success and -1 if an error occurs, including overflow. On overflow, the global variable `errno` will be set to `PGTYPES_NUM_OVERFLOW` additionally.

`PGTYPESnumeric_to_int`

Convert a variable of type numeric to int.

```
int PGTYPESnumeric_to_int(numeric *nv, int *ip);
```

The function converts the numeric value from the variable that `nv` points to into the integer variable that `ip` points to. It returns 0 on success and -1 if an error occurs, including overflow. On overflow, the global variable `errno` will be set to `PGTYPES_NUM_OVERFLOW` additionally.

`PGTYPESnumeric_to_long`

Convert a variable of type numeric to long.

```
int PGTYPESnumeric_to_long(numeric *nv, long *lp);
```

The function converts the numeric value from the variable that `nv` points to into the long integer variable that `lp` points to. It returns 0 on success and -1 if an error occurs, including overflow. On overflow, the global variable `errno` will be set to `PGTYPES_NUM_OVERFLOW` additionally.

`PGTYPESnumeric_to_decimal`

Convert a variable of type numeric to decimal.

```
int PGTYPESnumeric_to_decimal(numeric *src, decimal *dst);
```

The function converts the numeric value from the variable that `src` points to into the decimal variable that `dst` points to. It returns 0 on success and -1 if an error occurs, including overflow. On overflow, the global variable `errno` will be set to `PGTYPES_NUM_OVERFLOW` additionally.

`PGTYPESnumeric_from_decimal`

Convert a variable of type decimal to numeric.

```
int PGTYPESnumeric_from_decimal(decimal *src, numeric *dst);
```

The function converts the decimal value from the variable that `src` points to into the numeric variable that `dst` points to. It returns 0 on success and -1 if an error occurs. Since the decimal type is implemented as a limited version of the numeric type, overflow can not occur with this conversion.

31.8.2. The date type

The date type in C enables your programs to deal with data of the SQL type date. See Section 8.5 for the equivalent type in the PostgreSQL server.

The following functions can be used to work with the date type:

`PGTYPESdate_from_timestamp`

Extract the date part from a timestamp.

```
date PGTYPESdate_from_timestamp(timestamp dt);
```

The function receives a timestamp as its only argument and returns the extracted date part from this timestamp.

`PGTYPESdate_from_asc`

Parse a date from its textual representation.

```
date PGTYPESdate_from_asc(char *str, char **endptr);
```

The function receives a C char* string `str` and a pointer to a C char* string `endptr`. At the moment ecpg always parses the complete string and so it currently does not support to store the address of the first invalid character in `*endptr`. You can safely set `endptr` to NULL.

Note that the function always assumes MDY-formatted dates and there is currently no variable to change that within ecpg.

The following input formats are allowed:

Table 31-1. Valid input formats for `PGTYPESdate_from_asc`

Input	Result
January 8, 1999	January 8, 1999
1999-01-08	January 8, 1999
1/8/1999	January 8, 1999
1/18/1999	January 18, 1999
01/02/03	February 1, 2003
1999-Jan-08	January 8, 1999
Jan-08-1999	January 8, 1999
08-Jan-1999	January 8, 1999
99-Jan-08	January 8, 1999
08-Jan-99	January 8, 1999
08-Jan-06	January 8, 2006
Jan-08-99	January 8, 1999
19990108	ISO 8601; January 8, 1999
990108	ISO 8601; January 8, 1999
1999.008	year and day of year
J2451187	Julian day
January 8, 99 BC	year 99 before the Common Era

`PGTYPESdate_to_asc`

Return the textual representation of a date variable.

```
char *PGTYPESdate_to_asc(date dDate);
```

The function receives the date `dDate` as its only parameter. It will output the date in the form `1999-01-18`, i.e. in the `YYYY-MM-DD` format.

`PGTYPESdate_julmdy`

Extract the values for the day, the month and the year from a variable of type date.

```
void PGTYPESdate_julmdy(date d, int *mdy);
```

The function receives the date `d` and a pointer to an array of 3 integer values `mdy`. The variable name indicates the sequential order: `mdy[0]` will be set to contain the number of the month, `mdy[1]` will be set to the value of the day and `mdy[2]` will contain the year.

`PGTYPESdate_mdyjul`

Create a date value from an array of 3 integers that specify the day, the month and the year of the date.

```
void PGTYPESdate_mdyjul(int *mdy, date *jdate);
```

The function receives the array of the 3 integers (`mdy`) as its first argument and as its second argument a pointer to a variable of type date that should hold the result of the operation.

`PGTYPESdate_dayofweek`

Return a number representing the day of the week for a date value.

```
int PGTYPESdate_dayofweek(date d);
```

The function receives the date variable `d` as its only argument and returns an integer that indicates the day of the week for this date.

- 0 - Sunday
- 1 - Monday
- 2 - Tuesday
- 3 - Wednesday
- 4 - Thursday
- 5 - Friday
- 6 - Saturday

`PGTYPESdate_today`

Get the current date.

```
void PGTYPESdate_today(date *d);
```

The function receives a pointer to a date variable (`d`) that it sets to the current date.

`PGTYPESdate_fmt_asc`

Convert a variable of type date to its textual representation using a format mask.

```
int PGTYPESdate_fmt_asc(date dDate, char *fmtstring, char *outbuf);
```

The function receives the date to convert (`dDate`), the format mask (`fmtstring`) and the string that will hold the textual representation of the date (`outbuf`).

On success, 0 is returned and a negative value if an error occurred.

The following literals are the field specifiers you can use:

- `dd` - The number of the day of the month.
- `mm` - The number of the month of the year.
- `yy` - The number of the year as a two digit number.
- `yyyy` - The number of the year as a four digit number.
- `ddd` - The name of the day (abbreviated).

- mmm - The name of the month (abbreviated).

All other characters are copied 1:1 to the output string.

The following table indicates a few possible formats. This will give you an idea of how to use this function. All output lines are based on the same date: November, 23rd, 1959.

Table 31-2. Valid input formats for PGTYPESdate_fmt_asc

fmt	result
mmddyy	112359
ddmmyy	231159
yymmdd	591123
yy/mm/dd	59/11/23
yy mm dd	59 11 23
yy.mm.dd	59.11.23
.mm.yyyy.dd.	.11.1959.23.
mmm. dd, yyyy	Nov. 23, 1959
mmm dd yyyy	Nov 23 1959
yyyy dd mm	1959 23 11
ddd, mmm. dd, yyyy	Mon, Nov. 23, 1959
(ddd) mmm. dd, yyyy	(Mon) Nov. 23, 1959

PGTYPESdate_defmt_asc

Use a format mask to convert a C char* string to a value of type date.

```
int PGTYPESdate_defmt_asc(date *d, char *fmt, char *str);
```

The function receives a pointer to the date value that should hold the result of the operation (d), the format mask to use for parsing the date (fmt) and the C char* string containing the textual representation of the date (str). The textual representation is expected to match the format mask. However you do not need to have a 1:1 mapping of the string to the format mask. The function only analyzes the sequential order and looks for the literals yy or yyyy that indicate the position of the year, mm to indicate the position of the month and dd to indicate the position of the day.

The following table indicates a few possible formats. This will give you an idea of how to use this function.

Table 31-3. Valid input formats for rdefmtdate

fmt	str	result
ddmmyy	21-2-54	1954-02-21
ddmmyy	2-12-54	1954-12-02
ddmmyy	20111954	1954-11-20
ddmmyy	130464	1964-04-13
mmm.dd.yyyy	MAR-12-1967	1967-03-12
yy/mm/dd	1954, February 3rd	1954-02-03

fmt	str	result
mmm.dd.yyyy	041269	1969-04-12
yy/mm/dd	In the year 2525, in the month of July, mankind will be alive on the 28th day	2525-07-28
dd-mm-yy	I said on the 28th of July in the year 2525	2525-07-28
mmm.dd.yyyy	9/14/58	1958-09-14
yy/mm/dd	47/03/29	1947-03-29
mmm.dd.yyyy	oct 28 1975	1975-10-28
mmddyy	Nov 14th, 1985	1985-11-14

31.8.3. The timestamp type

The timestamp type in C enables your programs to deal with data of the SQL type timestamp. See Section 8.5 for the equivalent type in the PostgreSQL server.

The following functions can be used to work with the timestamp type:

`PGTYPEStimestamp_from_asc`

Parse a timestamp from its textual representation into a timestamp variable.

```
timestamp PGTYPEStimestamp_from_asc(char *str, char **endptr);
```

The function receives the string to parse (`str`) and a pointer to a C char* (`endptr`). At the moment ecpg always parses the complete string and so it currently does not support to store the address of the first invalid character in `*endptr`. You can safely set `endptr` to NULL.

The function returns the parsed timestamp on success. On error, `PGTYPESInvalidTimestamp` is returned and errno is set to `PGTYPES_TS_BAD_TIMESTAMP`. See *`PGTYPESInvalidTimestamp`* for important notes on this value.

In general, the input string can contain any combination of an allowed date specification, a whitespace character and an allowed time specification. Note that timezones are not supported by ecpg. It can parse them but does not apply any calculation as the PostgreSQL server does for example. Timezone specifiers are silently discarded.

The following table contains a few examples for input strings:

Table 31-4. Valid input formats for `PGTYPEStimestamp_from_asc`

Input	Result
1999-01-08 04:05:06	1999-01-08 04:05:06
January 8 04:05:06 1999 PST	1999-01-08 04:05:06

Input	Result
1999-Jan-08 04:05:06.789-8	1999-01-08 04:05:06.789 (time zone specifier ignored)
J2451187 04:05-08:00	1999-01-08 04:05:00 (time zone specifier ignored)

PGTYPEStimestamp_to_asc

Converts a date to a C char* string.

```
char *PGTYPEStimestamp_to_asc(timestamp tstamp);
```

The function receives the timestamp `tstamp` as its only argument and returns an allocated string that contains the textual representation of the timestamp.

PGTYPEStimestamp_current

Retrieve the current timestamp.

```
void PGTYPEStimestamp_current(timestamp *ts);
```

The function retrieves the current timestamp and saves it into the timestamp variable that `ts` points to.

PGTYPEStimestamp_fmt_asc

Convert a timestamp variable to a C char* using a format mask.

```
int PGTYPEStimestamp_fmt_asc(timestamp *ts, char *output, int str_len, char *fmtstr);
```

The function receives a pointer to the timestamp to convert as its first argument (`ts`), a pointer to the output buffer (`output`), the maximal length that has been allocated for the output buffer (`str_len`) and the format mask to use for the conversion (`fmtstr`).

Upon success, the function returns 0 and a negative value if an error occurred.

You can use the following format specifiers for the format mask. The format specifiers are the same ones that are used in the `strftime` function in libc. Any non-format specifier will be copied into the output buffer.

- `%A` - is replaced by national representation of the full weekday name.
- `%a` - is replaced by national representation of the abbreviated weekday name.
- `%B` - is replaced by national representation of the full month name.
- `%b` - is replaced by national representation of the abbreviated month name.
- `%C` - is replaced by (year / 100) as decimal number; single digits are preceded by a zero.
- `%c` - is replaced by national representation of time and date.
- `%D` - is equivalent to `%m/%d/%y`.
- `%d` - is replaced by the day of the month as a decimal number (01-31).
- `%E* %O*` - POSIX locale extensions. The sequences `%Ec %EC %Ex %EX %Ey %EY %Od %Oe %OH %OI %Om %OM %OS %Ou %OU %OV %Ow %OW %Oy` are supposed to provide alternate representations.

  Additionally `%OB` implemented to represent alternative months names (used standalone, without day mentioned).

- `%e` - is replaced by the day of month as a decimal number (1-31); single digits are preceded by a blank.
- `%F` - is equivalent to `%Y-%m-%d`.
- `%G` - is replaced by a year as a decimal number with century. This year is the one that contains the greater part of the week (Monday as the first day of the week).
- `%g` - is replaced by the same year as in `%G`, but as a decimal number without century (00-99).
- `%H` - is replaced by the hour (24-hour clock) as a decimal number (00-23).
- `%h` - the same as `%b`.
- `%I` - is replaced by the hour (12-hour clock) as a decimal number (01-12).
- `%j` - is replaced by the day of the year as a decimal number (001-366).
- `%k` - is replaced by the hour (24-hour clock) as a decimal number (0-23); single digits are preceded by a blank.
- `%l` - is replaced by the hour (12-hour clock) as a decimal number (1-12); single digits are preceded by a blank.
- `%M` - is replaced by the minute as a decimal number (00-59).
- `%m` - is replaced by the month as a decimal number (01-12).
- `%n` - is replaced by a newline.
- `%O*` - the same as `%E*`.
- `%p` - is replaced by national representation of either "ante meridiem" or "post meridiem" as appropriate.
- `%R` - is equivalent to `%H:%M`.
- `%r` - is equivalent to `%I:%M:%S %p`.
- `%S` - is replaced by the second as a decimal number (00-60).
- `%s` - is replaced by the number of seconds since the Epoch, UTC.
- `%T` - is equivalent to `%H:%M:%S`
- `%t` - is replaced by a tab.
- `%U` - is replaced by the week number of the year (Sunday as the first day of the week) as a decimal number (00-53).
- `%u` - is replaced by the weekday (Monday as the first day of the week) as a decimal number (1-7).
- `%V` - is replaced by the week number of the year (Monday as the first day of the week) as a decimal number (01-53). If the week containing January 1 has four or more days in the new year, then it is week 1; otherwise it is the last week of the previous year, and the next week is week 1.
- `%v` - is equivalent to `%e-%b-%Y`.
- `%W` - is replaced by the week number of the year (Monday as the first day of the week) as a decimal number (00-53).
- `%w` - is replaced by the weekday (Sunday as the first day of the week) as a decimal number (0-6).
- `%X` - is replaced by national representation of the time.

- `%x` - is replaced by national representation of the date.
- `%Y` - is replaced by the year with century as a decimal number.
- `%y` - is replaced by the year without century as a decimal number (00-99).
- `%Z` - is replaced by the time zone name.
- `%z` - is replaced by the time zone offset from UTC; a leading plus sign stands for east of UTC, a minus sign for west of UTC, hours and minutes follow with two digits each and no delimiter between them (common form for RFC 822 date headers).
- `%+` - is replaced by national representation of the date and time.
- `%-*` - GNU libc extension. Do not do any padding when performing numerical outputs.
- $_* - GNU libc extension. Explicitly specify space for padding.
- `%0*` - GNU libc extension. Explicitly specify zero for padding.
- `%%` - is replaced by `%`.

`PGTYPEStimestamp_sub`

Subtract one timestamp from another one and save the result in a variable of type interval.

```
int PGTYPEStimestamp_sub(timestamp *ts1, timestamp *ts2, interval *iv);
```

The function will subtract the timestamp variable that `ts2` points to from the timestamp variable that `ts1` points to and will store the result in the interval variable that `iv` points to.

Upon success, the function returns 0 and a negative value if an error occurred.

`PGTYPEStimestamp_defmt_asc`

Parse a timestamp value from its textual representation using a formatting mask.

```
int PGTYPEStimestamp_defmt_asc(char *str, char *fmt, timestamp *d);
```

The function receives the textual representation of a timestamp in the variable `str` as well as the formatting mask to use in the variable `fmt`. The result will be stored in the variable that `d` points to.

If the formatting mask `fmt` is NULL, the function will fall back to the default formatting mask which is `%Y-%m-%d %H:%M:%S`.

This is the reverse function to *`PGTYPEStimestamp_fmt_asc`*. See the documentation there in order to find out about the possible formatting mask entries.

`PGTYPEStimestamp_add_interval`

Add an interval variable to a timestamp variable.

```
int PGTYPEStimestamp_add_interval(timestamp *tin, interval *span, timestamp *tout);
```

The function receives a pointer to a timestamp variable `tin` and a pointer to an interval variable `span`. It adds the interval to the timestamp and saves the resulting timestamp in the variable that `tout` points to.

Upon success, the function returns 0 and a negative value if an error occurred.

`PGTYPEStimestamp_sub_interval`

Subtract an interval variable from a timestamp variable.

```
int PGTYPEStimestamp_sub_interval(timestamp *tin, interval *span, timestamp *tout);
```

The function subtracts the interval variable that `span` points to from the timestamp variable that `tin` points to and saves the result into the variable that `tout` points to.

Upon success, the function returns 0 and a negative value if an error occurred.

31.8.4. The interval type

The interval type in C enables your programs to deal with data of the SQL type interval. See Section 8.5 for the equivalent type in the PostgreSQL server.

The following functions can be used to work with the interval type:

`PGTYPESinterval_new`

Return a pointer to a newly allocated interval variable.

```
interval *PGTYPESinterval_new(void);
```

`PGTYPESinterval_free`

Release the memory of a previously allocated interval variable.

```
void PGTYPESinterval_new(interval *intvl);
```

`PGTYPESinterval_from_asc`

Parse an interval from its textual representation.

```
interval *PGTYPESinterval_from_asc(char *str, char **endptr);
```

The function parses the input string `str` and returns a pointer to an allocated interval variable. At the moment ecpg always parses the complete string and so it currently does not support to store the address of the first invalid character in `*endptr`. You can safely set `endptr` to NULL.

`PGTYPESinterval_to_asc`

Convert a variable of type interval to its textual representation.

```
char *PGTYPESinterval_to_asc(interval *span);
```

The function converts the interval variable that `span` points to into a C char*. The output looks like this example: `@ 1 day 12 hours 59 mins 10 secs`.

`PGTYPESinterval_copy`

Copy a variable of type interval.

```
int PGTYPESinterval_copy(interval *intvlsrc, interval *intvldest);
```

The function copies the interval variable that `intvlsrc` points to into the variable that `intvldest` points to. Note that you need to allocate the memory for the destination variable before.

31.8.5. The decimal type

The decimal type is similar to the numeric type. However it is limited to a maximal precision of 30 significant digits. In contrast to the numeric type which can be created on the heap only, the decimal type can be created either on the stack or on the heap (by means of the functions PGTYPESdecimal_new()

and PGTYPESdecimal_free(). There are a lot of other functions that deal with the decimal type in the Informix compatibility mode described in Section 31.9.

The following functions can be used to work with the decimal type and are not only contained in the `libcompat` library.

`PGTYPESdecimal_new`

Request a pointer to a newly allocated decimal variable.

```
decimal *PGTYPESdecimal_new(void);
```

`PGTYPESdecimal_free`

Free a decimal type, release all of its memory.

```
void PGTYPESdecimal_free(decimal *var);
```

31.8.6. errno values of pgtypeslib

`PGTYPES_NUM_BAD_NUMERIC`

An argument should contain a numeric variable (or point to a numeric variable) but in fact its in-memory representation was invalid.

`PGTYPES_NUM_OVERFLOW`

An overflow occurred. Since the numeric type can deal with almost arbitrary precision, converting a numeric variable into other types might cause overflow.

`PGTYPES_NUM_OVERFLOW`

An underflow occurred. Since the numeric type can deal with almost arbitrary precision, converting a numeric variable into other types might cause underflow.

`PGTYPES_NUM_DIVIDE_ZERO`

A division by zero has been attempted.

`PGTYPES_DATE_BAD_DATE`

`PGTYPES_DATE_ERR_EARGS`

`PGTYPES_DATE_ERR_ENOSHORTDATE`

`PGTYPES_INTVL_BAD_INTERVAL`

`PGTYPES_DATE_ERR_ENOTDMY`

`PGTYPES_DATE_BAD_DAY`

`PGTYPES_DATE_BAD_MONTH`

PGTYPES_TS_BAD_TIMESTAMP

31.8.7. Special constants of pgtypeslib

PGTYPESInvalidTimestamp

A value of type timestamp representing an invalid time stamp. This is returned by the function PGTYPEStimestamp_from_asc on parse error. Note that due to the internal representation of the timestamp datatype, PGTYPESInvalidTimestamp is also a valid timestamp at the same time. It is set to 1899-12-31 23:59:59. In order to detect errors, make sure that your application does not only test for PGTYPESInvalidTimestamp but also for errno != 0 after each call to PGTYPEStimestamp_from_asc.

31.9. Informix compatibility mode

ecpg can be run in a so-called *Informix compatibility mode*. If this mode is active, it tries to behave as if it were the Informix precompiler for Informix E/SQL. Generally spoken this will allow you to use the dollar sign instead of the EXEC SQL primitive to introduce embedded SQL commands.

```
$int j = 3;
$CONNECT TO :dbname;
$CREATE TABLE test(i INT PRIMARY KEY, j INT);
$INSERT INTO test(i, j) VALUES (7, :j);
$COMMIT;
```

There are two compatiblity modes: INFORMIX, INFORMIX_SE

When linking programs that use this compatibility mode, remember to link against libcompat that is shipped with ecpg.

Besides the previously explained syntactic sugar, the Informix compatibility mode ports some functions for input, output and transformation of data as well as embedded SQL statements known from E/SQL to ecpg.

Informix compatibility mode is closely connected to the pgtypeslib library of ecpg. pgtypeslib maps SQL data types to data types within the C host program and most of the additional functions of the Informix compatibility mode allow you to operate on those C host program types. Note however that the extent of the compatibility is limited. It does not try to copy Informix behaviour; it allows you to do more or less the same operations and gives you functions that have the same name and the same basic behavior but it is no drop-in replacement if you are using Informix at the moment. Moreover, some of the data types are different. For example, PostgreSQL's datetime and interval types do not know about ranges like for example YEAR TO MINUTE so you won't find support in ecpg for that either.

31.9.1. Additional embedded SQL statements

`CLOSE DATABASE`

This statement closes the current connection. In fact, this is a synonym for ecpg's `DISCONNECT CURRENT`.

```
$CLOSE DATABASE;                /* close the current connection */
EXEC SQL CLOSE DATABASE;
```

31.9.2. Additional functions

`decadd`

Add two decimal type values.

```
int decadd(decimal *arg1, decimal *arg2, decimal *sum);
```

The function receives a pointer to the first operand of type decimal (`arg1`), a pointer to the second operand of type decimal (`arg2`) and a pointer to a value of type decimal that will contain the sum (`sum`). On success, the function returns 0. ECPG_INFORMIX_NUM_OVERFLOW is returned in case of overflow and ECPG_INFORMIX_NUM_UNDERFLOW in case of underflow. -1 is returned for other failures and errno is set to the respective errno number of the pgtypeslib.

`deccmp`

Compare two variables of type decimal.

```
int deccmp(decimal *arg1, decimal *arg2);
```

The function receives a pointer to the first decimal value (`arg1`), a pointer to the second decimal value (`arg2`) and returns an integer value that indicates which is the bigger value.

- 1, if the value that `arg1` points to is bigger than the value that `var2` points to
- -1, if the value that `arg1` points to is smaller than the value that `arg2` points to
- 0, if the value that `arg1` points to and the value that `arg2` points to are equal

`deccopy`

Copy a decimal value.

```
void deccopy(decimal *src, decimal *target);
```

The function receives a pointer to the decimal value that should be copied as the first argument (`src`) and a pointer to the target structure of type decimal (`target`) as the second argument.

`deccvasc`

Convert a value from its ASCII representation into a decimal type.

```
int deccvasc(char *cp, int len, decimal *np);
```

The function receives a pointer to string that contains the string representation of the number to be converted (`cp`) as well as its length `len`. `np` is a pointer to the decimal value that saves the result of the operation.

Valid formats are for example: `-2`, `.794`, `+3.44`, `592.49E07` or `-32.84e-4`.

The function returns 0 on success. If overflow or underflow occurred, `ECPG_INFORMIX_NUM_OVERFLOW` or `ECPG_INFORMIX_NUM_UNDERFLOW` is returned. If the ASCII representation could not be parsed, `ECPG_INFORMIX_BAD_NUMERIC` is returned or `ECPG_INFORMIX_BAD_EXPONENT` if this problem occurred while parsing the exponent.

`deccvdbl`

Convert a value of type double to a value of type decimal.

```
int deccvdbl(double dbl, decimal *np);
```

The function receives the variable of type double that should be converted as its first argument (`dbl`). As the second argument (`np`), the function receives a pointer to the decimal variable that should hold the result of the operation.

The function returns 0 on success and a negative value if the conversion failed.

`deccvint`

Convert a value of type int to a value of type decimal.

```
int deccvint(int in, decimal *np);
```

The function receives the variable of type int that should be converted as its first argument (`in`). As the second argument (`np`), the function receives a pointer to the decimal variable that should hold the result of the operation.

The function returns 0 on success and a negative value if the conversion failed.

`deccvlong`

Convert a value of type long to a value of type decimal.

```
int deccvlong(long lng, decimal *np);
```

The function receives the variable of type long that should be converted as its first argument (`lng`). As the second argument (`np`), the function receives a pointer to the decimal variable that should hold the result of the operation.

The function returns 0 on success and a negative value if the conversion failed.

`decdiv`

Divide two variables of type decimal.

```
int decdiv(decimal *n1, decimal *n2, decimal *result);
```

The function receives pointers to the variables that are the first (`n1`) and the second (`n2`) operands and calculates `n1`/`n2`. `result` is a pointer to the variable that should hold the result of the operation.

On success, 0 is returned and a negative value if the division fails. If overflow or underflow occurred, the function returns `ECPG_INFORMIX_NUM_OVERFLOW` or `ECPG_INFORMIX_NUM_UNDERFLOW` respectively. If an attempt to divide by zero is observed, the function returns `ECPG_INFORMIX_DIVIDE_ZERO`.

`decmul`

Multiply two decimal values.

```
int decmul(decimal *n1, decimal *n2, decimal *result);
```

The function receives pointers to the variables that are the first (`n1`) and the second (`n2`) operands and calculates `n1`*`n2`. `result` is a pointer to the variable that should hold the result of the operation.

On success, 0 is returned and a negative value if the multiplication fails. If overflow or underflow occurred, the function returns `ECPG_INFORMIX_NUM_OVERFLOW` or `ECPG_INFORMIX_NUM_UNDERFLOW` respectively.

`decsub`

Subtract one decimal value from another.

```
int decsub(decimal *n1, decimal *n2, decimal *result);
```

The function receives pointers to the variables that are the first (`n1`) and the second (`n2`) operands and calculates `n1-n2`. `result` is a pointer to the variable that should hold the result of the operation.

On success, 0 is returned and a negative value if the subtraction fails. If overflow or underflow occurred, the function returns `ECPG_INFORMIX_NUM_OVERFLOW` or `ECPG_INFORMIX_NUM_UNDERFLOW` respectively.

`dectoasc`

Convert a variable of type decimal to its ASCII representation in a C char* string.

```
int dectoasc(decimal *np, char *cp, int len, int right)
```

The function receives a pointer to a variable of type decimal (`np`) that it converts to its textual representation. `cp` is the buffer that should hold the result of the operation. The parameter `right` specifies, how many digits right of the decimal point should be included in the output. The result will be rounded to this number of decimal digits. Setting `right` to -1 indicates that all available decimal digits should be included in the output. If the length of the output buffer, which is indicated by `len` is not sufficient to hold the textual representation including the trailing NUL character, only a single `*` character is stored in the result and -1 is returned.

The function returns either -1 if the buffer `cp` was too small or `ECPG_INFORMIX_OUT_OF_MEMORY` if memory was exhausted.

`dectodbl`

Convert a variable of type decimal to a double.

```
int dectodbl(decimal *np, double *dblp);
```

The function receives a pointer to the decimal value to convert (`np`) and a pointer to the double variable that should hold the result of the operation (`dblp`).

On success, 0 is returned and a negative value if the conversion failed.

`dectoint`

Convert a variable to type decimal to an integer.

```
int dectoint(decimal *np, int *ip);
```

The function receives a pointer to the decimal value to convert (`np`) and a pointer to the integer variable that should hold the result of the operation (`ip`).

On success, 0 is returned and a negative value if the conversion failed. If an overflow occurred, `ECPG_INFORMIX_NUM_OVERFLOW` is returned.

Note that the ecpg implementation differs from the Informix implementation. Informix limits an integer to the range from -32767 to 32767, while the limits in the ecpg implementation depend on the architecture (`-INT_MAX .. INT_MAX`).

`dectolong`

Convert a variable to type decimal to a long integer.

```
int dectolong(decimal *np, long *lngp);
```

The function receives a pointer to the decimal value to convert (`np`) and a pointer to the long variable that should hold the result of the operation (`lngp`).

On success, 0 is returned and a negative value if the conversion failed. If an overflow occurred, `ECPG_INFORMIX_NUM_OVERFLOW` is returned.

Note that the ecpg implementation differs from the Informix implementation. Informix limits a long integer to the range from -2,147,483,647 to 2,147,483,647, while the limits in the ecpg implementation depend on the architecture (`-LONG_MAX .. LONG_MAX`).

`rdatestr`

Converts a date to a C char* string.

```
int rdatestr(date d, char *str);
```

The function receives two arguments, the first one is the date to convert (`d` and the second one is a pointer to the target string. The output format is always `yyyy-mm-dd`, so you need to allocate at least 11 bytes (including the NUL-terminator) for the string.

The function returns 0 on success and a negative value in case of error.

Note that ecpg's implementation differs from the Informix implementation. In Informix the format can be influenced by setting environment variables. In ecpg however, you cannot change the output format.

`rstrdate`

Parse the textual representation of a date.

```
int rstrdate(char *str, date *d);
```

The function receives the textual representation of the date to convert (`str`) and a pointer to a variable of type date (`d`). This function does not allow you to specify a format mask. It uses the default format mask of Informix which is `mm/dd/yyyy`. Internally, this function is implemented by means of `rdefmtdate`. Therefore, `rstrdate` is not faster and if you have the choice you should opt for `rdefmtdate` which allows you to specify the format mask explicitly.

The function returns the same values as `rdefmtdate`.

`rtoday`

Get the current date.

```
void rtoday(date *d);
```

The function receives a pointer to a date variable (`d`) that it sets to the current date.

Internally this function uses the *`PGTYPESdate_today`* function.

`rjulmdy`

Extract the values for the day, the month and the year from a variable of type date.

```
int rjulmdy(date d, short mdy[3]);
```

The function receives the date `d` and a pointer to an array of 3 short integer values `mdy`. The variable name indicates the sequential order: `mdy[0]` will be set to contain the number of the month, `mdy[1]` will be set to the value of the day and `mdy[2]` will contain the year.

The function always returns 0 at the moment.

Internally the function uses the *PGTYPESdate_julmdy* function.

`rdefmtdate`

Use a format mask to convert a character string to a value of type date.

```
int rdefmtdate(date *d, char *fmt, char *str);
```

The function receives a pointer to the date value that should hold the result of the operation (`d`), the format mask to use for parsing the date (`fmt`) and the C char* string containing the textual representation of the date (`str`). The textual representation is expected to match the format mask. However you do not need to have a 1:1 mapping of the string to the format mask. The function only analyzes the sequential order and looks for the literals `yy` or `yyyy` that indicate the position of the year, `mm` to indicate the position of the month and `dd` to indicate the position of the day.

The function returns the following values:

- 0 - The function terminated successfully.
- `ECPG_INFORMIX_ENOSHORTDATE` - The date does not contain delimiters between day, month and year. In this case the input string must be exactly 6 or 8 bytes long but isn't.
- `ECPG_INFORMIX_ENOTDMY` - The format string did not correctly indicate the sequential order of year, month and day.
- `ECPG_INFORMIX_BAD_DAY` - The input string does not contain a valid day.
- `ECPG_INFORMIX_BAD_MONTH` - The input string does not contain a valid month.
- `ECPG_INFORMIX_BAD_YEAR` - The input string does not contain a valid year.

Internally this function is implemented to use the *PGTYPESdate_defmt_asc* function. See the reference there for a table of example input.

`rfmtdate`

Convert a variable of type date to its textual representation using a format mask.

```
int rfmtdate(date d, char *fmt, char *str);
```

The function receives the date to convert (`d`), the format mask (`fmt`) and the string that will hold the textual representation of the date (`str`).

On success, 0 is returned and a negative value if an error occurred.

Internally this function uses the *PGTYPESdate_fmt_asc* function, see the reference there for examples.

`rmdyjul`

Create a date value from an array of 3 short integers that specify the day, the month and the year of the date.

```
int rmdyjul(short mdy[3], date *d);
```

The function receives the array of the 3 short integers (`mdy`) and a pointer to a variable of type date that should hold the result of the operation.

Currently the function returns always 0.

Internally the function is implemented to use the function *PGTYPESdate_mdyjul*.

`rdayofweek`

Return a number representing the day of the week for a date value.

```
int rdayofweek(date d);
```

The function receives the date variable `d` as its only argument and returns an integer that indicates the day of the week for this date.

- 0 - Sunday
- 1 - Monday
- 2 - Tuesday
- 3 - Wednesday
- 4 - Thursday
- 5 - Friday
- 6 - Saturday

Internally the function is implemented to use the function *`PGTYPESdate_dayofweek`*.

`dtcurrent`

Retrieve the current timestamp.

```
void dtcurrent(timestamp *ts);
```

The function retrieves the current timestamp and saves it into the timestamp variable that `ts` points to.

`dtcvasc`

Parses a timestamp from its textual representation in ANSI standard into a timestamp variable.

```
int dtcvasc(char *str, timestamp *ts);
```

The function receives the string to parse (`str`) and a pointer to the timestamp variable that should hold the result of the operation (`ts`).

The function returns 0 on success and a negative value in case of error.

Internally this function uses the *`PGTYPEStimestamp_from_asc`* function. See the reference there for a table with example inputs.

`dtcvfmtasc`

Parses a timestamp from its textual representation in ANSI standard using a format mask into a timestamp variable.

```
dtcvfmtasc(char *inbuf, char *fmtstr, timestamp *dtvalue)
```

The function receives the string to parse (`inbuf`), the format mask to use (`fmtstr`) and a pointer to the timestamp variable that should hold the result of the operation (`ts`).

This functions is implemented by means of the *`PGTYPEStimestamp_defmt_asc`*. See the documentation there for a list of format specifiers that can be used.

The function returns 0 on success and a negative value in case of error.

`dtsub`

Subtract one timestamp from another and return a variable of type interval.

```
int dtsub(timestamp *ts1, timestamp *ts2, interval *iv);
```

The function will subtract the timestamp variable that `ts2` points to from the timestamp variable that `ts1` points to and will store the result in the interval variable that `iv` points to.

Upon success, the function returns 0 and a negative value if an error occurred.

`dttoasc`

Convert a timestamp variable to a C char* string.

```
int dttoasc(timestamp *ts, char *output);
```

The function receives a pointer to the timestamp variable to convert (`ts`) and the string that should hold the result of the operation `output`). It converts `ts` to its textual representation in the ANSI SQL standard which is defined to be `YYYY-MM-DD HH:MM:SS`.

Upon success, the function returns 0 and a negative value if an error occurred.

`dttofmtasc`

Convert a timestamp variable to a C char* using a format mask.

```
int dttofmtasc(timestamp *ts, char *output, int str_len, char *fmtstr);
```

The function receives a pointer to the timestamp to convert as its first argument (`ts`), a pointer to the output buffer (`output`), the maximal length that has been allocated for the output buffer (`str_len`) and the format mask to use for the conversion (`fmtstr`).

Upon success, the function returns 0 and a negative value if an error occurred.

Internally, this function uses the *`PGTYPEStimestamp_fmt_asc`* function. See the reference there for information on what format mask specifiers can be used.

`intoasc`

Convert an interval variable to a C char* string.

```
int intoasc(interval *i, char *str);
```

The function receives a pointer to the interval variable to convert (`i`) and the string that should hold the result of the operation `str`). It converts `i` to its textual representation in the ANSI SQL standard which is defined to be `YYYY-MM-DD HH:MM:SS`.

Upon success, the function returns 0 and a negative value if an error occurred.

`rfmtlong`

Convert a long integer value to its textual representation using a format mask.

```
int rfmtlong(long lng_val, char *fmt, char *outbuf);
```

The function receives the long value `lng_val`, the format mask `fmt` and a pointer to the output buffer `outbuf`. It converts the long value according to the format mask to its textual representation.

The format mask can be composed of the following format specifying characters:

- `*` (asterisk) - if this position would be blank otherwise, fill it with an asterisk.
- `&` (ampersand) - if this position would be blank otherwise, fill it with a zero.
- `#` - turn leading zeroes into blanks.
- `<` - left-justify the number in the string.

- , (comma) - group numbers of four or more digits into groups of three digits separated by a comma.
- . (period) - this character separates the whole-number part of the number from the fractional part.
- - (minus) - the minus sign appears if the number is a negative value.
- + (plus) - the plus sign appears if the number is a positive value.
- (- this replaces the minus sign in front of the negative number. The minus sign will not appear.
-) - this character replaces the minus and is printed behind the negative value.
- $ - the currency symbol.

`rupshift`

Convert a string to upper case.

```
void rupshift(char *str);
```

The function receives a pointer to the string and transforms every lower case character to upper case.

`byleng`

Return the number of characters in a string without counting trailing blanks.

```
int byleng(char *str, int len);
```

The function expects a fixed-length string as its first argument (`str`) and its length as its second argument (`len`). It returns the number of significant characters, that is the length of the string without trailing blanks.

`ldchar`

Copy a fixed-length string into a null-terminated string.

```
void ldchar(char *src, int len, char *dest);
```

The function receives the fixed-length string to copy (`src`), its length (`len`) and a pointer to the destination memory (`dest`). Note that you need to reserve at least `len+1` bytes for the string that `dest` points to. The function copies at most `len` bytes to the new location (less if the source string has trailing blanks) and adds the null-terminator.

`rgetmsg`

```
int rgetmsg(int msgnum, char *s, int maxsize);
```

This function exists but is not implemented at the moment!

`rtypalign`

```
int rtypalign(int offset, int type);
```

This function exists but is not implemented at the moment!

`rtypmsize`

```
int rtypmsize(int type, int len);
```

This function exists but is not implemented at the moment!

`rtypwidth`

```
int rtypwidth(int sqltype, int sqllen);
```

This function exists but is not implemented at the moment!

`rsetnull`

Set a variable to NULL.

```
int rsetnull(int t, char *ptr);
```

The function receives an integer that indicates the type of the variable and a pointer to the variable itself that is casted to a C char* pointer.

The following types exist:

- `CCHARTYPE` - For a variable of type `char` or `char*`
- `CSHORTTYPE` - For a variable of type `short int`
- `CINTTYPE` - For a variable of type `int`
- `CBOOLTYPE` - For a variable of type `boolean`
- `CFLOATTYPE` - For a variable of type `float`
- `CLONGTYPE` - For a variable of type `long`
- `CDOUBLETYPE` - For a variable of type `double`
- `CDECIMALTYPE` - For a variable of type `decimal`
- `CDATETYPE` - For a variable of type `date`
- `CDTIMETYPE` - For a variable of type `timestamp`

Here is an example of a call to this function:

```
$char c[] = "abc       ";
$short s = 17;
$int i = -74874;

rsetnull(CCHARTYPE, (char *) c);
rsetnull(CSHORTTYPE, (char *) &s);
rsetnull(CINTTYPE, (char *) &i);
```

`risnull`

Test if a variable is NULL.

```
int risnull(int t, char *ptr);
```

The function receives the type of the variable to test (`t`) as well a pointer to this variable (`ptr`). Note that the latter needs to be casted to a char*. See the function *`rsetnull`* for a list of possible variable types.

Here is an example of how to use this function:

```
$char c[] = "abc       ";
$short s = 17;
$int i = -74874;

risnull(CCHARTYPE, (char *) c);
risnull(CSHORTTYPE, (char *) &s);
risnull(CINTTYPE, (char *) &i);
```

31.9.3. Additional constants

Note that all constants here describe errors and all of them are defined to represent negative values. In the descriptions of the different constants you can also find the value that the constants represent in the current implementation. However you should not rely on this number. You can however rely on the fact all of them are defined to represent negative values.

`ECPG_INFORMIX_NUM_OVERFLOW`

Functions return this value if an overflow occurred in a calculation. Internally it is defined to -1200 (the Informix definition).

`ECPG_INFORMIX_NUM_UNDERFLOW`

Functions return this value if an underflow occurred in a calculation. Internally it is defined to -1201 (the Informix definition).

`ECPG_INFORMIX_DIVIDE_ZERO`

Functions return this value if an attempt to divide by zero is observed. Internally it is defined to -1202 (the Informix definition).

`ECPG_INFORMIX_BAD_YEAR`

Functions return this value if a bad value for a year was found while parsing a date. Internally it is defined to -1204 (the Informix definition).

`ECPG_INFORMIX_BAD_MONTH`

Functions return this value if a bad value for a month was found while parsing a date. Internally it is defined to -1205 (the Informix definition).

`ECPG_INFORMIX_BAD_DAY`

Functions return this value if a bad value for a day was found while parsing a date. Internally it is defined to -1206 (the Informix definition).

`ECPG_INFORMIX_ENOSHORTDATE`

Functions return this value if a parsing routine needs a short date representation but did not get the date string in the right length. Internally it is defined to -1209 (the Informix definition).

`ECPG_INFORMIX_DATE_CONVERT`

Functions return this value if Internally it is defined to -1210 (the Informix definition).

`ECPG_INFORMIX_OUT_OF_MEMORY`

Functions return this value if Internally it is defined to -1211 (the Informix definition).

`ECPG_INFORMIX_ENOTDMY`

Functions return this value if a parsing routine was supposed to get a format mask (like `mmddyy`) but not all fields were listed correctly. Internally it is defined to -1212 (the Informix definition).

`ECPG_INFORMIX_BAD_NUMERIC`

Functions return this value either if a parsing routine cannot parse the textual representation for a numeric value because it contains errors or if a routine cannot complete a calculation involving numeric variables because at least one of the numeric variables is invalid. Internally it is defined to -1213 (the Informix definition).

ECPG_INFORMIX_BAD_EXPONENT

Functions return this value if Internally it is defined to -1216 (the Informix definition).

ECPG_INFORMIX_BAD_DATE

Functions return this value if Internally it is defined to -1218 (the Informix definition).

ECPG_INFORMIX_EXTRA_CHARS

Functions return this value if Internally it is defined to -1264 (the Informix definition).

31.10. Using SQL Descriptor Areas

An SQL descriptor area is a more sophisticated method for processing the result of a SELECT or FETCH statement. An SQL descriptor area groups the data of one row of data together with metadata items into one data structure. The metadata is particularly useful when executing dynamic SQL statements, where the nature of the result columns may not be known ahead of time.

An SQL descriptor area consists of a header, which contains information concerning the entire descriptor, and one or more item descriptor areas, which basically each describe one column in the result row.

Before you can use an SQL descriptor area, you need to allocate one:

```
EXEC SQL ALLOCATE DESCRIPTOR identifier;
```

The identifier serves as the "variable name" of the descriptor area. When you don't need the descriptor anymore, you should deallocate it:

```
EXEC SQL DEALLOCATE DESCRIPTOR identifier;
```

To use a descriptor area, specify it as the storage target in an INTO clause, instead of listing host variables:

```
EXEC SQL FETCH NEXT FROM mycursor INTO DESCRIPTOR mydesc;
```

Now how do you get the data out of the descriptor area? You can think of the descriptor area as a structure with named fields. To retrieve the value of a field from the header and store it into a host variable, use the following command:

```
EXEC SQL GET DESCRIPTOR name :hostvar = field;
```

Currently, there is only one header field defined: *COUNT*, which tells how many item descriptor areas exist (that is, how many columns are contained in the result). The host variable needs to be of an integer type. To get a field from the item descriptor area, use the following command:

```
EXEC SQL GET DESCRIPTOR name VALUE num :hostvar = field;
```

num can be a literal integer or a host variable containing an integer. Possible fields are:

`CARDINALITY` (integer)

number of rows in the result set

`DATA`

actual data item (therefore, the data type of this field depends on the query)

`DATETIME_INTERVAL_CODE` (integer)

?

`DATETIME_INTERVAL_PRECISION` (integer)

not implemented

`INDICATOR` (integer)

the indicator (indicating a null value or a value truncation)

`KEY_MEMBER` (integer)

not implemented

`LENGTH` (integer)

length of the datum in characters

`NAME` (string)

name of the column

`NULLABLE` (integer)

not implemented

`OCTET_LENGTH` (integer)

length of the character representation of the datum in bytes

`PRECISION` (integer)

precision (for type `numeric`)

`RETURNED_LENGTH` (integer)

length of the datum in characters

`RETURNED_OCTET_LENGTH` (integer)

length of the character representation of the datum in bytes

`SCALE` (integer)

scale (for type `numeric`)

`TYPE` (integer)

numeric code of the data type of the column

31.11. Error Handling

This section describes how you can handle exceptional conditions and warnings in an embedded SQL program. There are several nonexclusive facilities for this.

31.11.1. Setting Callbacks

One simple method to catch errors and warnings is to set a specific action to be executed whenever a particular condition occurs. In general:

```
EXEC SQL WHENEVER condition action;
```

condition can be one of the following:

`SQLERROR`

The specified action is called whenever an error occurs during the execution of an SQL statement.

`SQLWARNING`

The specified action is called whenever a warning occurs during the execution of an SQL statement.

`NOT FOUND`

The specified action is called whenever an SQL statement retrieves or affects zero rows. (This condition is not an error, but you might be interested in handling it specially.)

action can be one of the following:

`CONTINUE`

This effectively means that the condition is ignored. This is the default.

`GOTO` *label*
`GO TO` *label*

Jump to the specified label (using a C `goto` statement).

`SQLPRINT`

Print a message to standard error. This is useful for simple programs or during prototyping. The details of the message cannot be configured.

`STOP`

Call `exit(1)`, which will terminate the program.

`DO BREAK`

Execute the C statement `break`. This should only be used in loops or `switch` statements.

`CALL` *name* `(`*args*`)`
`DO` *name* `(`*args*`)`

Call the specified C functions with the specified arguments.

The SQL standard only provides for the actions `CONTINUE` and `GOTO` (and `GO TO`).

Here is an example that you might want to use in a simple program. It prints a simple message when a warning occurs and aborts the program when an error happens.

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLERROR STOP;
```

The statement `EXEC SQL WHENEVER` is a directive of the SQL preprocessor, not a C statement. The error or warning actions that it sets apply to all embedded SQL statements that appear below the point where the handler is set, unless a different action was set for the same condition between the first `EXEC SQL WHENEVER` and the SQL statement causing the condition, regardless of the flow of control in the C program. So neither of the two following C program excerpts will have the desired effect.

```
/*
 * WRONG
 */
int main(int argc, char *argv[])
{
    ...
    if (verbose) {
        EXEC SQL WHENEVER SQLWARNING SQLPRINT;
    }
    ...
    EXEC SQL SELECT ...;
    ...
}

/*
 * WRONG
 */
int main(int argc, char *argv[])
{
    ...
    set_error_handler();
    ...
    EXEC SQL SELECT ...;
    ...
}

static void set_error_handler(void)
{
    EXEC SQL WHENEVER SQLERROR STOP;
}
```

31.11.2. sqlca

For more powerful error handling, the embedded SQL interface provides a global variable with the name `sqlca` that has the following structure:

```
struct
{
    char sqlcaid[8];
    long sqlabc;
    long sqlcode;
    struct
    {
        int sqlerrml;
        char sqlerrmc[70];
    } sqlerrm;
    char sqlerrp[8];
    long sqlerrd[6];
    char sqlwarn[8];
    char sqlstate[5];
} sqlca;
```

(In a multithreaded program, every thread automatically gets its own copy of `sqlca`. This works similarly to the handling of the standard C global variable `errno`.)

`sqlca` covers both warnings and errors. If multiple warnings or errors occur during the execution of a statement, then `sqlca` will only contain information about the last one.

If no error occurred in the last SQL statement, `sqlca.sqlcode` will be 0 and `sqlca.sqlstate` will be `"00000"`. If a warning or error occurred, then `sqlca.sqlcode` will be negative and `sqlca.sqlstate` will be different from `"00000"`. A positive `sqlca.sqlcode` indicates a harmless condition, such as that the last query returned zero rows. `sqlcode` and `sqlstate` are two different error code schemes; details appear below.

If the last SQL statement was successful, then `sqlca.sqlerrd[1]` contains the OID of the processed row, if applicable, and `sqlca.sqlerrd[2]` contains the number of processed or returned rows, if applicable to the command.

In case of an error or warning, `sqlca.sqlerrm.sqlerrmc` will contain a string that describes the error. The field `sqlca.sqlerrm.sqlerrml` contains the length of the error message that is stored in `sqlca.sqlerrm.sqlerrmc` (the result of `strlen()`, not really interesting for a C programmer). Note that some messages are too long to fit in the fixed-size `sqlerrmc` array; they will be truncated.

In case of a warning, `sqlca.sqlwarn[2]` is set to `W`. (In all other cases, it is set to something different from `W`.) If `sqlca.sqlwarn[1]` is set to `W`, then a value was truncated when it was stored in a host variable. `sqlca.sqlwarn[0]` is set to `W` if any of the other elements are set to indicate a warning.

The fields `sqlcaid`, `sqlcabc`, `sqlerrp`, and the remaining elements of `sqlerrd` and `sqlwarn` currently contain no useful information.

The structure `sqlca` is not defined in the SQL standard, but is implemented in several other SQL database systems. The definitions are similar at the core, but if you want to write portable applications, then you should investigate the different implementations carefully.

31.11.3. SQLSTATE VS SQLCODE

The fields sqlca.sqlstate and sqlca.sqlcode are two different schemes that provide error codes. Both are derived from the SQL standard, but SQLCODE has been marked deprecated in the SQL-92 edition of the standard and has been dropped in later editions. Therefore, new applications are strongly encouraged to use SQLSTATE.

SQLSTATE is a five-character array. The five characters contain digits or upper-case letters that represent codes of various error and warning conditions. SQLSTATE has a hierarchical scheme: the first two characters indicate the general class of the condition, the last three characters indicate a subclass of the general condition. A successful state is indicated by the code 00000. The SQLSTATE codes are for the most part defined in the SQL standard. The PostgreSQL server natively supports SQLSTATE error codes; therefore a high degree of consistency can be achieved by using this error code scheme throughout all applications. For further information see Appendix A.

SQLCODE, the deprecated error code scheme, is a simple integer. A value of 0 indicates success, a positive value indicates success with additional information, a negative value indicates an error. The SQL standard only defines the positive value +100, which indicates that the last command returned or affected zero rows, and no specific negative values. Therefore, this scheme can only achieve poor portability and does not have a hierarchical code assignment. Historically, the embedded SQL processor for PostgreSQL has assigned some specific SQLCODE values for its use, which are listed below with their numeric value and their symbolic name. Remember that these are not portable to other SQL implementations. To simplify the porting of applications to the SQLSTATE scheme, the corresponding SQLSTATE is also listed. There is, however, no one-to-one or one-to-many mapping between the two schemes (indeed it is many-to-many), so you should consult the global SQLSTATE listing in Appendix A in each case.

These are the assigned SQLCODE values:

-12 (ECPG_OUT_OF_MEMORY)

Indicates that your virtual memory is exhausted. (SQLSTATE YE001)

-200 (ECPG_UNSUPPORTED)

Indicates the preprocessor has generated something that the library does not know about. Perhaps you are running incompatible versions of the preprocessor and the library. (SQLSTATE YE002)

-201 (ECPG_TOO_MANY_ARGUMENTS)

This means that the command specified more host variables than the command expected. (SQLSTATE 07001 or 07002)

-202 (ECPG_TOO_FEW_ARGUMENTS)

This means that the command specified fewer host variables than the command expected. (SQLSTATE 07001 or 07002)

-203 (ECPG_TOO_MANY_MATCHES)

This means a query has returned multiple rows but the statement was only prepared to store one result row (for example, because the specified variables are not arrays). (SQLSTATE 21000)

-204 (ECPG_INT_FORMAT)

The host variable is of type int and the datum in the database is of a different type and contains a value that cannot be interpreted as an int. The library uses strtol() for this conversion. (SQLSTATE 42804)

-205 (`ECPG_UINT_FORMAT`)

The host variable is of type `unsigned int` and the datum in the database is of a different type and contains a value that cannot be interpreted as an `unsigned int`. The library uses `strtoul()` for this conversion. (SQLSTATE 42804)

-206 (`ECPG_FLOAT_FORMAT`)

The host variable is of type `float` and the datum in the database is of another type and contains a value that cannot be interpreted as a `float`. The library uses `strtod()` for this conversion. (SQLSTATE 42804)

-207 (`ECPG_CONVERT_BOOL`)

This means the host variable is of type `bool` and the datum in the database is neither `'t'` nor `'f'`. (SQLSTATE 42804)

-208 (`ECPG_EMPTY`)

The statement sent to the PostgreSQL server was empty. (This cannot normally happen in an embedded SQL program, so it may point to an internal error.) (SQLSTATE YE002)

-209 (`ECPG_MISSING_INDICATOR`)

A null value was returned and no null indicator variable was supplied. (SQLSTATE 22002)

-210 (`ECPG_NO_ARRAY`)

An ordinary variable was used in a place that requires an array. (SQLSTATE 42804)

-211 (`ECPG_DATA_NOT_ARRAY`)

The database returned an ordinary variable in a place that requires array value. (SQLSTATE 42804)

-220 (`ECPG_NO_CONN`)

The program tried to access a connection that does not exist. (SQLSTATE 08003)

-221 (`ECPG_NOT_CONN`)

The program tried to access a connection that does exist but is not open. (This is an internal error.) (SQLSTATE YE002)

-230 (`ECPG_INVALID_STMT`)

The statement you are trying to use has not been prepared. (SQLSTATE 26000)

-240 (`ECPG_UNKNOWN_DESCRIPTOR`)

The descriptor specified was not found. The statement you are trying to use has not been prepared. (SQLSTATE 33000)

-241 (`ECPG_INVALID_DESCRIPTOR_INDEX`)

The descriptor index specified was out of range. (SQLSTATE 07009)

-242 (`ECPG_UNKNOWN_DESCRIPTOR_ITEM`)

An invalid descriptor item was requested. (This is an internal error.) (SQLSTATE YE002)

-243 (`ECPG_VAR_NOT_NUMERIC`)

During the execution of a dynamic statement, the database returned a numeric value and the host variable was not numeric. (SQLSTATE 07006)

-244 (ECPG_VAR_NOT_CHAR)

During the execution of a dynamic statement, the database returned a non-numeric value and the host variable was numeric. (SQLSTATE 07006)

-400 (ECPG_PGSQL)

Some error caused by the PostgreSQL server. The message contains the error message from the PostgreSQL server.

-401 (ECPG_TRANS)

The PostgreSQL server signaled that we cannot start, commit, or rollback the transaction. (SQLSTATE 08007)

-402 (ECPG_CONNECT)

The connection attempt to the database did not succeed. (SQLSTATE 08001)

100 (ECPG_NOT_FOUND)

This is a harmless condition indicating that the last command retrieved or processed zero rows, or that you are at the end of the cursor. (SQLSTATE 02000)

31.12. Preprocessor directives

31.12.1. Including files

To include an external file into your embedded SQL program, use:

```
EXEC SQL INCLUDE filename;
```

The embedded SQL preprocessor will look for a file named *filename*.h, preprocess it, and include it in the resulting C output. Thus, embedded SQL statements in the included file are handled correctly.

Note that this is *not* the same as

```
#include <filename.h>
```

because this file would not be subject to SQL command preprocessing. Naturally, you can continue to use the C #include directive to include other header files.

Note: The include file name is case-sensitive, even though the rest of the EXEC SQL INCLUDE command follows the normal SQL case-sensitivity rules.

31.12.2. The #define and #undef directives

Similar to the directive `#define` that is known from C, embedded SQL has a similar concept:

```
EXEC SQL DEFINE name;
EXEC SQL DEFINE name value;
```

So you can define a name:

```
EXEC SQL DEFINE HAVE_FEATURE;
```

And you can also define constants:

```
EXEC SQL DEFINE MYNUMBER 12;
EXEC SQL DEFINE MYSTRING 'abc';
```

Use `undef` to remove a previous definition:

```
EXEC SQL UNDEF MYNUMBER;
```

Of course you can continue to use the C versions `#define` and `#undef` in your embedded SQL program. The difference is where your defined values get evaluated. If you use `EXEC SQL DEFINE` then the ecpg preprocessor evaluates the defines and substitutes the values. For example if you write:

```
EXEC SQL DEFINE MYNUMBER 12;
...
EXEC SQL UPDATE Tbl SET col = MYNUMBER;
```

then ecpg will already do the substitution and your C compiler will never see any name or identifier `MYNUMBER`. Note that you can not use `#define` for a constant that you are going to use in an embedded SQL query because in this case the embedded SQL precompiler is not able to see this declaration.

31.12.3. ifdef, ifndef, else, elif and endif directives

You can use the following directives to compile code sections conditionally:

`EXEC SQL ifdef name;`

Checks a *name* and processes subsequent lines if *name* has been created with `EXEC SQL define name`.

`EXEC SQL ifndef name;`

Checks a *name* and processes subsequent lines if *name* has *not* been created with `EXEC SQL define name`.

`EXEC SQL else;`

Starts processing an alternative section to a section introduced by either `EXEC SQL ifdef name` or `EXEC SQL ifndef name`.

`EXEC SQL elif name;`

Checks *name* and starts an alternative section if *name* has been created with `EXEC SQL define name`.

`EXEC SQL endif;`

Ends an alternative section.

Example:

```
exec sql ifndef TZVAR;
exec sql SET TIMEZONE TO 'GMT';
exec sql elif TZNAME;
exec sql SET TIMEZONE TO TZNAME;
exec sql else;
exec sql SET TIMEZONE TO TZVAR;
exec sql endif;
```

31.13. Processing Embedded SQL Programs

Now that you have an idea how to form embedded SQL C programs, you probably want to know how to compile them. Before compiling you run the file through the embedded SQL C preprocessor, which converts the SQL statements you used to special function calls. After compiling, you must link with a special library that contains the needed functions. These functions fetch information from the arguments, perform the SQL command using the libpq interface, and put the result in the arguments specified for output.

The preprocessor program is called `ecpg` and is included in a normal PostgreSQL installation. Embedded SQL programs are typically named with an extension `.pgc`. If you have a program file called `prog1.pgc`, you can preprocess it by simply calling

```
ecpg prog1.pgc
```

This will create a file called `prog1.c`. If your input files do not follow the suggested naming pattern, you can specify the output file explicitly using the `-o` option.

The preprocessed file can be compiled normally, for example:

```
cc -c prog1.c
```

The generated C source files include header files from the PostgreSQL installation, so if you installed PostgreSQL in a location that is not searched by default, you have to add an option such as `-I/usr/local/pgsql/include` to the compilation command line.

To link an embedded SQL program, you need to include the `libecpg` library, like so:

```
cc -o myprog prog1.o prog2.o ... -lecpg
```

Again, you might have to add an option like `-L/usr/local/pgsql/lib` to that command line.

If you manage the build process of a larger project using make, it may be convenient to include the following implicit rule to your makefiles:

```
ECPG = ecpg

%.c: %.pgc
        $(ECPG) $<
```

The complete syntax of the `ecpg` command is detailed in ecpg.

The ecpg library is thread-safe if it is built using the `--enable-thread-safety` command-line option to `configure`. (You might need to use other threading command-line options to compile your client code.)

31.14. Library Functions

The `libecpg` library primarily contains "hidden" functions that are used to implement the functionality expressed by the embedded SQL commands. But there are some functions that can usefully be called directly. Note that this makes your code unportable.

- `ECPGdebug(int` *`on`*`, FILE` *`*stream`*`)` turns on debug logging if called with the first argument non-zero. Debug logging is done on *`stream`*. The log contains all SQL statements with all the input variables inserted, and the results from the PostgreSQL server. This can be very useful when searching for errors in your SQL statements.

  Note: On Windows, if the ecpg libraries and an application are compiled with different flags, this function call will crash the application because the internal representation of the `FILE` pointers differ. Specifically, multithreaded/single-threaded, release/debug, and static/dynamic flags should be the same for the library and all applications using that library.

- `ECPGstatus(int` *`lineno`*`, const char*` *`connection_name`*`)` returns true if you are connected to a database and false if not. *`connection_name`* can be `NULL` if a single connection is being used.

31.15. Internals

This section explains how ECPG works internally. This information can occasionally be useful to help users understand how to use ECPG.

The first four lines written by `ecpg` to the output are fixed lines. Two are comments and two are include lines necessary to interface to the library. Then the preprocessor reads through the file and writes output. Normally it just echoes everything to the output.

When it sees an `EXEC SQL` statement, it intervenes and changes it. The command starts with `EXEC SQL` and ends with `;`. Everything in between is treated as an SQL statement and parsed for variable substitution.

Variable substitution occurs when a symbol starts with a colon (`:`). The variable with that name is looked up among the variables that were previously declared within a `EXEC SQL DECLARE` section.

The most important function in the library is `ECPGdo`, which takes care of executing most commands. It takes a variable number of arguments. This can easily add up to 50 or so arguments, and we hope this will not be a problem on any platform.

The arguments are:

A line number

This is the line number of the original line; used in error messages only.

A string

This is the SQL command that is to be issued. It is modified by the input variables, i.e., the variables that where not known at compile time but are to be entered in the command. Where the variables should go the string contains `?`.

Input variables

Every input variable causes ten arguments to be created. (See below.)

`ECPGt_EOIT`

An `enum` telling that there are no more input variables.

Output variables

Every output variable causes ten arguments to be created. (See below.) These variables are filled by the function.

`ECPGt_EORT`

An `enum` telling that there are no more variables.

For every variable that is part of the SQL command, the function gets ten arguments:

1. The type as a special symbol.
2. A pointer to the value or a pointer to the pointer.
3. The size of the variable if it is a `char` or `varchar`.
4. The number of elements in the array (for array fetches).
5. The offset to the next element in the array (for array fetches).
6. The type of the indicator variable as a special symbol.
7. A pointer to the indicator variable.
8. 0
9. The number of elements in the indicator array (for array fetches).
10. The offset to the next element in the indicator array (for array fetches).

Note that not all SQL commands are treated in this way. For instance, an open cursor statement like

```
EXEC SQL OPEN cursor;
```

is not copied to the output. Instead, the cursor's `DECLARE` command is used at the position of the `OPEN` command because it indeed opens the cursor.

Here is a complete example describing the output of the preprocessor of a file `foo.pgc` (details may change with each particular version of the preprocessor):

```
EXEC SQL BEGIN DECLARE SECTION;
int index;
int result;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT res INTO :result FROM mytable WHERE index = :index;
```

is translated into:

```
/* Processed by ecpg (2.6.0) */
/* These two include files are added by the preprocessor */
#include <ecpgtype.h>;
#include <ecpglib.h>;

/* exec sql begin declare section */

#line 1 "foo.pgc"

 int index;
 int result;
/* exec sql end declare section */
...
ECPGdo(__LINE__, NULL, "SELECT res FROM mytable WHERE index = ?     ",
        ECPGt_int,&(index),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EOIT,
        ECPGt_int,&(result),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EORT);
#line 147 "foo.pgc"
```

(The indentation here is added for readability and not something the preprocessor does.)

Chapter 32. The Information Schema

The information schema consists of a set of views that contain information about the objects defined in the current database. The information schema is defined in the SQL standard and can therefore be expected to be portable and remain stable — unlike the system catalogs, which are specific to PostgreSQL and are modelled after implementation concerns. The information schema views do not, however, contain information about PostgreSQL-specific features; to inquire about those you need to query the system catalogs or other PostgreSQL-specific views.

32.1. The Schema

The information schema itself is a schema named `information_schema`. This schema automatically exists in all databases. The owner of this schema is the initial database user in the cluster, and that user naturally has all the privileges on this schema, including the ability to drop it (but the space savings achieved by that are minuscule).

By default, the information schema is not in the schema search path, so you need to access all objects in it through qualified names. Since the names of some of the objects in the information schema are generic names that might occur in user applications, you should be careful if you want to put the information schema in the path.

32.2. Data Types

The columns of the information schema views use special data types that are defined in the information schema. These are defined as simple domains over ordinary built-in types. You should not use these types for work outside the information schema, but your applications must be prepared for them if they select from the information schema.

These types are:

`cardinal_number`

A nonnegative integer.

`character_data`

A character string (without specific maximum length).

`sql_identifier`

A character string. This type is used for SQL identifiers, the type `character_data` is used for any other kind of text data.

`time_stamp`

A domain over the type `timestamp with time zone`

Every column in the information schema has one of these four types.

Boolean (true/false) data is represented in the information schema by a column of type `character_data` that contains either `YES` or `NO`. (The information schema was invented before the type `boolean` was

added to the SQL standard, so this convention is necessary to keep the information schema backward compatible.)

32.3. information_schema_catalog_name

information_schema_catalog_name is a table that always contains one row and one column containing the name of the current database (current catalog, in SQL terminology).

Table 32-1. information_schema_catalog_name Columns

Name	Data Type	Description
catalog_name	sql_identifier	Name of the database that contains this information schema

32.4. administrable_role_authorizations

The view administrable_role_authorizations identifies all roles that the current user has the admin option for.

Table 32-2. administrable_role_authorizations Columns

Name	Data Type	Description
grantee	sql_identifier	Name of the role to which this role membership was granted (may be the current user, or a different role in case of nested role memberships)
role_name	sql_identifier	Name of a role
is_grantable	character_data	Always YES

32.5. applicable_roles

The view applicable_roles identifies all roles whose privileges the current user can use. This means there is some chain of role grants from the current user to the role in question. The current user itself is also an applicable role. The set of applicable roles is generally used for permission checking.

Table 32-3. applicable_roles Columns

Name	Data Type	Description

Name	Data Type	Description
grantee	sql_identifier	Name of the role to which this role membership was granted (may be the current user, or a different role in case of nested role memberships)
role_name	sql_identifier	Name of a role
is_grantable	character_data	YES if the grantee has the admin option on the role, NO if not

32.6. attributes

The view attributes contains information about the attributes of composite data types defined in the database. (Note that the view does not give information about table columns, which are sometimes called attributes in PostgreSQL contexts.)

Table 32-4. attributes Columns

Name	Data Type	Description
udt_catalog	sql_identifier	Name of the database containing the data type (always the current database)
udt_schema	sql_identifier	Name of the schema containing the data type
udt_name	sql_identifier	Name of the data type
attribute_name	sql_identifier	Name of the attribute
ordinal_position	cardinal_number	Ordinal position of the attribute within the data type (count starts at 1)
attribute_default	character_data	Default expression of the attribute
is_nullable	character_data	YES if the attribute is possibly nullable, NO if it is known not nullable.
data_type	character_data	Data type of the attribute, if it is a built-in type, or ARRAY if it is some array (in that case, see the view element_types), else USER-DEFINED (in that case, the type is identified in attribute_udt_name and associated columns).

Name	Data Type	Description
character_maximum_length	cardinal_number	If data_type identifies a character or bit string type, the declared maximum length; null for all other data types or if no maximum length was declared.
character_octet_length	cardinal_number	If data_type identifies a character type, the maximum possible length in octets (bytes) of a datum (this should not be of concern to PostgreSQL users); null for all other data types.
numeric_precision	cardinal_number	If data_type identifies a numeric type, this column contains the (declared or implicit) precision of the type for this attribute. The precision indicates the number of significant digits. It may be expressed in decimal (base 10) or binary (base 2) terms, as specified in the column numeric_precision_radix. For all other data types, this column is null.
numeric_precision_radix	cardinal_number	If data_type identifies a numeric type, this column indicates in which base the values in the columns numeric_precision and numeric_scale are expressed. The value is either 2 or 10. For all other data types, this column is null.

Name	Data Type	Description
numeric_scale	cardinal_number	If data_type identifies an exact numeric type, this column contains the (declared or implicit) scale of the type for this attribute. The scale indicates the number of significant digits to the right of the decimal point. It may be expressed in decimal (base 10) or binary (base 2) terms, as specified in the column numeric_precision_radix. For all other data types, this column is null.
datetime_precision	cardinal_number	If data_type identifies a date, time, or interval type, the declared precision; null for all other data types or if no precision was declared.
interval_type	character_data	Not yet implemented
interval_precision	character_data	Not yet implemented
attribute_udt_catalog	sql_identifier	Name of the database that the attribute data type is defined in (always the current database)
attribute_udt_schema	sql_identifier	Name of the schema that the attribute data type is defined in
attribute_udt_name	sql_identifier	Name of the attribute data type
scope_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
scope_schema	sql_identifier	Applies to a feature not available in PostgreSQL
scope_name	sql_identifier	Applies to a feature not available in PostgreSQL
maximum_cardinality	cardinal_number	Always null, because arrays always have unlimited maximum cardinality in PostgreSQL

Name	Data Type	Description
dtd_identifier	sql_identifier	An identifier of the data type descriptor of the column, unique among the data type descriptors pertaining to the table. This is mainly useful for joining with other instances of such identifiers. (The specific format of the identifier is not defined and not guaranteed to remain the same in future versions.)
is_derived_reference_attribute	character_data	Applies to a feature not available in PostgreSQL

See also under Section 32.12, a similarly structured view, for further information on some of the columns.

32.7. check_constraint_routine_usage

The view check_constraint_routine_usage identifies routines (functions and procedures) that are used by a check constraint. Only those routines are shown that are owned by a currently enabled role.

Table 32-5. check_constraint_routine_usage Columns

Name	Data Type	Description
constraint_catalog	sql_identifier	Name of the database containing the constraint (always the current database)
constraint_schema	sql_identifier	Name of the schema containing the constraint
constraint_name	sql_identifier	Name of the constraint
specific_catalog	sql_identifier	Name of the database containing the function (always the current database)
specific_schema	sql_identifier	Name of the schema containing the function
specific_name	sql_identifier	The "specific name" of the function. See Section 32.29 for more information.

32.8. check_constraints

The view check_constraints contains all check constraints, either defined on a table or on a domain, that are owned by a currently enabled role. (The owner of the table or domain is the owner of the con-

straint.)

Table 32-6. `check_constraints` Columns

Name	Data Type	Description
`constraint_catalog`	`sql_identifier`	Name of the database containing the constraint (always the current database)
`constraint_schema`	`sql_identifier`	Name of the schema containing the constraint
`constraint_name`	`sql_identifier`	Name of the constraint
`check_clause`	`character_data`	The check expression of the check constraint

32.9. `column_domain_usage`

The view `column_domain_usage` identifies all columns (of a table or a view) that make use of some domain defined in the current database and owned by a currently enabled role.

Table 32-7. `column_domain_usage` Columns

Name	Data Type	Description
`domain_catalog`	`sql_identifier`	Name of the database containing the domain (always the current database)
`domain_schema`	`sql_identifier`	Name of the schema containing the domain
`domain_name`	`sql_identifier`	Name of the domain
`table_catalog`	`sql_identifier`	Name of the database containing the table (always the current database)
`table_schema`	`sql_identifier`	Name of the schema containing the table
`table_name`	`sql_identifier`	Name of the table
`column_name`	`sql_identifier`	Name of the column

32.10. `column_privileges`

The view `column_privileges` identifies all privileges granted on columns to a currently enabled role or by a currently enabled role. There is one row for each combination of column, grantor, and grantee.

In PostgreSQL, you can only grant privileges on entire tables, not individual columns. Therefore, this

view contains the same information as table_privileges, just represented through one row for each column in each appropriate table, but it only covers privilege types where column granularity is possible: SELECT, INSERT, UPDATE, REFERENCES. If you want to make your applications fit for possible future developments, it is generally the right choice to use this view instead of table_privileges if one of those privilege types is concerned.

Table 32-8. column_privileges Columns

Name	Data Type	Description
grantor	sql_identifier	Name of the role that granted the privilege
grantee	sql_identifier	Name of the role that the privilege was granted to
table_catalog	sql_identifier	Name of the database that contains the table that contains the column (always the current database)
table_schema	sql_identifier	Name of the schema that contains the table that contains the column
table_name	sql_identifier	Name of the table that contains the column
column_name	sql_identifier	Name of the column
privilege_type	character_data	Type of the privilege: SELECT, INSERT, UPDATE, or REFERENCES
is_grantable	character_data	YES if the privilege is grantable, NO if not

32.11. column_udt_usage

The view column_udt_usage identifies all columns that use data types owned by a currently enabled role. Note that in PostgreSQL, built-in data types behave like user-defined types, so they are included here as well. See also Section 32.12 for details.

Table 32-9. column_udt_usage Columns

Name	Data Type	Description
udt_catalog	sql_identifier	Name of the database that the column data type (the underlying type of the domain, if applicable) is defined in (always the current database)

Name	Data Type	Description
udt_schema	sql_identifier	Name of the schema that the column data type (the underlying type of the domain, if applicable) is defined in
udt_name	sql_identifier	Name of the column data type (the underlying type of the domain, if applicable)
table_catalog	sql_identifier	Name of the database containing the table (always the current database)
table_schema	sql_identifier	Name of the schema containing the table
table_name	sql_identifier	Name of the table
column_name	sql_identifier	Name of the column

32.12. columns

The view columns contains information about all table columns (or view columns) in the database. System columns (oid, etc.) are not included. Only those columns are shown that the current user has access to (by way of being the owner or having some privilege).

Table 32-10. columns Columns

Name	Data Type	Description
table_catalog	sql_identifier	Name of the database containing the table (always the current database)
table_schema	sql_identifier	Name of the schema containing the table
table_name	sql_identifier	Name of the table
column_name	sql_identifier	Name of the column
ordinal_position	cardinal_number	Ordinal position of the column within the table (count starts at 1)
column_default	character_data	Default expression of the column
is_nullable	character_data	YES if the column is possibly nullable, NO if it is known not nullable. A not-null constraint is one way a column can be known not nullable, but there may be others.

Name	Data Type	Description
data_type	character_data	Data type of the column, if it is a built-in type, or ARRAY if it is some array (in that case, see the view element_types), else USER-DEFINED (in that case, the type is identified in udt_name and associated columns). If the column is based on a domain, this column refers to the type underlying the domain (and the domain is identified in domain_name and associated columns).
character_maximum_length	cardinal_number	If data_type identifies a character or bit string type, the declared maximum length; null for all other data types or if no maximum length was declared.
character_octet_length	cardinal_number	If data_type identifies a character type, the maximum possible length in octets (bytes) of a datum (this should not be of concern to PostgreSQL users); null for all other data types.
numeric_precision	cardinal_number	If data_type identifies a numeric type, this column contains the (declared or implicit) precision of the type for this column. The precision indicates the number of significant digits. It may be expressed in decimal (base 10) or binary (base 2) terms, as specified in the column numeric_precision_radix. For all other data types, this column is null.

Name	Data Type	Description
numeric_precision_radix	cardinal_number	If data_type identifies a numeric type, this column indicates in which base the values in the columns numeric_precision and numeric_scale are expressed. The value is either 2 or 10. For all other data types, this column is null.
numeric_scale	cardinal_number	If data_type identifies an exact numeric type, this column contains the (declared or implicit) scale of the type for this column. The scale indicates the number of significant digits to the right of the decimal point. It may be expressed in decimal (base 10) or binary (base 2) terms, as specified in the column numeric_precision_radix. For all other data types, this column is null.
datetime_precision	cardinal_number	If data_type identifies a date, time, or interval type, the declared precision; null for all other data types or if no precision was declared.
interval_type	character_data	Not yet implemented
interval_precision	character_data	Not yet implemented
character_set_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
character_set_schema	sql_identifier	Applies to a feature not available in PostgreSQL
character_set_name	sql_identifier	Applies to a feature not available in PostgreSQL
collation_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
collation_schema	sql_identifier	Applies to a feature not available in PostgreSQL
collation_name	sql_identifier	Applies to a feature not available in PostgreSQL

Name	Data Type	Description
domain_catalog	sql_identifier	If the column has a domain type, the name of the database that the domain is defined in (always the current database), else null.
domain_schema	sql_identifier	If the column has a domain type, the name of the schema that the domain is defined in, else null.
domain_name	sql_identifier	If the column has a domain type, the name of the domain, else null.
udt_catalog	sql_identifier	Name of the database that the column data type (the underlying type of the domain, if applicable) is defined in (always the current database)
udt_schema	sql_identifier	Name of the schema that the column data type (the underlying type of the domain, if applicable) is defined in
udt_name	sql_identifier	Name of the column data type (the underlying type of the domain, if applicable)
scope_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
scope_schema	sql_identifier	Applies to a feature not available in PostgreSQL
scope_name	sql_identifier	Applies to a feature not available in PostgreSQL
maximum_cardinality	cardinal_number	Always null, because arrays always have unlimited maximum cardinality in PostgreSQL
dtd_identifier	sql_identifier	An identifier of the data type descriptor of the column, unique among the data type descriptors pertaining to the table. This is mainly useful for joining with other instances of such identifiers. (The specific format of the identifier is not defined and not guaranteed to remain the same in future versions.)
is_self_referencing	character_data	Applies to a feature not available in PostgreSQL

Name	Data Type	Description
is_identity	character_data	Applies to a feature not available in PostgreSQL
identity_generation	character_data	Applies to a feature not available in PostgreSQL
identity_start	character_data	Applies to a feature not available in PostgreSQL
identity_increment	character_data	Applies to a feature not available in PostgreSQL
identity_maximum	character_data	Applies to a feature not available in PostgreSQL
identity_minimum	character_data	Applies to a feature not available in PostgreSQL
identity_cycle	character_data	Applies to a feature not available in PostgreSQL
is_generated	character_data	Applies to a feature not available in PostgreSQL
generation_expression	character_data	Applies to a feature not available in PostgreSQL
is_updatable	character_data	YES if the column is updatable, NO if not (Columns in base tables are always updatable, columns in views not necessarily)

Since data types can be defined in a variety of ways in SQL, and PostgreSQL contains additional ways to define data types, their representation in the information schema can be somewhat difficult. The column data_type is supposed to identify the underlying built-in type of the column. In PostgreSQL, this means that the type is defined in the system catalog schema pg_catalog. This column may be useful if the application can handle the well-known built-in types specially (for example, format the numeric types differently or use the data in the precision columns). The columns udt_name, udt_schema, and udt_catalog always identify the underlying data type of the column, even if the column is based on a domain. (Since PostgreSQL treats built-in types like user-defined types, built-in types appear here as well. This is an extension of the SQL standard.) These columns should be used if an application wants to process data differently according to the type, because in that case it wouldn't matter if the column is really based on a domain. If the column is based on a domain, the identity of the domain is stored in the columns domain_name, domain_schema, and domain_catalog. If you want to pair up columns with their associated data types and treat domains as separate types, you could write coalesce(domain_name, udt_name), etc.

32.13. constraint_column_usage

The view constraint_column_usage identifies all columns in the current database that are used by some constraint. Only those columns are shown that are contained in a table owned by a currently enabled role. For a check constraint, this view identifies the columns that are used in the check expression. For a

foreign key constraint, this view identifies the columns that the foreign key references. For a unique or primary key constraint, this view identifies the constrained columns.

Table 32-11. `constraint_column_usage` Columns

Name	Data Type	Description
`table_catalog`	`sql_identifier`	Name of the database that contains the table that contains the column that is used by some constraint (always the current database)
`table_schema`	`sql_identifier`	Name of the schema that contains the table that contains the column that is used by some constraint
`table_name`	`sql_identifier`	Name of the table that contains the column that is used by some constraint
`column_name`	`sql_identifier`	Name of the column that is used by some constraint
`constraint_catalog`	`sql_identifier`	Name of the database that contains the constraint (always the current database)
`constraint_schema`	`sql_identifier`	Name of the schema that contains the constraint
`constraint_name`	`sql_identifier`	Name of the constraint

32.14. `constraint_table_usage`

The view `constraint_table_usage` identifies all tables in the current database that are used by some constraint and are owned by a currently enabled role. (This is different from the view `table_constraints`, which identifies all table constraints along with the table they are defined on.) For a foreign key constraint, this view identifies the table that the foreign key references. For a unique or primary key constraint, this view simply identifies the table the constraint belongs to. Check constraints and not-null constraints are not included in this view.

Table 32-12. `constraint_table_usage` Columns

Name	Data Type	Description
`table_catalog`	`sql_identifier`	Name of the database that contains the table that is used by some constraint (always the current database)

Name	Data Type	Description
table_schema	sql_identifier	Name of the schema that contains the table that is used by some constraint
table_name	sql_identifier	Name of the table that is used by some constraint
constraint_catalog	sql_identifier	Name of the database that contains the constraint (always the current database)
constraint_schema	sql_identifier	Name of the schema that contains the constraint
constraint_name	sql_identifier	Name of the constraint

32.15. data_type_privileges

The view data_type_privileges identifies all data type descriptors that the current user has access to, by way of being the owner of the described object or having some privilege for it. A data type descriptor is generated whenever a data type is used in the definition of a table column, a domain, or a function (as parameter or return type) and stores some information about how the data type is used in that instance (for example, the declared maximum length, if applicable). Each data type descriptor is assigned an arbitrary identifier that is unique among the data type descriptor identifiers assigned for one object (table, domain, function). This view is probably not useful for applications, but it is used to define some other views in the information schema.

Table 32-13. data_type_privileges Columns

Name	Data Type	Description
object_catalog	sql_identifier	Name of the database that contains the described object (always the current database)
object_schema	sql_identifier	Name of the schema that contains the described object
object_name	sql_identifier	Name of the described object
object_type	character_data	The type of the described object: one of TABLE (the data type descriptor pertains to a column of that table), DOMAIN (the data type descriptors pertains to that domain), ROUTINE (the data type descriptor pertains to a parameter or the return data type of that function).

Name	Data Type	Description
dtd_identifier	sql_identifier	The identifier of the data type descriptor, which is unique among the data type descriptors for that same object.

32.16. domain_constraints

The view domain_constraints contains all constraints belonging to domains defined in the current database.

Table 32-14. domain_constraints Columns

Name	Data Type	Description
constraint_catalog	sql_identifier	Name of the database that contains the constraint (always the current database)
constraint_schema	sql_identifier	Name of the schema that contains the constraint
constraint_name	sql_identifier	Name of the constraint
domain_catalog	sql_identifier	Name of the database that contains the domain (always the current database)
domain_schema	sql_identifier	Name of the schema that contains the domain
domain_name	sql_identifier	Name of the domain
is_deferrable	character_data	YES if the constraint is deferrable, NO if not
initially_deferred	character_data	YES if the constraint is deferrable and initially deferred, NO if not

32.17. domain_udt_usage

The view domain_udt_usage identifies all domains that are based on data types owned by a currently enabled role. Note that in PostgreSQL, built-in data types behave like user-defined types, so they are included here as well.

Table 32-15. domain_udt_usage Columns

Name	Data Type	Description

Name	Data Type	Description
udt_catalog	sql_identifier	Name of the database that the domain data type is defined in (always the current database)
udt_schema	sql_identifier	Name of the schema that the domain data type is defined in
udt_name	sql_identifier	Name of the domain data type
domain_catalog	sql_identifier	Name of the database that contains the domain (always the current database)
domain_schema	sql_identifier	Name of the schema that contains the domain
domain_name	sql_identifier	Name of the domain

32.18. domains

The view domains contains all domains defined in the current database.

Table 32-16. domains Columns

Name	Data Type	Description
domain_catalog	sql_identifier	Name of the database that contains the domain (always the current database)
domain_schema	sql_identifier	Name of the schema that contains the domain
domain_name	sql_identifier	Name of the domain
data_type	character_data	Data type of the domain, if it is a built-in type, or ARRAY if it is some array (in that case, see the view element_types), else USER-DEFINED (in that case, the type is identified in udt_name and associated columns).
character_maximum_length	cardinal_number	If the domain has a character or bit string type, the declared maximum length; null for all other data types or if no maximum length was declared.

Name	Data Type	Description
character_octet_length	cardinal_number	If the domain has a character type, the maximum possible length in octets (bytes) of a datum (this should not be of concern to PostgreSQL users); null for all other data types.
character_set_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
character_set_schema	sql_identifier	Applies to a feature not available in PostgreSQL
character_set_name	sql_identifier	Applies to a feature not available in PostgreSQL
collation_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
collation_schema	sql_identifier	Applies to a feature not available in PostgreSQL
collation_name	sql_identifier	Applies to a feature not available in PostgreSQL
numeric_precision	cardinal_number	If the domain has a numeric type, this column contains the (declared or implicit) precision of the type for this column. The precision indicates the number of significant digits. It may be expressed in decimal (base 10) or binary (base 2) terms, as specified in the column numeric_precision_radix. For all other data types, this column is null.
numeric_precision_radix	cardinal_number	If the domain has a numeric type, this column indicates in which base the values in the columns numeric_precision and numeric_scale are expressed. The value is either 2 or 10. For all other data types, this column is null.

Name	Data Type	Description
numeric_scale	cardinal_number	If the domain has an exact numeric type, this column contains the (declared or implicit) scale of the type for this column. The scale indicates the number of significant digits to the right of the decimal point. It may be expressed in decimal (base 10) or binary (base 2) terms, as specified in the column numeric_precision_radix. For all other data types, this column is null.
datetime_precision	cardinal_number	If the domain has a date, time, or interval type, the declared precision; null for all other data types or if no precision was declared.
interval_type	character_data	Not yet implemented
interval_precision	character_data	Not yet implemented
domain_default	character_data	Default expression of the domain
udt_catalog	sql_identifier	Name of the database that the domain data type is defined in (always the current database)
udt_schema	sql_identifier	Name of the schema that the domain data type is defined in
udt_name	sql_identifier	Name of the domain data type
scope_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
scope_schema	sql_identifier	Applies to a feature not available in PostgreSQL
scope_name	sql_identifier	Applies to a feature not available in PostgreSQL
maximum_cardinality	cardinal_number	Always null, because arrays always have unlimited maximum cardinality in PostgreSQL

Name	Data Type	Description
dtd_identifier	sql_identifier	An identifier of the data type descriptor of the domain, unique among the data type descriptors pertaining to the domain (which is trivial, because a domain only contains one data type descriptor). This is mainly useful for joining with other instances of such identifiers. (The specific format of the identifier is not defined and not guaranteed to remain the same in future versions.)

32.19. element_types

The view element_types contains the data type descriptors of the elements of arrays. When a table column, domain, function parameter, or function return value is defined to be of an array type, the respective information schema view only contains ARRAY in the column data_type. To obtain information on the element type of the array, you can join the respective view with this view. For example, to show the columns of a table with data types and array element types, if applicable, you could do

```
SELECT c.column_name, c.data_type, e.data_type AS element_type
FROM information_schema.columns c LEFT JOIN information_schema.element_types e
     ON ((c.table_catalog, c.table_schema, c.table_name, 'TABLE', c.dtd_identifier)
       = (e.object_catalog, e.object_schema, e.object_name, e.object_type, e.dtd_identifier
WHERE c.table_schema = '...' AND c.table_name = '...'
ORDER BY c.ordinal_position;
```

This view only includes objects that the current user has access to, by way of being the owner or having some privilege.

Table 32-17. element_types Columns

Name	Data Type	Description
object_catalog	sql_identifier	Name of the database that contains the object that uses the array being described (always the current database)
object_schema	sql_identifier	Name of the schema that contains the object that uses the array being described
object_name	sql_identifier	Name of the object that uses the array being described

Name	Data Type	Description
object_type	character_data	The type of the object that uses the array being described: one of TABLE (the array is used by a column of that table), DOMAIN (the array is used by that domain), ROUTINE (the array is used by a parameter or the return data type of that function).
dtd_identifier	sql_identifier	The identifier of the data type descriptor of the array being described
data_type	character_data	Data type of the array elements, if it is a built-in type, else USER-DEFINED (in that case, the type is identified in udt_name and associated columns).
character_maximum_length	cardinal_number	Always null, since this information is not applied to array element data types in PostgreSQL
character_octet_length	cardinal_number	Always null, since this information is not applied to array element data types in PostgreSQL
character_set_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
character_set_schema	sql_identifier	Applies to a feature not available in PostgreSQL
character_set_name	sql_identifier	Applies to a feature not available in PostgreSQL
collation_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
collation_schema	sql_identifier	Applies to a feature not available in PostgreSQL
collation_name	sql_identifier	Applies to a feature not available in PostgreSQL
numeric_precision	cardinal_number	Always null, since this information is not applied to array element data types in PostgreSQL
numeric_precision_radix	cardinal_number	Always null, since this information is not applied to array element data types in PostgreSQL

Name	Data Type	Description
numeric_scale	cardinal_number	Always null, since this information is not applied to array element data types in PostgreSQL
datetime_precision	cardinal_number	Always null, since this information is not applied to array element data types in PostgreSQL
interval_type	character_data	Always null, since this information is not applied to array element data types in PostgreSQL
interval_precision	character_data	Always null, since this information is not applied to array element data types in PostgreSQL
domain_default	character_data	Not yet implemented
udt_catalog	sql_identifier	Name of the database that the data type of the elements is defined in (always the current database)
udt_schema	sql_identifier	Name of the schema that the data type of the elements is defined in
udt_name	sql_identifier	Name of the data type of the elements
scope_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
scope_schema	sql_identifier	Applies to a feature not available in PostgreSQL
scope_name	sql_identifier	Applies to a feature not available in PostgreSQL
maximum_cardinality	cardinal_number	Always null, because arrays always have unlimited maximum cardinality in PostgreSQL

32.20. enabled_roles

The view enabled_roles identifies the currently "enabled roles". The enabled roles are recursively defined as the current user together with all roles that have been granted to the enabled roles with automatic inheritance. In other words, these are all roles that the current user has direct or indirect, automatically inheriting membership in.

For permission checking, the set of "applicable roles" is applied, which may be broader than the set of enabled roles. So generally, it is better to use the view `applicable_roles` instead of this one; see also there.

Table 32-18. `enabled_roles` Columns

Name	Data Type	Description
`role_name`	`sql_identifier`	Name of a role

32.21. `key_column_usage`

The view `key_column_usage` identifies all columns in the current database that are restricted by some unique, primary key, or foreign key constraint. Check constraints are not included in this view. Only those columns are shown that the current user has access to, by way of being the owner or having some privilege.

Table 32-19. `key_column_usage` Columns

Name	Data Type	Description
`constraint_catalog`	`sql_identifier`	Name of the database that contains the constraint (always the current database)
`constraint_schema`	`sql_identifier`	Name of the schema that contains the constraint
`constraint_name`	`sql_identifier`	Name of the constraint
`table_catalog`	`sql_identifier`	Name of the database that contains the table that contains the column that is restricted by some constraint (always the current database)
`table_schema`	`sql_identifier`	Name of the schema that contains the table that contains the column that is restricted by some constraint
`table_name`	`sql_identifier`	Name of the table that contains the column that is restricted by some constraint
`column_name`	`sql_identifier`	Name of the column that is restricted by some constraint
`ordinal_position`	`cardinal_number`	Ordinal position of the column within the constraint key (count starts at 1)
`position_in_unique_constraint`	`cardinal_number`	Not yet implemented

32.22. parameters

The view parameters contains information about the parameters (arguments) of all functions in the current database. Only those functions are shown that the current user has access to (by way of being the owner or having some privilege).

Table 32-20. parameters Columns

Name	Data Type	Description
specific_catalog	sql_identifier	Name of the database containing the function (always the current database)
specific_schema	sql_identifier	Name of the schema containing the function
specific_name	sql_identifier	The "specific name" of the function. See Section 32.29 for more information.
ordinal_position	cardinal_number	Ordinal position of the parameter in the argument list of the function (count starts at 1)
parameter_mode	character_data	IN for input parameter, OUT for output parameter, and INOUT for input/output parameter.
is_result	character_data	Applies to a feature not available in PostgreSQL
as_locator	character_data	Applies to a feature not available in PostgreSQL
parameter_name	sql_identifier	Name of the parameter, or null if the parameter has no name
data_type	character_data	Data type of the parameter, if it is a built-in type, or ARRAY if it is some array (in that case, see the view element_types), else USER-DEFINED (in that case, the type is identified in udt_name and associated columns).
character_maximum_length	cardinal_number	Always null, since this information is not applied to parameter data types in PostgreSQL
character_octet_length	cardinal_number	Always null, since this information is not applied to parameter data types in PostgreSQL

Name	Data Type	Description
character_set_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
character_set_schema	sql_identifier	Applies to a feature not available in PostgreSQL
character_set_name	sql_identifier	Applies to a feature not available in PostgreSQL
collation_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
collation_schema	sql_identifier	Applies to a feature not available in PostgreSQL
collation_name	sql_identifier	Applies to a feature not available in PostgreSQL
numeric_precision	cardinal_number	Always null, since this information is not applied to parameter data types in PostgreSQL
numeric_precision_radix	cardinal_number	Always null, since this information is not applied to parameter data types in PostgreSQL
numeric_scale	cardinal_number	Always null, since this information is not applied to parameter data types in PostgreSQL
datetime_precision	cardinal_number	Always null, since this information is not applied to parameter data types in PostgreSQL
interval_type	character_data	Always null, since this information is not applied to parameter data types in PostgreSQL
interval_precision	character_data	Always null, since this information is not applied to parameter data types in PostgreSQL
udt_catalog	sql_identifier	Name of the database that the data type of the parameter is defined in (always the current database)
udt_schema	sql_identifier	Name of the schema that the data type of the parameter is defined in

Name	Data Type	Description
udt_name	sql_identifier	Name of the data type of the parameter
scope_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
scope_schema	sql_identifier	Applies to a feature not available in PostgreSQL
scope_name	sql_identifier	Applies to a feature not available in PostgreSQL
maximum_cardinality	cardinal_number	Always null, because arrays always have unlimited maximum cardinality in PostgreSQL
dtd_identifier	sql_identifier	An identifier of the data type descriptor of the parameter, unique among the data type descriptors pertaining to the function. This is mainly useful for joining with other instances of such identifiers. (The specific format of the identifier is not defined and not guaranteed to remain the same in future versions.)

32.23. referential_constraints

The view referential_constraints contains all referential (foreign key) constraints in the current database that belong to a table owned by a currently enabled role.

Table 32-21. referential_constraints Columns

Name	Data Type	Description
constraint_catalog	sql_identifier	Name of the database containing the constraint (always the current database)
constraint_schema	sql_identifier	Name of the schema containing the constraint
constraint_name	sql_identifier	Name of the constraint
unique_constraint_catalog	sql_identifier	Name of the database that contains the unique or primary key constraint that the foreign key constraint references (always the current database)

Name	Data Type	Description
unique_constraint_schema	sql_identifier	Name of the schema that contains the unique or primary key constraint that the foreign key constraint references
unique_constraint_name	sql_identifier	Name of the unique or primary key constraint that the foreign key constraint references
match_option	character_data	Match option of the foreign key constraint: FULL, PARTIAL, or NONE.
update_rule	character_data	Update rule of the foreign key constraint: CASCADE, SET NULL, SET DEFAULT, RESTRICT, or NO ACTION.
delete_rule	character_data	Delete rule of the foreign key constraint: CASCADE, SET NULL, SET DEFAULT, RESTRICT, or NO ACTION.

32.24. role_column_grants

The view role_column_grants identifies all privileges granted on columns where the grantor or grantee is a currently enabled role. Further information can be found under column_privileges.

Table 32-22. role_column_grants Columns

Name	Data Type	Description
grantor	sql_identifier	Name of the role that granted the privilege
grantee	sql_identifier	Name of the role that the privilege was granted to
table_catalog	sql_identifier	Name of the database that contains the table that contains the column (always the current database)
table_schema	sql_identifier	Name of the schema that contains the table that contains the column
table_name	sql_identifier	Name of the table that contains the column
column_name	sql_identifier	Name of the column

Name	Data Type	Description
privilege_type	character_data	Type of the privilege: SELECT, INSERT, UPDATE, or REFERENCES
is_grantable	character_data	YES if the privilege is grantable, NO if not

32.25. role_routine_grants

The view role_routine_grants identifies all privileges granted on functions where the grantor or grantee is a currently enabled role. Further information can be found under routine_privileges.

Table 32-23. role_routine_grants Columns

Name	Data Type	Description
grantor	sql_identifier	Name of the role that granted the privilege
grantee	sql_identifier	Name of the role that the privilege was granted to
specific_catalog	sql_identifier	Name of the database containing the function (always the current database)
specific_schema	sql_identifier	Name of the schema containing the function
specific_name	sql_identifier	The "specific name" of the function. See Section 32.29 for more information.
routine_catalog	sql_identifier	Name of the database containing the function (always the current database)
routine_schema	sql_identifier	Name of the schema containing the function
routine_name	sql_identifier	Name of the function (may be duplicated in case of overloading)
privilege_type	character_data	Always EXECUTE (the only privilege type for functions)
is_grantable	character_data	YES if the privilege is grantable, NO if not

32.26. role_table_grants

The view role_table_grants identifies all privileges granted on tables or views where the grantor or grantee is a currently enabled role. Further information can be found under table_privileges.

Table 32-24. role_table_grants Columns

Name	Data Type	Description
grantor	sql_identifier	Name of the role that granted the privilege
grantee	sql_identifier	Name of the role that the privilege was granted to
table_catalog	sql_identifier	Name of the database that contains the table (always the current database)
table_schema	sql_identifier	Name of the schema that contains the table
table_name	sql_identifier	Name of the table
privilege_type	character_data	Type of the privilege: SELECT, DELETE, INSERT, UPDATE, REFERENCES, or TRIGGER
is_grantable	character_data	YES if the privilege is grantable, NO if not
with_hierarchy	character_data	Applies to a feature not available in PostgreSQL

32.27. role_usage_grants

The view role_usage_grants is meant to identify USAGE privileges granted on various kinds of objects to a currently enabled role or by a currently enabled role. In PostgreSQL, this currently only applies to domains, and since domains do not have real privileges in PostgreSQL, this view is empty. Further information can be found under usage_privileges. In the future, this view may contain more useful information.

Table 32-25. role_usage_grants Columns

Name	Data Type	Description
grantor	sql_identifier	In the future, the name of the role that granted the privilege
grantee	sql_identifier	In the future, the name of the role that the privilege was granted to

Name	Data Type	Description
object_catalog	sql_identifier	Name of the database containing the object (always the current database)
object_schema	sql_identifier	Name of the schema containing the object
object_name	sql_identifier	Name of the object
object_type	character_data	In the future, the type of the object
privilege_type	character_data	Always USAGE
is_grantable	character_data	YES if the privilege is grantable, NO if not

32.28. routine_privileges

The view routine_privileges identifies all privileges granted to a currently enabled role or by a currently enabled role. There is one row for each combination of function, grantor, and grantee.

Table 32-26. routine_privileges Columns

Name	Data Type	Description
grantor	sql_identifier	Name of the role that granted the privilege
grantee	sql_identifier	Name of the role that the privilege was granted to
specific_catalog	sql_identifier	Name of the database containing the function (always the current database)
specific_schema	sql_identifier	Name of the schema containing the function
specific_name	sql_identifier	The "specific name" of the function. See Section 32.29 for more information.
routine_catalog	sql_identifier	Name of the database containing the function (always the current database)
routine_schema	sql_identifier	Name of the schema containing the function
routine_name	sql_identifier	Name of the function (may be duplicated in case of overloading)

Name	Data Type	Description
privilege_type	character_data	Always EXECUTE (the only privilege type for functions)
is_grantable	character_data	YES if the privilege is grantable, NO if not

32.29. routines

The view routines contains all functions in the current database. Only those functions are shown that the current user has access to (by way of being the owner or having some privilege).

Table 32-27. routines Columns

Name	Data Type	Description
specific_catalog	sql_identifier	Name of the database containing the function (always the current database)
specific_schema	sql_identifier	Name of the schema containing the function
specific_name	sql_identifier	The "specific name" of the function. This is a name that uniquely identifies the function in the schema, even if the real name of the function is overloaded. The format of the specific name is not defined, it should only be used to compare it to other instances of specific routine names.
routine_catalog	sql_identifier	Name of the database containing the function (always the current database)
routine_schema	sql_identifier	Name of the schema containing the function
routine_name	sql_identifier	Name of the function (may be duplicated in case of overloading)
routine_type	character_data	Always FUNCTION (In the future there might be other types of routines.)
module_catalog	sql_identifier	Applies to a feature not available in PostgreSQL

Name	Data Type	Description
module_schema	sql_identifier	Applies to a feature not available in PostgreSQL
module_name	sql_identifier	Applies to a feature not available in PostgreSQL
udt_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
udt_schema	sql_identifier	Applies to a feature not available in PostgreSQL
udt_name	sql_identifier	Applies to a feature not available in PostgreSQL
data_type	character_data	Return data type of the function, if it is a built-in type, or ARRAY if it is some array (in that case, see the view element_types), else USER-DEFINED (in that case, the type is identified in type_udt_name and associated columns).
character_maximum_length	cardinal_number	Always null, since this information is not applied to return data types in PostgreSQL
character_octet_length	cardinal_number	Always null, since this information is not applied to return data types in PostgreSQL
character_set_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
character_set_schema	sql_identifier	Applies to a feature not available in PostgreSQL
character_set_name	sql_identifier	Applies to a feature not available in PostgreSQL
collation_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
collation_schema	sql_identifier	Applies to a feature not available in PostgreSQL
collation_name	sql_identifier	Applies to a feature not available in PostgreSQL
numeric_precision	cardinal_number	Always null, since this information is not applied to return data types in PostgreSQL
numeric_precision_radix	cardinal_number	Always null, since this information is not applied to return data types in PostgreSQL

Name	Data Type	Description
numeric_scale	cardinal_number	Always null, since this information is not applied to return data types in PostgreSQL
datetime_precision	cardinal_number	Always null, since this information is not applied to return data types in PostgreSQL
interval_type	character_data	Always null, since this information is not applied to return data types in PostgreSQL
interval_precision	character_data	Always null, since this information is not applied to return data types in PostgreSQL
type_udt_catalog	sql_identifier	Name of the database that the return data type of the function is defined in (always the current database)
type_udt_schema	sql_identifier	Name of the schema that the return data type of the function is defined in
type_udt_name	sql_identifier	Name of the return data type of the function
scope_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
scope_schema	sql_identifier	Applies to a feature not available in PostgreSQL
scope_name	sql_identifier	Applies to a feature not available in PostgreSQL
maximum_cardinality	cardinal_number	Always null, because arrays always have unlimited maximum cardinality in PostgreSQL
dtd_identifier	sql_identifier	An identifier of the data type descriptor of the return data type of this function, unique among the data type descriptors pertaining to the function. This is mainly useful for joining with other instances of such identifiers. (The specific format of the identifier is not defined and not guaranteed to remain the same in future versions.)
routine_body	character_data	If the function is an SQL function, then SQL, else EXTERNAL.

Name	Data Type	Description
routine_definition	character_data	The source text of the function (null if the function is not owned by a currently enabled role). (According to the SQL standard, this column is only applicable if routine_body is SQL, but in PostgreSQL it will contain whatever source text was specified when the function was created.)
external_name	character_data	If this function is a C function, then the external name (link symbol) of the function; else null. (This works out to be the same value that is shown in routine_definition.)
external_language	character_data	The language the function is written in
parameter_style	character_data	Always GENERAL (The SQL standard defines other parameter styles, which are not available in PostgreSQL.)
is_deterministic	character_data	If the function is declared immutable (called deterministic in the SQL standard), then YES, else NO. (You cannot query the other volatility levels available in PostgreSQL through the information schema.)
sql_data_access	character_data	Always MODIFIES, meaning that the function possibly modifies SQL data. This information is not useful for PostgreSQL.
is_null_call	character_data	If the function automatically returns null if any of its arguments are null, then YES, else NO.
sql_path	character_data	Applies to a feature not available in PostgreSQL

Name	Data Type	Description
schema_level_routine	character_data	Always YES (The opposite would be a method of a user-defined type, which is a feature not available in PostgreSQL.)
max_dynamic_result_sets	cardinal_number	Applies to a feature not available in PostgreSQL
is_user_defined_cast	character_data	Applies to a feature not available in PostgreSQL
is_implicitly_invocable	character_data	Applies to a feature not available in PostgreSQL
security_type	character_data	If the function runs with the privileges of the current user, then INVOKER, if the function runs with the privileges of the user who defined it, then DEFINER.
to_sql_specific_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
to_sql_specific_schema	sql_identifier	Applies to a feature not available in PostgreSQL
to_sql_specific_name	sql_identifier	Applies to a feature not available in PostgreSQL
as_locator	character_data	Applies to a feature not available in PostgreSQL
created	time_stamp	Applies to a feature not available in PostgreSQL
last_altered	time_stamp	Applies to a feature not available in PostgreSQL
new_savepoint_level	character_data	Applies to a feature not available in PostgreSQL
is_udt_dependent	character_data	Applies to a feature not available in PostgreSQL
result_cast_from_data_type	character_data	Applies to a feature not available in PostgreSQL
result_cast_as_locator	character_data	Applies to a feature not available in PostgreSQL
result_cast_char_max_length	cardinal_number	Applies to a feature not available in PostgreSQL
result_cast_char_octet_length	character_data	Applies to a feature not available in PostgreSQL
result_cast_char_set_catalog	sql_identifier	Applies to a feature not available in PostgreSQL

Name	Data Type	Description
result_cast_char_set_schema	sql_identifier	Applies to a feature not available in PostgreSQL
result_cast_char_set_name	sql_identifier	Applies to a feature not available in PostgreSQL
result_cast_collation_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
result_cast_collation_schema	sql_identifier	Applies to a feature not available in PostgreSQL
result_cast_collation_name	sql_identifier	Applies to a feature not available in PostgreSQL
result_cast_numeric_precision	cardinal_number	Applies to a feature not available in PostgreSQL
result_cast_numeric_precision_radix	cardinal_number	Applies to a feature not available in PostgreSQL
result_cast_numeric_scale	cardinal_number	Applies to a feature not available in PostgreSQL
result_cast_datetime_precision	character_data	Applies to a feature not available in PostgreSQL
result_cast_interval_type	character_data	Applies to a feature not available in PostgreSQL
result_cast_interval_precision	character_data	Applies to a feature not available in PostgreSQL
result_cast_type_udt_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
result_cast_type_udt_schema	sql_identifier	Applies to a feature not available in PostgreSQL
result_cast_type_udt_name	sql_identifier	Applies to a feature not available in PostgreSQL
result_cast_scope_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
result_cast_scope_schema	sql_identifier	Applies to a feature not available in PostgreSQL
result_cast_scope_name	sql_identifier	Applies to a feature not available in PostgreSQL
result_cast_maximum_cardinality	cardinal_number	Applies to a feature not available in PostgreSQL
result_cast_dtd_identifier	sql_identifier	Applies to a feature not available in PostgreSQL

32.30. schemata

The view schemata contains all schemas in the current database that are owned by a currently enabled role.

Table 32-28. schemata Columns

Name	Data Type	Description
catalog_name	sql_identifier	Name of the database that the schema is contained in (always the current database)
schema_name	sql_identifier	Name of the schema
schema_owner	sql_identifier	Name of the owner of the schema
default_character_set_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
default_character_set_schema	sql_identifier	Applies to a feature not available in PostgreSQL
default_character_set_name	sql_identifier	Applies to a feature not available in PostgreSQL
sql_path	character_data	Applies to a feature not available in PostgreSQL

32.31. sequences

The view sequences contains all sequences defined in the current database. Only those sequences are shown that the current user has access to (by way of being the owner or having some privilege).

Table 32-29. sequences Columns

Name	Data Type	Description
sequence_catalog	sql_identifier	Name of the database that contains the sequence (always the current database)
sequence_schema	sql_identifier	Name of the schema that contains the sequence
sequence_name	sql_identifier	Name of the sequence
data_type	character_data	The data type of the sequence. In PostgreSQL, this is currently always bigint.

Name	Data Type	Description
numeric_precision	cardinal_number	This column contains the (declared or implicit) precision of the sequence data type (see above). The precision indicates the number of significant digits. It may be expressed in decimal (base 10) or binary (base 2) terms, as specified in the column numeric_precision_radix.
numeric_precision_radix	cardinal_number	This column indicates in which base the values in the columns numeric_precision and numeric_scale are expressed. The value is either 2 or 10.
numeric_scale	cardinal_number	This column contains the (declared or implicit) scale of the sequence data type (see above). The scale indicates the number of significant digits to the right of the decimal point. It may be expressed in decimal (base 10) or binary (base 2) terms, as specified in the column numeric_precision_radix.
maximum_value	cardinal_number	Not yet implemented
minimum_value	cardinal_number	Not yet implemented
increment	cardinal_number	Not yet implemented
cycle_option	character_data	Not yet implemented

32.32. sql_features

The table sql_features contains information about which formal features defined in the SQL standard are supported by PostgreSQL. This is the same information that is presented in Appendix D. There you can also find some additional background information.

Table 32-30. sql_features Columns

Name	Data Type	Description
feature_id	character_data	Identifier string of the feature
feature_name	character_data	Descriptive name of the feature

Name	Data Type	Description
sub_feature_id	character_data	Identifier string of the subfeature, or a zero-length string if not a subfeature
sub_feature_name	character_data	Descriptive name of the subfeature, or a zero-length string if not a subfeature
is_supported	character_data	YES if the feature is fully supported by the current version of PostgreSQL, NO if not
is_verified_by	character_data	Always null, since the PostgreSQL development group does not perform formal testing of feature conformance
comments	character_data	Possibly a comment about the supported status of the feature

32.33. sql_implementation_info

The table sql_implementation_info contains information about various aspects that are left implementation-defined by the SQL standard. This information is primarily intended for use in the context of the ODBC interface; users of other interfaces will probably find this information to be of little use. For this reason, the individual implementation information items are not described here; you will find them in the description of the ODBC interface.

Table 32-31. sql_implementation_info Columns

Name	Data Type	Description
implementation_info_id	character_data	Identifier string of the implementation information item
implementation_info_name	character_data	Descriptive name of the implementation information item
integer_value	cardinal_number	Value of the implementation information item, or null if the value is contained in the column character_value
character_value	character_data	Value of the implementation information item, or null if the value is contained in the column integer_value

Name	Data Type	Description
comments	character_data	Possibly a comment pertaining to the implementation information item

32.34. sql_languages

The table sql_languages contains one row for each SQL language binding that is supported by PostgreSQL. PostgreSQL supports direct SQL and embedded SQL in C; that is all you will learn from this table.

Table 32-32. sql_languages Columns

Name	Data Type	Description
sql_language_source	character_data	The name of the source of the language definition; always ISO 9075, that is, the SQL standard
sql_language_year	character_data	The year the standard referenced in sql_language_source was approved; currently 2003
sql_language_conformance	character_data	The standard conformance level for the language binding. For ISO 9075:2003 this is always CORE.
sql_language_integrity	character_data	Always null (This value is relevant to an earlier version of the SQL standard.)
sql_language_implementation	character_data	Always null
sql_language_binding_style	character_data	The language binding style, either DIRECT or EMBEDDED
sql_language_programming_language	character_data	The programming language, if the binding style is EMBEDDED, else null. PostgreSQL only supports the language C.

32.35. sql_packages

The table sql_packages contains information about which feature packages defined in the SQL standard are supported by PostgreSQL. Refer to Appendix D for background information on feature packages.

Table 32-33. `sql_packages` Columns

Name	Data Type	Description
feature_id	character_data	Identifier string of the package
feature_name	character_data	Descriptive name of the package
is_supported	character_data	YES if the package is fully supported by the current version of PostgreSQL, NO if not
is_verified_by	character_data	Always null, since the PostgreSQL development group does not perform formal testing of feature conformance
comments	character_data	Possibly a comment about the supported status of the package

32.36. sql_parts

The table sql_parts contains information about which of the several parts of the SQL standard are supported by PostgreSQL.

Table 32-34. `sql_parts` Columns

Name	Data Type	Description
feature_id	character_data	An identifier string containing the number of the part
feature_name	character_data	Descriptive name of the part
is_supported	character_data	YES if the part is fully supported by the current version of PostgreSQL, NO if not
is_verified_by	character_data	Always null, since the PostgreSQL development group does not perform formal testing of feature conformance
comments	character_data	Possibly a comment about the supported status of the part

32.37. sql_sizing

The table sql_sizing contains information about various size limits and maximum values in PostgreSQL. This information is primarily intended for use in the context of the ODBC interface; users of other interfaces will probably find this information to be of little use. For this reason, the individual sizing items are not described here; you will find them in the description of the ODBC interface.

Table 32-35. `sql_sizing` Columns

Name	Data Type	Description
`sizing_id`	`cardinal_number`	Identifier of the sizing item
`sizing_name`	`character_data`	Descriptive name of the sizing item
`supported_value`	`cardinal_number`	Value of the sizing item, or 0 if the size is unlimited or cannot be determined, or null if the features for which the sizing item is applicable are not supported
`comments`	`character_data`	Possibly a comment pertaining to the sizing item

32.38. `sql_sizing_profiles`

The table `sql_sizing_profiles` contains information about the `sql_sizing` values that are required by various profiles of the SQL standard. PostgreSQL does not track any SQL profiles, so this table is empty.

Table 32-36. `sql_sizing_profiles` Columns

Name	Data Type	Description
`sizing_id`	`cardinal_number`	Identifier of the sizing item
`sizing_name`	`character_data`	Descriptive name of the sizing item
`profile_id`	`character_data`	Identifier string of a profile
`required_value`	`cardinal_number`	The value required by the SQL profile for the sizing item, or 0 if the profile places no limit on the sizing item, or null if the profile does not require any of the features for which the sizing item is applicable
`comments`	`character_data`	Possibly a comment pertaining to the sizing item within the profile

32.39. `table_constraints`

The view `table_constraints` contains all constraints belonging to tables that the current user owns or has some privilege on.

Table 32-37. `table_constraints` Columns

Name	Data Type	Description
`constraint_catalog`	`sql_identifier`	Name of the database that contains the constraint (always the current database)
`constraint_schema`	`sql_identifier`	Name of the schema that contains the constraint
`constraint_name`	`sql_identifier`	Name of the constraint
`table_catalog`	`sql_identifier`	Name of the database that contains the table (always the current database)
`table_schema`	`sql_identifier`	Name of the schema that contains the table
`table_name`	`sql_identifier`	Name of the table
`constraint_type`	`character_data`	Type of the constraint: `CHECK`, `FOREIGN KEY`, `PRIMARY KEY`, or `UNIQUE`
`is_deferrable`	`character_data`	`YES` if the constraint is deferrable, `NO` if not
`initially_deferred`	`character_data`	`YES` if the constraint is deferrable and initially deferred, `NO` if not

32.40. `table_privileges`

The view `table_privileges` identifies all privileges granted on tables or views to a currently enabled role or by a currently enabled role. There is one row for each combination of table, grantor, and grantee.

Table 32-38. `table_privileges` Columns

Name	Data Type	Description
`grantor`	`sql_identifier`	Name of the role that granted the privilege
`grantee`	`sql_identifier`	Name of the role that the privilege was granted to
`table_catalog`	`sql_identifier`	Name of the database that contains the table (always the current database)
`table_schema`	`sql_identifier`	Name of the schema that contains the table
`table_name`	`sql_identifier`	Name of the table

Name	Data Type	Description
privilege_type	character_data	Type of the privilege: SELECT, DELETE, INSERT, UPDATE, REFERENCES, or TRIGGER
is_grantable	character_data	YES if the privilege is grantable, NO if not
with_hierarchy	character_data	Applies to a feature not available in PostgreSQL

32.41. tables

The view tables contains all tables and views defined in the current database. Only those tables and views are shown that the current user has access to (by way of being the owner or having some privilege).

Table 32-39. tables Columns

Name	Data Type	Description
table_catalog	sql_identifier	Name of the database that contains the table (always the current database)
table_schema	sql_identifier	Name of the schema that contains the table
table_name	sql_identifier	Name of the table
table_type	character_data	Type of the table: BASE TABLE for a persistent base table (the normal table type), VIEW for a view, or LOCAL TEMPORARY for a temporary table
self_referencing_column_name	sql_identifier	Applies to a feature not available in PostgreSQL
reference_generation	character_data	Applies to a feature not available in PostgreSQL
user_defined_type_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
user_defined_type_schema	sql_identifier	Applies to a feature not available in PostgreSQL
user_defined_type_name	sql_identifier	Applies to a feature not available in PostgreSQL
is_insertable_into	character_data	YES if the table is insertable into, NO if not (Base tables are always insertable into, views not necessarily.)

Name	Data Type	Description
is_typed	character_data	Applies to a feature not available in PostgreSQL
commit_action	character_data	If the table is a temporary table, then PRESERVE, else null. (The SQL standard defines other commit actions for temporary tables, which are not supported by PostgreSQL.)

32.42. triggers

The view triggers contains all triggers defined in the current database on tables that the current user owns or has some privilege on.

Table 32-40. triggers Columns

Name	Data Type	Description
trigger_catalog	sql_identifier	Name of the database that contains the trigger (always the current database)
trigger_schema	sql_identifier	Name of the schema that contains the trigger
trigger_name	sql_identifier	Name of the trigger
event_manipulation	character_data	Event that fires the trigger (INSERT, UPDATE, or DELETE)
event_object_catalog	sql_identifier	Name of the database that contains the table that the trigger is defined on (always the current database)
event_object_schema	sql_identifier	Name of the schema that contains the table that the trigger is defined on
event_object_table	sql_identifier	Name of the table that the trigger is defined on
action_order	cardinal_number	Not yet implemented
action_condition	character_data	Applies to a feature not available in PostgreSQL
action_statement	character_data	Statement that is executed by the trigger (currently always EXECUTE PROCEDURE *function*(...))

Name	Data Type	Description
action_orientation	character_data	Identifies whether the trigger fires once for each processed row or once for each statement (ROW or STATEMENT)
condition_timing	character_data	Time at which the trigger fires (BEFORE or AFTER)
condition_reference_old_table	sql_identifier	Applies to a feature not available in PostgreSQL
condition_reference_new_table	sql_identifier	Applies to a feature not available in PostgreSQL
condition_reference_old_row	sql_identifier	Applies to a feature not available in PostgreSQL
condition_reference_new_row	sql_identifier	Applies to a feature not available in PostgreSQL
created	time_stamp	Applies to a feature not available in PostgreSQL

Triggers in PostgreSQL have two incompatibilities with the SQL standard that affect the representation in the information schema. First, trigger names are local to the table in PostgreSQL, rather than being independent schema objects. Therefore there may be duplicate trigger names defined in one schema, as long as they belong to different tables. (trigger_catalog and trigger_schema are really the values pertaining to the table that the trigger is defined on.) Second, triggers can be defined to fire on multiple events in PostgreSQL (e.g., ON INSERT OR UPDATE), whereas the SQL standard only allows one. If a trigger is defined to fire on multiple events, it is represented as multiple rows in the information schema, one for each type of event. As a consequence of these two issues, the primary key of the view triggers is really (trigger_catalog, trigger_schema, trigger_name, event_object_table, event_manipulation) instead of (trigger_catalog, trigger_schema, trigger_name), which is what the SQL standard specifies. Nonetheless, if you define your triggers in a manner that conforms with the SQL standard (trigger names unique in the schema and only one event type per trigger), this will not affect you.

32.43. usage_privileges

The view usage_privileges is meant to identify USAGE privileges granted on various kinds of objects to a currently enabled role or by a currently enabled role. In PostgreSQL, this currently only applies to domains, and since domains do not have real privileges in PostgreSQL, this view shows implicit USAGE privileges granted to PUBLIC for all domains. In the future, this view may contain more useful information.

Table 32-41. usage_privileges Columns

Name	Data Type	Description
grantor	sql_identifier	Currently set to the name of the owner of the object

Name	Data Type	Description
grantee	sql_identifier	Currently always PUBLIC
object_catalog	sql_identifier	Name of the database containing the object (always the current database)
object_schema	sql_identifier	Name of the schema containing the object
object_name	sql_identifier	Name of the object
object_type	character_data	Currently always DOMAIN
privilege_type	character_data	Always USAGE
is_grantable	character_data	Currently always NO

32.44. view_column_usage

The view view_column_usage identifies all columns that are used in the query expression of a view (the SELECT statement that defines the view). A column is only included if the table that contains the column is owned by a currently enabled role.

Note: Columns of system tables are not included. This should be fixed sometime.

Table 32-42. view_column_usage Columns

Name	Data Type	Description
view_catalog	sql_identifier	Name of the database that contains the view (always the current database)
view_schema	sql_identifier	Name of the schema that contains the view
view_name	sql_identifier	Name of the view
table_catalog	sql_identifier	Name of the database that contains the table that contains the column that is used by the view (always the current database)
table_schema	sql_identifier	Name of the schema that contains the table that contains the column that is used by the view
table_name	sql_identifier	Name of the table that contains the column that is used by the view

Name	Data Type	Description
column_name	sql_identifier	Name of the column that is used by the view

32.45. view_routine_usage

The view view_routine_usage identifies all routines (functions and procedures) that are used in the query expression of a view (the SELECT statement that defines the view). A routine is only included if that routine is owned by a currently enabled role.

Table 32-43. view_routine_usage Columns

Name	Data Type	Description
table_catalog	sql_identifier	Name of the database containing the view (always the current database)
table_schema	sql_identifier	Name of the schema containing the view
table_name	sql_identifier	Name of the view
specific_catalog	sql_identifier	Name of the database containing the function (always the current database)
specific_schema	sql_identifier	Name of the schema containing the function
specific_name	sql_identifier	The "specific name" of the function. See Section 32.29 for more information.

32.46. view_table_usage

The view view_table_usage identifies all tables that are used in the query expression of a view (the SELECT statement that defines the view). A table is only included if that table is owned by a currently enabled role.

Note: System tables are not included. This should be fixed sometime.

Table 32-44. view_table_usage Columns

Name	Data Type	Description

Name	Data Type	Description
view_catalog	sql_identifier	Name of the database that contains the view (always the current database)
view_schema	sql_identifier	Name of the schema that contains the view
view_name	sql_identifier	Name of the view
table_catalog	sql_identifier	Name of the database that contains the table that is used by the view (always the current database)
table_schema	sql_identifier	Name of the schema that contains the table that is used by the view
table_name	sql_identifier	Name of the table that is used by the view

32.47. views

The view views contains all views defined in the current database. Only those views are shown that the current user has access to (by way of being the owner or having some privilege).

Table 32-45. views Columns

Name	Data Type	Description
table_catalog	sql_identifier	Name of the database that contains the view (always the current database)
table_schema	sql_identifier	Name of the schema that contains the view
table_name	sql_identifier	Name of the view
view_definition	character_data	Query expression defining the view (null if the view is not owned by a currently enabled role)
check_option	character_data	Applies to a feature not available in PostgreSQL
is_updatable	character_data	YES if the view is updatable (allows UPDATE and DELETE), NO if not
is_insertable_into	character_data	YES if the view is insertable into (allows INSERT), NO if not

V. Server Programming

This part is about extending the server functionality with user-defined functions, data types, triggers, etc. These are advanced topics which should probably be approached only after all the other user documentation about PostgreSQL has been understood. Later chapters in this part describe the server-side programming languages available in the PostgreSQL distribution as well as general issues concerning server-side programming languages. It is essential to read at least the earlier sections of Chapter 33 (covering functions) before diving into the material about server-side programming languages.

Chapter 33. Extending SQL

In the sections that follow, we will discuss how you can extend the PostgreSQL SQL query language by adding:

- functions (starting in Section 33.3)
- aggregates (starting in Section 33.10)
- data types (starting in Section 33.11)
- operators (starting in Section 33.12)
- operator classes for indexes (starting in Section 33.14)

33.1. How Extensibility Works

PostgreSQL is extensible because its operation is catalog-driven. If you are familiar with standard relational database systems, you know that they store information about databases, tables, columns, etc., in what are commonly known as system catalogs. (Some systems call this the data dictionary.) The catalogs appear to the user as tables like any other, but the DBMS stores its internal bookkeeping in them. One key difference between PostgreSQL and standard relational database systems is that PostgreSQL stores much more information in its catalogs: not only information about tables and columns, but also information about data types, functions, access methods, and so on. These tables can be modified by the user, and since PostgreSQL bases its operation on these tables, this means that PostgreSQL can be extended by users. By comparison, conventional database systems can only be extended by changing hardcoded procedures in the source code or by loading modules specially written by the DBMS vendor.

The PostgreSQL server can moreover incorporate user-written code into itself through dynamic loading. That is, the user can specify an object code file (e.g., a shared library) that implements a new type or function, and PostgreSQL will load it as required. Code written in SQL is even more trivial to add to the server. This ability to modify its operation "on the fly" makes PostgreSQL uniquely suited for rapid prototyping of new applications and storage structures.

33.2. The PostgreSQL Type System

PostgreSQL data types are divided into base types, composite types, domains, and pseudo-types.

33.2.1. Base Types

Base types are those, like `int4`, that are implemented below the level of the SQL language (typically in a low-level language such as C). They generally correspond to what are often known as abstract data types. PostgreSQL can only operate on such types through functions provided by the user and only understands the behavior of such types to the extent that the user describes them. Base types are further subdivided into scalar and array types. For each scalar type, a corresponding array type is automatically created that can hold variable-size arrays of that scalar type.

33.2.2. Composite Types

Composite types, or row types, are created whenever the user creates a table. It is also possible to use *CREATE TYPE* to define a "stand-alone" composite type with no associated table. A composite type is simply a list of types with associated field names. A value of a composite type is a row or record of field values. The user can access the component fields from SQL queries. Refer to Section 8.11 for more information on composite types.

33.2.3. Domains

A domain is based on a particular base type and for many purposes is interchangeable with its base type. However, a domain may have constraints that restrict its valid values to a subset of what the underlying base type would allow.

Domains can be created using the SQL command *CREATE DOMAIN*. Their creation and use is not discussed in this chapter.

33.2.4. Pseudo-Types

There are a few "pseudo-types" for special purposes. Pseudo-types cannot appear as columns of tables or attributes of composite types, but they can be used to declare the argument and result types of functions. This provides a mechanism within the type system to identify special classes of functions. Table 8-20 lists the existing pseudo-types.

33.2.5. Polymorphic Types

Two pseudo-types of special interest are `anyelement` and `anyarray`, which are collectively called *polymorphic types*. Any function declared using these types is said to be a *polymorphic function*. A polymorphic function can operate on many different data types, with the specific data type(s) being determined by the data types actually passed to it in a particular call.

Polymorphic arguments and results are tied to each other and are resolved to a specific data type when a query calling a polymorphic function is parsed. Each position (either argument or return value) declared as `anyelement` is allowed to have any specific actual data type, but in any given call they must all be the *same* actual type. Each position declared as `anyarray` can have any array data type, but similarly they must all be the same type. If there are positions declared `anyarray` and others declared `anyelement`, the actual array type in the `anyarray` positions must be an array whose elements are the same type appearing in the `anyelement` positions.

Thus, when more than one argument position is declared with a polymorphic type, the net effect is that only certain combinations of actual argument types are allowed. For example, a function declared as `equal(anyelement, anyelement)` will take any two input values, so long as they are of the same data type.

When the return value of a function is declared as a polymorphic type, there must be at least one argument position that is also polymorphic, and the actual data type supplied as the argument determines the actual result type for that call. For example, if there were not already an array subscripting mechanism, one could define a function that implements subscripting as `subscript(anyarray, integer) returns`

anyelement. This declaration constrains the actual first argument to be an array type, and allows the parser to infer the correct result type from the actual first argument's type.

33.3. User-Defined Functions

PostgreSQL provides four kinds of functions:

- query language functions (functions written in SQL) (Section 33.4)
- procedural language functions (functions written in, for example, PL/pgSQL or PL/Tcl) (Section 33.7)
- internal functions (Section 33.8)
- C-language functions (Section 33.9)

Every kind of function can take base types, composite types, or combinations of these as arguments (parameters). In addition, every kind of function can return a base type or a composite type. Functions may also be defined to return sets of base or composite values.

Many kinds of functions can take or return certain pseudo-types (such as polymorphic types), but the available facilities vary. Consult the description of each kind of function for more details.

It's easiest to define SQL functions, so we'll start by discussing those. Most of the concepts presented for SQL functions will carry over to the other types of functions.

Throughout this chapter, it can be useful to look at the reference page of the *CREATE FUNCTION* command to understand the examples better. Some examples from this chapter can be found in funcs.sql and funcs.c in the src/tutorial directory in the PostgreSQL source distribution.

33.4. Query Language (SQL) Functions

SQL functions execute an arbitrary list of SQL statements, returning the result of the last query in the list. In the simple (non-set) case, the first row of the last query's result will be returned. (Bear in mind that "the first row" of a multirow result is not well-defined unless you use ORDER BY.) If the last query happens to return no rows at all, the null value will be returned.

Alternatively, an SQL function may be declared to return a set, by specifying the function's return type as SETOF *sometype*. In this case all rows of the last query's result are returned. Further details appear below.

The body of an SQL function must be a list of SQL statements separated by semicolons. A semicolon after the last statement is optional. Unless the function is declared to return void, the last statement must be a SELECT.

Any collection of commands in the SQL language can be packaged together and defined as a function. Besides SELECT queries, the commands can include data modification queries (INSERT, UPDATE, and DELETE), as well as other SQL commands. (The only exception is that you can't put BEGIN, COMMIT, ROLLBACK, or SAVEPOINT commands into a SQL function.) However, the final command must be a SELECT that returns whatever is specified as the function's return type. Alternatively, if you want to define

a SQL function that performs actions but has no useful value to return, you can define it as returning `void`. In that case, the function body must not end with a `SELECT`. For example, this function removes rows with negative salaries from the `emp` table:

```
CREATE FUNCTION clean_emp() RETURNS void AS '
    DELETE FROM emp
        WHERE salary < 0;
' LANGUAGE SQL;

SELECT clean_emp();

 clean_emp
-----------

(1 row)
```

The syntax of the `CREATE FUNCTION` command requires the function body to be written as a string constant. It is usually most convenient to use dollar quoting (see Section 4.1.2.2) for the string constant. If you choose to use regular single-quoted string constant syntax, you must double single quote marks (`'`) and backslashes (`\`) (assuming escape string syntax) in the body of the function (see Section 4.1.2.1).

Arguments to the SQL function are referenced in the function body using the syntax `$n`: `$1` refers to the first argument, `$2` to the second, and so on. If an argument is of a composite type, then the dot notation, e.g., `$1.name`, may be used to access attributes of the argument. The arguments can only be used as data values, not as identifiers. Thus for example this is reasonable:

```
INSERT INTO mytable VALUES ($1);
```

but this will not work:

```
INSERT INTO $1 VALUES (42);
```

33.4.1. SQL Functions on Base Types

The simplest possible SQL function has no arguments and simply returns a base type, such as `integer`:

```
CREATE FUNCTION one() RETURNS integer AS $$
    SELECT 1 AS result;
$$ LANGUAGE SQL;

-- Alternative syntax for string literal:
CREATE FUNCTION one() RETURNS integer AS '
    SELECT 1 AS result;
' LANGUAGE SQL;

SELECT one();

 one
-----
```

```
   1
```

Notice that we defined a column alias within the function body for the result of the function (with the name `result`), but this column alias is not visible outside the function. Hence, the result is labeled `one` instead of `result`.

It is almost as easy to define SQL functions that take base types as arguments. In the example below, notice how we refer to the arguments within the function as `$1` and `$2`.

```
CREATE FUNCTION add_em(integer, integer) RETURNS integer AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;

SELECT add_em(1, 2) AS answer;

 answer
--------
      3
```

Here is a more useful function, which might be used to debit a bank account:

```
CREATE FUNCTION tf1 (integer, numeric) RETURNS integer AS $$
    UPDATE bank
        SET balance = balance - $2
        WHERE accountno = $1;
    SELECT 1;
$$ LANGUAGE SQL;
```

A user could execute this function to debit account 17 by $100.00 as follows:

```
SELECT tf1(17, 100.0);
```

In practice one would probably like a more useful result from the function than a constant 1, so a more likely definition is

```
CREATE FUNCTION tf1 (integer, numeric) RETURNS numeric AS $$
    UPDATE bank
        SET balance = balance - $2
        WHERE accountno = $1;
    SELECT balance FROM bank WHERE accountno = $1;
$$ LANGUAGE SQL;
```

which adjusts the balance and returns the new balance.

33.4.2. SQL Functions on Composite Types

When writing functions with arguments of composite types, we must not only specify which argument we want (as we did above with `$1` and `$2`) but also the desired attribute (field) of that argument. For example, suppose that `emp` is a table containing employee data, and therefore also the name of the composite type of each row of the table. Here is a function `double_salary` that computes what someone's salary would be if it were doubled:

```
CREATE TABLE emp (
    name        text,
    salary      numeric,
    age         integer,
    cubicle     point
);

CREATE FUNCTION double_salary(emp) RETURNS numeric AS $$
    SELECT $1.salary * 2 AS salary;
$$ LANGUAGE SQL;

SELECT name, double_salary(emp.*) AS dream
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';

 name | dream
------+-------
 Bill |  8400
```

Notice the use of the syntax `$1.salary` to select one field of the argument row value. Also notice how the calling `SELECT` command uses `*` to select the entire current row of a table as a composite value. The table row can alternatively be referenced using just the table name, like this:

```
SELECT name, double_salary(emp) AS dream
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';
```

but this usage is deprecated since it's easy to get confused.

Sometimes it is handy to construct a composite argument value on-the-fly. This can be done with the `ROW` construct. For example, we could adjust the data being passed to the function:

```
SELECT name, double_salary(ROW(name, salary*1.1, age, cubicle)) AS dream
    FROM emp;
```

It is also possible to build a function that returns a composite type. This is an example of a function that returns a single `emp` row:

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT text 'None' AS name,
        1000.0 AS salary,
        25 AS age,
```

```
        point '(2,2)' AS cubicle;
$$ LANGUAGE SQL;
```

In this example we have specified each of the attributes with a constant value, but any computation could have been substituted for these constants.

Note two important things about defining the function:

- The select list order in the query must be exactly the same as that in which the columns appear in the table associated with the composite type. (Naming the columns, as we did above, is irrelevant to the system.)
- You must typecast the expressions to match the definition of the composite type, or you will get errors like this:

  ```
  ERROR:  function declared to return emp returns varchar instead of text at column 1
  ```

A different way to define the same function is:

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT ROW('None', 1000.0, 25, '(2,2)')::emp;
$$ LANGUAGE SQL;
```

Here we wrote a `SELECT` that returns just a single column of the correct composite type. This isn't really better in this situation, but it is a handy alternative in some cases — for example, if we need to compute the result by calling another function that returns the desired composite value.

We could call this function directly in either of two ways:

```
SELECT new_emp();

          new_emp
--------------------------
 (None,1000.0,25,"(2,2)")

SELECT * FROM new_emp();

 name | salary | age | cubicle
------+--------+-----+---------
 None | 1000.0 |  25 | (2,2)
```

The second way is described more fully in Section 33.4.4.

When you use a function that returns a composite type, you might want only one field (attribute) from its result. You can do that with syntax like this:

```
SELECT (new_emp()).name;

 name
------
 None
```

The extra parentheses are needed to keep the parser from getting confused. If you try to do it without them, you get something like this:

```
SELECT new_emp().name;
ERROR:  syntax error at or near "." at character 17
LINE 1: SELECT new_emp().name;
                        ^
```

Another option is to use functional notation for extracting an attribute. The simple way to explain this is that we can use the notations `attribute(table)` and `table.attribute` interchangeably.

```
SELECT name(new_emp());

 name
------
 None

-- This is the same as:
-- SELECT emp.name AS youngster FROM emp WHERE emp.age < 30;

SELECT name(emp) AS youngster FROM emp WHERE age(emp) < 30;

 youngster
-----------
 Sam
 Andy
```

> **Tip:** The equivalence between functional notation and attribute notation makes it possible to use functions on composite types to emulate "computed fields". For example, using the previous definition for `double_salary(emp)`, we can write
>
> ```
> SELECT emp.name, emp.double_salary FROM emp;
> ```
>
> An application using this wouldn't need to be directly aware that `double_salary` isn't a real column of the table. (You can also emulate computed fields with views.)

Another way to use a function returning a composite type is to pass the result to another function that accepts the correct row type as input:

```
CREATE FUNCTION getname(emp) RETURNS text AS $$
    SELECT $1.name;
$$ LANGUAGE SQL;

SELECT getname(new_emp());
 getname
---------
 None
(1 row)
```

Still another way to use a function that returns a composite type is to call it as a table function, as described in Section 33.4.4.

33.4.3. Functions with Output Parameters

An alternative way of describing a function's results is to define it with *output parameters*, as in this example:

```
CREATE FUNCTION add_em (IN x int, IN y int, OUT sum int)
AS 'SELECT $1 + $2'
LANGUAGE SQL;

SELECT add_em(3,7);
 add_em
--------
     10
(1 row)
```

This is not essentially different from the version of `add_em` shown in Section 33.4.1. The real value of output parameters is that they provide a convenient way of defining functions that return several columns. For example,

```
CREATE FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int)
AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;

 SELECT * FROM sum_n_product(11,42);
 sum | product
-----+---------
  53 |     462
(1 row)
```

What has essentially happened here is that we have created an anonymous composite type for the result of the function. The above example has the same end result as

```
CREATE TYPE sum_prod AS (sum int, product int);

CREATE FUNCTION sum_n_product (int, int) RETURNS sum_prod
AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;
```

but not having to bother with the separate composite type definition is often handy.

Notice that output parameters are not included in the calling argument list when invoking such a function from SQL. This is because PostgreSQL considers only the input parameters to define the function's calling signature. That means also that only the input parameters matter when referencing the function for purposes such as dropping it. We could drop the above function with either of

```
DROP FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int);
DROP FUNCTION sum_n_product (int, int);
```

Parameters can be marked as IN (the default), OUT, or INOUT. An INOUT parameter serves as both an input parameter (part of the calling argument list) and an output parameter (part of the result record type).

33.4.4. SQL Functions as Table Sources

All SQL functions may be used in the FROM clause of a query, but it is particularly useful for functions returning composite types. If the function is defined to return a base type, the table function produces a one-column table. If the function is defined to return a composite type, the table function produces a column for each attribute of the composite type.

Here is an example:

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);
INSERT INTO foo VALUES (1, 1, 'Joe');
INSERT INTO foo VALUES (1, 2, 'Ed');
INSERT INTO foo VALUES (2, 1, 'Mary');

CREATE FUNCTION getfoo(int) RETURNS foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT *, upper(fooname) FROM getfoo(1) AS t1;

 fooid | foosubid | fooname | upper
-------+----------+---------+-------
     1 |        1 | Joe     | JOE
(1 row)
```

As the example shows, we can work with the columns of the function's result just the same as if they were columns of a regular table.

Note that we only got one row out of the function. This is because we did not use SETOF. That is described in the next section.

33.4.5. SQL Functions Returning Sets

When an SQL function is declared as returning SETOF *sometype*, the function's final SELECT query is executed to completion, and each row it outputs is returned as an element of the result set.

This feature is normally used when calling the function in the FROM clause. In this case each row returned by the function becomes a row of the table seen by the query. For example, assume that table foo has the same contents as above, and we say:

```
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;
```

Then we would get:

```
 fooid | foosubid | fooname
-------+----------+---------
     1 |        1 | Joe
     1 |        2 | Ed
(2 rows)
```

Currently, functions returning sets may also be called in the select list of a query. For each row that the query generates by itself, the function returning set is invoked, and an output row is generated for each element of the function's result set. Note, however, that this capability is deprecated and may be removed in future releases. The following is an example function returning a set from the select list:

```
CREATE FUNCTION listchildren(text) RETURNS SETOF text AS $$
    SELECT name FROM nodes WHERE parent = $1
$$ LANGUAGE SQL;

SELECT * FROM nodes;
   name    | parent
-----------+--------
 Top       |
 Child1    | Top
 Child2    | Top
 Child3    | Top
 SubChild1 | Child1
 SubChild2 | Child1
(6 rows)

SELECT listchildren('Top');
 listchildren
--------------
 Child1
 Child2
 Child3
(3 rows)

SELECT name, listchildren(name) FROM nodes;
  name   | listchildren
--------+--------------
 Top     | Child1
 Top     | Child2
 Top     | Child3
 Child1  | SubChild1
 Child1  | SubChild2
(5 rows)
```

In the last `SELECT`, notice that no output row appears for `Child2`, `Child3`, etc. This happens because `listchildren` returns an empty set for those arguments, so no result rows are generated.

33.4.6. Polymorphic SQL Functions

SQL functions may be declared to accept and return the polymorphic types `anyelement` and `anyarray`. See Section 33.2.5 for a more detailed explanation of polymorphic functions. Here is a polymorphic function `make_array` that builds up an array from two arbitrary data type elements:

```
CREATE FUNCTION make_array(anyelement, anyelement) RETURNS anyarray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;

SELECT make_array(1, 2) AS intarray, make_array('a'::text, 'b') AS textarray;
 intarray | textarray
----------+-----------
 {1,2}    | {a,b}
(1 row)
```

Notice the use of the typecast `'a'::text` to specify that the argument is of type `text`. This is required if the argument is just a string literal, since otherwise it would be treated as type `unknown`, and array of `unknown` is not a valid type. Without the typecast, you will get errors like this:

```
ERROR:  could not determine "anyarray"/"anyelement" type because input has type "unknown"
```

It is permitted to have polymorphic arguments with a fixed return type, but the converse is not. For example:

```
CREATE FUNCTION is_greater(anyelement, anyelement) RETURNS boolean AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;

SELECT is_greater(1, 2);
 is_greater
------------
 f
(1 row)

CREATE FUNCTION invalid_func() RETURNS anyelement AS $$
    SELECT 1;
$$ LANGUAGE SQL;
ERROR:  cannot determine result data type
DETAIL:  A function returning "anyarray" or "anyelement" must have at least one argument of
```

Polymorphism can be used with functions that have output arguments. For example:

```
CREATE FUNCTION dup (f1 anyelement, OUT f2 anyelement, OUT f3 anyarray)
AS 'select $1, array[$1,$1]' LANGUAGE sql;

SELECT * FROM dup(22);
 f2 |   f3
----+---------
```

```
 22 | {22,22}
(1 row)
```

33.5. Function Overloading

More than one function may be defined with the same SQL name, so long as the arguments they take are different. In other words, function names can be *overloaded*. When a query is executed, the server will determine which function to call from the data types and the number of the provided arguments. Overloading can also be used to simulate functions with a variable number of arguments, up to a finite maximum number.

When creating a family of overloaded functions, one should be careful not to create ambiguities. For instance, given the functions

```
CREATE FUNCTION test(int, real) RETURNS ...
CREATE FUNCTION test(smallint, double precision) RETURNS ...
```

it is not immediately clear which function would be called with some trivial input like `test(1, 1.5)`. The currently implemented resolution rules are described in Chapter 10, but it is unwise to design a system that subtly relies on this behavior.

A function that takes a single argument of a composite type should generally not have the same name as any attribute (field) of that type. Recall that `attribute(table)` is considered equivalent to `table.attribute`. In the case that there is an ambiguity between a function on a composite type and an attribute of the composite type, the attribute will always be used. It is possible to override that choice by schema-qualifying the function name (that is, `schema.func(table)`) but it's better to avoid the problem by not choosing conflicting names.

When overloading C-language functions, there is an additional constraint: The C name of each function in the family of overloaded functions must be different from the C names of all other functions, either internal or dynamically loaded. If this rule is violated, the behavior is not portable. You might get a run-time linker error, or one of the functions will get called (usually the internal one). The alternative form of the `AS` clause for the SQL `CREATE FUNCTION` command decouples the SQL function name from the function name in the C source code. For instance,

```
CREATE FUNCTION test(int) RETURNS int
    AS 'filename', 'test_1arg'
    LANGUAGE C;
CREATE FUNCTION test(int, int) RETURNS int
    AS 'filename', 'test_2arg'
    LANGUAGE C;
```

The names of the C functions here reflect one of many possible conventions.

33.6. Function Volatility Categories

Every function has a *volatility* classification, with the possibilities being `VOLATILE`, `STABLE`, or `IMMUTABLE`. `VOLATILE` is the default if the *CREATE FUNCTION* command does not specify a category. The volatility category is a promise to the optimizer about the behavior of the function:

- A `VOLATILE` function can do anything, including modifying the database. It can return different results on successive calls with the same arguments. The optimizer makes no assumptions about the behavior of such functions. A query using a volatile function will re-evaluate the function at every row where its value is needed.
- A `STABLE` function cannot modify the database and is guaranteed to return the same results given the same arguments for all rows within a single statement. This category allows the optimizer to optimize multiple calls of the function to a single call. In particular, it is safe to use an expression containing such a function in an index scan condition. (Since an index scan will evaluate the comparison value only once, not once at each row, it is not valid to use a `VOLATILE` function in an index scan condition.)
- An `IMMUTABLE` function cannot modify the database and is guaranteed to return the same results given the same arguments forever. This category allows the optimizer to pre-evaluate the function when a query calls it with constant arguments. For example, a query like `SELECT ... WHERE x = 2 + 2` can be simplified on sight to `SELECT ... WHERE x = 4`, because the function underlying the integer addition operator is marked `IMMUTABLE`.

For best optimization results, you should label your functions with the strictest volatility category that is valid for them.

Any function with side-effects *must* be labeled `VOLATILE`, so that calls to it cannot be optimized away. Even a function with no side-effects needs to be labeled `VOLATILE` if its value can change within a single query; some examples are `random()`, `currval()`, `timeofday()`.

There is relatively little difference between `STABLE` and `IMMUTABLE` categories when considering simple interactive queries that are planned and immediately executed: it doesn't matter a lot whether a function is executed once during planning or once during query execution startup. But there is a big difference if the plan is saved and reused later. Labeling a function `IMMUTABLE` when it really isn't may allow it to be prematurely folded to a constant during planning, resulting in a stale value being re-used during subsequent uses of the plan. This is a hazard when using prepared statements or when using function languages that cache plans (such as PL/pgSQL).

Because of the snapshotting behavior of MVCC (see Chapter 12) a function containing only `SELECT` commands can safely be marked `STABLE`, even if it selects from tables that might be undergoing modifications by concurrent queries. PostgreSQL will execute a `STABLE` function using the snapshot established for the calling query, and so it will see a fixed view of the database throughout that query. Also note that the `current_timestamp` family of functions qualify as stable, since their values do not change within a transaction.

The same snapshotting behavior is used for `SELECT` commands within `IMMUTABLE` functions. It is generally unwise to select from database tables within an `IMMUTABLE` function at all, since the immutability will be broken if the table contents ever change. However, PostgreSQL does not enforce that you do not do that.

A common error is to label a function IMMUTABLE when its results depend on a configuration parameter. For example, a function that manipulates timestamps might well have results that depend on the timezone setting. For safety, such functions should be labeled STABLE instead.

> **Note:** Before PostgreSQL release 8.0, the requirement that STABLE and IMMUTABLE functions cannot modify the database was not enforced by the system. Release 8.0 enforces it by requiring SQL functions and procedural language functions of these categories to contain no SQL commands other than SELECT. (This is not a completely bulletproof test, since such functions could still call VOLATILE functions that modify the database. If you do that, you will find that the STABLE or IMMUTABLE function does not notice the database changes applied by the called function.)

33.7. Procedural Language Functions

PostgreSQL allows user-defined functions to be written in other languages besides SQL and C. These other languages are generically called *procedural languages* (PLs). Procedural languages aren't built into the PostgreSQL server; they are offered by loadable modules. See Chapter 36 and following chapters for more information.

33.8. Internal Functions

Internal functions are functions written in C that have been statically linked into the PostgreSQL server. The "body" of the function definition specifies the C-language name of the function, which need not be the same as the name being declared for SQL use. (For reasons of backwards compatibility, an empty body is accepted as meaning that the C-language function name is the same as the SQL name.)

Normally, all internal functions present in the server are declared during the initialization of the database cluster (initdb), but a user could use CREATE FUNCTION to create additional alias names for an internal function. Internal functions are declared in CREATE FUNCTION with language name internal. For instance, to create an alias for the sqrt function:

```
CREATE FUNCTION square_root(double precision) RETURNS double precision
    AS 'dsqrt'
    LANGUAGE internal
    STRICT;
```

(Most internal functions expect to be declared "strict".)

> **Note:** Not all "predefined" functions are "internal" in the above sense. Some predefined functions are written in SQL.

33.9. C-Language Functions

User-defined functions can be written in C (or a language that can be made compatible with C, such as C++). Such functions are compiled into dynamically loadable objects (also called shared libraries) and are loaded by the server on demand. The dynamic loading feature is what distinguishes "C language" functions from "internal" functions — the actual coding conventions are essentially the same for both. (Hence, the standard internal function library is a rich source of coding examples for user-defined C functions.)

Two different calling conventions are currently used for C functions. The newer "version 1" calling convention is indicated by writing a `PG_FUNCTION_INFO_V1()` macro call for the function, as illustrated below. Lack of such a macro indicates an old-style ("version 0") function. The language name specified in `CREATE FUNCTION` is `C` in either case. Old-style functions are now deprecated because of portability problems and lack of functionality, but they are still supported for compatibility reasons.

33.9.1. Dynamic Loading

The first time a user-defined function in a particular loadable object file is called in a session, the dynamic loader loads that object file into memory so that the function can be called. The `CREATE FUNCTION` for a user-defined C function must therefore specify two pieces of information for the function: the name of the loadable object file, and the C name (link symbol) of the specific function to call within that object file. If the C name is not explicitly specified then it is assumed to be the same as the SQL function name.

The following algorithm is used to locate the shared object file based on the name given in the `CREATE FUNCTION` command:

1. If the name is an absolute path, the given file is loaded.
2. If the name starts with the string `$libdir`, that part is replaced by the PostgreSQL package library directory name, which is determined at build time.
3. If the name does not contain a directory part, the file is searched for in the path specified by the configuration variable dynamic_library_path.
4. Otherwise (the file was not found in the path, or it contains a non-absolute directory part), the dynamic loader will try to take the name as given, which will most likely fail. (It is unreliable to depend on the current working directory.)

If this sequence does not work, the platform-specific shared library file name extension (often `.so`) is appended to the given name and this sequence is tried again. If that fails as well, the load will fail.

It is recommended to locate shared libraries either relative to `$libdir` or through the dynamic library path. This simplifies version upgrades if the new installation is at a different location. The actual directory that `$libdir` stands for can be found out with the command `pg_config --pkglibdir`.

The user ID the PostgreSQL server runs as must be able to traverse the path to the file you intend to load. Making the file or a higher-level directory not readable and/or not executable by the postgres user is a common mistake.

In any case, the file name that is given in the `CREATE FUNCTION` command is recorded literally in the system catalogs, so if the file needs to be loaded again the same procedure is applied.

Note: PostgreSQL will not compile a C function automatically. The object file must be compiled before it is referenced in a `CREATE FUNCTION` command. See Section 33.9.6 for additional information.

To ensure that a dynamically loaded object file is not loaded into an incompatible server, PostgreSQL checks that the file contains a "magic block" with the appropriate contents. This allows the server to detect obvious incompatibilities, such as code compiled for a different major version of PostgreSQL. A magic block is required as of PostgreSQL 8.2. To include a magic block, write this in one (and only one) of the module source files, after having included the header `fmgr.h`:

```
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif
```

The `#ifdef` test can be omitted if the code doesn't need to compile against pre-8.2 PostgreSQL releases.

After it is used for the first time, a dynamically loaded object file is retained in memory. Future calls in the same session to the function(s) in that file will only incur the small overhead of a symbol table lookup. If you need to force a reload of an object file, for example after recompiling it, use the *LOAD* command or begin a fresh session.

Optionally, a dynamically loaded file can contain initialization and finalization functions. If the file includes a function named `_PG_init`, that function will be called immediately after loading the file. The function receives no parameters and should return void. If the file includes a function named `_PG_fini`, that function will be called immediately before unloading the file. Likewise, the function receives no parameters and should return void. Note that `_PG_fini` will only be called during an unload of the file, not during process termination. (Presently, an unload only happens in the context of re-loading the file due to an explicit `LOAD` command.)

33.9.2. Base Types in C-Language Functions

To know how to write C-language functions, you need to know how PostgreSQL internally represents base data types and how they can be passed to and from functions. Internally, PostgreSQL regards a base type as a "blob of memory". The user-defined functions that you define over a type in turn define the way that PostgreSQL can operate on it. That is, PostgreSQL will only store and retrieve the data from disk and use your user-defined functions to input, process, and output the data.

Base types can have one of three internal formats:

- pass by value, fixed-length
- pass by reference, fixed-length
- pass by reference, variable-length

By-value types can only be 1, 2, or 4 bytes in length (also 8 bytes, if `sizeof(Datum)` is 8 on your machine). You should be careful to define your types such that they will be the same size (in bytes) on all architectures. For example, the `long` type is dangerous because it is 4 bytes on some machines and 8 bytes on others, whereas `int` type is 4 bytes on most Unix machines. A reasonable implementation of the `int4` type on Unix machines might be:

```
/* 4-byte integer, passed by value */
typedef int int4;
```

On the other hand, fixed-length types of any size may be passed by-reference. For example, here is a sample implementation of a PostgreSQL type:

```
/* 16-byte structure, passed by reference */
typedef struct
{
    double  x, y;
} Point;
```

Only pointers to such types can be used when passing them in and out of PostgreSQL functions. To return a value of such a type, allocate the right amount of memory with `palloc`, fill in the allocated memory, and return a pointer to it. (Also, if you just want to return the same value as one of your input arguments that's of the same data type, you can skip the extra `palloc` and just return the pointer to the input value.)

Finally, all variable-length types must also be passed by reference. All variable-length types must begin with a length field of exactly 4 bytes, and all data to be stored within that type must be located in the memory immediately following that length field. The length field contains the total length of the structure, that is, it includes the size of the length field itself.

> **Warning**
>
> *Never* modify the contents of a pass-by-reference input value. If you do so you are likely to corrupt on-disk data, since the pointer you are given may well point directly into a disk buffer. The sole exception to this rule is explained in Section 33.10.

As an example, we can define the type `text` as follows:

```
typedef struct {
    int4 length;
    char data[1];
} text;
```

Obviously, the data field declared here is not long enough to hold all possible strings. Since it's impossible to declare a variable-size structure in C, we rely on the knowledge that the C compiler won't range-check array subscripts. We just allocate the necessary amount of space and then access the array as if it were declared the right length. (This is a common trick, which you can read about in many textbooks about C.)

When manipulating variable-length types, we must be careful to allocate the correct amount of memory and set the length field correctly. For example, if we wanted to store 40 bytes in a `text` structure, we might use a code fragment like this:

```
#include "postgres.h"
...
char buffer[40]; /* our source data */
...
text *destination = (text *) palloc(VARHDRSZ + 40);
destination->length = VARHDRSZ + 40;
memcpy(destination->data, buffer, 40);
```

...

VARHDRSZ is the same as sizeof(int4), but it's considered good style to use the macro VARHDRSZ to refer to the size of the overhead for a variable-length type.

Table 33-1 specifies which C type corresponds to which SQL type when writing a C-language function that uses a built-in type of PostgreSQL. The "Defined In" column gives the header file that needs to be included to get the type definition. (The actual definition may be in a different file that is included by the listed file. It is recommended that users stick to the defined interface.) Note that you should always include postgres.h first in any source file, because it declares a number of things that you will need anyway.

Table 33-1. Equivalent C Types for Built-In SQL Types

SQL Type	C Type	Defined In
abstime	AbsoluteTime	utils/nabstime.h
boolean	bool	postgres.h (maybe compiler built-in)
box	BOX*	utils/geo_decls.h
bytea	bytea*	postgres.h
"char"	char	(compiler built-in)
character	BpChar*	postgres.h
cid	CommandId	postgres.h
date	DateADT	utils/date.h
smallint (int2)	int2 or int16	postgres.h
int2vector	int2vector*	postgres.h
integer (int4)	int4 or int32	postgres.h
real (float4)	float4*	postgres.h
double precision (float8)	float8*	postgres.h
interval	Interval*	utils/timestamp.h
lseg	LSEG*	utils/geo_decls.h
name	Name	postgres.h
oid	Oid	postgres.h
oidvector	oidvector*	postgres.h
path	PATH*	utils/geo_decls.h
point	POINT*	utils/geo_decls.h
regproc	regproc	postgres.h
reltime	RelativeTime	utils/nabstime.h
text	text*	postgres.h
tid	ItemPointer	storage/itemptr.h
time	TimeADT	utils/date.h
time with time zone	TimeTzADT	utils/date.h
timestamp	Timestamp*	utils/timestamp.h
tinterval	TimeInterval	utils/nabstime.h

SQL Type	C Type	Defined In
varchar	VarChar*	postgres.h
xid	TransactionId	postgres.h

Now that we've gone over all of the possible structures for base types, we can show some examples of real functions.

33.9.3. Version 0 Calling Conventions

We present the "old style" calling convention first — although this approach is now deprecated, it's easier to get a handle on initially. In the version-0 method, the arguments and result of the C function are just declared in normal C style, but being careful to use the C representation of each SQL data type as shown above.

Here are some examples:

```
#include "postgres.h"
#include <string.h>

/* by value */

int
add_one(int arg)
{
    return arg + 1;
}

/* by reference, fixed length */

float8 *
add_one_float8(float8 *arg)
{
    float8    *result = (float8 *) palloc(sizeof(float8));

    *result = *arg + 1.0;

    return result;
}

Point *
makepoint(Point *pointx, Point *pointy)
{
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    return new_point;
}

/* by reference, variable length */
```

```
text *
copytext(text *t)
{
    /*
     * VARSIZE is the total size of the struct in bytes.
     */
    text *new_t = (text *) palloc(VARSIZE(t));
    VARATT_SIZEP(new_t) = VARSIZE(t);
    /*
     * VARDATA is a pointer to the data region of the struct.
     */
    memcpy((void *) VARDATA(new_t), /* destination */
           (void *) VARDATA(t),     /* source */
           VARSIZE(t) - VARHDRSZ);  /* how many bytes */
    return new_t;
}

text *
concat_text(text *arg1, text *arg2)
{
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    VARATT_SIZEP(new_text) = new_text_size;
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
    return new_text;
}
```

Supposing that the above code has been prepared in file `funcs.c` and compiled into a shared object, we could define the functions to PostgreSQL with commands like this:

```
CREATE FUNCTION add_one(integer) RETURNS integer
     AS 'DIRECTORY/funcs', 'add_one'
     LANGUAGE C STRICT;

-- note overloading of SQL function name "add_one"
CREATE FUNCTION add_one(double precision) RETURNS double precision
     AS 'DIRECTORY/funcs', 'add_one_float8'
     LANGUAGE C STRICT;

CREATE FUNCTION makepoint(point, point) RETURNS point
     AS 'DIRECTORY/funcs', 'makepoint'
     LANGUAGE C STRICT;

CREATE FUNCTION copytext(text) RETURNS text
     AS 'DIRECTORY/funcs', 'copytext'
     LANGUAGE C STRICT;

CREATE FUNCTION concat_text(text, text) RETURNS text
```

```
AS 'DIRECTORY/funcs', 'concat_text'
LANGUAGE C STRICT;
```

Here, *DIRECTORY* stands for the directory of the shared library file (for instance the PostgreSQL tutorial directory, which contains the code for the examples used in this section). (Better style would be to use just 'funcs' in the AS clause, after having added *DIRECTORY* to the search path. In any case, we may omit the system-specific extension for a shared library, commonly .so or .sl.)

Notice that we have specified the functions as "strict", meaning that the system should automatically assume a null result if any input value is null. By doing this, we avoid having to check for null inputs in the function code. Without this, we'd have to check for null values explicitly, by checking for a null pointer for each pass-by-reference argument. (For pass-by-value arguments, we don't even have a way to check!)

Although this calling convention is simple to use, it is not very portable; on some architectures there are problems with passing data types that are smaller than int this way. Also, there is no simple way to return a null result, nor to cope with null arguments in any way other than making the function strict. The version-1 convention, presented next, overcomes these objections.

33.9.4. Version 1 Calling Conventions

The version-1 calling convention relies on macros to suppress most of the complexity of passing arguments and results. The C declaration of a version-1 function is always

```
Datum funcname(PG_FUNCTION_ARGS)
```

In addition, the macro call

```
PG_FUNCTION_INFO_V1(funcname);
```

must appear in the same source file. (Conventionally. it's written just before the function itself.) This macro call is not needed for internal-language functions, since PostgreSQL assumes that all internal functions use the version-1 convention. It is, however, required for dynamically-loaded functions.

In a version-1 function, each actual argument is fetched using a PG_GETARG_*xxx*() macro that corresponds to the argument's data type, and the result is returned using a PG_RETURN_*xxx*() macro for the return type. PG_GETARG_*xxx*() takes as its argument the number of the function argument to fetch, where the count starts at 0. PG_RETURN_*xxx*() takes as its argument the actual value to return.

Here we show the same functions as above, coded in version-1 style:

```
#include "postgres.h"
#include <string.h>
#include "fmgr.h"

/* by value */

PG_FUNCTION_INFO_V1(add_one);

Datum
add_one(PG_FUNCTION_ARGS)
```

```
{
    int32   arg = PG_GETARG_INT32(0);

    PG_RETURN_INT32(arg + 1);
}

/* by reference, fixed length */

PG_FUNCTION_INFO_V1(add_one_float8);

Datum
add_one_float8(PG_FUNCTION_ARGS)
{
    /* The macros for FLOAT8 hide its pass-by-reference nature. */
    float8   arg = PG_GETARG_FLOAT8(0);

    PG_RETURN_FLOAT8(arg + 1.0);
}

PG_FUNCTION_INFO_V1(makepoint);

Datum
makepoint(PG_FUNCTION_ARGS)
{
    /* Here, the pass-by-reference nature of Point is not hidden. */
    Point     *pointx = PG_GETARG_POINT_P(0);
    Point     *pointy = PG_GETARG_POINT_P(1);
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    PG_RETURN_POINT_P(new_point);
}

/* by reference, variable length */

PG_FUNCTION_INFO_V1(copytext);

Datum
copytext(PG_FUNCTION_ARGS)
{
    text     *t = PG_GETARG_TEXT_P(0);
    /*
     * VARSIZE is the total size of the struct in bytes.
     */
    text     *new_t = (text *) palloc(VARSIZE(t));
    VARATT_SIZEP(new_t) = VARSIZE(t);
    /*
     * VARDATA is a pointer to the data region of the struct.
     */
    memcpy((void *) VARDATA(new_t), /* destination */
           (void *) VARDATA(t),     /* source */
```

```
            VARSIZE(t) - VARHDRSZ);  /* how many bytes */
    PG_RETURN_TEXT_P(new_t);
}

PG_FUNCTION_INFO_V1(concat_text);

Datum
concat_text(PG_FUNCTION_ARGS)
{
    text  *arg1 = PG_GETARG_TEXT_P(0);
    text  *arg2 = PG_GETARG_TEXT_P(1);
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    VARATT_SIZEP(new_text) = new_text_size;
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
    PG_RETURN_TEXT_P(new_text);
}
```

The `CREATE FUNCTION` commands are the same as for the version-0 equivalents.

At first glance, the version-1 coding conventions may appear to be just pointless obscurantism. They do, however, offer a number of improvements, because the macros can hide unnecessary detail. An example is that in coding `add_one_float8`, we no longer need to be aware that `float8` is a pass-by-reference type. Another example is that the `GETARG` macros for variable-length types allow for more efficient fetching of "toasted" (compressed or out-of-line) values.

One big improvement in version-1 functions is better handling of null inputs and results. The macro `PG_ARGISNULL(`*n*`)` allows a function to test whether each input is null. (Of course, doing this is only necessary in functions not declared "strict".) As with the `PG_GETARG_`*xxx*`()` macros, the input arguments are counted beginning at zero. Note that one should refrain from executing `PG_GETARG_`*xxx*`()` until one has verified that the argument isn't null. To return a null result, execute `PG_RETURN_NULL()`; this works in both strict and nonstrict functions.

Other options provided in the new-style interface are two variants of the `PG_GETARG_`*xxx*`()` macros. The first of these, `PG_GETARG_`*xxx*`_COPY()`, guarantees to return a copy of the specified argument that is safe for writing into. (The normal macros will sometimes return a pointer to a value that is physically stored in a table, which must not be written to. Using the `PG_GETARG_`*xxx*`_COPY()` macros guarantees a writable result.) The second variant consists of the `PG_GETARG_`*xxx*`_SLICE()` macros which take three arguments. The first is the number of the function argument (as above). The second and third are the offset and length of the segment to be returned. Offsets are counted from zero, and a negative length requests that the remainder of the value be returned. These macros provide more efficient access to parts of large values in the case where they have storage type "external". (The storage type of a column can be specified using `ALTER TABLE` *`tablename`* `ALTER COLUMN` *`colname`* `SET STORAGE` *`storagetype`*. *`storagetype`* is one of `plain`, `external`, `extended`, or `main`.)

Finally, the version-1 function call conventions make it possible to return set results (Section 33.9.10) and implement trigger functions (Chapter 34) and procedural-language call handlers (Chapter 47). Version-1

code is also more portable than version-0, because it does not break restrictions on function call protocol in the C standard. For more details see `src/backend/utils/fmgr/README` in the source distribution.

33.9.5. Writing Code

Before we turn to the more advanced topics, we should discuss some coding rules for PostgreSQL C-language functions. While it may be possible to load functions written in languages other than C into PostgreSQL, this is usually difficult (when it is possible at all) because other languages, such as C++, FORTRAN, or Pascal often do not follow the same calling convention as C. That is, other languages do not pass argument and return values between functions in the same way. For this reason, we will assume that your C-language functions are actually written in C.

The basic rules for writing and building C functions are as follows:

- Use `pg_config --includedir-server` to find out where the PostgreSQL server header files are installed on your system (or the system that your users will be running on).
- Compiling and linking your code so that it can be dynamically loaded into PostgreSQL always requires special flags. See Section 33.9.6 for a detailed explanation of how to do it for your particular operating system.
- Remember to define a "magic block" for your shared library, as described in Section 33.9.1.
- When allocating memory, use the PostgreSQL functions `palloc` and `pfree` instead of the corresponding C library functions `malloc` and `free`. The memory allocated by `palloc` will be freed automatically at the end of each transaction, preventing memory leaks.
- Always zero the bytes of your structures using `memset`. Without this, it's difficult to support hash indexes or hash joins, as you must pick out only the significant bits of your data structure to compute a hash. Even if you initialize all fields of your structure, there may be alignment padding (holes in the structure) that may contain garbage values.
- Most of the internal PostgreSQL types are declared in `postgres.h`, while the function manager interfaces (`PG_FUNCTION_ARGS`, etc.) are in `fmgr.h`, so you will need to include at least these two files. For portability reasons it's best to include `postgres.h` *first*, before any other system or user header files. Including `postgres.h` will also include `elog.h` and `palloc.h` for you.
- Symbol names defined within object files must not conflict with each other or with symbols defined in the PostgreSQL server executable. You will have to rename your functions or variables if you get error messages to this effect.

33.9.6. Compiling and Linking Dynamically-Loaded Functions

Before you are able to use your PostgreSQL extension functions written in C, they must be compiled and linked in a special way to produce a file that can be dynamically loaded by the server. To be precise, a *shared library* needs to be created.

For information beyond what is contained in this section you should read the documentation of your operating system, in particular the manual pages for the C compiler, `cc`, and the link editor, `ld`. In addition,

the PostgreSQL source code contains several working examples in the `contrib` directory. If you rely on these examples you will make your modules dependent on the availability of the PostgreSQL source code, however.

Creating shared libraries is generally analogous to linking executables: first the source files are compiled into object files, then the object files are linked together. The object files need to be created as *position-independent code* (PIC), which conceptually means that they can be placed at an arbitrary location in memory when they are loaded by the executable. (Object files intended for executables are usually not compiled that way.) The command to link a shared library contains special flags to distinguish it from linking an executable (at least in theory — on some systems the practice is much uglier).

In the following examples we assume that your source code is in a file `foo.c` and we will create a shared library `foo.so`. The intermediate object file will be called `foo.o` unless otherwise noted. A shared library can contain more than one object file, but we only use one here.

BSD/OS

The compiler flag to create PIC is `-fpic`. The linker flag to create shared libraries is `-shared`.

```
gcc -fpic -c foo.c
ld -shared -o foo.so foo.o
```

This is applicable as of version 4.0 of BSD/OS.

FreeBSD

The compiler flag to create PIC is `-fpic`. To create shared libraries the compiler flag is `-shared`.

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

This is applicable as of version 3.0 of FreeBSD.

HP-UX

The compiler flag of the system compiler to create PIC is `+z`. When using GCC it's `-fpic`. The linker flag for shared libraries is `-b`. So

```
cc +z -c foo.c
```

or

```
gcc -fpic -c foo.c
```

and then

```
ld -b -o foo.sl foo.o
```

HP-UX uses the extension `.sl` for shared libraries, unlike most other systems.

IRIX

PIC is the default, no special compiler options are necessary. The linker option to produce shared libraries is `-shared`.

```
cc -c foo.c
ld -shared -o foo.so foo.o
```

Linux

The compiler flag to create PIC is `-fpic`. On some platforms in some situations `-fPIC` must be used if `-fpic` does not work. Refer to the GCC manual for more information. The compiler flag to create a shared library is `-shared`. A complete example looks like this:

```
cc -fpic -c foo.c
```

```
cc -shared -o foo.so foo.o
```

MacOS X

Here is an example. It assumes the developer tools are installed.

```
cc -c foo.c
cc -bundle -flat_namespace -undefined suppress -o foo.so foo.o
```

NetBSD

The compiler flag to create PIC is `-fpic`. For ELF systems, the compiler with the flag `-shared` is used to link shared libraries. On the older non-ELF systems, `ld -Bshareable` is used.

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

OpenBSD

The compiler flag to create PIC is `-fpic`. `ld -Bshareable` is used to link shared libraries.

```
gcc -fpic -c foo.c
ld -Bshareable -o foo.so foo.o
```

Solaris

The compiler flag to create PIC is `-KPIC` with the Sun compiler and `-fpic` with GCC. To link shared libraries, the compiler option is `-G` with either compiler or alternatively `-shared` with GCC.

```
cc -KPIC -c foo.c
cc -G -o foo.so foo.o
```

or

```
gcc -fpic -c foo.c
gcc -G -o foo.so foo.o
```

Tru64 UNIX

PIC is the default, so the compilation command is the usual one. `ld` with special options is used to do the linking:

```
cc -c foo.c
ld -shared -expect_unresolved '*' -o foo.so foo.o
```

The same procedure is used with GCC instead of the system compiler; no special options are required.

UnixWare

The compiler flag to create PIC is `-K PIC` with the SCO compiler and `-fpic` with GCC. To link shared libraries, the compiler option is `-G` with the SCO compiler and `-shared` with GCC.

```
cc -K PIC -c foo.c
cc -G -o foo.so foo.o
```

or

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

Tip: If this is too complicated for you, you should consider using GNU Libtool[1], which hides the platform differences behind a uniform interface.

1. http://www.gnu.org/software/libtool/

The resulting shared library file can then be loaded into PostgreSQL. When specifying the file name to the `CREATE FUNCTION` command, one must give it the name of the shared library file, not the intermediate object file. Note that the system's standard shared-library extension (usually `.so` or `.sl`) can be omitted from the `CREATE FUNCTION` command, and normally should be omitted for best portability.

Refer back to Section 33.9.1 about where the server expects to find the shared library files.

33.9.7. Extension Building Infrastructure

If you are thinking about distributing your PostgreSQL extension modules, setting up a portable build system for them can be fairly difficult. Therefore the PostgreSQL installation provides a build infrastructure for extensions, called PGXS, so that simple extension modules can be built simply against an already installed server. Note that this infrastructure is not intended to be a universal build system framework that can be used to build all software interfacing to PostgreSQL; it simply automates common build rules for simple server extension modules. For more complicated packages, you need to write your own build system.

To use the infrastructure for your extension, you must write a simple makefile. In that makefile, you need to set some variables and finally include the global PGXS makefile. Here is an example that builds an extension module named `isbn_issn` consisting of a shared library, an SQL script, and a documentation text file:

```
MODULES = isbn_issn
DATA_built = isbn_issn.sql
DOCS = README.isbn_issn

PGXS := $(shell pg_config --pgxs)
include $(PGXS)
```

The last two lines should always be the same. Earlier in the file, you assign variables or add custom make rules.

The following variables can be set:

`MODULES`

list of shared objects to be built from source file with same stem (do not include suffix in this list)

`DATA`

random files to install into *`prefix`*`/share/contrib`

`DATA_built`

random files to install into *`prefix`*`/share/contrib`, which need to be built first

`DOCS`

random files to install under *`prefix`*`/doc/contrib`

`SCRIPTS`

script files (not binaries) to install into *`prefix`*`/bin`

`SCRIPTS_built`

script files (not binaries) to install into *`prefix`*`/bin`, which need to be built first

`REGRESS`

list of regression test cases (without suffix), see below

or at most one of these two:

`PROGRAM`

a binary program to build (list objects files in `OBJS`)

`MODULE_big`

a shared object to build (list object files in `OBJS`)

The following can also be set:

`EXTRA_CLEAN`

extra files to remove in `make clean`

`PG_CPPFLAGS`

will be added to `CPPFLAGS`

`PG_LIBS`

will be added to `PROGRAM` link line

`SHLIB_LINK`

will be added to `MODULE_big` link line

Put this makefile as `Makefile` in the directory which holds your extension. Then you can do `make` to compile, and later `make install` to install your module. The extension is compiled and installed for the PostgreSQL installation that corresponds to the first `pg_config` command found in your path.

The scripts listed in the `REGRESS` variable are used for regression testing of your module, just like `make installcheck` is used for the main PostgreSQL server. For this to work you need to have a subdirectory named `sql/` in your extension's directory, within which you put one file for each group of tests you want to run. The files should have extension `.sql`, which should not be included in the `REGRESS` list in the makefile. For each test there should be a file containing the expected result in a subdirectory named `expected/`, with extension `.out`. The tests are run by executing `make installcheck`, and the resulting output will be compared to the expected files. The differences will be written to the file `regression.diffs` in `diff -c` format. Note that trying to run a test which is missing the expected file will be reported as "trouble", so make sure you have all expected files.

> **Tip:** The easiest way of creating the expected files is creating empty files, then carefully inspecting the result files after a test run (to be found in the `results/` directory), and copying them to `expected/` if they match what you want from the test.

33.9.8. Composite-Type Arguments

Composite types do not have a fixed layout like C structures. Instances of a composite type may contain null fields. In addition, composite types that are part of an inheritance hierarchy may have different fields than other members of the same inheritance hierarchy. Therefore, PostgreSQL provides a function interface for accessing fields of composite types from C.

Suppose we want to write a function to answer the query

```
SELECT name, c_overpaid(emp, 1500) AS overpaid
    FROM emp
    WHERE name = 'Bill' OR name = 'Sam';
```

Using call conventions version 0, we can define `c_overpaid` as:

```
#include "postgres.h"
#include "executor/executor.h"  /* for GetAttributeByName() */

bool
c_overpaid(HeapTupleHeader t, /* the current row of emp */
           int32 limit)
{
    bool isnull;
    int32 salary;

    salary = DatumGetInt32(GetAttributeByName(t, "salary", &isnull));
    if (isnull)
        return false;
    return salary > limit;
}
```

In version-1 coding, the above would look like this:

```
#include "postgres.h"
#include "executor/executor.h"  /* for GetAttributeByName() */

PG_FUNCTION_INFO_V1(c_overpaid);

Datum
c_overpaid(PG_FUNCTION_ARGS)
{
    HeapTupleHeader  t = PG_GETARG_HEAPTUPLEHEADER(0);
    int32            limit = PG_GETARG_INT32(1);
    bool isnull;
    Datum salary;

    salary = GetAttributeByName(t, "salary", &isnull);
    if (isnull)
        PG_RETURN_BOOL(false);
    /* Alternatively, we might prefer to do PG_RETURN_NULL() for null salary. */

    PG_RETURN_BOOL(DatumGetInt32(salary) > limit);
}
```

GetAttributeByName is the PostgreSQL system function that returns attributes out of the specified row. It has three arguments: the argument of type HeapTupleHeader passed into the function, the name of the desired attribute, and a return parameter that tells whether the attribute is null. GetAttributeByName returns a Datum value that you can convert to the proper data type by using the appropriate DatumGet*XXX*() macro. Note that the return value is meaningless if the null flag is set; always check the null flag before trying to do anything with the result.

There is also GetAttributeByNum, which selects the target attribute by column number instead of name.

The following command declares the function c_overpaid in SQL:

```
CREATE FUNCTION c_overpaid(emp, integer) RETURNS boolean
    AS 'DIRECTORY/funcs', 'c_overpaid'
    LANGUAGE C STRICT;
```

Notice we have used STRICT so that we did not have to check whether the input arguments were NULL.

33.9.9. Returning Rows (Composite Types)

To return a row or composite-type value from a C-language function, you can use a special API that provides macros and functions to hide most of the complexity of building composite data types. To use this API, the source file must include:

```
#include "funcapi.h"
```

There are two ways you can build a composite data value (henceforth a "tuple"): you can build it from an array of Datum values, or from an array of C strings that can be passed to the input conversion functions of the tuple's column data types. In either case, you first need to obtain or construct a TupleDesc descriptor for the tuple structure. When working with Datums, you pass the TupleDesc to BlessTupleDesc, and then call heap_form_tuple for each row. When working with C strings, you pass the TupleDesc to TupleDescGetAttInMetadata, and then call BuildTupleFromCStrings for each row. In the case of a function returning a set of tuples, the setup steps can all be done once during the first call of the function.

Several helper functions are available for setting up the needed TupleDesc. The recommended way to do this in most functions returning composite values is to call

```
TypeFuncClass get_call_result_type(FunctionCallInfo fcinfo,
                                   Oid *resultTypeId,
                                   TupleDesc *resultTupleDesc)
```

passing the same fcinfo struct passed to the calling function itself. (This of course requires that you use the version-1 calling conventions.) resultTypeId can be specified as NULL or as the address of a local variable to receive the function's result type OID. resultTupleDesc should be the address of a local TupleDesc variable. Check that the result is TYPEFUNC_COMPOSITE; if so, resultTupleDesc has been filled with the needed TupleDesc. (If it is not, you can report an error along the lines of "function returning record called in context that cannot accept type record".)

Tip: `get_call_result_type` can resolve the actual type of a polymorphic function result; so it is useful in functions that return scalar polymorphic results, not only functions that return composites. The `resultTypeId` output is primarily useful for functions returning polymorphic scalars.

Note: `get_call_result_type` has a sibling `get_expr_result_type`, which can be used to resolve the expected output type for a function call represented by an expression tree. This can be used when trying to determine the result type from outside the function itself. There is also `get_func_result_type`, which can be used when only the function's OID is available. However these functions are not able to deal with functions declared to return `record`, and `get_func_result_type` cannot resolve polymorphic types, so you should preferentially use `get_call_result_type`.

Older, now-deprecated functions for obtaining `TupleDesc`s are

```
TupleDesc RelationNameGetTupleDesc(const char *relname)
```

to get a `TupleDesc` for the row type of a named relation, and

```
TupleDesc TypeGetTupleDesc(Oid typeoid, List *colaliases)
```

to get a `TupleDesc` based on a type OID. This can be used to get a `TupleDesc` for a base or composite type. It will not work for a function that returns `record`, however, and it cannot resolve polymorphic types.

Once you have a `TupleDesc`, call

```
TupleDesc BlessTupleDesc(TupleDesc tupdesc)
```

if you plan to work with Datums, or

```
AttInMetadata *TupleDescGetAttInMetadata(TupleDesc tupdesc)
```

if you plan to work with C strings. If you are writing a function returning set, you can save the results of these functions in the `FuncCallContext` structure — use the `tuple_desc` or `attinmeta` field respectively.

When working with Datums, use

```
HeapTuple heap_form_tuple(TupleDesc tupdesc, Datum *values, bool *isnull)
```

to build a `HeapTuple` given user data in Datum form.

When working with C strings, use

```
HeapTuple BuildTupleFromCStrings(AttInMetadata *attinmeta, char **values)
```

to build a `HeapTuple` given user data in C string form. `values` is an array of C strings, one for each attribute of the return row. Each C string should be in the form expected by the input function of the attribute data type. In order to return a null value for one of the attributes, the corresponding pointer in the `values` array should be set to `NULL`. This function will need to be called again for each row you return.

Once you have built a tuple to return from your function, it must be converted into a `Datum`. Use

```
HeapTupleGetDatum(HeapTuple tuple)
```

to convert a `HeapTuple` into a valid Datum. This `Datum` can be returned directly if you intend to return just a single row, or it can be used as the current return value in a set-returning function.

An example appears in the next section.

33.9.10. Returning Sets

There is also a special API that provides support for returning sets (multiple rows) from a C-language function. A set-returning function must follow the version-1 calling conventions. Also, source files must include `funcapi.h`, as above.

A set-returning function (SRF) is called once for each item it returns. The SRF must therefore save enough state to remember what it was doing and return the next item on each call. The structure `FuncCallContext` is provided to help control this process. Within a function, `fcinfo->flinfo->fn_extra` is used to hold a pointer to `FuncCallContext` across calls.

```
typedef struct
{
    /*
     * Number of times we've been called before
     *
     * call_cntr is initialized to 0 for you by SRF_FIRSTCALL_INIT(), and
     * incremented for you every time SRF_RETURN_NEXT() is called.
     */
    uint32 call_cntr;

    /*
     * OPTIONAL maximum number of calls
     *
     * max_calls is here for convenience only and setting it is optional.
     * If not set, you must provide alternative means to know when the
     * function is done.
     */
    uint32 max_calls;

    /*
     * OPTIONAL pointer to result slot
     *
     * This is obsolete and only present for backwards compatibility, viz,
     * user-defined SRFs that use the deprecated TupleDescGetSlot().
     */
    TupleTableSlot *slot;

    /*
     * OPTIONAL pointer to miscellaneous user-provided context information
     *
     * user_fctx is for use as a pointer to your own data to retain
     * arbitrary context information between calls of your function.
```

```
     */
    void *user_fctx;

    /*
     * OPTIONAL pointer to struct containing attribute type input metadata
     *
     * attinmeta is for use when returning tuples (i.e., composite data types)
     * and is not used when returning base data types. It is only needed
     * if you intend to use BuildTupleFromCStrings() to create the return
     * tuple.
     */
    AttInMetadata *attinmeta;

    /*
     * memory context used for structures that must live for multiple calls
     *
     * multi_call_memory_ctx is set by SRF_FIRSTCALL_INIT() for you, and used
     * by SRF_RETURN_DONE() for cleanup. It is the most appropriate memory
     * context for any memory that is to be reused across multiple calls
     * of the SRF.
     */
    MemoryContext multi_call_memory_ctx;

    /*
     * OPTIONAL pointer to struct containing tuple description
     *
     * tuple_desc is for use when returning tuples (i.e. composite data types)
     * and is only needed if you are going to build the tuples with
     * heap_form_tuple() rather than with BuildTupleFromCStrings().  Note that
     * the TupleDesc pointer stored here should usually have been run through
     * BlessTupleDesc() first.
     */
    TupleDesc tuple_desc;

} FuncCallContext;
```

An SRF uses several functions and macros that automatically manipulate the `FuncCallContext` structure (and expect to find it via `fn_extra`). Use

```
SRF_IS_FIRSTCALL()
```

to determine if your function is being called for the first or a subsequent time. On the first call (only) use

```
SRF_FIRSTCALL_INIT()
```

to initialize the `FuncCallContext`. On every function call, including the first, use

```
SRF_PERCALL_SETUP()
```

to properly set up for using the `FuncCallContext` and clearing any previously returned data left over from the previous pass.

If your function has data to return, use

```
SRF_RETURN_NEXT(funcctx, result)
```

to return it to the caller. (`result` must be of type `Datum`, either a single value or a tuple prepared as described above.) Finally, when your function is finished returning data, use

```
SRF_RETURN_DONE(funcctx)
```

to clean up and end the SRF.

The memory context that is current when the SRF is called is a transient context that will be cleared between calls. This means that you do not need to call `pfree` on everything you allocated using `palloc`; it will go away anyway. However, if you want to allocate any data structures to live across calls, you need to put them somewhere else. The memory context referenced by `multi_call_memory_ctx` is a suitable location for any data that needs to survive until the SRF is finished running. In most cases, this means that you should switch into `multi_call_memory_ctx` while doing the first-call setup.

A complete pseudo-code example looks like the following:

```
Datum
my_set_returning_function(PG_FUNCTION_ARGS)
{
    FuncCallContext  *funcctx;
    Datum             result;
    MemoryContext     oldcontext;
    further declarations as needed

    if (SRF_IS_FIRSTCALL())
    {
        funcctx = SRF_FIRSTCALL_INIT();
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);
        /* One-time setup code appears here: */
        user code
        if returning composite
            build TupleDesc, and perhaps AttInMetadata
        endif returning composite
        user code
        MemoryContextSwitchTo(oldcontext);
    }

    /* Each-time setup code appears here: */
    user code
    funcctx = SRF_PERCALL_SETUP();
    user code

    /* this is just one way we might test whether we are done: */
    if (funcctx->call_cntr < funcctx->max_calls)
    {
        /* Here we want to return another item: */
        user code
        obtain result Datum
        SRF_RETURN_NEXT(funcctx, result);
    }
```

```
    else
    {
        /* Here we are done returning items and just need to clean up: */
        user code
        SRF_RETURN_DONE(funcctx);
    }
}
```

A complete example of a simple SRF returning a composite type looks like:

```
PG_FUNCTION_INFO_V1(retcomposite);

Datum
retcomposite(PG_FUNCTION_ARGS)
{
    FuncCallContext     *funcctx;
    int                  call_cntr;
    int                  max_calls;
    TupleDesc            tupdesc;
    AttInMetadata       *attinmeta;

     /* stuff done only on the first call of the function */
     if (SRF_IS_FIRSTCALL())
     {
        MemoryContext   oldcontext;

        /* create a function context for cross-call persistence */
        funcctx = SRF_FIRSTCALL_INIT();

        /* switch to memory context appropriate for multiple function calls */
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);

        /* total number of tuples to be returned */
        funcctx->max_calls = PG_GETARG_UINT32(0);

        /* Build a tuple descriptor for our result type */
        if (get_call_result_type(fcinfo, NULL, &tupdesc) != TYPEFUNC_COMPOSITE)
            ereport(ERROR,
                    (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
                     errmsg("function returning record called in context "
                            "that cannot accept type record")));

        /*
         * generate attribute metadata needed later to produce tuples from raw
         * C strings
         */
        attinmeta = TupleDescGetAttInMetadata(tupdesc);
        funcctx->attinmeta = attinmeta;

        MemoryContextSwitchTo(oldcontext);
    }
```

```
    /* stuff done on every call of the function */
    funcctx = SRF_PERCALL_SETUP();

    call_cntr = funcctx->call_cntr;
    max_calls = funcctx->max_calls;
    attinmeta = funcctx->attinmeta;

    if (call_cntr < max_calls)    /* do when there is more left to send */
    {
        char       **values;
        HeapTuple    tuple;
        Datum        result;

        /*
         * Prepare a values array for building the returned tuple.
         * This should be an array of C strings which will
         * be processed later by the type input functions.
         */
        values = (char **) palloc(3 * sizeof(char *));
        values[0] = (char *) palloc(16 * sizeof(char));
        values[1] = (char *) palloc(16 * sizeof(char));
        values[2] = (char *) palloc(16 * sizeof(char));

        snprintf(values[0], 16, "%d", 1 * PG_GETARG_INT32(1));
        snprintf(values[1], 16, "%d", 2 * PG_GETARG_INT32(1));
        snprintf(values[2], 16, "%d", 3 * PG_GETARG_INT32(1));

        /* build a tuple */
        tuple = BuildTupleFromCStrings(attinmeta, values);

        /* make the tuple into a datum */
        result = HeapTupleGetDatum(tuple);

        /* clean up (this is not really necessary) */
        pfree(values[0]);
        pfree(values[1]);
        pfree(values[2]);
        pfree(values);

        SRF_RETURN_NEXT(funcctx, result);
    }
    else    /* do when there is no more left */
    {
        SRF_RETURN_DONE(funcctx);
    }
}
```

One way to declare this function in SQL is:

```
CREATE TYPE __retcomposite AS (f1 integer, f2 integer, f3 integer);

CREATE OR REPLACE FUNCTION retcomposite(integer, integer)
    RETURNS SETOF __retcomposite
```

```
    AS 'filename', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;
```

A different way is to use OUT parameters:

```
CREATE OR REPLACE FUNCTION retcomposite(IN integer, IN integer,
    OUT f1 integer, OUT f2 integer, OUT f3 integer)
    RETURNS SETOF record
    AS 'filename', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;
```

Notice that in this method the output type of the function is formally an anonymous `record` type.

The directory `contrib/tablefunc` in the source distribution contains more examples of set-returning functions.

33.9.11. Polymorphic Arguments and Return Types

C-language functions may be declared to accept and return the polymorphic types `anyelement` and `anyarray`. See Section 33.2.5 for a more detailed explanation of polymorphic functions. When function arguments or return types are defined as polymorphic types, the function author cannot know in advance what data type it will be called with, or need to return. There are two routines provided in `fmgr.h` to allow a version-1 C function to discover the actual data types of its arguments and the type it is expected to return. The routines are called `get_fn_expr_rettype(FmgrInfo *flinfo)` and `get_fn_expr_argtype(FmgrInfo *flinfo, int argnum)`. They return the result or argument type OID, or `InvalidOid` if the information is not available. The structure `flinfo` is normally accessed as `fcinfo->flinfo`. The parameter `argnum` is zero based. `get_call_result_type` can also be used as an alternative to `get_fn_expr_rettype`.

For example, suppose we want to write a function to accept a single element of any type, and return a one-dimensional array of that type:

```
PG_FUNCTION_INFO_V1(make_array);
Datum
make_array(PG_FUNCTION_ARGS)
{
    ArrayType  *result;
    Oid         element_type = get_fn_expr_argtype(fcinfo->flinfo, 0);
    Datum       element;
    bool        isnull;
    int16       typlen;
    bool        typbyval;
    char        typalign;
    int         ndims;
    int         dims[MAXDIM];
    int         lbs[MAXDIM];

    if (!OidIsValid(element_type))
        elog(ERROR, "could not determine data type of input");

    /* get the provided element, being careful in case it's NULL */
    isnull = PG_ARGISNULL(0);
```

```
    if (isnull)
        element = (Datum) 0;
    else
        element = PG_GETARG_DATUM(0);

    /* we have one dimension */
    ndims = 1;
    /* and one element */
    dims[0] = 1;
    /* and lower bound is 1 */
    lbs[0] = 1;

    /* get required info about the element type */
    get_typlenbyvalalign(element_type, &typlen, &typbyval, &typalign);

    /* now build the array */
    result = construct_md_array(&element, &isnull, ndims, dims, lbs,
                                element_type, typlen, typbyval, typalign);

    PG_RETURN_ARRAYTYPE_P(result);
}
```

The following command declares the function `make_array` in SQL:

```
CREATE FUNCTION make_array(anyelement) RETURNS anyarray
    AS 'DIRECTORY/funcs', 'make_array'
    LANGUAGE C IMMUTABLE;
```

33.9.12. Shared Memory and LWLocks

Add-ins may reserve LWLocks and an allocation of shared memory on server startup. The add-in's shared library must be preloaded by specifying it in shared_preload_libraries. Shared memory is reserved by calling:

```
void RequestAddinShmemSpace(int size)
```

from your `_PG_init` function.

LWLocks are reserved by calling:

```
void RequestAddinLWLocks(int n)
```

from `_PG_init`.

To avoid possible race-conditions, each backend should use the LWLock `AddinShmemInitLock` when connecting to and initializing its allocation of shared memory, as shown here:

```
        static mystruct *ptr = NULL;
```

```
if (!ptr)
{
        bool    found;

        LWLockAcquire(AddinShmemInitLock, LW_EXCLUSIVE);
        ptr = ShmemInitStruct("my struct name", size, &found);
        if (!ptr)
                elog(ERROR, "out of shared memory");
        if (!found)
        {
                initialize contents of shmem area;
                acquire any requested LWLocks using:
                ptr->mylockid = LWLockAssign();
        }
        LWLockRelease(AddinShmemInitLock);
}
```

33.10. User-Defined Aggregates

Aggregate functions in PostgreSQL are expressed in terms of *state values* and *state transition functions*. That is, an aggregate operates using a state value that is updated as each successive input row is processed. To define a new aggregate function, one selects a data type for the state value, an initial value for the state, and a state transition function. The state transition function is just an ordinary function that could also be used outside the context of the aggregate. A *final function* can also be specified, in case the desired result of the aggregate is different from the data that needs to be kept in the running state value.

Thus, in addition to the argument and result data types seen by a user of the aggregate, there is an internal state-value data type that may be different from both the argument and result types.

If we define an aggregate that does not use a final function, we have an aggregate that computes a running function of the column values from each row. `sum` is an example of this kind of aggregate. `sum` starts at zero and always adds the current row's value to its running total. For example, if we want to make a `sum` aggregate to work on a data type for complex numbers, we only need the addition function for that data type. The aggregate definition would be:

```
CREATE AGGREGATE sum (complex)
(
    sfunc = complex_add,
    stype = complex,
    initcond = '(0,0)'
);

SELECT sum(a) FROM test_complex;

   sum
-----------
 (34,53.9)
```

(Notice that we are relying on function overloading: there is more than one aggregate named `sum`, but PostgreSQL can figure out which kind of sum applies to a column of type `complex`.)

The above definition of `sum` will return zero (the initial state condition) if there are no nonnull input values. Perhaps we want to return null in that case instead — the SQL standard expects `sum` to behave that way. We can do this simply by omitting the `initcond` phrase, so that the initial state condition is null. Ordinarily this would mean that the `sfunc` would need to check for a null state-condition input, but for `sum` and some other simple aggregates like `max` and `min`, it is sufficient to insert the first nonnull input value into the state variable and then start applying the transition function at the second nonnull input value. PostgreSQL will do that automatically if the initial condition is null and the transition function is marked "strict" (i.e., not to be called for null inputs).

Another bit of default behavior for a "strict" transition function is that the previous state value is retained unchanged whenever a null input value is encountered. Thus, null values are ignored. If you need some other behavior for null inputs, do not declare your transition function as strict; instead code it to test for null inputs and do whatever is needed.

`avg` (average) is a more complex example of an aggregate. It requires two pieces of running state: the sum of the inputs and the count of the number of inputs. The final result is obtained by dividing these quantities. Average is typically implemented by using a two-element array as the state value. For example, the built-in implementation of `avg(float8)` looks like:

```
CREATE AGGREGATE avg (float8)
(
    sfunc = float8_accum,
    stype = float8[],
    finalfunc = float8_avg,
    initcond = '{0,0}'
);
```

Aggregate functions may use polymorphic state transition functions or final functions, so that the same functions can be used to implement multiple aggregates. See Section 33.2.5 for an explanation of polymorphic functions. Going a step further, the aggregate function itself may be specified with polymorphic input type(s) and state type, allowing a single aggregate definition to serve for multiple input data types. Here is an example of a polymorphic aggregate:

```
CREATE AGGREGATE array_accum (anyelement)
(
    sfunc = array_append,
    stype = anyarray,
    initcond = '{}'
);
```

Here, the actual state type for any aggregate call is the array type having the actual input type as elements.

Here's the output using two different actual data types as arguments:

```
SELECT attrelid::regclass, array_accum(attname)
    FROM pg_attribute
    WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
    GROUP BY attrelid;
```

```
   attrelid    |              array_accum
---------------+---------------------------------------
 pg_tablespace | {spcname,spcowner,spclocation,spcacl}
(1 row)

SELECT attrelid::regclass, array_accum(atttypid)
    FROM pg_attribute
    WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
    GROUP BY attrelid;

   attrelid    |   array_accum
---------------+-----------------
 pg_tablespace | {19,26,25,1034}
(1 row)
```

A function written in C can detect that it is being called as an aggregate transition or final function by seeing if it was passed an `AggState` node as the function call "context", for example by

```
        if (fcinfo->context && IsA(fcinfo->context, AggState))
```

One reason for checking this is that when it is true, the first input must be a temporary transition value and can therefore safely be modified in-place rather than allocating a new copy. (This is the *only* case where it is safe for a function to modify a pass-by-reference input.) See `int8inc()` for an example.

For further details see the *CREATE AGGREGATE* command.

33.11. User-Defined Types

As described in Section 33.2, PostgreSQL can be extended to support new data types. This section describes how to define new base types, which are data types defined below the level of the SQL language. Creating a new base type requires implementing functions to operate on the type in a low-level language, usually C.

The examples in this section can be found in `complex.sql` and `complex.c` in the `src/tutorial` directory of the source distribution. See the `README` file in that directory for instructions about running the examples.

A user-defined type must always have input and output functions. These functions determine how the type appears in strings (for input by the user and output to the user) and how the type is organized in memory. The input function takes a null-terminated character string as its argument and returns the internal (in memory) representation of the type. The output function takes the internal representation of the type as argument and returns a null-terminated character string. If we want to do anything more with the type than merely store it, we must provide additional functions to implement whatever operations we'd like to have for the type.

Suppose we want to define a type `complex` that represents complex numbers. A natural way to represent a complex number in memory would be the following C structure:

```
typedef struct Complex {
    double      x;
```

```
    double      y;
} Complex;
```

We will need to make this a pass-by-reference type, since it's too large to fit into a single `Datum` value.

As the external string representation of the type, we choose a string of the form `(x,y)`.

The input and output functions are usually not hard to write, especially the output function. But when defining the external string representation of the type, remember that you must eventually write a complete and robust parser for that representation as your input function. For instance:

```
PG_FUNCTION_INFO_V1(complex_in);

Datum
complex_in(PG_FUNCTION_ARGS)
{
    char       *str = PG_GETARG_CSTRING(0);
    double      x,
                y;
    Complex    *result;

    if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                 errmsg("invalid input syntax for complex: \"%s\"",
                        str)));

    result = (Complex *) palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    PG_RETURN_POINTER(result);
}
```

The output function can simply be:

```
PG_FUNCTION_INFO_V1(complex_out);

Datum
complex_out(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    char       *result;

    result = (char *) palloc(100);
    snprintf(result, 100, "(%g,%g)", complex->x, complex->y);
    PG_RETURN_CSTRING(result);
}
```

You should be careful to make the input and output functions inverses of each other. If you do not, you will have severe problems when you need to dump your data into a file and then read it back in. This is a particularly common problem when floating-point numbers are involved.

Optionally, a user-defined type can provide binary input and output routines. Binary I/O is normally faster but less portable than textual I/O. As with textual I/O, it is up to you to define exactly what the external binary representation is. Most of the built-in data types try to provide a machine-independent binary representation. For `complex`, we will piggy-back on the binary I/O converters for type `float8`:

```
PG_FUNCTION_INFO_V1(complex_recv);

Datum
complex_recv(PG_FUNCTION_ARGS)
{
    StringInfo  buf = (StringInfo) PG_GETARG_POINTER(0);
    Complex    *result;

    result = (Complex *) palloc(sizeof(Complex));
    result->x = pq_getmsgfloat8(buf);
    result->y = pq_getmsgfloat8(buf);
    PG_RETURN_POINTER(result);
}

PG_FUNCTION_INFO_V1(complex_send);

Datum
complex_send(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    StringInfoData buf;

    pq_begintypsend(&buf);
    pq_sendfloat8(&buf, complex->x);
    pq_sendfloat8(&buf, complex->y);
    PG_RETURN_BYTEA_P(pq_endtypsend(&buf));
}
```

Once we have written the I/O functions and compiled them into a shared library, we can define the `complex` type in SQL. First we declare it as a shell type:

```
CREATE TYPE complex;
```

This serves as a placeholder that allows us to reference the type while defining its I/O functions. Now we can define the I/O functions:

```
CREATE FUNCTION complex_in(cstring)
    RETURNS complex
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_out(complex)
    RETURNS cstring
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;
```

```
CREATE FUNCTION complex_recv(internal)
   RETURNS complex
   AS 'filename'
   LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_send(complex)
   RETURNS bytea
   AS 'filename'
   LANGUAGE C IMMUTABLE STRICT;
```

Finally, we can provide the full definition of the data type:

```
CREATE TYPE complex (
   internallength = 16,
   input = complex_in,
   output = complex_out,
   receive = complex_recv,
   send = complex_send,
   alignment = double
);
```

When you define a new base type, PostgreSQL automatically provides support for arrays of that type. For historical reasons, the array type has the same name as the base type with the underscore character (_) prepended.

Once the data type exists, we can declare additional functions to provide useful operations on the data type. Operators can then be defined atop the functions, and if needed, operator classes can be created to support indexing of the data type. These additional layers are discussed in following sections.

If the values of your data type might exceed a few hundred bytes in size (in internal form), you should make the data type TOAST-able (see Section 52.2). To do this, the internal representation must follow the standard layout for variable-length data: the first four bytes must be an `int32` containing the total length in bytes of the datum (including itself). The C functions operating on the data type must be careful to unpack any toasted values they are handed, by using `PG_DETOAST_DATUM`. (This detail is customarily hidden by defining type-specific `GETARG` macros.) Then, when running the `CREATE TYPE` command, specify the internal length as `variable` and select the appropriate storage option.

For further details see the description of the *CREATE TYPE* command.

33.12. User-Defined Operators

Every operator is "syntactic sugar" for a call to an underlying function that does the real work; so you must first create the underlying function before you can create the operator. However, an operator is *not merely* syntactic sugar, because it carries additional information that helps the query planner optimize queries that use the operator. The next section will be devoted to explaining that additional information.

PostgreSQL supports left unary, right unary, and binary operators. Operators can be overloaded; that is, the same operator name can be used for different operators that have different numbers and types of operands.

When a query is executed, the system determines the operator to call from the number and types of the provided operands.

Here is an example of creating an operator for adding two complex numbers. We assume we've already created the definition of type `complex` (see Section 33.11). First we need a function that does the work, then we can define the operator:

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS 'filename', 'complex_add'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    procedure = complex_add,
    commutator = +
);
```

Now we could execute a query like this:

```
SELECT (a + b) AS c FROM test_complex;

        c
-----------------
 (5.2,6.05)
 (133.42,144.95)
```

We've shown how to create a binary operator here. To create unary operators, just omit one of `leftarg` (for left unary) or `rightarg` (for right unary). The `procedure` clause and the argument clauses are the only required items in `CREATE OPERATOR`. The `commutator` clause shown in the example is an optional hint to the query optimizer. Further details about `commutator` and other optimizer hints appear in the next section.

33.13. Operator Optimization Information

A PostgreSQL operator definition can include several optional clauses that tell the system useful things about how the operator behaves. These clauses should be provided whenever appropriate, because they can make for considerable speedups in execution of queries that use the operator. But if you provide them, you must be sure that they are right! Incorrect use of an optimization clause can result in server process crashes, subtly wrong output, or other Bad Things. You can always leave out an optimization clause if you are not sure about it; the only consequence is that queries might run slower than they need to.

Additional optimization clauses might be added in future versions of PostgreSQL. The ones described here are all the ones that release 8.2.6 understands.

33.13.1. COMMUTATOR

The COMMUTATOR clause, if provided, names an operator that is the commutator of the operator being defined. We say that operator A is the commutator of operator B if (x A y) equals (y B x) for all possible input values x, y. Notice that B is also the commutator of A. For example, operators < and > for a particular data type are usually each others' commutators, and operator + is usually commutative with itself. But operator – is usually not commutative with anything.

The left operand type of a commutable operator is the same as the right operand type of its commutator, and vice versa. So the name of the commutator operator is all that PostgreSQL needs to be given to look up the commutator, and that's all that needs to be provided in the COMMUTATOR clause.

It's critical to provide commutator information for operators that will be used in indexes and join clauses, because this allows the query optimizer to "flip around" such a clause to the forms needed for different plan types. For example, consider a query with a WHERE clause like `tab1.x = tab2.y`, where `tab1.x` and `tab2.y` are of a user-defined type, and suppose that `tab2.y` is indexed. The optimizer cannot generate an index scan unless it can determine how to flip the clause around to `tab2.y = tab1.x`, because the index-scan machinery expects to see the indexed column on the left of the operator it is given. PostgreSQL will *not* simply assume that this is a valid transformation — the creator of the = operator must specify that it is valid, by marking the operator with commutator information.

When you are defining a self-commutative operator, you just do it. When you are defining a pair of commutative operators, things are a little trickier: how can the first one to be defined refer to the other one, which you haven't defined yet? There are two solutions to this problem:

- One way is to omit the COMMUTATOR clause in the first operator that you define, and then provide one in the second operator's definition. Since PostgreSQL knows that commutative operators come in pairs, when it sees the second definition it will automatically go back and fill in the missing COMMUTATOR clause in the first definition.
- The other, more straightforward way is just to include COMMUTATOR clauses in both definitions. When PostgreSQL processes the first definition and realizes that COMMUTATOR refers to a nonexistent operator, the system will make a dummy entry for that operator in the system catalog. This dummy entry will have valid data only for the operator name, left and right operand types, and result type, since that's all that PostgreSQL can deduce at this point. The first operator's catalog entry will link to this dummy entry. Later, when you define the second operator, the system updates the dummy entry with the additional information from the second definition. If you try to use the dummy operator before it's been filled in, you'll just get an error message.

33.13.2. NEGATOR

The NEGATOR clause, if provided, names an operator that is the negator of the operator being defined. We say that operator A is the negator of operator B if both return Boolean results and (x A y) equals NOT (x B y) for all possible inputs x, y. Notice that B is also the negator of A. For example, < and >= are a negator pair for most data types. An operator can never validly be its own negator.

Unlike commutators, a pair of unary operators could validly be marked as each others' negators; that would mean (A x) equals NOT (B x) for all x, or the equivalent for right unary operators.

An operator's negator must have the same left and/or right operand types as the operator to be defined, so just as with COMMUTATOR, only the operator name need be given in the NEGATOR clause.

Providing a negator is very helpful to the query optimizer since it allows expressions like NOT (x = y) to be simplified into x <> y. This comes up more often than you might think, because NOT operations can be inserted as a consequence of other rearrangements.

Pairs of negator operators can be defined using the same methods explained above for commutator pairs.

33.13.3. RESTRICT

The RESTRICT clause, if provided, names a restriction selectivity estimation function for the operator. (Note that this is a function name, not an operator name.) RESTRICT clauses only make sense for binary operators that return boolean. The idea behind a restriction selectivity estimator is to guess what fraction of the rows in a table will satisfy a WHERE-clause condition of the form

```
column OP constant
```

for the current operator and a particular constant value. This assists the optimizer by giving it some idea of how many rows will be eliminated by WHERE clauses that have this form. (What happens if the constant is on the left, you may be wondering? Well, that's one of the things that COMMUTATOR is for...)

Writing new restriction selectivity estimation functions is far beyond the scope of this chapter, but fortunately you can usually just use one of the system's standard estimators for many of your own operators. These are the standard restriction estimators:

eqsel for =
neqsel for <>
scalarltsel for < or <=
scalargtsel for > or >=

It might seem a little odd that these are the categories, but they make sense if you think about it. = will typically accept only a small fraction of the rows in a table; <> will typically reject only a small fraction. < will accept a fraction that depends on where the given constant falls in the range of values for that table column (which, it just so happens, is information collected by ANALYZE and made available to the selectivity estimator). <= will accept a slightly larger fraction than < for the same comparison constant, but they're close enough to not be worth distinguishing, especially since we're not likely to do better than a rough guess anyhow. Similar remarks apply to > and >=.

You can frequently get away with using either eqsel or neqsel for operators that have very high or very low selectivity, even if they aren't really equality or inequality. For example, the approximate-equality geometric operators use eqsel on the assumption that they'll usually only match a small fraction of the entries in a table.

You can use scalarltsel and scalargtsel for comparisons on data types that have some sensible means of being converted into numeric scalars for range comparisons. If possible, add the data type to those understood by the function convert_to_scalar() in src/backend/utils/adt/selfuncs.c. (Eventually, this function should be replaced by per-data-type functions identified through a column of the pg_type system catalog; but that hasn't happened yet.) If you do not do this, things will still work, but the optimizer's estimates won't be as good as they could be.

There are additional selectivity estimation functions designed for geometric operators in src/backend/utils/adt/geo_selfuncs.c: areasel, positionsel, and contsel. At this writing these are just stubs, but you may want to use them (or even better, improve them) anyway.

33.13.4. JOIN

The JOIN clause, if provided, names a join selectivity estimation function for the operator. (Note that this is a function name, not an operator name.) JOIN clauses only make sense for binary operators that return boolean. The idea behind a join selectivity estimator is to guess what fraction of the rows in a pair of tables will satisfy a WHERE-clause condition of the form

```
table1.column1 OP table2.column2
```

for the current operator. As with the RESTRICT clause, this helps the optimizer very substantially by letting it figure out which of several possible join sequences is likely to take the least work.

As before, this chapter will make no attempt to explain how to write a join selectivity estimator function, but will just suggest that you use one of the standard estimators if one is applicable:

eqjoinsel for =
neqjoinsel for <>
scalarltjoinsel for < or <=
scalargtjoinsel for > or >=
areajoinsel for 2D area-based comparisons
positionjoinsel for 2D position-based comparisons
contjoinsel for 2D containment-based comparisons

33.13.5. HASHES

The HASHES clause, if present, tells the system that it is permissible to use the hash join method for a join based on this operator. HASHES only makes sense for a binary operator that returns boolean, and in practice the operator had better be equality for some data type.

The assumption underlying hash join is that the join operator can only return true for pairs of left and right values that hash to the same hash code. If two values get put in different hash buckets, the join will never compare them at all, implicitly assuming that the result of the join operator must be false. So it never makes sense to specify HASHES for operators that do not represent equality.

To be marked HASHES, the join operator must appear in a hash index operator class. This is not enforced when you create the operator, since of course the referencing operator class couldn't exist yet. But attempts to use the operator in hash joins will fail at run time if no such operator class exists. The system needs the operator class to find the data-type-specific hash function for the operator's input data type. Of course, you must also supply a suitable hash function before you can create the operator class.

Care should be exercised when preparing a hash function, because there are machine-dependent ways in which it might fail to do the right thing. For example, if your data type is a structure in which there may be uninteresting pad bits, you can't simply pass the whole structure to hash_any. (Unless you write your other operators and functions to ensure that the unused bits are always zero, which is the recommended

strategy.) Another example is that on machines that meet the IEEE floating-point standard, negative zero and positive zero are different values (different bit patterns) but they are defined to compare equal. If a float value might contain negative zero then extra steps are needed to ensure it generates the same hash value as positive zero.

Note: The function underlying a hash-joinable operator must be marked immutable or stable. If it is volatile, the system will never attempt to use the operator for a hash join.

Note: If a hash-joinable operator has an underlying function that is marked strict, the function must also be complete: that is, it should return true or false, never null, for any two nonnull inputs. If this rule is not followed, hash-optimization of `IN` operations may generate wrong results. (Specifically, `IN` might return false where the correct answer according to the standard would be null; or it might yield an error complaining that it wasn't prepared for a null result.)

33.13.6. `MERGES` (`SORT1`, `SORT2`, `LTCMP`, `GTCMP`)

The `MERGES` clause, if present, tells the system that it is permissible to use the merge-join method for a join based on this operator. `MERGES` only makes sense for a binary operator that returns `boolean`, and in practice the operator must represent equality for some data type or pair of data types.

Merge join is based on the idea of sorting the left- and right-hand tables into order and then scanning them in parallel. So, both data types must be capable of being fully ordered, and the join operator must be one that can only succeed for pairs of values that fall at the "same place" in the sort order. In practice this means that the join operator must behave like equality. But unlike hash join, where the left and right data types had better be the same (or at least bitwise equivalent), it is possible to merge-join two distinct data types so long as they are logically compatible. For example, the `smallint`-versus-`integer` equality operator is merge-joinable. We only need sorting operators that will bring both data types into a logically compatible sequence.

Execution of a merge join requires that the system be able to identify four operators related to the merge-join equality operator: less-than comparison for the left operand data type, less-than comparison for the right operand data type, less-than comparison between the two data types, and greater-than comparison between the two data types. (These are actually four distinct operators if the merge-joinable operator has two different operand data types; but when the operand types are the same the three less-than operators are all the same operator.) It is possible to specify these operators individually by name, as the `SORT1`, `SORT2`, `LTCMP`, and `GTCMP` options respectively. The system will fill in the default names $<$, $<$, $<$, $>$ respectively if any of these are omitted when `MERGES` is specified. Also, `MERGES` will be assumed to be implied if any of these four operator options appear, so it is possible to specify just some of them and let the system fill in the rest.

The operand data types of the four comparison operators can be deduced from the operand types of the merge-joinable operator, so just as with `COMMUTATOR`, only the operator names need be given in these clauses. Unless you are using peculiar choices of operator names, it's sufficient to write `MERGES` and let the system fill in the details. (As with `COMMUTATOR` and `NEGATOR`, the system is able to make dummy operator entries if you happen to define the equality operator before the other ones.)

There are additional restrictions on operators that you mark merge-joinable. These restrictions are not currently checked by CREATE OPERATOR, but errors may occur when the operator is used if any are not true:

- A merge-joinable equality operator must have a merge-joinable commutator (itself if the two operand data types are the same, or a related equality operator if they are different).
- If there is a merge-joinable operator relating any two data types A and B, and another merge-joinable operator relating B to any third data type C, then A and C must also have a merge-joinable operator; in other words, having a merge-joinable operator must be transitive.
- Bizarre results will ensue at run time if the four comparison operators you name do not sort the data values compatibly.

Note: The function underlying a merge-joinable operator must be marked immutable or stable. If it is volatile, the system will never attempt to use the operator for a merge join.

Note: In PostgreSQL versions before 7.3, the MERGES shorthand was not available: to make a merge-joinable operator one had to write both SORT1 and SORT2 explicitly. Also, the LTCMP and GTCMP options did not exist; the names of those operators were hardwired as < and > respectively.

33.14. Interfacing Extensions To Indexes

The procedures described thus far let you define new types, new functions, and new operators. However, we cannot yet define an index on a column of a new data type. To do this, we must define an *operator class* for the new data type. Later in this section, we will illustrate this concept in an example: a new operator class for the B-tree index method that stores and sorts complex numbers in ascending absolute value order.

Note: Prior to PostgreSQL release 7.3, it was necessary to make manual additions to the system catalogs pg_amop, pg_amproc, and pg_opclass in order to create a user-defined operator class. That approach is now deprecated in favor of using *CREATE OPERATOR CLASS*, which is a much simpler and less error-prone way of creating the necessary catalog entries.

33.14.1. Index Methods and Operator Classes

The pg_am table contains one row for every index method (internally known as access method). Support for regular access to tables is built into PostgreSQL, but all index methods are described in pg_am. It is possible to add a new index method by defining the required interface routines and then creating a row in pg_am — but that is beyond the scope of this chapter (see Chapter 49).

The routines for an index method do not directly know anything about the data types that the index method will operate on. Instead, an *operator class* identifies the set of operations that the index method needs to use to work with a particular data type. Operator classes are so called because one thing they specify is the set of `WHERE`-clause operators that can be used with an index (i.e., can be converted into an index-scan qualification). An operator class may also specify some *support procedures* that are needed by the internal operations of the index method, but do not directly correspond to any `WHERE`-clause operator that can be used with the index.

It is possible to define multiple operator classes for the same data type and index method. By doing this, multiple sets of indexing semantics can be defined for a single data type. For example, a B-tree index requires a sort ordering to be defined for each data type it works on. It might be useful for a complex-number data type to have one B-tree operator class that sorts the data by complex absolute value, another that sorts by real part, and so on. Typically, one of the operator classes will be deemed most commonly useful and will be marked as the default operator class for that data type and index method.

The same operator class name can be used for several different index methods (for example, both B-tree and hash index methods have operator classes named `int4_ops`), but each such class is an independent entity and must be defined separately.

33.14.2. Index Method Strategies

The operators associated with an operator class are identified by "strategy numbers", which serve to identify the semantics of each operator within the context of its operator class. For example, B-trees impose a strict ordering on keys, lesser to greater, and so operators like "less than" and "greater than or equal to" are interesting with respect to a B-tree. Because PostgreSQL allows the user to define operators, PostgreSQL cannot look at the name of an operator (e.g., < or >=) and tell what kind of comparison it is. Instead, the index method defines a set of "strategies", which can be thought of as generalized operators. Each operator class specifies which actual operator corresponds to each strategy for a particular data type and interpretation of the index semantics.

The B-tree index method defines five strategies, shown in Table 33-2.

Table 33-2. B-tree Strategies

Operation	Strategy Number
less than	1
less than or equal	2
equal	3
greater than or equal	4
greater than	5

Hash indexes express only bitwise equality, and so they use only one strategy, shown in Table 33-3.

Table 33-3. Hash Strategies

Operation	Strategy Number
equal	1

GiST indexes are even more flexible: they do not have a fixed set of strategies at all. Instead, the "consistency" support routine of each particular GiST operator class interprets the strategy numbers however it likes. As an example, several of the built-in GiST index operator classes index two-dimensional geometric objects, providing the "R-tree" strategies shown in Table 33-4. Four of these are true two-dimensional tests (overlaps, same, contains, contained by); four of them consider only the X direction; and the other four provide the same tests in the Y direction.

Table 33-4. GiST Two-Dimensional "R-tree" Strategies

Operation	Strategy Number
strictly left of	1
does not extend to right of	2
overlaps	3
does not extend to left of	4
strictly right of	5
same	6
contains	7
contained by	8
does not extend above	9
strictly below	10
strictly above	11
does not extend below	12

GIN indexes are similar to GiST indexes in flexibility: they don't have a fixed set of strategies. Instead the support routines of each operator class interpret the strategy numbers according to the operator class's definition. As an example, the strategy numbers used by the built-in operator classes for arrays are shown in Table 33-5.

Table 33-5. GIN Array Strategies

Operation	Strategy Number
overlap	1
contains	2
is contained by	3
equal	4

Note that all strategy operators return Boolean values. In practice, all operators defined as index method strategies must return type `boolean`, since they must appear at the top level of a `WHERE` clause to be used with an index.

By the way, the `amorderstrategy` column in `pg_am` tells whether the index method supports ordered scans. Zero means it doesn't; if it does, `amorderstrategy` is the strategy number that corresponds to the ordering operator. For example, B-tree has `amorderstrategy` = 1, which is its "less than" strategy number.

33.14.3. Index Method Support Routines

Strategies aren't usually enough information for the system to figure out how to use an index. In practice, the index methods require additional support routines in order to work. For example, the B-tree index method must be able to compare two keys and determine whether one is greater than, equal to, or less than the other. Similarly, the hash index method must be able to compute hash codes for key values. These operations do not correspond to operators used in qualifications in SQL commands; they are administrative routines used by the index methods, internally.

Just as with strategies, the operator class identifies which specific functions should play each of these roles for a given data type and semantic interpretation. The index method defines the set of functions it needs, and the operator class identifies the correct functions to use by assigning them to the "support function numbers".

B-trees require a single support function, shown in Table 33-6.

Table 33-6. B-tree Support Functions

Function	Support Number
Compare two keys and return an integer less than zero, zero, or greater than zero, indicating whether the first key is less than, equal to, or greater than the second.	1

Hash indexes likewise require one support function, shown in Table 33-7.

Table 33-7. Hash Support Functions

Function	Support Number
Compute the hash value for a key	1

GiST indexes require seven support functions, shown in Table 33-8.

Table 33-8. GiST Support Functions

Function	Support Number
consistent - determine whether key satisfies the query qualifier	1
union - compute union of a set of keys	2
compress - compute a compressed representation of a key or value to be indexed	3
decompress - compute a decompressed representation of a compressed key	4
penalty - compute penalty for inserting new key into subtree with given subtree's key	5
picksplit - determine which entries of a page are to be moved to the new page and compute the union keys for resulting pages	6

Function	Support Number
equal - compare two keys and return true if they are equal	7

GIN indexes require four support functions, shown in Table 33-9.

Table 33-9. GIN Support Functions

Function	Support Number
compare - compare two keys and return an integer less than zero, zero, or greater than zero, indicating whether the first key is less than, equal to, or greater than the second	1
extractValue - extract keys from a value to be indexed	2
extractQuery - extract keys from a query condition	3
consistent - determine whether value matches query condition	4

Unlike strategy operators, support functions return whichever data type the particular index method expects; for example in the case of the comparison function for B-trees, a signed integer.

33.14.4. An Example

Now that we have seen the ideas, here is the promised example of creating a new operator class. (You can find a working copy of this example in `src/tutorial/complex.c` and `src/tutorial/complex.sql` in the source distribution.) The operator class encapsulates operators that sort complex numbers in absolute value order, so we choose the name `complex_abs_ops`. First, we need a set of operators. The procedure for defining operators was discussed in Section 33.12. For an operator class on B-trees, the operators we require are:

- absolute-value less-than (strategy 1)
- absolute-value less-than-or-equal (strategy 2)
- absolute-value equal (strategy 3)
- absolute-value greater-than-or-equal (strategy 4)
- absolute-value greater-than (strategy 5)

The least error-prone way to define a related set of comparison operators is to write the B-tree comparison support function first, and then write the other functions as one-line wrappers around the support function. This reduces the odds of getting inconsistent results for corner cases. Following this approach, we first write

```
#define Mag(c)  ((c)->x*(c)->x + (c)->y*(c)->y)

static int
```

```
complex_abs_cmp_internal(Complex *a, Complex *b)
{
    double      amag = Mag(a),
                bmag = Mag(b);

    if (amag < bmag)
        return -1;
    if (amag > bmag)
        return 1;
    return 0;
}
```

Now the less-than function looks like

```
PG_FUNCTION_INFO_V1(complex_abs_lt);

Datum
complex_abs_lt(PG_FUNCTION_ARGS)
{
    Complex    *a = (Complex *) PG_GETARG_POINTER(0);
    Complex    *b = (Complex *) PG_GETARG_POINTER(1);

    PG_RETURN_BOOL(complex_abs_cmp_internal(a, b) < 0);
}
```

The other four functions differ only in how they compare the internal function's result to zero.

Next we declare the functions and the operators based on the functions to SQL:

```
CREATE FUNCTION complex_abs_lt(complex, complex) RETURNS bool
    AS 'filename', 'complex_abs_lt'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR < (
   leftarg = complex, rightarg = complex, procedure = complex_abs_lt,
   commutator = > , negator = >= ,
   restrict = scalarltsel, join = scalarltjoinsel
);
```

It is important to specify the correct commutator and negator operators, as well as suitable restriction and join selectivity functions, otherwise the optimizer will be unable to make effective use of the index. Note that the less-than, equal, and greater-than cases should use different selectivity functions.

Other things worth noting are happening here:

- There can only be one operator named, say, = and taking type `complex` for both operands. In this case we don't have any other operator = for `complex`, but if we were building a practical data type we'd probably want = to be the ordinary equality operation for complex numbers (and not the equality of the absolute values). In that case, we'd need to use some other operator name for `complex_abs_eq`.
- Although PostgreSQL can cope with functions having the same SQL name as long as they have different argument data types, C can only cope with one global function having a given name. So we shouldn't

name the C function something simple like `abs_eq`. Usually it's a good practice to include the data type name in the C function name, so as not to conflict with functions for other data types.

- We could have made the SQL name of the function `abs_eq`, relying on PostgreSQL to distinguish it by argument data types from any other SQL function of the same name. To keep the example simple, we make the function have the same names at the C level and SQL level.

The next step is the registration of the support routine required by B-trees. The example C code that implements this is in the same file that contains the operator functions. This is how we declare the function:

```
CREATE FUNCTION complex_abs_cmp(complex, complex)
    RETURNS integer
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;
```

Now that we have the required operators and support routine, we can finally create the operator class:

```
CREATE OPERATOR CLASS complex_abs_ops
    DEFAULT FOR TYPE complex USING btree AS
        OPERATOR        1       < ,
        OPERATOR        2       <= ,
        OPERATOR        3       = ,
        OPERATOR        4       >= ,
        OPERATOR        5       > ,
        FUNCTION        1       complex_abs_cmp(complex, complex);
```

And we're done! It should now be possible to create and use B-tree indexes on `complex` columns.

We could have written the operator entries more verbosely, as in

```
        OPERATOR        1       < (complex, complex) ,
```

but there is no need to do so when the operators take the same data type we are defining the operator class for.

The above example assumes that you want to make this new operator class the default B-tree operator class for the `complex` data type. If you don't, just leave out the word `DEFAULT`.

33.14.5. Cross-Data-Type Operator Classes

So far we have implicitly assumed that an operator class deals with only one data type. While there certainly can be only one data type in a particular index column, it is often useful to index operations that compare an indexed column to a value of a different data type. This is presently supported by the B-tree and GiST index methods.

B-trees require the left-hand operand of each operator to be the indexed data type, but the right-hand operand can be of a different type. There must be a support function having a matching signature. For

example, the built-in operator class for type `bigint` (`int8`) allows cross-type comparisons to `int4` and `int2`. It could be duplicated by this definition:

```
CREATE OPERATOR CLASS int8_ops
DEFAULT FOR TYPE int8 USING btree AS
  -- standard int8 comparisons
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint8cmp(int8, int8) ,

  -- cross-type comparisons to int2 (smallint)
  OPERATOR 1 < (int8, int2) ,
  OPERATOR 2 <= (int8, int2) ,
  OPERATOR 3 = (int8, int2) ,
  OPERATOR 4 >= (int8, int2) ,
  OPERATOR 5 > (int8, int2) ,
  FUNCTION 1 btint82cmp(int8, int2) ,

  -- cross-type comparisons to int4 (integer)
  OPERATOR 1 < (int8, int4) ,
  OPERATOR 2 <= (int8, int4) ,
  OPERATOR 3 = (int8, int4) ,
  OPERATOR 4 >= (int8, int4) ,
  OPERATOR 5 > (int8, int4) ,
  FUNCTION 1 btint84cmp(int8, int4) ;
```

Notice that this definition "overloads" the operator strategy and support function numbers. This is allowed (for B-tree operator classes only) so long as each instance of a particular number has a different right-hand data type. The instances that are not cross-type are the default or primary operators of the operator class.

GiST indexes do not allow overloading of strategy or support function numbers, but it is still possible to get the effect of supporting multiple right-hand data types, by assigning a distinct strategy number to each operator that needs to be supported. The `consistent` support function must determine what it needs to do based on the strategy number, and must be prepared to accept comparison values of the appropriate data types.

33.14.6. System Dependencies on Operator Classes

PostgreSQL uses operator classes to infer the properties of operators in more ways than just whether they can be used with indexes. Therefore, you might want to create operator classes even if you have no intention of indexing any columns of your data type.

In particular, there are SQL features such as `ORDER BY` and `DISTINCT` that require comparison and sorting of values. To implement these features on a user-defined data type, PostgreSQL looks for the default B-tree operator class for the data type. The "equals" member of this operator class defines the system's notion of equality of values for `GROUP BY` and `DISTINCT`, and the sort ordering imposed by the operator class defines the default `ORDER BY` ordering.

Comparison of arrays of user-defined types also relies on the semantics defined by the default B-tree operator class.

If there is no default B-tree operator class for a data type, the system will look for a default hash operator class. But since that kind of operator class only provides equality, in practice it is only enough to support array equality.

When there is no default operator class for a data type, you will get errors like "could not identify an ordering operator" if you try to use these SQL features with the data type.

> **Note:** In PostgreSQL versions before 7.4, sorting and grouping operations would implicitly use operators named =, <, and >. The new behavior of relying on default operator classes avoids having to make any assumption about the behavior of operators with particular names.

33.14.7. Special Features of Operator Classes

There are two special features of operator classes that we have not discussed yet, mainly because they are not useful with the most commonly used index methods.

Normally, declaring an operator as a member of an operator class means that the index method can retrieve exactly the set of rows that satisfy a `WHERE` condition using the operator. For example,

```
SELECT * FROM table WHERE integer_column < 4;
```

can be satisfied exactly by a B-tree index on the integer column. But there are cases where an index is useful as an inexact guide to the matching rows. For example, if a GiST index stores only bounding boxes for objects, then it cannot exactly satisfy a `WHERE` condition that tests overlap between nonrectangular objects such as polygons. Yet we could use the index to find objects whose bounding box overlaps the bounding box of the target object, and then do the exact overlap test only on the objects found by the index. If this scenario applies, the index is said to be "lossy" for the operator, and we add `RECHECK` to the `OPERATOR` clause in the `CREATE OPERATOR CLASS` command. `RECHECK` is valid if the index is guaranteed to return all the required rows, plus perhaps some additional rows, which can be eliminated by performing the original operator invocation.

Consider again the situation where we are storing in the index only the bounding box of a complex object such as a polygon. In this case there's not much value in storing the whole polygon in the index entry — we may as well store just a simpler object of type `box`. This situation is expressed by the `STORAGE` option in `CREATE OPERATOR CLASS`: we'd write something like

```
CREATE OPERATOR CLASS polygon_ops
    DEFAULT FOR TYPE polygon USING gist AS
        ...
        STORAGE box;
```

At present, only the GiST and GIN index methods support a `STORAGE` type that's different from the column data type. The GiST `compress` and `decompress` support routines must deal with data-type conversion when `STORAGE` is used. In GIN, the `STORAGE` type identifies the type of the "key" values, which normally is different from the type of the indexed column — for example, an operator class for integer ar-

ray columns might have keys that are just integers. The GIN `extractValue` and `extractQuery` support routines are responsible for extracting keys from indexed values.

Chapter 34. Triggers

This chapter provides general information about writing trigger functions. Trigger functions can be written in most of the available procedural languages, including PL/pgSQL (Chapter 37), PL/Tcl (Chapter 38), PL/Perl (Chapter 39), and PL/Python (Chapter 40). After reading this chapter, you should consult the chapter for your favorite procedural language to find out the language-specific details of writing a trigger in it.

It is also possible to write a trigger function in C, although most people find it easier to use one of the procedural languages. It is not currently possible to write a trigger function in the plain SQL function language.

34.1. Overview of Trigger Behavior

A trigger is a specification that the database should automatically execute a particular function whenever a certain type of operation is performed. Triggers can be defined to execute either before or after any `INSERT`, `UPDATE`, or `DELETE` operation, either once per modified row, or once per SQL statement. If a trigger event occurs, the trigger's function is called at the appropriate time to handle the event.

The trigger function must be defined before the trigger itself can be created. The trigger function must be declared as a function taking no arguments and returning type `trigger`. (The trigger function receives its input through a specially-passed `TriggerData` structure, not in the form of ordinary function arguments.)

Once a suitable trigger function has been created, the trigger is established with *CREATE TRIGGER*. The same trigger function can be used for multiple triggers.

PostgreSQL offers both *per-row* triggers and *per-statement* triggers. With a per-row trigger, the trigger function is invoked once for each row that is affected by the statement that fired the trigger. In contrast, a per-statement trigger is invoked only once when an appropriate statement is executed, regardless of the number of rows affected by that statement. In particular, a statement that affects zero rows will still result in the execution of any applicable per-statement triggers. These two types of triggers are sometimes called *row-level* triggers and *statement-level* triggers, respectively.

Triggers are also classified as *before* triggers and *after* triggers. Statement-level before triggers naturally fire before the statement starts to do anything, while statement-level after triggers fire at the very end of the statement. Row-level before triggers fire immediately before a particular row is operated on, while row-level after triggers fire at the end of the statement (but before any statement-level after triggers).

Trigger functions invoked by per-statement triggers should always return `NULL`. Trigger functions invoked by per-row triggers can return a table row (a value of type `HeapTuple`) to the calling executor, if they choose. A row-level trigger fired before an operation has the following choices:

- It can return `NULL` to skip the operation for the current row. This instructs the executor to not perform the row-level operation that invoked the trigger (the insertion or modification of a particular table row).
- For row-level `INSERT` and `UPDATE` triggers only, the returned row becomes the row that will be inserted or will replace the row being updated. This allows the trigger function to modify the row being inserted or updated.

A row-level before trigger that does not intend to cause either of these behaviors must be careful to return as its result the same row that was passed in (that is, the NEW row for INSERT and UPDATE triggers, the OLD row for DELETE triggers).

The return value is ignored for row-level triggers fired after an operation, and so they may as well return NULL.

If more than one trigger is defined for the same event on the same relation, the triggers will be fired in alphabetical order by trigger name. In the case of before triggers, the possibly-modified row returned by each trigger becomes the input to the next trigger. If any before trigger returns NULL, the operation is abandoned for that row and subsequent triggers are not fired.

Typically, row before triggers are used for checking or modifying the data that will be inserted or updated. For example, a before trigger might be used to insert the current time into a timestamp column, or to check that two elements of the row are consistent. Row after triggers are most sensibly used to propagate the updates to other tables, or make consistency checks against other tables. The reason for this division of labor is that an after trigger can be certain it is seeing the final value of the row, while a before trigger cannot; there might be other before triggers firing after it. If you have no specific reason to make a trigger before or after, the before case is more efficient, since the information about the operation doesn't have to be saved until end of statement.

If a trigger function executes SQL commands then these commands may fire triggers again. This is known as cascading triggers. There is no direct limitation on the number of cascade levels. It is possible for cascades to cause a recursive invocation of the same trigger; for example, an INSERT trigger might execute a command that inserts an additional row into the same table, causing the INSERT trigger to be fired again. It is the trigger programmer's responsibility to avoid infinite recursion in such scenarios.

When a trigger is being defined, arguments can be specified for it. The purpose of including arguments in the trigger definition is to allow different triggers with similar requirements to call the same function. As an example, there could be a generalized trigger function that takes as its arguments two column names and puts the current user in one and the current time stamp in the other. Properly written, this trigger function would be independent of the specific table it is triggering on. So the same function could be used for INSERT events on any table with suitable columns, to automatically track creation of records in a transaction table for example. It could also be used to track last-update events if defined as an UPDATE trigger.

Each programming language that supports triggers has its own method for making the trigger input data available to the trigger function. This input data includes the type of trigger event (e.g., INSERT or UPDATE) as well as any arguments that were listed in CREATE TRIGGER. For a row-level trigger, the input data also includes the NEW row for INSERT and UPDATE triggers, and/or the OLD row for UPDATE and DELETE triggers. Statement-level triggers do not currently have any way to examine the individual row(s) modified by the statement.

34.2. Visibility of Data Changes

If you execute SQL commands in your trigger function, and these commands access the table that the trigger is for, then you need to be aware of the data visibility rules, because they determine whether these

SQL commands will see the data change that the trigger is fired for. Briefly:

- Statement-level triggers follow simple visibility rules: none of the changes made by a statement are visible to statement-level triggers that are invoked before the statement, whereas all modifications are visible to statement-level after triggers.
- The data change (insertion, update, or deletion) causing the trigger to fire is naturally *not* visible to SQL commands executed in a row-level before trigger, because it hasn't happened yet.
- However, SQL commands executed in a row-level before trigger *will* see the effects of data changes for rows previously processed in the same outer command. This requires caution, since the ordering of these change events is not in general predictable; a SQL command that affects multiple rows may visit the rows in any order.
- When a row-level after trigger is fired, all data changes made by the outer command are already complete, and are visible to the invoked trigger function.

Further information about data visibility rules can be found in Section 41.4. The example in Section 34.4 contains a demonstration of these rules.

34.3. Writing Trigger Functions in C

This section describes the low-level details of the interface to a trigger function. This information is only needed when writing trigger functions in C. If you are using a higher-level language then these details are handled for you. In most cases you should consider using a procedural language before writing your triggers in C. The documentation of each procedural language explains how to write a trigger in that language.

Trigger functions must use the "version 1" function manager interface.

When a function is called by the trigger manager, it is not passed any normal arguments, but it is passed a "context" pointer pointing to a `TriggerData` structure. C functions can check whether they were called from the trigger manager or not by executing the macro

```
CALLED_AS_TRIGGER(fcinfo)
```

which expands to

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, TriggerData))
```

If this returns true, then it is safe to cast `fcinfo->context` to type `TriggerData *` and make use of the pointed-to `TriggerData` structure. The function must *not* alter the `TriggerData` structure or any of the data it points to.

`struct TriggerData` is defined in `commands/trigger.h`:

```
typedef struct TriggerData
{
    NodeTag       type;
    TriggerEvent  tg_event;
```

```
    Relation      tg_relation;
    HeapTuple     tg_trigtuple;
    HeapTuple     tg_newtuple;
    Trigger      *tg_trigger;
    Buffer        tg_trigtuplebuf;
    Buffer        tg_newtuplebuf;
} TriggerData;
```

where the members are defined as follows:

`type`

Always `T_TriggerData`.

`tg_event`

Describes the event for which the function is called. You may use the following macros to examine `tg_event`:

`TRIGGER_FIRED_BEFORE(tg_event)`

Returns true if the trigger fired before the operation.

`TRIGGER_FIRED_AFTER(tg_event)`

Returns true if the trigger fired after the operation.

`TRIGGER_FIRED_FOR_ROW(tg_event)`

Returns true if the trigger fired for a row-level event.

`TRIGGER_FIRED_FOR_STATEMENT(tg_event)`

Returns true if the trigger fired for a statement-level event.

`TRIGGER_FIRED_BY_INSERT(tg_event)`

Returns true if the trigger was fired by an `INSERT` command.

`TRIGGER_FIRED_BY_UPDATE(tg_event)`

Returns true if the trigger was fired by an `UPDATE` command.

`TRIGGER_FIRED_BY_DELETE(tg_event)`

Returns true if the trigger was fired by a `DELETE` command.

`tg_relation`

A pointer to a structure describing the relation that the trigger fired for. Look at `utils/rel.h` for details about this structure. The most interesting things are `tg_relation->rd_att` (descriptor of the relation tuples) and `tg_relation->rd_rel->relname` (relation name; the type is not `char*` but `NameData`; use `SPI_getrelname(tg_relation)` to get a `char*` if you need a copy of the name).

`tg_trigtuple`

A pointer to the row for which the trigger was fired. This is the row being inserted, updated, or deleted. If this trigger was fired for an `INSERT` or `DELETE` then this is what you should return from the function if you don't want to replace the row with a different one (in the case of `INSERT`) or skip the operation.

`tg_newtuple`

A pointer to the new version of the row, if the trigger was fired for an `UPDATE`, and `NULL` if it is for an `INSERT` or a `DELETE`. This is what you have to return from the function if the event is an `UPDATE` and you don't want to replace this row by a different one or skip the operation.

`tg_trigger`

A pointer to a structure of type `Trigger`, defined in `utils/rel.h`:

```
typedef struct Trigger
{
    Oid         tgoid;
    char       *tgname;
    Oid         tgfoid;
    int16       tgtype;
    bool        tgenabled;
    bool        tgisconstraint;
    Oid         tgconstrrelid;
    bool        tgdeferrable;
    bool        tginitdeferred;
    int16       tgnargs;
    int16       tgnattr;
    int16      *tgattr;
    char      **tgargs;
} Trigger;
```

where `tgname` is the trigger's name, `tgnargs` is number of arguments in `tgargs`, and `tgargs` is an array of pointers to the arguments specified in the `CREATE TRIGGER` statement. The other members are for internal use only.

`tg_trigtuplebuf`

The buffer containing `tg_trigtuple`, or `InvalidBuffer` if there is no such tuple or it is not stored in a disk buffer.

`tg_newtuplebuf`

The buffer containing `tg_newtuple`, or `InvalidBuffer` if there is no such tuple or it is not stored in a disk buffer.

A trigger function must return either a `HeapTuple` pointer or a `NULL` pointer (*not* an SQL null value, that is, do not set `isNull` true). Be careful to return either `tg_trigtuple` or `tg_newtuple`, as appropriate, if you don't want to modify the row being operated on.

34.4. A Complete Example

Here is a very simple example of a trigger function written in C. (Examples of triggers written in procedural languages may be found in the documentation of the procedural languages.)

The function `trigf` reports the number of rows in the table `ttest` and skips the actual operation if the command attempts to insert a null value into the column `x`. (So the trigger acts as a not-null constraint but doesn't abort the transaction.)

First, the table definition:

```
CREATE TABLE ttest (
    x integer
);
```

This is the source code of the trigger function:

```
#include "postgres.h"
#include "executor/spi.h"       /* this is what you need to work with SPI */
#include "commands/trigger.h"   /* ... and triggers */

extern Datum trigf(PG_FUNCTION_ARGS);

PG_FUNCTION_INFO_V1(trigf);

Datum
trigf(PG_FUNCTION_ARGS)
{
    TriggerData *trigdata = (TriggerData *) fcinfo->context;
    TupleDesc   tupdesc;
    HeapTuple   rettuple;
    char       *when;
    bool        checknull = false;
    bool        isnull;
    int         ret, i;

    /* make sure it's called as a trigger at all */
    if (!CALLED_AS_TRIGGER(fcinfo))
        elog(ERROR, "trigf: not called by trigger manager");

    /* tuple to return to executor */
    if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))
        rettuple = trigdata->tg_newtuple;
    else
        rettuple = trigdata->tg_trigtuple;

    /* check for null values */
    if (!TRIGGER_FIRED_BY_DELETE(trigdata->tg_event)
        && TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        checknull = true;

    if (TRIGGER_FIRED_BEFORE(trigdata->tg_event))
```

```
        when = "before";
    else
        when = "after ";

    tupdesc = trigdata->tg_relation->rd_att;

    /* connect to SPI manager */
    if ((ret = SPI_connect()) < 0)
        elog(INFO, "trigf (fired %s): SPI_connect returned %d", when, ret);

    /* get number of rows in table */
    ret = SPI_exec("SELECT count(*) FROM ttest", 0);

    if (ret < 0)
        elog(NOTICE, "trigf (fired %s): SPI_exec returned %d", when, ret);

    /* count(*) returns int8, so be careful to convert */
    i = DatumGetInt64(SPI_getbinval(SPI_tuptable->vals[0],
                                    SPI_tuptable->tupdesc,
                                    1,
                                    &isnull));

    elog (INFO, "trigf (fired %s): there are %d rows in ttest", when, i);

    SPI_finish();

    if (checknull)
    {
        SPI_getbinval(rettuple, tupdesc, 1, &isnull);
        if (isnull)
            rettuple = NULL;
    }

    return PointerGetDatum(rettuple);
}
```

After you have compiled the source code, declare the function and the triggers:

```
CREATE FUNCTION trigf() RETURNS trigger
    AS 'filename'
    LANGUAGE C;

CREATE TRIGGER tbefore BEFORE INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE PROCEDURE trigf();

CREATE TRIGGER tafter AFTER INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE PROCEDURE trigf();
```

Now you can test the operation of the trigger:

```
=> INSERT INTO ttest VALUES (NULL);
INFO:  trigf (fired before): there are 0 rows in ttest
INSERT 0 0

-- Insertion skipped and AFTER trigger is not fired

=> SELECT * FROM ttest;
 x
---
(0 rows)

=> INSERT INTO ttest VALUES (1);
INFO:  trigf (fired before): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 1 rows in ttest
                                       ^^^^^^^^
                             remember what we said about visibility.
INSERT 167793 1
vac=> SELECT * FROM ttest;
 x
---
 1
(1 row)

=> INSERT INTO ttest SELECT x * 2 FROM ttest;
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
                                       ^^^^^^
                             remember what we said about visibility.
INSERT 167794 1
=> SELECT * FROM ttest;
 x
---
 1
 2
(2 rows)

=> UPDATE ttest SET x = NULL WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
UPDATE 0
=> UPDATE ttest SET x = 4 WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
UPDATE 1
vac=> SELECT * FROM ttest;
 x
---
 1
 4
(2 rows)

=> DELETE FROM ttest;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired before): there are 1 rows in ttest
```

```
INFO:  trigf (fired after ): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
                                       ^^^^^^
                             remember what we said about visibility.
DELETE 2
=> SELECT * FROM ttest;
 x
---
(0 rows)
```

There are more complex examples in `src/test/regress/regress.c` and in `contrib/spi`.

Chapter 35. The Rule System

This chapter discusses the rule system in PostgreSQL. Production rule systems are conceptually simple, but there are many subtle points involved in actually using them.

Some other database systems define active database rules, which are usually stored procedures and triggers. In PostgreSQL, these can be implemented using functions and triggers as well.

The rule system (more precisely speaking, the query rewrite rule system) is totally different from stored procedures and triggers. It modifies queries to take rules into consideration, and then passes the modified query to the query planner for planning and execution. It is very powerful, and can be used for many things such as query language procedures, views, and versions. The theoretical foundations and the power of this rule system are also discussed in *On Rules, Procedures, Caching and Views in Database Systems* and *A Unified Framework for Version Modeling Using Production Rules in a Database System*.

35.1. The Query Tree

To understand how the rule system works it is necessary to know when it is invoked and what its input and results are.

The rule system is located between the parser and the planner. It takes the output of the parser, one query tree, and the user-defined rewrite rules, which are also query trees with some extra information, and creates zero or more query trees as result. So its input and output are always things the parser itself could have produced and thus, anything it sees is basically representable as an SQL statement.

Now what is a query tree? It is an internal representation of an SQL statement where the single parts that it is built from are stored separately. These query trees can be shown in the server log if you set the configuration parameters `debug_print_parse`, `debug_print_rewritten`, or `debug_print_plan`. The rule actions are also stored as query trees, in the system catalog `pg_rewrite`. They are not formatted like the log output, but they contain exactly the same information.

Reading a raw query tree requires some experience. But since SQL representations of query trees are sufficient to understand the rule system, this chapter will not teach how to read them.

When reading the SQL representations of the query trees in this chapter it is necessary to be able to identify the parts the statement is broken into when it is in the query tree structure. The parts of a query tree are

the command type

This is a simple value telling which command (`SELECT`, `INSERT`, `UPDATE`, `DELETE`) produced the query tree.

the range table

The range table is a list of relations that are used in the query. In a `SELECT` statement these are the relations given after the `FROM` key word.

Every range table entry identifies a table or view and tells by which name it is called in the other parts of the query. In the query tree, the range table entries are referenced by number rather than by name, so here it doesn't matter if there are duplicate names as it would in an SQL statement. This

can happen after the range tables of rules have been merged in. The examples in this chapter will not have this situation.

the result relation

This is an index into the range table that identifies the relation where the results of the query go.

`SELECT` queries normally don't have a result relation. The special case of a `SELECT INTO` is mostly identical to a `CREATE TABLE` followed by a `INSERT ... SELECT` and is not discussed separately here.

For `INSERT`, `UPDATE`, and `DELETE` commands, the result relation is the table (or view!) where the changes are to take effect.

the target list

The target list is a list of expressions that define the result of the query. In the case of a `SELECT`, these expressions are the ones that build the final output of the query. They correspond to the expressions between the key words `SELECT` and `FROM`. (`*` is just an abbreviation for all the column names of a relation. It is expanded by the parser into the individual columns, so the rule system never sees it.)

`DELETE` commands don't need a target list because they don't produce any result. In fact, the planner will add a special CTID entry to the empty target list, but this is after the rule system and will be discussed later; for the rule system, the target list is empty.

For `INSERT` commands, the target list describes the new rows that should go into the result relation. It consists of the expressions in the `VALUES` clause or the ones from the `SELECT` clause in `INSERT ... SELECT`. The first step of the rewrite process adds target list entries for any columns that were not assigned to by the original command but have defaults. Any remaining columns (with neither a given value nor a default) will be filled in by the planner with a constant null expression.

For `UPDATE` commands, the target list describes the new rows that should replace the old ones. In the rule system, it contains just the expressions from the `SET column = expression` part of the command. The planner will handle missing columns by inserting expressions that copy the values from the old row into the new one. And it will add the special CTID entry just as for `DELETE`, too.

Every entry in the target list contains an expression that can be a constant value, a variable pointing to a column of one of the relations in the range table, a parameter, or an expression tree made of function calls, constants, variables, operators, etc.

the qualification

The query's qualification is an expression much like one of those contained in the target list entries. The result value of this expression is a Boolean that tells whether the operation (`INSERT`, `UPDATE`, `DELETE`, or `SELECT`) for the final result row should be executed or not. It corresponds to the `WHERE` clause of an SQL statement.

the join tree

The query's join tree shows the structure of the `FROM` clause. For a simple query like `SELECT ... FROM a, b, c`, the join tree is just a list of the `FROM` items, because we are allowed to join them in any order. But when `JOIN` expressions, particularly outer joins, are used, we have to join in the order shown by the joins. In that case, the join tree shows the structure of the `JOIN` expressions. The restrictions associated with particular `JOIN` clauses (from `ON` or `USING` expressions) are stored as qualification expressions attached to those join-tree nodes. It turns out to be convenient to store the

top-level WHERE expression as a qualification attached to the top-level join-tree item, too. So really the join tree represents both the FROM and WHERE clauses of a SELECT.

the others

The other parts of the query tree like the ORDER BY clause aren't of interest here. The rule system substitutes some entries there while applying rules, but that doesn't have much to do with the fundamentals of the rule system.

35.2. Views and the Rule System

Views in PostgreSQL are implemented using the rule system. In fact, there is essentially no difference between

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

compared against the two commands

```
CREATE TABLE myview (same column list as mytab);
CREATE RULE "_RETURN" AS ON SELECT TO myview DO INSTEAD
    SELECT * FROM mytab;
```

because this is exactly what the CREATE VIEW command does internally. This has some side effects. One of them is that the information about a view in the PostgreSQL system catalogs is exactly the same as it is for a table. So for the parser, there is absolutely no difference between a table and a view. They are the same thing: relations.

35.2.1. How SELECT Rules Work

Rules ON SELECT are applied to all queries as the last step, even if the command given is an INSERT, UPDATE or DELETE. And they have different semantics from rules on the other command types in that they modify the query tree in place instead of creating a new one. So SELECT rules are described first.

Currently, there can be only one action in an ON SELECT rule, and it must be an unconditional SELECT action that is INSTEAD. This restriction was required to make rules safe enough to open them for ordinary users, and it restricts ON SELECT rules to act like views.

The examples for this chapter are two join views that do some calculations and some more views using them in turn. One of the two first views is customized later by adding rules for INSERT, UPDATE, and DELETE operations so that the final result will be a view that behaves like a real table with some magic functionality. This is not such a simple example to start from and this makes things harder to get into. But it's better to have one example that covers all the points discussed step by step rather than having many different ones that might mix up in mind.

For the example, we need a little min function that returns the lower of 2 integer values. We create that as

```
CREATE FUNCTION min(integer, integer) RETURNS integer AS $$
    SELECT CASE WHEN $1 < $2 THEN $1 ELSE $2 END
$$ LANGUAGE SQL STRICT;
```

The real tables we need in the first two rule system descriptions are these:

```
CREATE TABLE shoe_data (
    shoename   text,          -- primary key
    sh_avail   integer,       -- available number of pairs
    slcolor    text,          -- preferred shoelace color
    slminlen   real,          -- minimum shoelace length
    slmaxlen   real,          -- maximum shoelace length
    slunit     text           -- length unit
);

CREATE TABLE shoelace_data (
    sl_name    text,          -- primary key
    sl_avail   integer,       -- available number of pairs
    sl_color   text,          -- shoelace color
    sl_len     real,          -- shoelace length
    sl_unit    text           -- length unit
);

CREATE TABLE unit (
    un_name    text,          -- primary key
    un_fact    real           -- factor to transform to cm
);
```

As you can see, they represent shoe-store data.

The views are created as

```
CREATE VIEW shoe AS
    SELECT sh.shoename,
           sh.sh_avail,
           sh.slcolor,
           sh.slminlen,
           sh.slminlen * un.un_fact AS slminlen_cm,
           sh.slmaxlen,
           sh.slmaxlen * un.un_fact AS slmaxlen_cm,
           sh.slunit
      FROM shoe_data sh, unit un
     WHERE sh.slunit = un.un_name;

CREATE VIEW shoelace AS
    SELECT s.sl_name,
           s.sl_avail,
           s.sl_color,
           s.sl_len,
           s.sl_unit,
           s.sl_len * u.un_fact AS sl_len_cm
      FROM shoelace_data s, unit u
     WHERE s.sl_unit = u.un_name;

CREATE VIEW shoe_ready AS
    SELECT rsh.shoename,
```

```
        rsh.sh_avail,
        rsl.sl_name,
        rsl.sl_avail,
        min(rsh.sh_avail, rsl.sl_avail) AS total_avail
   FROM shoe rsh, shoelace rsl
  WHERE rsl.sl_color = rsh.slcolor
    AND rsl.sl_len_cm >= rsh.slminlen_cm
    AND rsl.sl_len_cm <= rsh.slmaxlen_cm;
```

The `CREATE VIEW` command for the `shoelace` view (which is the simplest one we have) will create a relation `shoelace` and an entry in `pg_rewrite` that tells that there is a rewrite rule that must be applied whenever the relation `shoelace` is referenced in a query's range table. The rule has no rule qualification (discussed later, with the non-`SELECT` rules, since `SELECT` rules currently cannot have them) and it is `INSTEAD`. Note that rule qualifications are not the same as query qualifications. The action of our rule has a query qualification. The action of the rule is one query tree that is a copy of the `SELECT` statement in the view creation command.

> **Note:** The two extra range table entries for `NEW` and `OLD` (named `*NEW*` and `*OLD*` for historical reasons in the printed query tree) you can see in the `pg_rewrite` entry aren't of interest for `SELECT` rules.

Now we populate `unit`, `shoe_data` and `shoelace_data` and run a simple query on a view:

```
INSERT INTO unit VALUES ('cm', 1.0);
INSERT INTO unit VALUES ('m', 100.0);
INSERT INTO unit VALUES ('inch', 2.54);

INSERT INTO shoe_data VALUES ('sh1', 2, 'black', 70.0, 90.0, 'cm');
INSERT INTO shoe_data VALUES ('sh2', 0, 'black', 30.0, 40.0, 'inch');
INSERT INTO shoe_data VALUES ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
INSERT INTO shoe_data VALUES ('sh4', 3, 'brown', 40.0, 50.0, 'inch');

INSERT INTO shoelace_data VALUES ('sl1', 5, 'black', 80.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl2', 6, 'black', 100.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl3', 0, 'black', 35.0 , 'inch');
INSERT INTO shoelace_data VALUES ('sl4', 8, 'black', 40.0 , 'inch');
INSERT INTO shoelace_data VALUES ('sl5', 4, 'brown', 1.0 , 'm');
INSERT INTO shoelace_data VALUES ('sl6', 0, 'brown', 0.9 , 'm');
INSERT INTO shoelace_data VALUES ('sl7', 7, 'brown', 60 , 'cm');
INSERT INTO shoelace_data VALUES ('sl8', 1, 'brown', 40 , 'inch');

SELECT * FROM shoelace;

 sl_name   | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
-----------+----------+----------+--------+---------+-----------
 sl1       |        5 | black    |     80 | cm      |        80
 sl2       |        6 | black    |    100 | cm      |       100
 sl7       |        7 | brown    |     60 | cm      |        60
 sl3       |        0 | black    |     35 | inch    |      88.9
 sl4       |        8 | black    |     40 | inch    |     101.6
 sl8       |        1 | brown    |     40 | inch    |     101.6
```

```
 sl5       |        4 | brown    |      1 | m       |       100
 sl6       |        0 | brown    |    0.9 | m       |        90
(8 rows)
```

This is the simplest `SELECT` you can do on our views, so we take this opportunity to explain the basics of view rules. The `SELECT * FROM shoelace` was interpreted by the parser and produced the query tree

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM shoelace shoelace;
```

and this is given to the rule system. The rule system walks through the range table and checks if there are rules for any relation. When processing the range table entry for `shoelace` (the only one up to now) it finds the `_RETURN` rule with the query tree

```
SELECT s.sl_name, s.sl_avail,
       s.sl_color, s.sl_len, s.sl_unit,
       s.sl_len * u.un_fact AS sl_len_cm
  FROM shoelace *OLD*, shoelace *NEW*,
       shoelace_data s, unit u
 WHERE s.sl_unit = u.un_name;
```

To expand the view, the rewriter simply creates a subquery range-table entry containing the rule's action query tree, and substitutes this range table entry for the original one that referenced the view. The resulting rewritten query tree is almost the same as if you had typed

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM (SELECT s.sl_name,
               s.sl_avail,
               s.sl_color,
               s.sl_len,
               s.sl_unit,
               s.sl_len * u.un_fact AS sl_len_cm
          FROM shoelace_data s, unit u
         WHERE s.sl_unit = u.un_name) shoelace;
```

There is one difference however: the subquery's range table has two extra entries `shoelace *OLD*` and `shoelace *NEW*`. These entries don't participate directly in the query, since they aren't referenced by the subquery's join tree or target list. The rewriter uses them to store the access privilege check information that was originally present in the range-table entry that referenced the view. In this way, the executor will still check that the user has proper privileges to access the view, even though there's no direct use of the view in the rewritten query.

That was the first rule applied. The rule system will continue checking the remaining range-table entries in the top query (in this example there are no more), and it will recursively check the range-table entries in the added subquery to see if any of them reference views. (But it won't expand `*OLD*` or `*NEW*` —

otherwise we'd have infinite recursion!) In this example, there are no rewrite rules for `shoelace_data` or `unit`, so rewriting is complete and the above is the final result given to the planner.

Now we want to write a query that finds out for which shoes currently in the store we have the matching shoelaces (color and length) and where the total number of exactly matching pairs is greater or equal to two.

```
SELECT * FROM shoe_ready WHERE total_avail >= 2;

 shoename | sh_avail | sl_name | sl_avail | total_avail
----------+----------+---------+----------+-------------
 sh1      |        2 | sl1     |        5 |           2
 sh3      |        4 | sl7     |        7 |           4
(2 rows)
```

The output of the parser this time is the query tree

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM shoe_ready shoe_ready
 WHERE shoe_ready.total_avail >= 2;
```

The first rule applied will be the one for the `shoe_ready` view and it results in the query tree

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               min(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM shoe rsh, shoelace rsl
         WHERE rsl.sl_color = rsh.slcolor
           AND rsl.sl_len_cm >= rsh.slminlen_cm
           AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
 WHERE shoe_ready.total_avail >= 2;
```

Similarly, the rules for `shoe` and `shoelace` are substituted into the range table of the subquery, leading to a three-level final query tree:

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               min(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM (SELECT sh.shoename,
                       sh.sh_avail,
```

```
                        sh.slcolor,
                        sh.slminlen,
                        sh.slminlen * un.un_fact AS slminlen_cm,
                        sh.slmaxlen,
                        sh.slmaxlen * un.un_fact AS slmaxlen_cm,
                        sh.slunit
                   FROM shoe_data sh, unit un
                  WHERE sh.slunit = un.un_name) rsh,
                (SELECT s.sl_name,
                        s.sl_avail,
                        s.sl_color,
                        s.sl_len,
                        s.sl_unit,
                        s.sl_len * u.un_fact AS sl_len_cm
                   FROM shoelace_data s, unit u
                  WHERE s.sl_unit = u.un_name) rsl
          WHERE rsl.sl_color = rsh.slcolor
            AND rsl.sl_len_cm >= rsh.slminlen_cm
            AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
 WHERE shoe_ready.total_avail > 2;
```

It turns out that the planner will collapse this tree into a two-level query tree: the bottommost `SELECT` commands will be "pulled up" into the middle `SELECT` since there's no need to process them separately. But the middle `SELECT` will remain separate from the top, because it contains aggregate functions. If we pulled those up it would change the behavior of the topmost `SELECT`, which we don't want. However, collapsing the query tree is an optimization that the rewrite system doesn't have to concern itself with.

35.2.2. View Rules in Non-`SELECT` Statements

Two details of the query tree aren't touched in the description of view rules above. These are the command type and the result relation. In fact, view rules don't need this information.

There are only a few differences between a query tree for a `SELECT` and one for any other command. Obviously, they have a different command type and for a command other than a `SELECT`, the result relation points to the range-table entry where the result should go. Everything else is absolutely the same. So having two tables `t1` and `t2` with columns `a` and `b`, the query trees for the two statements

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

```
UPDATE t1 SET b = t2.b FROM t2 WHERE t1.a = t2.a;
```

are nearly identical. In particular:

- The range tables contain entries for the tables `t1` and `t2`.
- The target lists contain one variable that points to column `b` of the range table entry for table `t2`.
- The qualification expressions compare the columns `a` of both range-table entries for equality.

- The join trees show a simple join between t1 and t2.

The consequence is, that both query trees result in similar execution plans: They are both joins over the two tables. For the UPDATE the missing columns from t1 are added to the target list by the planner and the final query tree will read as

```
UPDATE t1 SET a = t1.a, b = t2.b FROM t2 WHERE t1.a = t2.a;
```

and thus the executor run over the join will produce exactly the same result set as a

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

will do. But there is a little problem in UPDATE: The executor does not care what the results from the join it is doing are meant for. It just produces a result set of rows. The difference that one is a SELECT command and the other is an UPDATE is handled in the caller of the executor. The caller still knows (looking at the query tree) that this is an UPDATE, and it knows that this result should go into table t1. But which of the rows that are there has to be replaced by the new row?

To resolve this problem, another entry is added to the target list in UPDATE (and also in DELETE) statements: the current tuple ID (CTID). This is a system column containing the file block number and position in the block for the row. Knowing the table, the CTID can be used to retrieve the original row of t1 to be updated. After adding the CTID to the target list, the query actually looks like

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

Now another detail of PostgreSQL enters the stage. Old table rows aren't overwritten, and this is why ROLLBACK is fast. In an UPDATE, the new result row is inserted into the table (after stripping the CTID) and in the row header of the old row, which the CTID pointed to, the cmax and xmax entries are set to the current command counter and current transaction ID. Thus the old row is hidden, and after the transaction commits the vacuum cleaner can really remove it.

Knowing all that, we can simply apply view rules in absolutely the same way to any command. There is no difference.

35.2.3. The Power of Views in PostgreSQL

The above demonstrates how the rule system incorporates view definitions into the original query tree. In the second example, a simple SELECT from one view created a final query tree that is a join of 4 tables (unit was used twice with different names).

The benefit of implementing views with the rule system is, that the planner has all the information about which tables have to be scanned plus the relationships between these tables plus the restrictive qualifications from the views plus the qualifications from the original query in one single query tree. And this is still the situation when the original query is already a join over views. The planner has to decide which is the best path to execute the query, and the more information the planner has, the better this decision can be. And the rule system as implemented in PostgreSQL ensures, that this is all information available about the query up to that point.

35.2.4. Updating a View

What happens if a view is named as the target relation for an INSERT, UPDATE, or DELETE? After doing the substitutions described above, we will have a query tree in which the result relation points at a subquery range-table entry. This will not work, so the rewriter throws an error if it sees it has produced such a thing.

To change this, we can define rules that modify the behavior of these kinds of commands. This is the topic of the next section.

35.3. Rules on INSERT, UPDATE, and DELETE

Rules that are defined on INSERT, UPDATE, and DELETE are significantly different from the view rules described in the previous section. First, their CREATE RULE command allows more:

- They are allowed to have no action.
- They can have multiple actions.
- They can be INSTEAD or ALSO (the default).
- The pseudorelations NEW and OLD become useful.
- They can have rule qualifications.

Second, they don't modify the query tree in place. Instead they create zero or more new query trees and can throw away the original one.

35.3.1. How Update Rules Work

Keep the syntax

```
CREATE [ OR REPLACE ] RULE name AS ON event
    TO table [ WHERE condition ]
    DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }
```

in mind. In the following, *update rules* means rules that are defined on INSERT, UPDATE, or DELETE.

Update rules get applied by the rule system when the result relation and the command type of a query tree are equal to the object and event given in the CREATE RULE command. For update rules, the rule system creates a list of query trees. Initially the query-tree list is empty. There can be zero (NOTHING key word), one, or multiple actions. To simplify, we will look at a rule with one action. This rule can have a qualification or not and it can be INSTEAD or ALSO (the default).

What is a rule qualification? It is a restriction that tells when the actions of the rule should be done and when not. This qualification can only reference the pseudorelations NEW and/or OLD, which basically represent the relation that was given as object (but with a special meaning).

So we have three cases that produce the following query trees for a one-action rule.

No qualification, with either ALSO or INSTEAD

the query tree from the rule action with the original query tree's qualification added

Qualification given and ALSO

the query tree from the rule action with the rule qualification and the original query tree's qualification added

Qualification given and INSTEAD

the query tree from the rule action with the rule qualification and the original query tree's qualification; and the original query tree with the negated rule qualification added

Finally, if the rule is ALSO, the unchanged original query tree is added to the list. Since only qualified INSTEAD rules already add the original query tree, we end up with either one or two output query trees for a rule with one action.

For ON INSERT rules, the original query (if not suppressed by INSTEAD) is done before any actions added by rules. This allows the actions to see the inserted row(s). But for ON UPDATE and ON DELETE rules, the original query is done after the actions added by rules. This ensures that the actions can see the to-be-updated or to-be-deleted rows; otherwise, the actions might do nothing because they find no rows matching their qualifications.

The query trees generated from rule actions are thrown into the rewrite system again, and maybe more rules get applied resulting in more or less query trees. So a rule's actions must have either a different command type or a different result relation than the rule itself is on, otherwise this recursive process will end up in an infinite loop. (Recursive expansion of a rule will be detected and reported as an error.)

The query trees found in the actions of the pg_rewrite system catalog are only templates. Since they can reference the range-table entries for NEW and OLD, some substitutions have to be made before they can be used. For any reference to NEW, the target list of the original query is searched for a corresponding entry. If found, that entry's expression replaces the reference. Otherwise, NEW means the same as OLD (for an UPDATE) or is replaced by a null value (for an INSERT). Any reference to OLD is replaced by a reference to the range-table entry that is the result relation.

After the system is done applying update rules, it applies view rules to the produced query tree(s). Views cannot insert new update actions so there is no need to apply update rules to the output of view rewriting.

35.3.1.1. A First Rule Step by Step

Say we want to trace changes to the sl_avail column in the shoelace_data relation. So we set up a log table and a rule that conditionally writes a log entry when an UPDATE is performed on shoelace_data.

```
CREATE TABLE shoelace_log (
    sl_name    text,          -- shoelace changed
    sl_avail   integer,       -- new available value
    log_who    text,          -- who did it
    log_when   timestamp      -- when
);

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
    WHERE NEW.sl_avail <> OLD.sl_avail
    DO INSERT INTO shoelace_log VALUES (
                                    NEW.sl_name,
```

```
                                    NEW.sl_avail,
                                    current_user,
                                    current_timestamp
                                );
```

Now someone does:

```
UPDATE shoelace_data SET sl_avail = 6 WHERE sl_name = 'sl7';
```

and we look at the log table:

```
SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who | log_when
---------+----------+---------+----------------------------------
 sl7     |        6 | Al      | Tue Oct 20 16:14:45 1998 MET DST
(1 row)
```

That's what we expected. What happened in the background is the following. The parser created the query tree

```
UPDATE shoelace_data SET sl_avail = 6
  FROM shoelace_data shoelace_data
 WHERE shoelace_data.sl_name = 'sl7';
```

There is a rule `log_shoelace` that is `ON UPDATE` with the rule qualification expression

```
NEW.sl_avail <> OLD.sl_avail
```

and the action

```
INSERT INTO shoelace_log VALUES (
       *NEW*.sl_name, *NEW*.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data *NEW*, shoelace_data *OLD*;
```

(This looks a little strange since you can't normally write `INSERT ... VALUES ... FROM`. The `FROM` clause here is just to indicate that there are range-table entries in the query tree for `*NEW*` and `*OLD*`. These are needed so that they can be referenced by variables in the `INSERT` command's query tree.)

The rule is a qualified `ALSO` rule, so the rule system has to return two query trees: the modified rule action and the original query tree. In step 1, the range table of the original query is incorporated into the rule's action query tree. This results in:

```
INSERT INTO shoelace_log VALUES (
       *NEW*.sl_name, *NEW*.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data;
```

In step 2, the rule qualification is added to it, so the result set is restricted to rows where `sl_avail` changes:

```
INSERT INTO shoelace_log VALUES (
       *NEW*.sl_name, *NEW*.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data
 WHERE *NEW*.sl_avail <> *OLD*.sl_avail;
```

(This looks even stranger, since `INSERT ... VALUES` doesn't have a `WHERE` clause either, but the planner and executor will have no difficulty with it. They need to support this same functionality anyway for `INSERT ... SELECT`.)

In step 3, the original query tree's qualification is added, restricting the result set further to only the rows that would have been touched by the original query:

```
INSERT INTO shoelace_log VALUES (
       *NEW*.sl_name, *NEW*.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data
 WHERE *NEW*.sl_avail <> *OLD*.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

Step 4 replaces references to `NEW` by the target list entries from the original query tree or by the matching variable references from the result relation:

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data
 WHERE 6 <> *OLD*.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

Step 5 changes `OLD` references into result relation references:

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data
 WHERE 6 <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

That's it. Since the rule is `ALSO`, we also output the original query tree. In short, the output from the rule system is a list of two query trees that correspond to these statements:

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data
 WHERE 6 <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';

UPDATE shoelace_data SET sl_avail = 6
 WHERE sl_name = 'sl7';
```

These are executed in this order, and that is exactly what the rule was meant to do.

The substitutions and the added qualifications ensure that, if the original query would be, say,

```
UPDATE shoelace_data SET sl_color = 'green'
 WHERE sl_name = 'sl7';
```

no log entry would get written. In that case, the original query tree does not contain a target list entry for `sl_avail`, so `NEW.sl_avail` will get replaced by `shoelace_data.sl_avail`. Thus, the extra command generated by the rule is

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, shoelace_data.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data
 WHERE shoelace_data.sl_avail <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

and that qualification will never be true.

It will also work if the original query modifies multiple rows. So if someone issued the command

```
UPDATE shoelace_data SET sl_avail = 0
 WHERE sl_color = 'black';
```

four rows in fact get updated (`sl1`, `sl2`, `sl3`, and `sl4`). But `sl3` already has `sl_avail = 0`. In this case, the original query trees qualification is different and that results in the extra query tree

```
INSERT INTO shoelace_log
SELECT shoelace_data.sl_name, 0,
       current_user, current_timestamp
  FROM shoelace_data
 WHERE 0 <> shoelace_data.sl_avail
   AND shoelace_data.sl_color = 'black';
```

being generated by the rule. This query tree will surely insert three new log entries. And that's absolutely correct.

Here we can see why it is important that the original query tree is executed last. If the `UPDATE` had been executed first, all the rows would have already been set to zero, so the logging `INSERT` would not find any row where `0 <> shoelace_data.sl_avail`.

35.3.2. Cooperation with Views

A simple way to protect view relations from the mentioned possibility that someone can try to run INSERT, UPDATE, or DELETE on them is to let those query trees get thrown away. So we could create the rules

```
CREATE RULE shoe_ins_protect AS ON INSERT TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_upd_protect AS ON UPDATE TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_del_protect AS ON DELETE TO shoe
    DO INSTEAD NOTHING;
```

If someone now tries to do any of these operations on the view relation shoe, the rule system will apply these rules. Since the rules have no actions and are INSTEAD, the resulting list of query trees will be empty and the whole query will become nothing because there is nothing left to be optimized or executed after the rule system is done with it.

A more sophisticated way to use the rule system is to create rules that rewrite the query tree into one that does the right operation on the real tables. To do that on the shoelace view, we create the following rules:

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
    DO INSTEAD
    INSERT INTO shoelace_data VALUES (
           NEW.sl_name,
           NEW.sl_avail,
           NEW.sl_color,
           NEW.sl_len,
           NEW.sl_unit
    );

CREATE RULE shoelace_upd AS ON UPDATE TO shoelace
    DO INSTEAD
    UPDATE shoelace_data
       SET sl_name = NEW.sl_name,
           sl_avail = NEW.sl_avail,
           sl_color = NEW.sl_color,
           sl_len = NEW.sl_len,
           sl_unit = NEW.sl_unit
     WHERE sl_name = OLD.sl_name;

CREATE RULE shoelace_del AS ON DELETE TO shoelace
    DO INSTEAD
    DELETE FROM shoelace_data
     WHERE sl_name = OLD.sl_name;
```

If you want to support RETURNING queries on the view, you need to make the rules include RETURNING clauses that compute the view rows. This is usually pretty trivial for views on a single table, but it's a bit tedious for join views such as shoelace. An example for the insert case is

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
    DO INSTEAD
```

```
    INSERT INTO shoelace_data VALUES (
           NEW.sl_name,
           NEW.sl_avail,
           NEW.sl_color,
           NEW.sl_len,
           NEW.sl_unit
    )
    RETURNING
           shoelace_data.*,
           (SELECT shoelace_data.sl_len * u.un_fact
            FROM unit u WHERE shoelace_data.sl_unit = u.un_name);
```

Note that this one rule supports both `INSERT` and `INSERT RETURNING` queries on the view — the `RETURNING` clause is simply ignored for `INSERT`.

Now assume that once in a while, a pack of shoelaces arrives at the shop and a big parts list along with it. But you don't want to manually update the `shoelace` view every time. Instead we setup two little tables: one where you can insert the items from the part list, and one with a special trick. The creation commands for these are:

```
CREATE TABLE shoelace_arrive (
    arr_name    text,
    arr_quant   integer
);

CREATE TABLE shoelace_ok (
    ok_name     text,
    ok_quant    integer
);

CREATE RULE shoelace_ok_ins AS ON INSERT TO shoelace_ok
    DO INSTEAD
    UPDATE shoelace
       SET sl_avail = sl_avail + NEW.ok_quant
     WHERE sl_name = NEW.ok_name;
```

Now you can fill the table `shoelace_arrive` with the data from the parts list:

```
SELECT * FROM shoelace_arrive;

 arr_name | arr_quant
----------+-----------
 sl3      |        10
 sl6      |        20
 sl8      |        20
(3 rows)
```

Take a quick look at the current data:

```
SELECT * FROM shoelace;

 sl_name  | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
----------+----------+----------+--------+---------+-----------
 sl1      |        5 | black    |     80 | cm      |        80
```

```
 sl2      |        6 | black    |    100 | cm      |       100
 sl7      |        6 | brown    |     60 | cm      |        60
 sl3      |        0 | black    |     35 | inch    |      88.9
 sl4      |        8 | black    |     40 | inch    |     101.6
 sl8      |        1 | brown    |     40 | inch    |     101.6
 sl5      |        4 | brown    |      1 | m       |       100
 sl6      |        0 | brown    |    0.9 | m       |        90
(8 rows)
```

Now move the arrived shoelaces in:

```
INSERT INTO shoelace_ok SELECT * FROM shoelace_arrive;
```

and check the results:

```
SELECT * FROM shoelace ORDER BY sl_name;

 sl_name  | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
----------+----------+----------+--------+---------+-----------
 sl1      |        5 | black    |     80 | cm      |        80
 sl2      |        6 | black    |    100 | cm      |       100
 sl7      |        6 | brown    |     60 | cm      |        60
 sl4      |        8 | black    |     40 | inch    |     101.6
 sl3      |       10 | black    |     35 | inch    |      88.9
 sl8      |       21 | brown    |     40 | inch    |     101.6
 sl5      |        4 | brown    |      1 | m       |       100
 sl6      |       20 | brown    |    0.9 | m       |        90
(8 rows)

SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who| log_when
---------+----------+--------+----------------------------------
 sl7     |        6 | Al     | Tue Oct 20 19:14:45 1998 MET DST
 sl3     |       10 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl6     |       20 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl8     |       21 | Al     | Tue Oct 20 19:25:16 1998 MET DST
(4 rows)
```

It's a long way from the one `INSERT ... SELECT` to these results. And the description of the query-tree transformation will be the last in this chapter. First, there is the parser's output

```
INSERT INTO shoelace_ok
SELECT shoelace_arrive.arr_name, shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok;
```

Now the first rule `shoelace_ok_ins` is applied and turns this into

```
UPDATE shoelace
   SET sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
```

```
       shoelace shoelace
 WHERE shoelace.sl_name = shoelace_arrive.arr_name;
```

and throws away the original INSERT on shoelace_ok. This rewritten query is passed to the rule system again, and the second applied rule shoelace_upd produces

```
UPDATE shoelace_data
   SET sl_name = shoelace.sl_name,
       sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant,
       sl_color = shoelace.sl_color,
       sl_len = shoelace.sl_len,
       sl_unit = shoelace.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
       shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data shoelace_data
 WHERE shoelace.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = shoelace.sl_name;
```

Again it's an INSTEAD rule and the previous query tree is trashed. Note that this query still uses the view shoelace. But the rule system isn't finished with this step, so it continues and applies the _RETURN rule on it, and we get

```
UPDATE shoelace_data
   SET sl_name = s.sl_name,
       sl_avail = s.sl_avail + shoelace_arrive.arr_quant,
       sl_color = s.sl_color,
       sl_len = s.sl_len,
       sl_unit = s.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
       shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data shoelace_data,
       shoelace *OLD*, shoelace *NEW*,
       shoelace_data s, unit u
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name;
```

Finally, the rule log_shoelace gets applied, producing the extra query tree

```
INSERT INTO shoelace_log
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
       current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
       shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data shoelace_data,
       shoelace *OLD*, shoelace *NEW*,
       shoelace_data s, unit u,
       shoelace_data *OLD*, shoelace_data *NEW*
       shoelace_log shoelace_log
```

```
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND (s.sl_avail + shoelace_arrive.arr_quant) <> s.sl_avail;
```

After that the rule system runs out of rules and returns the generated query trees.

So we end up with two final query trees that are equivalent to the SQL statements

```
INSERT INTO shoelace_log
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
       current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND s.sl_avail + shoelace_arrive.arr_quant <> s.sl_avail;

UPDATE shoelace_data
   SET sl_avail = shoelace_data.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive,
       shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.sl_name
   AND shoelace_data.sl_name = s.sl_name;
```

The result is that data coming from one relation inserted into another, changed into updates on a third, changed into updating a fourth plus logging that final update in a fifth gets reduced into two queries.

There is a little detail that's a bit ugly. Looking at the two queries, it turns out that the `shoelace_data` relation appears twice in the range table where it could definitely be reduced to one. The planner does not handle it and so the execution plan for the rule systems output of the `INSERT` will be

```
Nested Loop
  ->  Merge Join
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on s
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on shoelace_arrive
  ->  Seq Scan on shoelace_data
```

while omitting the extra range table entry would result in a

```
Merge Join
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on s
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on shoelace_arrive
```

which produces exactly the same entries in the log table. Thus, the rule system caused one extra scan on the table `shoelace_data` that is absolutely not necessary. And the same redundant scan is done once more in the `UPDATE`. But it was a really hard job to make that all possible at all.

Now we make a final demonstration of the PostgreSQL rule system and its power. Say you add some shoelaces with extraordinary colors to your database:

```
INSERT INTO shoelace VALUES ('sl9', 0, 'pink', 35.0, 'inch', 0.0);
INSERT INTO shoelace VALUES ('sl10', 1000, 'magenta', 40.0, 'inch', 0.0);
```

We would like to make a view to check which `shoelace` entries do not fit any shoe in color. The view for this is

```
CREATE VIEW shoelace_mismatch AS
    SELECT * FROM shoelace WHERE NOT EXISTS
        (SELECT shoename FROM shoe WHERE slcolor = sl_color);
```

Its output is

```
SELECT * FROM shoelace_mismatch;

 sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
---------+----------+----------+--------+---------+-----------
 sl9     |        0 | pink     |     35 | inch    |      88.9
 sl10    |     1000 | magenta  |     40 | inch    |     101.6
```

Now we want to set it up so that mismatching shoelaces that are not in stock are deleted from the database. To make it a little harder for PostgreSQL, we don't delete it directly. Instead we create one more view

```
CREATE VIEW shoelace_can_delete AS
    SELECT * FROM shoelace_mismatch WHERE sl_avail = 0;
```

and do it this way:

```
DELETE FROM shoelace WHERE EXISTS
    (SELECT * FROM shoelace_can_delete
             WHERE sl_name = shoelace.sl_name);
```

Voilà:

```
SELECT * FROM shoelace;

 sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
---------+----------+----------+--------+---------+-----------
 sl1     |        5 | black    |     80 | cm      |         80
 sl2     |        6 | black    |    100 | cm      |        100
 sl7     |        6 | brown    |     60 | cm      |         60
 sl4     |        8 | black    |     40 | inch    |      101.6
 sl3     |       10 | black    |     35 | inch    |       88.9
 sl8     |       21 | brown    |     40 | inch    |      101.6
 sl10    |     1000 | magenta  |     40 | inch    |      101.6
 sl5     |        4 | brown    |      1 | m       |        100
```

```
 sl6     |       20 | brown    |    0.9 | m       |        90
(9 rows)
```

A DELETE on a view, with a subquery qualification that in total uses 4 nesting/joined views, where one of them itself has a subquery qualification containing a view and where calculated view columns are used, gets rewritten into one single query tree that deletes the requested data from a real table.

There are probably only a few situations out in the real world where such a construct is necessary. But it makes you feel comfortable that it works.

35.4. Rules and Privileges

Due to rewriting of queries by the PostgreSQL rule system, other tables/views than those used in the original query get accessed. When update rules are used, this can include write access to tables.

Rewrite rules don't have a separate owner. The owner of a relation (table or view) is automatically the owner of the rewrite rules that are defined for it. The PostgreSQL rule system changes the behavior of the default access control system. Relations that are used due to rules get checked against the privileges of the rule owner, not the user invoking the rule. This means that a user only needs the required privileges for the tables/views that he names explicitly in his queries.

For example: A user has a list of phone numbers where some of them are private, the others are of interest for the secretary of the office. He can construct the following:

```
CREATE TABLE phone_data (person text, phone text, private boolean);
CREATE VIEW phone_number AS
    SELECT person, phone FROM phone_data WHERE NOT private;
GRANT SELECT ON phone_number TO secretary;
```

Nobody except him (and the database superusers) can access the phone_data table. But because of the GRANT, the secretary can run a SELECT on the phone_number view. The rule system will rewrite the SELECT from phone_number into a SELECT from phone_data and add the qualification that only entries where private is false are wanted. Since the user is the owner of phone_number and therefore the owner of the rule, the read access to phone_data is now checked against his privileges and the query is permitted. The check for accessing phone_number is also performed, but this is done against the invoking user, so nobody but the user and the secretary can use it.

The privileges are checked rule by rule. So the secretary is for now the only one who can see the public phone numbers. But the secretary can setup another view and grant access to that to the public. Then, anyone can see the phone_number data through the secretary's view. What the secretary cannot do is to create a view that directly accesses phone_data. (Actually he can, but it will not work since every access will be denied during the permission checks.) And as soon as the user will notice, that the secretary opened his phone_number view, he can revoke his access. Immediately, any access to the secretary's view would fail.

One might think that this rule-by-rule checking is a security hole, but in fact it isn't. But if it did not work this way, the secretary could set up a table with the same columns as phone_number and copy the data to there once per day. Then it's his own data and he can grant access to everyone he wants. A GRANT

command means, "I trust you". If someone you trust does the thing above, it's time to think it over and then use REVOKE.

This mechanism also works for update rules. In the examples of the previous section, the owner of the tables in the example database could grant the privileges SELECT, INSERT, UPDATE, and DELETE on the shoelace view to someone else, but only SELECT on shoelace_log. The rule action to write log entries will still be executed successfully, and that other user could see the log entries. But he cannot create fake entries, nor could he manipulate or remove existing ones.

35.5. Rules and Command Status

The PostgreSQL server returns a command status string, such as INSERT 149592 1, for each command it receives. This is simple enough when there are no rules involved, but what happens when the query is rewritten by rules?

Rules affect the command status as follows:

- If there is no unconditional INSTEAD rule for the query, then the originally given query will be executed, and its command status will be returned as usual. (But note that if there were any conditional INSTEAD rules, the negation of their qualifications will have been added to the original query. This may reduce the number of rows it processes, and if so the reported status will be affected.)
- If there is any unconditional INSTEAD rule for the query, then the original query will not be executed at all. In this case, the server will return the command status for the last query that was inserted by an INSTEAD rule (conditional or unconditional) and is of the same command type (INSERT, UPDATE, or DELETE) as the original query. If no query meeting those requirements is added by any rule, then the returned command status shows the original query type and zeroes for the row-count and OID fields.

(This system was established in PostgreSQL 7.3. In versions before that, the command status might show different results when rules exist.)

The programmer can ensure that any desired INSTEAD rule is the one that sets the command status in the second case, by giving it the alphabetically last rule name among the active rules, so that it gets applied last.

35.6. Rules versus Triggers

Many things that can be done using triggers can also be implemented using the PostgreSQL rule system. One of the things that cannot be implemented by rules are some kinds of constraints, especially foreign keys. It is possible to place a qualified rule that rewrites a command to NOTHING if the value of a column does not appear in another table. But then the data is silently thrown away and that's not a good idea. If checks for valid values are required, and in the case of an invalid value an error message should be generated, it must be done by a trigger.

On the other hand, a trigger that is fired on INSERT on a view can do the same as a rule: put the data somewhere else and suppress the insert in the view. But it cannot do the same thing on UPDATE or DELETE, because there is no real data in the view relation that could be scanned, and thus the trigger would never get called. Only a rule will help.

For the things that can be implemented by both, which is best depends on the usage of the database. A trigger is fired for any affected row once. A rule manipulates the query or generates an additional query. So if many rows are affected in one statement, a rule issuing one extra command is likely to be faster than a trigger that is called for every single row and must execute its operations many times. However, the trigger approach is conceptually far simpler than the rule approach, and is easier for novices to get right.

Here we show an example of how the choice of rules versus triggers plays out in one situation. There are two tables:

```
CREATE TABLE computer (
    hostname        text,    -- indexed
    manufacturer    text     -- indexed
);

CREATE TABLE software (
    software        text,    -- indexed
    hostname        text     -- indexed
);
```

Both tables have many thousands of rows and the indexes on `hostname` are unique. The rule or trigger should implement a constraint that deletes rows from `software` that reference a deleted computer. The trigger would use this command:

```
DELETE FROM software WHERE hostname = $1;
```

Since the trigger is called for each individual row deleted from `computer`, it can prepare and save the plan for this command and pass the `hostname` value in the parameter. The rule would be written as

```
CREATE RULE computer_del AS ON DELETE TO computer
    DO DELETE FROM software WHERE hostname = OLD.hostname;
```

Now we look at different types of deletes. In the case of a

```
DELETE FROM computer WHERE hostname = 'mypc.local.net';
```

the table `computer` is scanned by index (fast), and the command issued by the trigger would also use an index scan (also fast). The extra command from the rule would be

```
DELETE FROM software WHERE computer.hostname = 'mypc.local.net'
                       AND software.hostname = computer.hostname;
```

Since there are appropriate indexes setup, the planner will create a plan of

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

So there would be not that much difference in speed between the trigger and the rule implementation.

With the next delete we want to get rid of all the 2000 computers where the `hostname` starts with `old`. There are two possible commands to do that. One is

```
DELETE FROM computer WHERE hostname >= 'old'
                       AND hostname <  'ole'
```

The command added by the rule will be

```
DELETE FROM software WHERE computer.hostname >= 'old' AND computer.hostname < 'ole'
                       AND software.hostname = computer.hostname;
```

with the plan

```
Hash Join
  ->  Seq Scan on software
  ->  Hash
    ->  Index Scan using comp_hostidx on computer
```

The other possible command is

```
DELETE FROM computer WHERE hostname ~ '^old';
```

which results in the following executing plan for the command added by the rule:

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

This shows, that the planner does not realize that the qualification for `hostname` in `computer` could also be used for an index scan on `software` when there are multiple qualification expressions combined with `AND`, which is what it does in the regular-expression version of the command. The trigger will get invoked once for each of the 2000 old computers that have to be deleted, and that will result in one index scan over `computer` and 2000 index scans over `software`. The rule implementation will do it with two commands that use indexes. And it depends on the overall size of the table `software` whether the rule will still be faster in the sequential scan situation. 2000 command executions from the trigger over the SPI manager take some time, even if all the index blocks will soon be in the cache.

The last command we look at is

```
DELETE FROM computer WHERE manufacturer = 'bim';
```

Again this could result in many rows to be deleted from `computer`. So the trigger will again run many commands through the executor. The command generated by the rule will be

```
DELETE FROM software WHERE computer.manufacturer = 'bim'
                       AND software.hostname = computer.hostname;
```

The plan for that command will again be the nested loop over two index scans, only using a different index on `computer`:

```
Nestloop
  ->  Index Scan using comp_manufidx on computer
  ->  Index Scan using soft_hostidx on software
```

In any of these cases, the extra commands from the rule system will be more or less independent from the number of affected rows in a command.

The summary is, rules will only be significantly slower than triggers if their actions result in large and badly qualified joins, a situation where the planner fails.

Chapter 36. Procedural Languages

PostgreSQL allows user-defined functions to be written in other languages besides SQL and C. These other languages are generically called *procedural languages* (PLs). For a function written in a procedural language, the database server has no built-in knowledge about how to interpret the function's source text. Instead, the task is passed to a special handler that knows the details of the language. The handler could either do all the work of parsing, syntax analysis, execution, etc. itself, or it could serve as "glue" between PostgreSQL and an existing implementation of a programming language. The handler itself is a C language function compiled into a shared object and loaded on demand, just like any other C function.

There are currently four procedural languages available in the standard PostgreSQL distribution: PL/pgSQL (Chapter 37), PL/Tcl (Chapter 38), PL/Perl (Chapter 39), and PL/Python (Chapter 40). There are additional procedural languages available that are not included in the core distribution. Appendix H has information about finding them. In addition other languages can be defined by users; the basics of developing a new procedural language are covered in Chapter 47.

36.1. Installing Procedural Languages

A procedural language must be "installed" into each database where it is to be used. But procedural languages installed in the database `template1` are automatically available in all subsequently created databases, since their entries in `template1` will be copied by `CREATE DATABASE`. So the database administrator can decide which languages are available in which databases and can make some languages available by default if he chooses.

For the languages supplied with the standard distribution, it is only necessary to execute `CREATE LANGUAGE` *`language_name`* to install the language into the current database. Alternatively, the program createlang may be used to do this from the shell command line. For example, to install the language PL/pgSQL into the database `template1`, use

```
createlang plpgsql template1
```

The manual procedure described below is only recommended for installing custom languages that `CREATE LANGUAGE` does not know about.

Manual Procedural Language Installation

A procedural language is installed in a database in four steps, which must be carried out by a database superuser. (For languages known to `CREATE LANGUAGE`, the second and third steps can be omitted, because they will be carried out automatically if needed.)

1. The shared object for the language handler must be compiled and installed into an appropriate library directory. This works in the same way as building and installing modules with regular user-defined C functions does; see Section 33.9.6. Often, the language handler will depend on an external library that provides the actual programming language engine; if so, that must be installed as well.

2. The handler must be declared with the command

   ```
   CREATE FUNCTION handler_function_name()
       RETURNS language_handler
       AS 'path-to-shared-object'
   ```

```
    LANGUAGE C;
```

The special return type of `language_handler` tells the database system that this function does not return one of the defined SQL data types and is not directly usable in SQL statements.

3. Optionally, the language handler may provide a "validator" function that checks a function definition for correctness without actually executing it. The validator function is called by `CREATE FUNCTION` if it exists. If a validator function is provided by the handler, declare it with a command like

```
CREATE FUNCTION validator_function_name(oid)
    RETURNS void
    AS 'path-to-shared-object'
    LANGUAGE C;
```

4. The PL must be declared with the command

```
CREATE [TRUSTED] [PROCEDURAL] LANGUAGE language-name
    HANDLER handler_function_name
    [VALIDATOR validator_function_name] ;
```

The optional key word `TRUSTED` specifies that ordinary database users that have no superuser privileges should be allowed to use this language to create functions and trigger procedures. Since PL functions are executed inside the database server, the `TRUSTED` flag should only be given for languages that do not allow access to database server internals or the file system. The languages PL/pgSQL, PL/Tcl, and PL/Perl are considered trusted; the languages PL/TclU, PL/PerlU, and PL/PythonU are designed to provide unlimited functionality and should *not* be marked trusted.

Example 36-1 shows how the manual installation procedure would work with the language PL/pgSQL.

Example 36-1. Manual Installation of PL/pgSQL

The following command tells the database server where to find the shared object for the PL/pgSQL language's call handler function.

```
CREATE FUNCTION plpgsql_call_handler() RETURNS language_handler AS
    '$libdir/plpgsql' LANGUAGE C;
```

PL/pgSQL has a validator function, so we declare that too:

```
CREATE FUNCTION plpgsql_validator(oid) RETURNS void AS
    '$libdir/plpgsql' LANGUAGE C;
```

The command

```
CREATE TRUSTED PROCEDURAL LANGUAGE plpgsql
    HANDLER plpgsql_call_handler
    VALIDATOR plpgsql_validator;
```

then defines that the previously declared functions should be invoked for functions and trigger procedures where the language attribute is `plpgsql`.

In a default PostgreSQL installation, the handler for the PL/pgSQL language is built and installed into the "library" directory. If Tcl support is configured in, the handlers for PL/Tcl and PL/TclU are also built and installed in the same location. Likewise, the PL/Perl and PL/PerlU handlers are built and installed if Perl support is configured, and the PL/PythonU handler is installed if Python support is configured.

Chapter 37. PL/pgSQL - SQL Procedural Language

PL/pgSQL is a loadable procedural language for the PostgreSQL database system. The design goals of PL/pgSQL were to create a loadable procedural language that

- can be used to create functions and trigger procedures,
- adds control structures to the SQL language,
- can perform complex computations,
- inherits all user-defined types, functions, and operators,
- can be defined to be trusted by the server,
- is easy to use.

Except for input/output conversion and calculation functions for user-defined types, anything that can be defined in C language functions can also be done with PL/pgSQL. For example, it is possible to create complex conditional computation functions and later use them to define operators or use them in index expressions.

37.1. Overview

The PL/pgSQL call handler parses the function's source text and produces an internal binary instruction tree the first time the function is called (within each session). The instruction tree fully translates the PL/pgSQL statement structure, but individual SQL expressions and SQL commands used in the function are not translated immediately.

As each expression and SQL command is first used in the function, the PL/pgSQL interpreter creates a prepared execution plan (using the SPI manager's `SPI_prepare` and `SPI_saveplan` functions). Subsequent visits to that expression or command reuse the prepared plan. Thus, a function with conditional code that contains many statements for which execution plans might be required will only prepare and save those plans that are really used during the lifetime of the database connection. This can substantially reduce the total amount of time required to parse and generate execution plans for the statements in a PL/pgSQL function. A disadvantage is that errors in a specific expression or command may not be detected until that part of the function is reached in execution.

Once PL/pgSQL has made an execution plan for a particular command in a function, it will reuse that plan for the life of the database connection. This is usually a win for performance, but it can cause some problems if you dynamically alter your database schema. For example:

```
CREATE FUNCTION populate() RETURNS integer AS $$
DECLARE
    -- declarations
BEGIN
    PERFORM my_function();
```

```
END;
$$ LANGUAGE plpgsql;
```

If you execute the above function, it will reference the OID for `my_function()` in the execution plan produced for the `PERFORM` statement. Later, if you drop and recreate `my_function()`, then `populate()` will not be able to find `my_function()` anymore. You would then have to recreate `populate()`, or at least start a new database session so that it will be compiled afresh. Another way to avoid this problem is to use `CREATE OR REPLACE FUNCTION` when updating the definition of `my_function` (when a function is "replaced", its OID is not changed).

Because PL/pgSQL saves execution plans in this way, SQL commands that appear directly in a PL/pgSQL function must refer to the same tables and columns on every execution; that is, you cannot use a parameter as the name of a table or column in an SQL command. To get around this restriction, you can construct dynamic commands using the PL/pgSQL `EXECUTE` statement — at the price of constructing a new execution plan on every execution.

> **Note:** The PL/pgSQL `EXECUTE` statement is not related to the *EXECUTE* SQL statement supported by the PostgreSQL server. The server's `EXECUTE` statement cannot be used within PL/pgSQL functions (and is not needed).

37.1.1. Advantages of Using PL/pgSQL

SQL is the language PostgreSQL and most other relational databases use as query language. It's portable and easy to learn. But every SQL statement must be executed individually by the database server.

That means that your client application must send each query to the database server, wait for it to be processed, receive and process the results, do some computation, then send further queries to the server. All this incurs interprocess communication and will also incur network overhead if your client is on a different machine than the database server.

With PL/pgSQL you can group a block of computation and a series of queries *inside* the database server, thus having the power of a procedural language and the ease of use of SQL, but with considerable savings because you don't have the whole client/server communication overhead.

- Elimination of additional round trips between client and server
- Intermediate results that the client does not need do not need to be marshaled or transferred between server and client
- There is no need for additional rounds of query parsing

This can allow for a considerable performance increase as compared to an application that does not use stored functions.

Also, with PL/pgSQL you can use all the data types, operators and functions of SQL.

37.1.2. Supported Argument and Result Data Types

Functions written in PL/pgSQL can accept as arguments any scalar or array data type supported by the server, and they can return a result of any of these types. They can also accept or return any composite type (row type) specified by name. It is also possible to declare a PL/pgSQL function as returning `record`, which means that the result is a row type whose columns are determined by specification in the calling query, as discussed in Section 7.2.1.4.

PL/pgSQL functions may also be declared to accept and return the polymorphic types `anyelement` and `anyarray`. The actual data types handled by a polymorphic function can vary from call to call, as discussed in Section 33.2.5. An example is shown in Section 37.4.1.

PL/pgSQL functions can also be declared to return a "set", or table, of any data type they can return a single instance of. Such a function generates its output by executing `RETURN NEXT` for each desired element of the result set.

Finally, a PL/pgSQL function may be declared to return `void` if it has no useful return value.

PL/pgSQL functions can also be declared with output parameters in place of an explicit specification of the return type. This does not add any fundamental capability to the language, but it is often convenient, especially for returning multiple values.

Specific examples appear in Section 37.4.1 and Section 37.7.1.

37.2. Tips for Developing in PL/pgSQL

One good way to develop in PL/pgSQL is to use the text editor of your choice to create your functions, and in another window, use psql to load and test those functions. If you are doing it this way, it is a good idea to write the function using `CREATE OR REPLACE FUNCTION`. That way you can just reload the file to update the function definition. For example:

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $$
          ....
$$ LANGUAGE plpgsql;
```

While running psql, you can load or reload such a function definition file with

```
\i filename.sql
```

and then immediately issue SQL commands to test the function.

Another good way to develop in PL/pgSQL is with a GUI database access tool that facilitates development in a procedural language. One example of such as a tool is PgAccess, although others exist. These tools often provide convenient features such as escaping single quotes and making it easier to recreate and debug functions.

37.2.1. Handling of Quotation Marks

The code of a PL/pgSQL function is specified in `CREATE FUNCTION` as a string literal. If you write the string literal in the ordinary way with surrounding single quotes, then any single quotes inside the function body must be doubled; likewise any backslashes must be doubled (assuming escape string syntax is used). Doubling quotes is at best tedious, and in more complicated cases the code can become downright incomprehensible, because you can easily find yourself needing half a dozen or more adjacent quote marks. It's recommended that you instead write the function body as a "dollar-quoted" string literal (see Section 4.1.2.2). In the dollar-quoting approach, you never double any quote marks, but instead take care to choose a different dollar-quoting delimiter for each level of nesting you need. For example, you might write the `CREATE FUNCTION` command as

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $PROC$
        ....
$PROC$ LANGUAGE plpgsql;
```

Within this, you might use quote marks for simple literal strings in SQL commands and `$$` to delimit fragments of SQL commands that you are assembling as strings. If you need to quote text that includes `$$`, you could use `$Q$`, and so on.

The following chart shows what you have to do when writing quote marks without dollar quoting. It may be useful when translating pre-dollar quoting code into something more comprehensible.

1 quotation mark

To begin and end the function body, for example:

```
CREATE FUNCTION foo() RETURNS integer AS '
          ....
' LANGUAGE plpgsql;
```

Anywhere within a single-quoted function body, quote marks *must* appear in pairs.

2 quotation marks

For string literals inside the function body, for example:

```
a_output := ”Blah”;
SELECT * FROM users WHERE f_name=”foobar”;
```

In the dollar-quoting approach, you'd just write

```
a_output := 'Blah';
SELECT * FROM users WHERE f_name='foobar';
```

which is exactly what the PL/pgSQL parser would see in either case.

4 quotation marks

When you need a single quotation mark in a string constant inside the function body, for example:

```
a_output := a_output || ” AND name LIKE ””foobar”” AND xyz”
```

The value actually appended to `a_output` would be: `AND name LIKE 'foobar' AND xyz`.

In the dollar-quoting approach, you'd write

```
a_output := a_output || $$ AND name LIKE 'foobar' AND xyz$$
```

being careful that any dollar-quote delimiters around this are not just `$$`.

6 quotation marks

When a single quotation mark in a string inside the function body is adjacent to the end of that string constant, for example:

```
a_output := a_output || '' AND name LIKE ''foobar''''
```

The value appended to `a_output` would then be: `AND name LIKE 'foobar'`.

In the dollar-quoting approach, this becomes

```
a_output := a_output || $$ AND name LIKE 'foobar'$$
```

10 quotation marks

When you want two single quotation marks in a string constant (which accounts for 8 quotation marks) and this is adjacent to the end of that string constant (2 more). You will probably only need that if you are writing a function that generates other functions, as in Example 37-6. For example:

```
a_output := a_output || '' if v_'' ||
    referrer_keys.kind || '' like ''''''
    || referrer_keys.key_string || ''''''
    then return ''''  || referrer_keys.referrer_type
    || ''''; end if;'';
```

The value of `a_output` would then be:

```
if v_... like ''...'' then return ''...''; end if;
```

In the dollar-quoting approach, this becomes

```
a_output := a_output || $$ if v_$$ || referrer_keys.kind || $$ like '$$
    || referrer_keys.key_string || $$'
    then return '$$  || referrer_keys.referrer_type
    || $$'; end if;$$;
```

where we assume we only need to put single quote marks into `a_output`, because it will be re-quoted before use.

37.3. Structure of PL/pgSQL

PL/pgSQL is a block-structured language. The complete text of a function definition must be a *block*. A block is defined as:

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END [ label ];
```

Each declaration and each statement within a block is terminated by a semicolon. A block that appears within another block must have a semicolon after `END`, as shown above; however the final `END` that concludes a function body does not require a semicolon.

All key words and identifiers can be written in mixed upper and lower case. Identifiers are implicitly converted to lowercase unless double-quoted.

There are two types of comments in PL/pgSQL. A double dash (--) starts a comment that extends to the end of the line. A /* starts a block comment that extends to the next occurrence of */. Block comments cannot be nested, but double dash comments can be enclosed into a block comment and a double dash can hide the block comment delimiters /* and */.

Any statement in the statement section of a block can be a *subblock*. Subblocks can be used for logical grouping or to localize variables to a small group of statements.

The variables declared in the declarations section preceding a block are initialized to their default values every time the block is entered, not only once per function call. For example:

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Quantity here is %', quantity;  -- Quantity here is 30
    quantity := 50;
    --
    -- Create a subblock
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Quantity here is %', quantity;  -- Quantity here is 80
    END;

    RAISE NOTICE 'Quantity here is %', quantity;  -- Quantity here is 50

    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

It is important not to confuse the use of BEGIN/END for grouping statements in PL/pgSQL with the database commands for transaction control. PL/pgSQL's BEGIN/END are only for grouping; they do not start or end a transaction. Functions and trigger procedures are always executed within a transaction established by an outer query — they cannot start or commit that transaction, since there would be no context for them to execute in. However, a block containing an EXCEPTION clause effectively forms a subtransaction that can be rolled back without affecting the outer transaction. For more about that see Section 37.7.5.

37.4. Declarations

All variables used in a block must be declared in the declarations section of the block. (The only exception is that the loop variable of a FOR loop iterating over a range of integer values is automatically declared as an integer variable.)

PL/pgSQL variables can have any SQL data type, such as integer, varchar, and char.

Here are some examples of variable declarations:

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

The general syntax of a variable declaration is:

```
name [ CONSTANT ] type [ NOT NULL ] [ { DEFAULT | := } expression ];
```

The DEFAULT clause, if given, specifies the initial value assigned to the variable when the block is entered. If the DEFAULT clause is not given then the variable is initialized to the SQL null value. The CONSTANT option prevents the variable from being assigned to, so that its value remains constant for the duration of the block. If NOT NULL is specified, an assignment of a null value results in a run-time error. All variables declared as NOT NULL must have a nonnull default value specified.

The default value is evaluated every time the block is entered. So, for example, assigning now() to a variable of type timestamp causes the variable to have the time of the current function call, not the time when the function was precompiled.

Examples:

```
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;
```

37.4.1. Aliases for Function Parameters

Parameters passed to functions are named with the identifiers $1, $2, etc. Optionally, aliases can be declared for $*n* parameter names for increased readability. Either the alias or the numeric identifier can then be used to refer to the parameter value.

There are two ways to create an alias. The preferred way is to give a name to the parameter in the CREATE FUNCTION command, for example:

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

The other way, which was the only way available before PostgreSQL 8.0, is to explicitly declare an alias, using the declaration syntax

```
name ALIAS FOR $n;
```

The same example in this style looks like

```
CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Some more examples:

```
CREATE FUNCTION instr(varchar, integer) RETURNS integer AS $$
DECLARE
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN
    -- some computations using v_string and index here
END;
$$ LANGUAGE plpgsql;


CREATE FUNCTION concat_selected_fields(in_t sometablename) RETURNS text AS $$
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;
```

When a PL/pgSQL function is declared with output parameters, the output parameters are given `$n` names and optional aliases in just the same way as the normal input parameters. An output parameter is effectively a variable that starts out NULL; it should be assigned to during the execution of the function. The final value of the parameter is what is returned. For instance, the sales-tax example could also be done this way:

```
CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
    tax := subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Notice that we omitted `RETURNS real` — we could have included it, but it would be redundant.

Output parameters are most useful when returning multiple values. A trivial example is:

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int) AS $$
BEGIN
    sum := x + y;
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

As discussed in Section 33.4.3, this effectively creates an anonymous record type for the function's results. If a `RETURNS` clause is given, it must say `RETURNS record`.

When the return type of a PL/pgSQL function is declared as a polymorphic type (`anyelement` or `anyarray`), a special parameter `$0` is created. Its data type is the actual return type of the function, as deduced from the actual input types (see Section 33.2.5). This allows the function to access its actual return type as shown in Section 37.4.2. `$0` is initialized to null and can be modified by the function, so it can be used to hold the return value if desired, though that is not required. `$0` can also be given an alias. For example, this function works on any data type that has a + operator:

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement)
RETURNS anyelement AS $$
DECLARE
    result ALIAS FOR $0;
BEGIN
    result := v1 + v2 + v3;
    RETURN result;
END;
$$ LANGUAGE plpgsql;
```

The same effect can be had by declaring one or more output parameters as `anyelement` or `anyarray`. In this case the special `$0` parameter is not used; the output parameters themselves serve the same purpose. For example:

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement,
                                 OUT sum anyelement)
AS $$
BEGIN
    sum := v1 + v2 + v3;
END;
$$ LANGUAGE plpgsql;
```

37.4.2. Copying Types

variable`%TYPE`

`%TYPE` provides the data type of a variable or table column. You can use this to declare variables that will hold database values. For example, let's say you have a column named `user_id` in your `users` table. To declare a variable with the same data type as `users.user_id` you write:

```
user_id users.user_id%TYPE;
```

By using `%TYPE` you don't need to know the data type of the structure you are referencing, and most importantly, if the data type of the referenced item changes in the future (for instance: you change the type of `user_id` from `integer` to `real`), you may not need to change your function definition.

`%TYPE` is particularly valuable in polymorphic functions, since the data types needed for internal variables may change from one call to the next. Appropriate variables can be created by applying `%TYPE` to the function's arguments or result placeholders.

37.4.3. Row Types

```
name table_name%ROWTYPE;
name composite_type_name;
```

A variable of a composite type is called a *row* variable (or *row-type* variable). Such a variable can hold a whole row of a `SELECT` or `FOR` query result, so long as that query's column set matches the declared type of the variable. The individual fields of the row value are accessed using the usual dot notation, for example `rowvar.field`.

A row variable can be declared to have the same type as the rows of an existing table or view, by using the *table_name*`%ROWTYPE` notation; or it can be declared by giving a composite type's name. (Since every table has an associated composite type of the same name, it actually does not matter in PostgreSQL whether you write `%ROWTYPE` or not. But the form with `%ROWTYPE` is more portable.)

Parameters to a function can be composite types (complete table rows). In that case, the corresponding identifier $n will be a row variable, and fields can be selected from it, for example `$1.user_id`.

Only the user-defined columns of a table row are accessible in a row-type variable, not the OID or other system columns (because the row could be from a view). The fields of the row type inherit the table's field size or precision for data types such as `char(`*n*`)`.

Here is an example of using composite types. `table1` and `table2` are existing tables having at least the mentioned fields:

```
CREATE FUNCTION merge_fields(t_row table1) RETURNS text AS $$
DECLARE
    t2_row table2%ROWTYPE;
BEGIN
    SELECT * INTO t2_row FROM table2 WHERE ... ;
    RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE plpgsql;

SELECT merge_fields(t.*) FROM table1 t WHERE ... ;
```

37.4.4. Record Types

```
name RECORD;
```

Record variables are similar to row-type variables, but they have no predefined structure. They take on the actual row structure of the row they are assigned during a `SELECT` or `FOR` command. The substructure of a record variable can change each time it is assigned to. A consequence of this is that until a record variable is first assigned to, it has no substructure, and any attempt to access a field in it will draw a run-time error.

Note that `RECORD` is not a true data type, only a placeholder. One should also realize that when a PL/pgSQL function is declared to return type `record`, this is not quite the same concept as a record variable, even though such a function may well use a record variable to hold its result. In both cases the actual row structure is unknown when the function is written, but for a function returning `record` the

actual structure is determined when the calling query is parsed, whereas a record variable can change its row structure on-the-fly.

37.4.5. RENAME

```
RENAME oldname TO newname;
```

Using the RENAME declaration you can change the name of a variable, record or row. This is primarily useful if NEW or OLD should be referenced by another name inside a trigger procedure. See also ALIAS.

Examples:

```
RENAME id TO user_id;
RENAME this_var TO that_var;
```

> **Note:** RENAME appears to be broken as of PostgreSQL 7.3. Fixing this is of low priority, since ALIAS covers most of the practical uses of RENAME.

37.5. Expressions

All expressions used in PL/pgSQL statements are processed using the server's regular SQL executor. In effect, a query like

```
SELECT expression
```

is executed using the SPI manager. Before evaluation, occurrences of PL/pgSQL variable identifiers are replaced by parameters, and the actual values from the variables are passed to the executor in the parameter array. This allows the query plan for the SELECT to be prepared just once and then reused for subsequent evaluations.

The evaluation done by the PostgreSQL main parser has some side effects on the interpretation of constant values. In detail there is a difference between what these two functions do:

```
CREATE FUNCTION logfunc1(logtxt text) RETURNS timestamp AS $$
    BEGIN
        INSERT INTO logtable VALUES (logtxt, 'now');
        RETURN 'now';
    END;
$$ LANGUAGE plpgsql;
```

and

```
CREATE FUNCTION logfunc2(logtxt text) RETURNS timestamp AS $$
    DECLARE
        curtime timestamp;
```

```
    BEGIN
        curtime := 'now';
        INSERT INTO logtable VALUES (logtxt, curtime);
        RETURN curtime;
    END;
$$ LANGUAGE plpgsql;
```

In the case of `logfunc1`, the PostgreSQL main parser knows when preparing the plan for the `INSERT` that the string `'now'` should be interpreted as `timestamp` because the target column of `logtable` is of that type. Thus, `'now'` will be converted to a constant when the `INSERT` is planned, and then used in all invocations of `logfunc1` during the lifetime of the session. Needless to say, this isn't what the programmer wanted.

In the case of `logfunc2`, the PostgreSQL main parser does not know what type `'now'` should become and therefore it returns a data value of type `text` containing the string `now`. During the ensuing assignment to the local variable `curtime`, the PL/pgSQL interpreter casts this string to the `timestamp` type by calling the `text_out` and `timestamp_in` functions for the conversion. So, the computed time stamp is updated on each execution as the programmer expects.

The mutable nature of record variables presents a problem in this connection. When fields of a record variable are used in expressions or statements, the data types of the fields must not change between calls of one and the same expression, since the expression will be planned using the data type that is present when the expression is first reached. Keep this in mind when writing trigger procedures that handle events for more than one table. (`EXECUTE` can be used to get around this problem when necessary.)

37.6. Basic Statements

In this section and the following ones, we describe all the statement types that are explicitly understood by PL/pgSQL. Anything not recognized as one of these statement types is presumed to be an SQL command and is sent to the main database engine to execute, as described in Section 37.6.2 and Section 37.6.3.

37.6.1. Assignment

An assignment of a value to a PL/pgSQL variable or row/record field is written as:

```
identifier := expression;
```

As explained above, the expression in such a statement is evaluated by means of an SQL `SELECT` command sent to the main database engine. The expression must yield a single value.

If the expression's result data type doesn't match the variable's data type, or the variable has a specific size/precision (like `char(20)`), the result value will be implicitly converted by the PL/pgSQL interpreter using the result type's output-function and the variable type's input-function. Note that this could potentially result in run-time errors generated by the input function, if the string form of the result value is not acceptable to the input function.

Examples:

```
user_id := 20;
tax := subtotal * 0.06;
```

37.6.2. Executing a Query With No Result

For any SQL query that does not return rows, for example `INSERT` without a `RETURNING` clause, you can execute the query within a PL/pgSQL function just by writing the query.

Any PL/pgSQL variable name appearing in the query text is replaced by a parameter symbol, and then the current value of the variable is provided as the parameter value at run time. This allows the same textual query to do different things in different calls of the function.

> **Note:** This two-step process allows PL/pgSQL to plan the query just once and re-use the plan on subsequent executions. As an example, if you write
>
> ```
> DECLARE
>     key TEXT;
>     delta INTEGER;
> BEGIN
>     ...
>     UPDATE mytab SET val = val + delta WHERE id = key;
> ```
>
> the query text seen by the main SQL engine will look like
>
> ```
>     UPDATE mytab SET val = val + $1 WHERE id = $2;
> ```
>
> Although you don't normally have to think about this, it's helpful to know it when you need to make sense of syntax-error messages.

> **Caution**
>
> PL/pgSQL will substitute for any identifier matching one of the function's declared variables; it is not bright enough to know whether that's what you meant! Thus, it is a bad idea to use a variable name that is the same as any table or column name that you need to reference in queries within the function. Sometimes you can work around this by using qualified names in the query: PL/pgSQL will not substitute in a qualified name *foo*.*bar*, even if *foo* or *bar* is a declared variable name.

Sometimes it is useful to evaluate an expression or `SELECT` query but discard the result, for example when calling a function that has side-effects but no useful result value. To do this in PL/pgSQL, use the `PERFORM` statement:

```
PERFORM query;
```

This executes *query* and discards the result. Write the *query* the same way you would write an SQL `SELECT` command, but replace the initial keyword `SELECT` with `PERFORM`. PL/pgSQL variables will be substituted into the query as usual. Also, the special variable `FOUND` is set to true if the query produced at least one row, or false if it produced no rows.

Note: One might expect that writing SELECT directly would accomplish this result, but at present the only accepted way to do it is PERFORM. A SQL command that can return rows, such as SELECT, will be rejected as an error unless it has an INTO clause as discussed in the next section.

An example:

```
PERFORM create_mv('cs_session_page_requests_mv', my_query);
```

37.6.3. Executing a Query with a Single-Row Result

The result of a SQL command yielding a single row (possibly of multiple columns) can be assigned to a record variable, row-type variable, or list of scalar variables. This is done by writing the base SQL command and adding an INTO clause. For example,

```
SELECT select_expressions INTO [STRICT] target FROM ...;
INSERT ... RETURNING expressions INTO [STRICT] target;
UPDATE ... RETURNING expressions INTO [STRICT] target;
DELETE ... RETURNING expressions INTO [STRICT] target;
```

where *target* can be a record variable, a row variable, or a comma-separated list of simple variables and record/row fields. PL/pgSQL variables will be substituted into the rest of the query as usual. This works for SELECT, INSERT/UPDATE/DELETE with RETURNING, and utility commands that return row-set results (such as EXPLAIN). Except for the INTO clause, the SQL command is the same as it would be written outside PL/pgSQL.

Tip: Note that this interpretation of SELECT with INTO is quite different from PostgreSQL's regular SELECT INTO command, wherein the INTO target is a newly created table. If you want to create a table from a SELECT result inside a PL/pgSQL function, use the syntax CREATE TABLE ... AS SELECT.

If a row or a variable list is used as target, the query's result columns must exactly match the structure of the target as to number and data types, or a run-time error occurs. When a record variable is the target, it automatically configures itself to the row type of the query result columns.

The INTO clause can appear almost anywhere in the SQL command. Customarily it is written either just before or just after the list of *select_expressions* in a SELECT command, or at the end of the command for other command types. It is recommended that you follow this convention in case the PL/pgSQL parser becomes stricter in future versions.

If STRICT is not specified, then *target* will be set to the first row returned by the query, or to nulls if the query returned no rows. (Note that "the first row" is not well-defined unless you've used ORDER BY.) Any result rows after the first row are discarded. You can check the special FOUND variable (see Section 37.6.6) to determine whether a row was returned:

```
SELECT * INTO myrec FROM emp WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION 'employee % not found', myname;
```

```
END IF;
```

If the STRICT option is specified, the query must return exactly one row or a run-time error will be reported, either NO_DATA_FOUND (no rows) or TOO_MANY_ROWS (more than one row). You can use an exception block if you wish to catch the error, for example:

```
BEGIN
    SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RAISE EXCEPTION 'employee % not found', myname;
        WHEN TOO_MANY_ROWS THEN
            RAISE EXCEPTION 'employee % not unique', myname;
END;
```

Successful execution of a command with STRICT always sets FOUND to true.

For INSERT/UPDATE/DELETE with RETURNING, PL/pgSQL reports an error for more than one returned row, even when STRICT is not specified. This is because there is no option such as ORDER BY with which to determine which affected row would be returned.

> **Note:** The STRICT option matches the behavior of Oracle PL/SQL's SELECT INTO and related statements.

To handle cases where you need to process multiple result rows from a SQL query, see Section 37.7.4.

37.6.4. Doing Nothing At All

Sometimes a placeholder statement that does nothing is useful. For example, it can indicate that one arm of an if/then/else chain is deliberately empty. For this purpose, use the NULL statement:

```
NULL;
```

For example, the following two fragments of code are equivalent:

```
BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        NULL;  -- ignore the error
END;

BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN  -- ignore the error
END;
```

Which is preferable is a matter of taste.

Note: In Oracle's PL/SQL, empty statement lists are not allowed, and so `NULL` statements are *required* for situations such as this. PL/pgSQL allows you to just write nothing, instead.

37.6.5. Executing Dynamic Commands

Oftentimes you will want to generate dynamic commands inside your PL/pgSQL functions, that is, commands that will involve different tables or different data types each time they are executed. PL/pgSQL's normal attempts to cache plans for commands will not work in such scenarios. To handle this sort of problem, the `EXECUTE` statement is provided:

```
EXECUTE command-string [ INTO [STRICT] target ];
```

where *command-string* is an expression yielding a string (of type `text`) containing the command to be executed and *target* is a record variable, row variable, or a comma-separated list of simple variables and record/row fields.

Note in particular that no substitution of PL/pgSQL variables is done on the computed command string. The values of variables must be inserted in the command string as it is constructed.

Unlike all other commands in PL/pgSQL, a command run by an `EXECUTE` statement is not prepared and saved just once during the life of the session. Instead, the command is prepared each time the statement is run. The command string can be dynamically created within the function to perform actions on different tables and columns.

The `INTO` clause specifies where the results of a SQL command returning rows should be assigned. If a row or variable list is provided, it must exactly match the structure of the query's results (when a record variable is used, it will configure itself to match the result structure automatically). If multiple rows are returned, only the first will be assigned to the `INTO` variable. If no rows are returned, NULL is assigned to the `INTO` variable. If no `INTO` clause is specified, the query results are discarded.

If the `STRICT` option is given, an error is reported unless the query produces exactly one row.

`SELECT INTO` is not currently supported within `EXECUTE`.

When working with dynamic commands you will often have to handle escaping of single quotes. The recommended method for quoting fixed text in your function body is dollar quoting. (If you have legacy code that does not use dollar quoting, please refer to the overview in Section 37.2.1, which can save you some effort when translating said code to a more reasonable scheme.)

Dynamic values that are to be inserted into the constructed query require special handling since they might themselves contain quote characters. An example (this assumes that you are using dollar quoting for the function as a whole, so the quote marks need not be doubled):

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = '
        || quote_literal(newvalue)
        || ' WHERE key = '
        || quote_literal(keyvalue);
```

This example demonstrates the use of the quote_ident and quote_literal functions. For safety, expressions containing column and table identifiers should be passed to quote_ident. Expressions containing values that should be literal strings in the constructed command should be passed to quote_literal. Both take the appropriate steps to return the input text enclosed in double or single quotes respectively, with any embedded special characters properly escaped.

Note that dollar quoting is only useful for quoting fixed text. It would be a very bad idea to try to do the above example as

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = $$'
        || newvalue
        || '$$ WHERE key = '
        || quote_literal(keyvalue);
```

because it would break if the contents of newvalue happened to contain $$. The same objection would apply to any other dollar-quoting delimiter you might pick. So, to safely quote text that is not known in advance, you *must* use quote_literal.

A much larger example of a dynamic command and EXECUTE can be seen in Example 37-6, which builds and executes a CREATE FUNCTION command to define a new function.

37.6.6. Obtaining the Result Status

There are several ways to determine the effect of a command. The first method is to use the GET DIAGNOSTICS command, which has the form:

```
GET DIAGNOSTICS variable = item [ , ... ];
```

This command allows retrieval of system status indicators. Each *item* is a key word identifying a state value to be assigned to the specified variable (which should be of the right data type to receive it). The currently available status items are ROW_COUNT, the number of rows processed by the last SQL command sent down to the SQL engine, and RESULT_OID, the OID of the last row inserted by the most recent SQL command. Note that RESULT_OID is only useful after an INSERT command into a table containing OIDs.

An example:

```
GET DIAGNOSTICS integer_var = ROW_COUNT;
```

The second method to determine the effects of a command is to check the special variable named FOUND, which is of type boolean. FOUND starts out false within each PL/pgSQL function call. It is set by each of the following types of statements:

- A SELECT INTO statement sets FOUND true if a row is assigned, false if no row is returned.
- A PERFORM statement sets FOUND true if it produces (and discards) a row, false if no row is produced.
- UPDATE, INSERT, and DELETE statements set FOUND true if at least one row is affected, false if no row is affected.

- A FETCH statement sets FOUND true if it returns a row, false if no row is returned.
- A FOR statement sets FOUND true if it iterates one or more times, else false. This applies to all three variants of the FOR statement (integer FOR loops, record-set FOR loops, and dynamic record-set FOR loops). FOUND is set this way when the FOR loop exits; inside the execution of the loop, FOUND is not modified by the FOR statement, although it may be changed by the execution of other statements within the loop body.

FOUND is a local variable within each PL/pgSQL function; any changes to it affect only the current function.

37.7. Control Structures

Control structures are probably the most useful (and important) part of PL/pgSQL. With PL/pgSQL's control structures, you can manipulate PostgreSQL data in a very flexible and powerful way.

37.7.1. Returning From a Function

There are two commands available that allow you to return data from a function: RETURN and RETURN NEXT.

37.7.1.1. RETURN

```
RETURN expression;
```

RETURN with an expression terminates the function and returns the value of *expression* to the caller. This form is to be used for PL/pgSQL functions that do not return a set.

When returning a scalar type, any expression can be used. The expression's result will be automatically cast into the function's return type as described for assignments. To return a composite (row) value, you must write a record or row variable as the *expression*.

If you declared the function with output parameters, write just RETURN with no expression. The current values of the output parameter variables will be returned.

If you declared the function to return void, a RETURN statement can be used to exit the function early; but do not write an expression following RETURN.

The return value of a function cannot be left undefined. If control reaches the end of the top-level block of the function without hitting a RETURN statement, a run-time error will occur. This restriction does not apply to functions with output parameters and functions returning void, however. In those cases a RETURN statement is automatically executed if the top-level block finishes.

37.7.1.2. RETURN NEXT

```
RETURN NEXT expression;
```

When a PL/pgSQL function is declared to return SETOF *sometype*, the procedure to follow is slightly different. In that case, the individual items to return are specified in RETURN NEXT commands, and then a final RETURN command with no argument is used to indicate that the function has finished executing. RETURN NEXT can be used with both scalar and composite data types; with a composite result type, an entire "table" of results will be returned.

RETURN NEXT does not actually return from the function — it simply saves away the value of the expression. Execution then continues with the next statement in the PL/pgSQL function. As successive RETURN NEXT commands are executed, the result set is built up. A final RETURN, which should have no argument, causes control to exit the function (or you can just let control reach the end of the function).

If you declared the function with output parameters, write just RETURN NEXT with no expression. The current values of the output parameter variable(s) will be saved for eventual return. Note that you must declare the function as returning SETOF record when there are multiple output parameters, or SETOF *sometype* when there is just one output parameter of type *sometype*, in order to create a set-returning function with output parameters.

Functions that use RETURN NEXT should be called in the following fashion:

```
SELECT * FROM some_func();
```

That is, the function must be used as a table source in a FROM clause.

> **Note:** The current implementation of RETURN NEXT for PL/pgSQL stores the entire result set before returning from the function, as discussed above. That means that if a PL/pgSQL function produces a very large result set, performance may be poor: data will be written to disk to avoid memory exhaustion, but the function itself will not return until the entire result set has been generated. A future version of PL/pgSQL may allow users to define set-returning functions that do not have this limitation. Currently, the point at which data begins being written to disk is controlled by the work_mem configuration variable. Administrators who have sufficient memory to store larger result sets in memory should consider increasing this parameter.

37.7.2. Conditionals

IF statements let you execute commands based on certain conditions. PL/pgSQL has five forms of IF:

- IF ... THEN
- IF ... THEN ... ELSE
- IF ... THEN ... ELSE IF
- IF ... THEN ... ELSIF ... THEN ... ELSE
- IF ... THEN ... ELSEIF ... THEN ... ELSE

37.7.2.1. IF-THEN

```
IF boolean-expression THEN
    statements
END IF;
```

IF-THEN statements are the simplest form of IF. The statements between THEN and END IF will be executed if the condition is true. Otherwise, they are skipped.

Example:

```
IF v_user_id <> 0 THEN
    UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

37.7.2.2. IF-THEN-ELSE

```
IF boolean-expression THEN
    statements
ELSE
    statements
END IF;
```

IF-THEN-ELSE statements add to IF-THEN by letting you specify an alternative set of statements that should be executed if the condition evaluates to false.

Examples:

```
IF parentid IS NULL OR parentid = ''
THEN
    RETURN fullname;
ELSE
    RETURN hp_true_filename(parentid) || '/' || fullname;
END IF;
```

```
IF v_count > 0 THEN
    INSERT INTO users_count (count) VALUES (v_count);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;
```

37.7.2.3. IF-THEN-ELSE IF

IF statements can be nested, as in the following example:

```
IF demo_row.sex = 'm' THEN
    pretty_sex := 'man';
```

```
ELSE
    IF demo_row.sex = 'f' THEN
        pretty_sex := 'woman';
    END IF;
END IF;
```

When you use this form, you are actually nesting an IF statement inside the ELSE part of an outer IF statement. Thus you need one END IF statement for each nested IF and one for the parent IF-ELSE. This is workable but grows tedious when there are many alternatives to be checked. Hence the next form.

37.7.2.4. IF-THEN-ELSIF-ELSE

```
IF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
    ...]]
[ ELSE
    statements ]
END IF;
```

IF-THEN-ELSIF-ELSE provides a more convenient method of checking many alternatives in one statement. Formally it is equivalent to nested IF-THEN-ELSE-IF-THEN commands, but only one END IF is needed.

Here is an example:

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    -- hmm, the only other possibility is that number is null
    result := 'NULL';
END IF;
```

37.7.2.5. IF-THEN-ELSEIF-ELSE

ELSEIF is an alias for ELSIF.

37.7.3. Simple Loops

With the LOOP, EXIT, CONTINUE, WHILE, and FOR statements, you can arrange for your PL/pgSQL function to repeat a series of commands.

37.7.3.1. LOOP

```
[ <<label>> ]
LOOP
    statements
END LOOP [ label ];
```

LOOP defines an unconditional loop that is repeated indefinitely until terminated by an EXIT or RETURN statement. The optional *label* can be used by EXIT and CONTINUE statements in nested loops to specify which loop the statement should be applied to.

37.7.3.2. EXIT

```
EXIT [ label ] [ WHEN expression ];
```

If no *label* is given, the innermost loop is terminated and the statement following END LOOP is executed next. If *label* is given, it must be the label of the current or some outer level of nested loop or block. Then the named loop or block is terminated and control continues with the statement after the loop's/block's corresponding END.

If WHEN is specified, the loop exit occurs only if *expression* is true. Otherwise, control passes to the statement after EXIT.

EXIT can be used with all types of loops; it is not limited to use with unconditional loops. When used with a BEGIN block, EXIT passes control to the next statement after the end of the block.

Examples:

```
LOOP
    -- some computations
    IF count > 0 THEN
        EXIT;  -- exit loop
    END IF;
END LOOP;

LOOP
    -- some computations
    EXIT WHEN count > 0;  -- same result as previous example
END LOOP;

BEGIN
    -- some computations
    IF stocks > 100000 THEN
        EXIT;  -- causes exit from the BEGIN block
    END IF;
END;
```

37.7.3.3. CONTINUE

```
CONTINUE [ label ] [ WHEN expression ];
```

If no *label* is given, the next iteration of the innermost loop is begun. That is, control is passed back to the loop control expression (if any), and the body of the loop is re-evaluated. If *label* is present, it specifies the label of the loop whose execution will be continued.

If WHEN is specified, the next iteration of the loop is begun only if *expression* is true. Otherwise, control passes to the statement after CONTINUE.

CONTINUE can be used with all types of loops; it is not limited to use with unconditional loops.

Examples:

```
LOOP
    -- some computations
    EXIT WHEN count > 100;
    CONTINUE WHEN count < 50;
    -- some computations for count IN [50 .. 100]
END LOOP;
```

37.7.3.4. WHILE

```
[ <<label>> ]
WHILE expression LOOP
    statements
END LOOP [ label ];
```

The WHILE statement repeats a sequence of statements so long as the condition expression evaluates to true. The condition is checked just before each entry to the loop body.

For example:

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
    -- some computations here
END LOOP;

WHILE NOT boolean_expression LOOP
    -- some computations here
END LOOP;
```

37.7.3.5. FOR (integer variant)

```
[ <<label>> ]
FOR name IN [ REVERSE ] expression .. expression [ BY expression ] LOOP
    statements
END LOOP [ label ];
```

This form of FOR creates a loop that iterates over a range of integer values. The variable *name* is automatically defined as type integer and exists only inside the loop (any existing definition of the variable name is ignored within the loop). The two expressions giving the lower and upper bound of the range are evaluated once when entering the loop. If the BY clause isn't specified the iteration step is 1 otherwise it's the value specified in the BY clause. If REVERSE is specified then the step value is considered negative.

Some examples of integer FOR loops:

```
FOR i IN 1..10 LOOP
    -- some computations here
    RAISE NOTICE 'i is %', i;
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- some computations here
END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP
    -- some computations here
    RAISE NOTICE 'i is %', i;
END LOOP;
```

If the lower bound is greater than the upper bound (or less than, in the REVERSE case), the loop body is not executed at all. No error is raised.

37.7.4. Looping Through Query Results

Using a different type of FOR loop, you can iterate through the results of a query and manipulate that data accordingly. The syntax is:

```
[ <<label>> ]
FOR target IN query LOOP
    statements
END LOOP [ label ];
```

The *target* is a record variable, row variable, or comma-separated list of scalar variables. The *target* is successively assigned each row resulting from the *query* and the loop body is executed for each row. Here is an example:

```
CREATE FUNCTION cs_refresh_mviews() RETURNS integer AS $$
DECLARE
    mviews RECORD;
```

```
BEGIN
    PERFORM cs_log('Refreshing materialized views...');

    FOR mviews IN SELECT * FROM cs_materialized_views ORDER BY sort_key LOOP

        -- Now "mviews" has one record from cs_materialized_views

        PERFORM cs_log('Refreshing materialized view ' || quote_ident(mviews.mv_name) || '
        EXECUTE 'TRUNCATE TABLE ' || quote_ident(mviews.mv_name);
        EXECUTE 'INSERT INTO ' || quote_ident(mviews.mv_name) || ' ' || mviews.mv_query;
    END LOOP;

    PERFORM cs_log('Done refreshing materialized views.');
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

If the loop is terminated by an EXIT statement, the last assigned row value is still accessible after the loop.

The *query* used in this type of FOR statement can be any SQL command that returns rows to the caller: SELECT is the most common case, but you can also use INSERT, UPDATE, or DELETE with a RETURNING clause. Some utility commands such as EXPLAIN will work too.

The FOR-IN-EXECUTE statement is another way to iterate over rows:

```
[ <<label>> ]
FOR target IN EXECUTE text_expression LOOP
    statements
END LOOP [ label ];
```

This is like the previous form, except that the source query is specified as a string expression, which is evaluated and replanned on each entry to the FOR loop. This allows the programmer to choose the speed of a preplanned query or the flexibility of a dynamic query, just as with a plain EXECUTE statement.

> **Note:** The PL/pgSQL parser presently distinguishes the two kinds of FOR loops (integer or query result) by checking whether .. appears outside any parentheses between IN and LOOP. If .. is not seen then the loop is presumed to be a loop over rows. Mistyping the .. is thus likely to lead to a complaint along the lines of "loop variable of loop over rows must be a record or row variable or list of scalar variables", rather than the simple syntax error one might expect to get.

37.7.5. Trapping Errors

By default, any error occurring in a PL/pgSQL function aborts execution of the function, and indeed of the surrounding transaction as well. You can trap errors and recover from them by using a BEGIN block with an EXCEPTION clause. The syntax is an extension of the normal syntax for a BEGIN block:

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
```

```
    statements
EXCEPTION
    WHEN condition [ OR condition ... ] THEN
        handler_statements
    [ WHEN condition [ OR condition ... ] THEN
          handler_statements
      ... ]
END;
```

If no error occurs, this form of block simply executes all the *statements*, and then control passes to the next statement after END. But if an error occurs within the *statements*, further processing of the *statements* is abandoned, and control passes to the EXCEPTION list. The list is searched for the first *condition* matching the error that occurred. If a match is found, the corresponding *handler_statements* are executed, and then control passes to the next statement after END. If no match is found, the error propagates out as though the EXCEPTION clause were not there at all: the error can be caught by an enclosing block with EXCEPTION, or if there is none it aborts processing of the function.

The *condition* names can be any of those shown in Appendix A. A category name matches any error within its category. The special condition name OTHERS matches every error type except QUERY_CANCELED. (It is possible, but often unwise, to trap QUERY_CANCELED by name.) Condition names are not case-sensitive.

If a new error occurs within the selected *handler_statements*, it cannot be caught by this EXCEPTION clause, but is propagated out. A surrounding EXCEPTION clause could catch it.

When an error is caught by an EXCEPTION clause, the local variables of the PL/pgSQL function remain as they were when the error occurred, but all changes to persistent database state within the block are rolled back. As an example, consider this fragment:

```
INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');
BEGIN
    UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
    x := x + 1;
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'caught division_by_zero';
        RETURN x;
END;
```

When control reaches the assignment to y, it will fail with a division_by_zero error. This will be caught by the EXCEPTION clause. The value returned in the RETURN statement will be the incremented value of x, but the effects of the UPDATE command will have been rolled back. The INSERT command preceding the block is not rolled back, however, so the end result is that the database contains Tom Jones not Joe Jones.

> **Tip:** A block containing an EXCEPTION clause is significantly more expensive to enter and exit than a block without one. Therefore, don't use EXCEPTION without need.

Within an exception handler, the SQLSTATE variable contains the error code that corresponds to the exception that was raised (refer to Table A-1 for a list of possible error codes). The SQLERRM variable contains the error message associated with the exception. These variables are undefined outside exception handlers.

Example 37-1. Exceptions with UPDATE/INSERT

This example uses exception handling to perform either UPDATE or INSERT, as appropriate.

```
CREATE TABLE db (a INT PRIMARY KEY, b TEXT);

CREATE FUNCTION merge_db(key INT, data TEXT) RETURNS VOID AS
$$
BEGIN
    LOOP
        UPDATE db SET b = data WHERE a = key;
        IF found THEN
            RETURN;
        END IF;

        BEGIN
            INSERT INTO db(a,b) VALUES (key, data);
            RETURN;
        EXCEPTION WHEN unique_violation THEN
            -- do nothing
        END;
    END LOOP;
END;
$$
LANGUAGE plpgsql;

SELECT merge_db(1, 'david');
SELECT merge_db(1, 'dennis');
```

37.8. Cursors

Rather than executing a whole query at once, it is possible to set up a *cursor* that encapsulates the query, and then read the query result a few rows at a time. One reason for doing this is to avoid memory overrun when the result contains a large number of rows. (However, PL/pgSQL users do not normally need to worry about that, since FOR loops automatically use a cursor internally to avoid memory problems.) A more interesting usage is to return a reference to a cursor that a function has created, allowing the caller to read the rows. This provides an efficient way to return large row sets from functions.

37.8.1. Declaring Cursor Variables

All access to cursors in PL/pgSQL goes through cursor variables, which are always of the special data type `refcursor`. One way to create a cursor variable is just to declare it as a variable of type `refcursor`. Another way is to use the cursor declaration syntax, which in general is:

```
name CURSOR [ ( arguments ) ] FOR query;
```

(`FOR` may be replaced by `IS` for Oracle compatibility.) *`arguments`*, if specified, is a comma-separated list of pairs *`name datatype`* that define names to be replaced by parameter values in the given query. The actual values to substitute for these names will be specified later, when the cursor is opened.

Some examples:

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (key integer) IS SELECT * FROM tenk1 WHERE unique1 = key;
```

All three of these variables have the data type `refcursor`, but the first may be used with any query, while the second has a fully specified query already *bound* to it, and the last has a parameterized query bound to it. (`key` will be replaced by an integer parameter value when the cursor is opened.) The variable `curs1` is said to be *unbound* since it is not bound to any particular query.

37.8.2. Opening Cursors

Before a cursor can be used to retrieve rows, it must be *opened*. (This is the equivalent action to the SQL command `DECLARE CURSOR`.) PL/pgSQL has three forms of the `OPEN` statement, two of which use unbound cursor variables while the third uses a bound cursor variable.

37.8.2.1. `OPEN FOR` *`query`*

```
OPEN unbound_cursor FOR query;
```

The cursor variable is opened and given the specified query to execute. The cursor cannot be open already, and it must have been declared as an unbound cursor (that is, as a simple `refcursor` variable). The query must be a `SELECT`, or something else that returns rows (such as `EXPLAIN`). The query is treated in the same way as other SQL commands in PL/pgSQL: PL/pgSQL variable names are substituted, and the query plan is cached for possible reuse.

An example:

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

37.8.2.2. `OPEN FOR EXECUTE`

```
OPEN unbound_cursor FOR EXECUTE query_string;
```

The cursor variable is opened and given the specified query to execute. The cursor cannot be open already, and it must have been declared as an unbound cursor (that is, as a simple `refcursor` variable). The query is specified as a string expression, in the same way as in the `EXECUTE` command. As usual, this gives flexibility so the query can vary from one run to the next.

An example:

```
OPEN curs1 FOR EXECUTE 'SELECT * FROM ' || quote_ident($1);
```

37.8.2.3. Opening a Bound Cursor

```
OPEN bound_cursor [ ( argument_values ) ];
```

This form of `OPEN` is used to open a cursor variable whose query was bound to it when it was declared. The cursor cannot be open already. A list of actual argument value expressions must appear if and only if the cursor was declared to take arguments. These values will be substituted in the query. The query plan for a bound cursor is always considered cacheable; there is no equivalent of `EXECUTE` in this case.

Examples:

```
OPEN curs2;
OPEN curs3(42);
```

37.8.3. Using Cursors

Once a cursor has been opened, it can be manipulated with the statements described here.

These manipulations need not occur in the same function that opened the cursor to begin with. You can return a `refcursor` value out of a function and let the caller operate on the cursor. (Internally, a `refcursor` value is simply the string name of a so-called portal containing the active query for the cursor. This name can be passed around, assigned to other `refcursor` variables, and so on, without disturbing the portal.)

All portals are implicitly closed at transaction end. Therefore a `refcursor` value is usable to reference an open cursor only until the end of the transaction.

37.8.3.1. `FETCH`

```
FETCH cursor INTO target;
```

`FETCH` retrieves the next row from the cursor into a target, which may be a row variable, a record variable, or a comma-separated list of simple variables, just like `SELECT INTO`. As with `SELECT INTO`, the special variable `FOUND` may be checked to see whether a row was obtained or not.

An example:

```
FETCH curs1 INTO rowvar;
```

```
FETCH curs2 INTO foo, bar, baz;
```

37.8.3.2. CLOSE

```
CLOSE cursor;
```

CLOSE closes the portal underlying an open cursor. This can be used to release resources earlier than end of transaction, or to free up the cursor variable to be opened again.

An example:

```
CLOSE curs1;
```

37.8.3.3. Returning Cursors

PL/pgSQL functions can return cursors to the caller. This is useful to return multiple rows or columns, especially with very large result sets. To do this, the function opens the cursor and returns the cursor name to the caller (or simply opens the cursor using a portal name specified by or otherwise known to the caller). The caller can then fetch rows from the cursor. The cursor can be closed by the caller, or it will be closed automatically when the transaction closes.

The portal name used for a cursor can be specified by the programmer or automatically generated. To specify a portal name, simply assign a string to the refcursor variable before opening it. The string value of the refcursor variable will be used by OPEN as the name of the underlying portal. However, if the refcursor variable is null, OPEN automatically generates a name that does not conflict with any existing portal, and assigns it to the refcursor variable.

> **Note:** A bound cursor variable is initialized to the string value representing its name, so that the portal name is the same as the cursor variable name, unless the programmer overrides it by assignment before opening the cursor. But an unbound cursor variable defaults to the null value initially, so it will receive an automatically-generated unique name, unless overridden.

The following example shows one way a cursor name can be supplied by the caller:

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
    OPEN $1 FOR SELECT col FROM test;
    RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;
```

```
SELECT reffunc('funccursor');
FETCH ALL IN funccursor;
COMMIT;
```

The following example uses automatic cursor name generation:

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM test;
    RETURN ref;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc2();

      reffunc2
--------------------
 <unnamed cursor 1>
(1 row)

FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

The following example shows one way to return multiple cursors from a single function:

```
CREATE FUNCTION myfunc(refcursor, refcursor) RETURNS SETOF refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM table_1;
    RETURN NEXT $1;
    OPEN $2 FOR SELECT * FROM table_2;
    RETURN NEXT $2;
END;
$$ LANGUAGE plpgsql;

-- need to be in a transaction to use cursors.
BEGIN;

SELECT * FROM myfunc('a', 'b');

FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

37.9. Errors and Messages

Use the RAISE statement to report messages and raise errors.

```
RAISE level 'format' [, expression [, ...]];
```

Possible levels are DEBUG, LOG, INFO, NOTICE, WARNING, and EXCEPTION. EXCEPTION raises an error (which normally aborts the current transaction); the other levels only generate messages of different priority levels. Whether messages of a particular priority are reported to the client, written to the server log, or both is controlled by the log_min_messages and client_min_messages configuration variables. See Chapter 17 for more information.

Inside the format string, % is replaced by the next optional argument's string representation. Write %% to emit a literal %. Arguments can be simple variables or expressions, and the format must be a simple string literal.

In this example, the value of v_job_id will replace the % in the string:

```
RAISE NOTICE 'Calling cs_create_job(%)', v_job_id;
```

This example will abort the transaction with the given error message:

```
RAISE EXCEPTION 'Nonexistent ID --> %', user_id;
```

RAISE EXCEPTION presently always generates the same SQLSTATE code, P0001, no matter what message it is invoked with. It is possible to trap this exception with EXCEPTION ... WHEN RAISE_EXCEPTION THEN ... but there is no way to tell one RAISE from another.

37.10. Trigger Procedures

PL/pgSQL can be used to define trigger procedures. A trigger procedure is created with the CREATE FUNCTION command, declaring it as a function with no arguments and a return type of trigger. Note that the function must be declared with no arguments even if it expects to receive arguments specified in CREATE TRIGGER — trigger arguments are passed via TG_ARGV, as described below.

When a PL/pgSQL function is called as a trigger, several special variables are created automatically in the top-level block. They are:

NEW

Data type RECORD; variable holding the new database row for INSERT/UPDATE operations in row-level triggers. This variable is NULL in statement-level triggers.

OLD

Data type RECORD; variable holding the old database row for UPDATE/DELETE operations in row-level triggers. This variable is NULL in statement-level triggers.

TG_NAME

Data type name; variable that contains the name of the trigger actually fired.

TG_WHEN

Data type text; a string of either BEFORE or AFTER depending on the trigger's definition.

TG_LEVEL

Data type text; a string of either ROW or STATEMENT depending on the trigger's definition.

TG_OP

Data type text; a string of INSERT, UPDATE, or DELETE telling for which operation the trigger was fired.

TG_RELID

Data type oid; the object ID of the table that caused the trigger invocation.

TG_RELNAME

Data type name; the name of the table that caused the trigger invocation. This is now deprecated, and could disappear in a future release. Use TG_TABLE_NAME instead.

TG_TABLE_NAME

Data type name; the name of the table that caused the trigger invocation.

TG_TABLE_SCHEMA

Data type name; the name of the schema of the table that caused the trigger invocation.

TG_NARGS

Data type integer; the number of arguments given to the trigger procedure in the CREATE TRIGGER statement.

TG_ARGV[]

Data type array of text; the arguments from the CREATE TRIGGER statement. The index counts from 0. Invalid indices (less than 0 or greater than or equal to tg_nargs) result in a null value.

A trigger function must return either NULL or a record/row value having exactly the structure of the table the trigger was fired for.

Row-level triggers fired BEFORE may return null to signal the trigger manager to skip the rest of the operation for this row (i.e., subsequent triggers are not fired, and the INSERT/UPDATE/DELETE does not occur for this row). If a nonnull value is returned then the operation proceeds with that row value. Returning a row value different from the original value of NEW alters the row that will be inserted or updated (but has no direct effect in the DELETE case). To alter the row to be stored, it is possible to replace single values directly in NEW and return the modified NEW, or to build a complete new record/row to return.

The return value of a BEFORE or AFTER statement-level trigger or an AFTER row-level trigger is always ignored; it may as well be null. However, any of these types of triggers can still abort the entire operation by raising an error.

Example 37-2 shows an example of a trigger procedure in PL/pgSQL.

Example 37-2. A PL/pgSQL Trigger Procedure

This example trigger ensures that any time a row is inserted or updated in the table, the current user name and time are stamped into the row. And it checks that an employee's name is given and that the salary is a positive value.

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN
        -- Check that empname and salary are given
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION 'empname cannot be null';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION '% cannot have null salary', NEW.empname;
        END IF;

        -- Who works for us when she must pay for it?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
        END IF;

        -- Remember who changed the payroll when
        NEW.last_date := current_timestamp;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
$emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

Another way to log changes to a table involves creating a new table that holds a row for each insert, update, or delete that occurs. This approach can be thought of as auditing changes to a table. Example 37-3 shows an example of an audit trigger procedure in PL/pgSQL.

Example 37-3. A PL/pgSQL Trigger Procedure For Auditing

This example trigger ensures that any insert, update or delete of a row in the `emp` table is recorded (i.e., audited) in the `emp_audit` table. The current time and user name are stamped into the row, together with the type of operation performed on it.

```
CREATE TABLE emp (
    empname           text NOT NULL,
    salary            integer
);
```

```
CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN
        --
        -- Create a row in emp_audit to reflect the operation performed on emp,
        -- make use of the special variable TG_OP to work out the operation.
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
            RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
            RETURN NEW;
        END IF;
        RETURN NULL; -- result is ignored since this is an AFTER trigger
    END;
$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();
```

One use of triggers is to maintain a summary table of another table. The resulting summary can be used in place of the original table for certain queries — often with vastly reduced run times. This technique is commonly used in Data Warehousing, where the tables of measured or observed data (called fact tables) can be extremely large. Example 37-4 shows an example of a trigger procedure in PL/pgSQL that maintains a summary table for a fact table in a data warehouse.

Example 37-4. A PL/pgSQL Trigger Procedure For Maintaining A Summary Table

The schema detailed here is partly based on the *Grocery Store* example from *The Data Warehouse Toolkit* by Ralph Kimball.

```
--
-- Main tables - time dimension and sales fact.
--
CREATE TABLE time_dimension (
    time_key                    integer NOT NULL,
    day_of_week                 integer NOT NULL,
    day_of_month                integer NOT NULL,
    month                       integer NOT NULL,
    quarter                     integer NOT NULL,
    year                        integer NOT NULL
```

```
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

CREATE TABLE sales_fact (
    time_key                    integer NOT NULL,
    product_key                 integer NOT NULL,
    store_key                   integer NOT NULL,
    amount_sold                 numeric(12,2) NOT NULL,
    units_sold                  integer NOT NULL,
    amount_cost                 numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);

--
-- Summary table - sales by time.
--
CREATE TABLE sales_summary_bytime (
    time_key                    integer NOT NULL,
    amount_sold                 numeric(15,2) NOT NULL,
    units_sold                  numeric(12) NOT NULL,
    amount_cost                 numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON sales_summary_bytime(time_key);

--
-- Function and trigger to amend summarized column(s) on UPDATE, INSERT, DELETE.
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER AS $maint_sales_sum
    DECLARE
        delta_time_key          integer;
        delta_amount_sold       numeric(15,2);
        delta_units_sold        numeric(12);
        delta_amount_cost       numeric(15,2);
    BEGIN

        -- Work out the increment/decrement amount(s).
        IF (TG_OP = 'DELETE') THEN

            delta_time_key = OLD.time_key;
            delta_amount_sold = -1 * OLD.amount_sold;
            delta_units_sold = -1 * OLD.units_sold;
            delta_amount_cost = -1 * OLD.amount_cost;

        ELSIF (TG_OP = 'UPDATE') THEN

            -- forbid updates that change the time_key -
            -- (probably not too onerous, as DELETE + INSERT is how most
            -- changes will be made).
            IF ( OLD.time_key != NEW.time_key) THEN
                RAISE EXCEPTION 'Update of time_key : % -> % not allowed', OLD.time_key, NE
            END IF;

            delta_time_key = OLD.time_key;
```

```
            delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
            delta_units_sold = NEW.units_sold - OLD.units_sold;
            delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

        ELSIF (TG_OP = 'INSERT') THEN

            delta_time_key = NEW.time_key;
            delta_amount_sold = NEW.amount_sold;
            delta_units_sold = NEW.units_sold;
            delta_amount_cost = NEW.amount_cost;

        END IF;


        -- Insert or update the summary row with the new values.
        <<insert_update>>
        LOOP
            UPDATE sales_summary_bytime
                SET amount_sold = amount_sold + delta_amount_sold,
                    units_sold = units_sold + delta_units_sold,
                    amount_cost = amount_cost + delta_amount_cost
                WHERE time_key = delta_time_key;

            EXIT insert_update WHEN found;

            BEGIN
                INSERT INTO sales_summary_bytime (
                            time_key,
                            amount_sold,
                            units_sold,
                            amount_cost)
                    VALUES (
                            delta_time_key,
                            delta_amount_sold,
                            delta_units_sold,
                            delta_amount_cost
                           );

                EXIT insert_update;

            EXCEPTION
                WHEN UNIQUE_VIOLATION THEN
                    -- do nothing
            END;
        END LOOP insert_update;

        RETURN NULL;

    END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint_sales_summary_bytime
AFTER INSERT OR UPDATE OR DELETE ON sales_fact
```

```
    FOR EACH ROW EXECUTE PROCEDURE maint_sales_summary_bytime();

INSERT INTO sales_fact VALUES(1,1,1,10,3,15);
INSERT INTO sales_fact VALUES(1,2,1,20,5,35);
INSERT INTO sales_fact VALUES(2,2,1,40,15,135);
INSERT INTO sales_fact VALUES(2,3,1,10,1,13);
SELECT * FROM sales_summary_bytime;
DELETE FROM sales_fact WHERE product_key = 1;
SELECT * FROM sales_summary_bytime;
UPDATE sales_fact SET units_sold = units_sold * 2;
SELECT * FROM sales_summary_bytime;
```

37.11. Porting from Oracle PL/SQL

This section explains differences between PostgreSQL's PL/pgSQL language and Oracle's PL/SQL language, to help developers who port applications from Oracle® to PostgreSQL.

PL/pgSQL is similar to PL/SQL in many aspects. It is a block-structured, imperative language, and all variables have to be declared. Assignments, loops, conditionals are similar. The main differences you should keep in mind when porting from PL/SQL to PL/pgSQL are:

- There are no default values for parameters in PostgreSQL.
- You can overload function names in PostgreSQL. This is often used to work around the lack of default parameters.
- You cannot use parameter names that are the same as columns that are referenced in the function. Oracle allows you to do this if you qualify the parameter name using `function_name.parameter_name`. In PL/pgSQL, you can instead avoid a conflict by qualifying the column or table name.
- No need for cursors in PL/pgSQL, just put the query in the `FOR` statement. (See Example 37-6.)
- In PostgreSQL the function body must be written as a string literal. Therefore you need to use dollar quoting or escape single quotes in the function body. See Section 37.2.1.
- Instead of packages, use schemas to organize your functions into groups.
- Since there are no packages, there are no package-level variables either. This is somewhat annoying. You can keep per-session state in temporary tables instead.

37.11.1. Porting Examples

Example 37-5 shows how to port a simple function from PL/SQL to PL/pgSQL.

Example 37-5. Porting a Simple Function from PL/SQL to PL/pgSQL

Here is an Oracle PL/SQL function:

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar,
                                                  v_version varchar)
```

```
RETURN varchar IS
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
/
show errors;
```

Let's go through this function and see the differences compared to PL/pgSQL:

- The `RETURN` key word in the function prototype (not the function body) becomes `RETURNS` in PostgreSQL. Also, `IS` becomes `AS`, and you need to add a `LANGUAGE` clause because PL/pgSQL is not the only possible function language.
- In PostgreSQL, the function body is considered to be a string literal, so you need to use quote marks or dollar quotes around it. This substitutes for the terminating `/` in the Oracle approach.
- The `show errors` command does not exist in PostgreSQL, and is not needed since errors are reported automatically.

This is how this function would look when ported to PostgreSQL:

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar,
                                                  v_version varchar)
RETURNS varchar AS $$
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
$$ LANGUAGE plpgsql;
```

Example 37-6 shows how to port a function that creates another function and how to handle the ensuing quoting problems.

Example 37-6. Porting a Function that Creates Another Function from PL/SQL to PL/pgSQL

The following procedure grabs rows from a `SELECT` statement and builds a large function with the results in `IF` statements, for the sake of efficiency. Notice particularly the differences in the cursor and the `FOR` loop.

This is the Oracle version:

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc IS
    CURSOR referrer_keys IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;

    func_cmd VARCHAR(4000);
```

```
BEGIN
    func_cmd := 'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host IN VARCHAR,
                 v_domain IN VARCHAR, v_url IN VARCHAR) RETURN VARCHAR IS BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_cmd := func_cmd ||
          ' IF v_' || referrer_key.kind
          || ' LIKE ''' || referrer_key.key_string
          || ''' THEN RETURN ''' || referrer_key.referrer_type
          || '''; END IF;';
    END LOOP;

    func_cmd := func_cmd || ' RETURN NULL; END;';

    EXECUTE IMMEDIATE func_cmd;
END;
/
show errors;
```

Here is how this function would end up in PostgreSQL:

```
CREATE OR REPLACE FUNCTION cs_update_referrer_type_proc() RETURNS void AS $func$
DECLARE
    referrer_key RECORD;  -- declare a generic record to be used in a FOR
    func_body text;
    func_cmd text;
BEGIN
    func_body := 'BEGIN';

    -- Notice how we scan through the results of a query in a FOR loop
    -- using the FOR <record> construct.

    FOR referrer_key IN SELECT * FROM cs_referrer_keys ORDER BY try_order LOOP
        func_body := func_body ||
          ' IF v_' || referrer_key.kind
          || ' LIKE ' || quote_literal(referrer_key.key_string)
          || ' THEN RETURN ' || quote_literal(referrer_key.referrer_type)
          || '; END IF;' ;
    END LOOP;

    func_body := func_body || ' RETURN NULL; END;';

    func_cmd :=
      'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host varchar,
                                                        v_domain varchar,
                                                        v_url varchar)
        RETURNS varchar AS '
      || quote_literal(func_body)
      || ' LANGUAGE plpgsql;' ;

    EXECUTE func_cmd;
END;
$func$ LANGUAGE plpgsql;
```

Notice how the body of the function is built separately and passed through `quote_literal` to double any quote marks in it. This technique is needed because we cannot safely use dollar quoting for defining the new function: we do not know for sure what strings will be interpolated from the `referrer_key.key_string` field. (We are assuming here that `referrer_key.kind` can be trusted to always be `host`, `domain`, or `url`, but `referrer_key.key_string` might be anything, in particular it might contain dollar signs.) This function is actually an improvement on the Oracle original, because it will not generate broken code when `referrer_key.key_string` or `referrer_key.referrer_type` contain quote marks.

Example 37-7 shows how to port a function with `OUT` parameters and string manipulation. PostgreSQL does not have a built-in `instr` function, but you can create one using a combination of other functions. In Section 37.11.3 there is a PL/pgSQL implementation of `instr` that you can use to make your porting easier.

Example 37-7. Porting a Procedure With String Manipulation and `OUT` Parameters from PL/SQL to PL/pgSQL

The following Oracle PL/SQL procedure is used to parse a URL and return several elements (host, path, and query).

This is the Oracle version:

```
CREATE OR REPLACE PROCEDURE cs_parse_url(
    v_url IN VARCHAR,
    v_host OUT VARCHAR,  -- This will be passed back
    v_path OUT VARCHAR,  -- This one too
    v_query OUT VARCHAR) -- And this one
IS
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
```

```
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
/
show errors;
```

Here is a possible translation into PL/pgSQL:

```
CREATE OR REPLACE FUNCTION cs_parse_url(
    v_url IN VARCHAR,
    v_host OUT VARCHAR,  -- This will be passed back
    v_path OUT VARCHAR,  -- This one too
    v_query OUT VARCHAR) -- And this one
AS $$
DECLARE
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
$$ LANGUAGE plpgsql;
```

This function could be used like this:

```
SELECT * FROM cs_parse_url('http://foobar.com/query.cgi?baz');
```

Example 37-8 shows how to port a procedure that uses numerous features that are specific to Oracle.

Example 37-8. Porting a Procedure from PL/SQL to PL/pgSQL

The Oracle version:

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
    a_running_job_count INTEGER;
    PRAGMA AUTONOMOUS_TRANSACTION;❶
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;❷

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- free lock❸
        raise_application_error(-20000, 'Unable to create a new job: a job is currently run
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, sysdate);
    EXCEPTION
        WHEN dup_val_on_index THEN NULL; -- don't worry if it already exists
    END;
    COMMIT;
END;
/
show errors
```

Procedures like this can easily be converted into PostgreSQL functions returning `void`. This procedure in particular is interesting because it can teach us some things:

❶ There is no `PRAGMA` statement in PostgreSQL.

❷ If you do a `LOCK TABLE` in PL/pgSQL, the lock will not be released until the calling transaction is finished.

❸ You cannot issue `COMMIT` in a PL/pgSQL function. The function is running within some outer transaction and so `COMMIT` would imply terminating the function's execution. However, in this particular case it is not necessary anyway, because the lock obtained by the `LOCK TABLE` will be released when we raise an error.

This is how we could port this procedure to PL/pgSQL:

```
CREATE OR REPLACE FUNCTION cs_create_job(v_job_id integer) RETURNS void AS $$
DECLARE
    a_running_job_count integer;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
```

```
        RAISE EXCEPTION 'Unable to create a new job: a job is currently running';❶
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
    EXCEPTION
        WHEN unique_violation THEN ❷
            -- don't worry if it already exists
    END;
END;
$$ LANGUAGE plpgsql;
```

❶ The syntax of `RAISE` is considerably different from Oracle's similar statement.

❷ The exception names supported by PL/pgSQL are different from Oracle's. The set of built-in exception names is much larger (see Appendix A). There is not currently a way to declare user-defined exception names.

The main functional difference between this procedure and the Oracle equivalent is that the exclusive lock on the `cs_jobs` table will be held until the calling transaction completes. Also, if the caller later aborts (for example due to an error), the effects of this procedure will be rolled back.

37.11.2. Other Things to Watch For

This section explains a few other things to watch for when porting Oracle PL/SQL functions to PostgreSQL.

37.11.2.1. Implicit Rollback after Exceptions

In PL/pgSQL, when an exception is caught by an `EXCEPTION` clause, all database changes since the block's `BEGIN` are automatically rolled back. That is, the behavior is equivalent to what you'd get in Oracle with

```
BEGIN
    SAVEPOINT s1;
    ... code here ...
EXCEPTION
    WHEN ... THEN
        ROLLBACK TO s1;
        ... code here ...
    WHEN ... THEN
        ROLLBACK TO s1;
        ... code here ...
END;
```

If you are translating an Oracle procedure that uses SAVEPOINT and ROLLBACK TO in this style, your task is easy: just omit the SAVEPOINT and ROLLBACK TO. If you have a procedure that uses SAVEPOINT and ROLLBACK TO in a different way then some actual thought will be required.

37.11.2.2. EXECUTE

The PL/pgSQL version of EXECUTE works similarly to the PL/SQL version, but you have to remember to use quote_literal and quote_ident as described in Section 37.6.5. Constructs of the type EXECUTE 'SELECT * FROM $1'; will not work reliably unless you use these functions.

37.11.2.3. Optimizing PL/pgSQL Functions

PostgreSQL gives you two function creation modifiers to optimize execution: "volatility" (whether the function always returns the same result when given the same arguments) and "strictness" (whether the function returns null if any argument is null). Consult the *CREATE FUNCTION* reference page for details.

When making use of these optimization attributes, your CREATE FUNCTION statement might look something like this:

```
CREATE FUNCTION foo(...) RETURNS integer AS $$
...
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

37.11.3. Appendix

This section contains the code for a set of Oracle-compatible instr functions that you can use to simplify your porting efforts.

```
--
-- instr functions that mimic Oracle's counterpart
-- Syntax: instr(string1, string2, [n], [m]) where [] denotes optional parameters.
--
-- Searches string1 beginning at the nth character for the mth occurrence
-- of string2.  If n is negative, search backwards.  If m is not passed,
-- assume 1 (search starts at first character).
--

CREATE FUNCTION instr(varchar, varchar) RETURNS integer AS $$
DECLARE
    pos integer;
BEGIN
    pos:= instr($1, $2, 1);
    RETURN pos;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

```
CREATE FUNCTION instr(string varchar, string_to_search varchar, beg_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        temp_str := substring(string FROM beg_index);
        pos := position(string_to_search IN temp_str);

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN pos + beg_index - 1;
        END IF;
    ELSE
        ss_length := char_length(string_to_search);
        length := char_length(string);
        beg := length + beg_index - ss_length + 2;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            pos := position(string_to_search IN temp_str);

            IF pos > 0 THEN
                RETURN beg;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;


CREATE FUNCTION instr(string varchar, string_to_search varchar,
                      beg_index integer, occur_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    occur_number integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    i integer;
    length integer;
    ss_length integer;
BEGIN
```

```
    IF beg_index > 0 THEN
        beg := beg_index;
        temp_str := substring(string FROM beg_index);

        FOR i IN 1..occur_index LOOP
            pos := position(string_to_search IN temp_str);

            IF i = 1 THEN
                beg := beg + pos - 1;
            ELSE
                beg := beg + pos;
            END IF;

            temp_str := substring(string FROM beg + 1);
        END LOOP;

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN beg;
        END IF;
    ELSE
        ss_length := char_length(string_to_search);
        length := char_length(string);
        beg := length + beg_index - ss_length + 2;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            pos := position(string_to_search IN temp_str);

            IF pos > 0 THEN
                occur_number := occur_number + 1;

                IF occur_number = occur_index THEN
                    RETURN beg;
                END IF;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

Chapter 38. PL/Tcl - Tcl Procedural Language

PL/Tcl is a loadable procedural language for the PostgreSQL database system that enables the Tcl language[1] to be used to write functions and trigger procedures.

38.1. Overview

PL/Tcl offers most of the capabilities a function writer has in the C language, with a few restrictions, and with the addition of the powerful string processing libraries that are available for Tcl.

One compelling *good* restriction is that everything is executed from within the safety of the context of a Tcl interpreter. In addition to the limited command set of safe Tcl, only a few commands are available to access the database via SPI and to raise messages via `elog()`. PL/Tcl provides no way to access internals of the database server or to gain OS-level access under the permissions of the PostgreSQL server process, as a C function can do. Thus, unprivileged database users may be trusted to use this language; it does not give them unlimited authority.

The other notable implementation restriction is that Tcl functions may not be used to create input/output functions for new data types.

Sometimes it is desirable to write Tcl functions that are not restricted to safe Tcl. For example, one might want a Tcl function that sends email. To handle these cases, there is a variant of PL/Tcl called `PL/TclU` (for untrusted Tcl). This is the exact same language except that a full Tcl interpreter is used. *If PL/TclU is used, it must be installed as an untrusted procedural language* so that only database superusers can create functions in it. The writer of a PL/TclU function must take care that the function cannot be used to do anything unwanted, since it will be able to do anything that could be done by a user logged in as the database administrator.

The shared object code for the PL/Tcl and PL/TclU call handlers is automatically built and installed in the PostgreSQL library directory if Tcl support is specified in the configuration step of the installation procedure. To install PL/Tcl and/or PL/TclU in a particular database, use the `createlang` program, for example `createlang pltcl` *`dbname`* or `createlang pltclu` *`dbname`*.

38.2. PL/Tcl Functions and Arguments

To create a function in the PL/Tcl language, use the standard *CREATE FUNCTION* syntax:

```
CREATE FUNCTION funcname (argument-types) RETURNS return-type AS $$
    # PL/Tcl function body
$$ LANGUAGE pltcl;
```

PL/TclU is the same, except that the language has to be specified as `pltclu`.

The body of the function is simply a piece of Tcl script. When the function is called, the argument values are passed as variables `$1` ... `$`*`n`* to the Tcl script. The result is returned from the Tcl code in the usual way, with a `return` statement.

1. http://www.tcl.tk/

For example, a function returning the greater of two integer values could be defined as:

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl STRICT;
```

Note the clause `STRICT`, which saves us from having to think about null input values: if a null value is passed, the function will not be called at all, but will just return a null result automatically.

In a nonstrict function, if the actual value of an argument is null, the corresponding `$n` variable will be set to an empty string. To detect whether a particular argument is null, use the function `argisnull`. For example, suppose that we wanted `tcl_max` with one null and one nonnull argument to return the nonnull argument, rather than null:

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {[argisnull 1]} {
        if {[argisnull 2]} { return_null }
        return $2
    }
    if {[argisnull 2]} { return $1 }
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl;
```

As shown above, to return a null value from a PL/Tcl function, execute `return_null`. This can be done whether the function is strict or not.

Composite-type arguments are passed to the function as Tcl arrays. The element names of the array are the attribute names of the composite type. If an attribute in the passed row has the null value, it will not appear in the array. Here is an example:

```
CREATE TABLE employee (
    name text,
    salary integer,
    age integer
);

CREATE FUNCTION overpaid(employee) RETURNS boolean AS $$
    if {200000.0 < $1(salary)} {
        return "t"
    }
    if {$1(age) < 30 && 100000.0 < $1(salary)} {
        return "t"
    }
    return "f"
$$ LANGUAGE pltcl;
```

There is currently no support for returning a composite-type result value, nor for returning sets.

PL/Tcl does not currently have full support for domain types: it treats a domain the same as the underlying scalar type. This means that constraints associated with the domain will not be enforced. This is not an issue for function arguments, but it is a hazard if you declare a PL/Tcl function as returning a domain type.

38.3. Data Values in PL/Tcl

The argument values supplied to a PL/Tcl function's code are simply the input arguments converted to text form (just as if they had been displayed by a `SELECT` statement). Conversely, the `return` command will accept any string that is acceptable input format for the function's declared return type. So, within the PL/Tcl function, all values are just text strings.

38.4. Global Data in PL/Tcl

Sometimes it is useful to have some global data that is held between two calls to a function or is shared between different functions. This is easily done since all PL/Tcl functions executed in one session share the same safe Tcl interpreter. So, any global Tcl variable is accessible to all PL/Tcl function calls and will persist for the duration of the SQL session. (Note that PL/TclU functions likewise share global data, but they are in a different Tcl interpreter and cannot communicate with PL/Tcl functions.)

To help protect PL/Tcl functions from unintentionally interfering with each other, a global array is made available to each function via the `upvar` command. The global name of this variable is the function's internal name, and the local name is `GD`. It is recommended that `GD` be used for persistent private data of a function. Use regular Tcl global variables only for values that you specifically intend to be shared among multiple functions.

An example of using `GD` appears in the `spi_execp` example below.

38.5. Database Access from PL/Tcl

The following commands are available to access the database from the body of a PL/Tcl function:

`spi_exec ?-count` *n*`? ?-array` *name*`?` *command* `?`*loop-body*`?`

Executes an SQL command given as a string. An error in the command causes an error to be raised. Otherwise, the return value of `spi_exec` is the number of rows processed (selected, inserted, updated, or deleted) by the command, or zero if the command is a utility statement. In addition, if the command is a `SELECT` statement, the values of the selected columns are placed in Tcl variables as described below.

The optional `-count` value tells `spi_exec` the maximum number of rows to process in the command. The effect of this is comparable to setting up a query as a cursor and then saying `FETCH` *n*.

If the command is a SELECT statement, the values of the result columns are placed into Tcl variables named after the columns. If the -array option is given, the column values are instead stored into the named associative array, with the column names used as array indexes.

If the command is a SELECT statement and no *loop-body* script is given, then only the first row of results are stored into Tcl variables; remaining rows, if any, are ignored. No storing occurs if the query returns no rows. (This case can be detected by checking the result of spi_exec.) For example,

```
spi_exec "SELECT count(*) AS cnt FROM pg_proc"
```

will set the Tcl variable $cnt to the number of rows in the pg_proc system catalog.

If the optional *loop-body* argument is given, it is a piece of Tcl script that is executed once for each row in the query result. (*loop-body* is ignored if the given command is not a SELECT.) The values of the current row's columns are stored into Tcl variables before each iteration. For example,

```
spi_exec -array C "SELECT * FROM pg_class" {
    elog DEBUG "have table $C(relname)"
}
```

will print a log message for every row of pg_class. This feature works similarly to other Tcl looping constructs; in particular continue and break work in the usual way inside the loop body.

If a column of a query result is null, the target variable for it is "unset" rather than being set.

spi_prepare *query typelist*

Prepares and saves a query plan for later execution. The saved plan will be retained for the life of the current session.

The query may use parameters, that is, placeholders for values to be supplied whenever the plan is actually executed. In the query string, refer to parameters by the symbols $1 ... $*n*. If the query uses parameters, the names of the parameter types must be given as a Tcl list. (Write an empty list for *typelist* if no parameters are used.) Presently, the parameter types must be identified by the internal type names shown in the system table pg_type; for example int4 not integer.

The return value from spi_prepare is a query ID to be used in subsequent calls to spi_execp. See spi_execp for an example.

spi_execp ?-count *n*? ?-array *name*? ?-nulls *string*? *queryid* ?*value-list*? ?*loop-body*?

Executes a query previously prepared with spi_prepare. *queryid* is the ID returned by spi_prepare. If the query references parameters, a *value-list* must be supplied. This is a Tcl list of actual values for the parameters. The list must be the same length as the parameter type list previously given to spi_prepare. Omit *value-list* if the query has no parameters.

The optional value for -nulls is a string of spaces and 'n' characters telling spi_execp which of the parameters are null values. If given, it must have exactly the same length as the *value-list*. If it is not given, all the parameter values are nonnull.

Except for the way in which the query and its parameters are specified, spi_execp works just like spi_exec. The -count, -array, and *loop-body* options are the same, and so is the result value.

Here's an example of a PL/Tcl function using a prepared plan:

```
CREATE FUNCTION t1_count(integer, integer) RETURNS integer AS $$
    if {![ info exists GD(plan) ]} {
        # prepare the saved plan on the first call
        set GD(plan) [ spi_prepare \
```

```
                    "SELECT count(*) AS cnt FROM t1 WHERE num >= \$1 AND num <= \$2" \
                    [ list int4 int4 ] ]
        }
        spi_execp -count 1 $GD(plan) [ list $1 $2 ]
        return $cnt
    $$ LANGUAGE pltcl;
```

We need backslashes inside the query string given to `spi_prepare` to ensure that the `$n` markers will be passed through to `spi_prepare` as-is, and not replaced by Tcl variable substitution.

`spi_lastoid`

Returns the OID of the row inserted by the last `spi_exec` or `spi_execp`, if the command was a single-row `INSERT` and the modified table contained OIDs. (If not, you get zero.)

`quote` *`string`*

Doubles all occurrences of single quote and backslash characters in the given string. This may be used to safely quote strings that are to be inserted into SQL commands given to `spi_exec` or `spi_prepare`. For example, think about an SQL command string like

```
"SELECT '$val' AS ret"
```

where the Tcl variable `val` actually contains `doesn't`. This would result in the final command string

```
SELECT 'doesn't' AS ret
```

which would cause a parse error during `spi_exec` or `spi_prepare`. To work properly, the submitted command should contain

```
SELECT 'doesn"t' AS ret
```

which can be formed in PL/Tcl using

```
"SELECT '[ quote $val ]' AS ret"
```

One advantage of `spi_execp` is that you don't have to quote parameter values like this, since the parameters are never parsed as part of an SQL command string.

`elog` *`level msg`*

Emits a log or error message. Possible levels are `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, and `FATAL`. `ERROR` raises an error condition; if this is not trapped by the surrounding Tcl code, the error propagates out to the calling query, causing the current transaction or subtransaction to be aborted. This is effectively the same as the Tcl `error` command. `FATAL` aborts the transaction and causes the current session to shut down. (There is probably no good reason to use this error level in PL/Tcl functions, but it's provided for completeness.) The other levels only generate messages of different priority levels. Whether messages of a particular priority are reported to the client, written to the server log, or both is controlled by the log_min_messages and client_min_messages configuration variables. See Chapter 17 for more information.

38.6. Trigger Procedures in PL/Tcl

Trigger procedures can be written in PL/Tcl. PostgreSQL requires that a procedure that is to be called as a trigger must be declared as a function with no arguments and a return type of `trigger`.

The information from the trigger manager is passed to the procedure body in the following variables:

`$TG_name`

The name of the trigger from the `CREATE TRIGGER` statement.

`$TG_relid`

The object ID of the table that caused the trigger procedure to be invoked.

`$TG_table_name`

The name of the table that caused the trigger procedure to be invoked.

`$TG_table_schema`

The schema of the table that caused the trigger procedure to be invoked.

`$TG_relatts`

A Tcl list of the table column names, prefixed with an empty list element. So looking up a column name in the list with Tcl's `lsearch` command returns the element's number starting with 1 for the first column, the same way the columns are customarily numbered in PostgreSQL. (Empty list elements also appear in the positions of columns that have been dropped, so that the attribute numbering is correct for columns to their right.)

`$TG_when`

The string `BEFORE` or `AFTER` depending on the type of trigger call.

`$TG_level`

The string `ROW` or `STATEMENT` depending on the type of trigger call.

`$TG_op`

The string `INSERT`, `UPDATE`, or `DELETE` depending on the type of trigger call.

`$NEW`

An associative array containing the values of the new table row for `INSERT` or `UPDATE` actions, or empty for `DELETE`. The array is indexed by column name. Columns that are null will not appear in the array.

`$OLD`

An associative array containing the values of the old table row for `UPDATE` or `DELETE` actions, or empty for `INSERT`. The array is indexed by column name. Columns that are null will not appear in the array.

`$args`

A Tcl list of the arguments to the procedure as given in the `CREATE TRIGGER` statement. These arguments are also accessible as `$1` ... `$`*n* in the procedure body.

The return value from a trigger procedure can be one of the strings `OK` or `SKIP`, or a list as returned by the `array get` Tcl command. If the return value is `OK`, the operation (`INSERT`/`UPDATE`/`DELETE`) that fired the trigger will proceed normally. `SKIP` tells the trigger manager to silently suppress the operation for this row. If a list is returned, it tells PL/Tcl to return a modified row to the trigger manager that will be inserted instead of the one given in `$NEW`. (This works for `INSERT` and `UPDATE` only.) Needless to say that all this is only meaningful when the trigger is `BEFORE` and `FOR EACH ROW`; otherwise the return value is ignored.

Here's a little example trigger procedure that forces an integer value in a table to keep track of the number of updates that are performed on the row. For new rows inserted, the value is initialized to 0 and then incremented on every update operation.

```
CREATE FUNCTION trigfunc_modcount() RETURNS trigger AS $$
    switch $TG_op {
        INSERT {
            set NEW($1) 0
        }
        UPDATE {
            set NEW($1) $OLD($1)
            incr NEW($1)
        }
        default {
            return OK
        }
    }
    return [array get NEW]
$$ LANGUAGE pltcl;

CREATE TABLE mytab (num integer, description text, modcnt integer);

CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON mytab
    FOR EACH ROW EXECUTE PROCEDURE trigfunc_modcount('modcnt');
```

Notice that the trigger procedure itself does not know the column name; that's supplied from the trigger arguments. This lets the trigger procedure be reused with different tables.

38.7. Modules and the `unknown` command

PL/Tcl has support for autoloading Tcl code when used. It recognizes a special table, `pltcl_modules`, which is presumed to contain modules of Tcl code. If this table exists, the module `unknown` is fetched from the table and loaded into the Tcl interpreter immediately after creating the interpreter.

While the `unknown` module could actually contain any initialization script you need, it normally defines a Tcl `unknown` procedure that is invoked whenever Tcl does not recognize an invoked procedure name. PL/Tcl's standard version of this procedure tries to find a module in `pltcl_modules` that will define the required procedure. If one is found, it is loaded into the interpreter, and then execution is allowed to proceed with the originally attempted procedure call. A secondary table `pltcl_modfuncs` provides an index of which functions are defined by which modules, so that the lookup is reasonably quick.

The PostgreSQL distribution includes support scripts to maintain these tables: `pltcl_loadmod`, `pltcl_listmod`, `pltcl_delmod`, as well as source for the standard `unknown` module in `share/unknown.pltcl`. This module must be loaded into each database initially to support the autoloading mechanism.

The tables `pltcl_modules` and `pltcl_modfuncs` must be readable by all, but it is wise to make them owned and writable only by the database administrator.

38.8. Tcl Procedure Names

In PostgreSQL, one and the same function name can be used for different functions as long as the number of arguments or their types differ. Tcl, however, requires all procedure names to be distinct. PL/Tcl deals with this by making the internal Tcl procedure names contain the object ID of the function from the system table `pg_proc` as part of their name. Thus, PostgreSQL functions with the same name and different argument types will be different Tcl procedures, too. This is not normally a concern for a PL/Tcl programmer, but it might be visible when debugging.

Chapter 39. PL/Perl - Perl Procedural Language

PL/Perl is a loadable procedural language that enables you to write PostgreSQL functions in the Perl programming language[1].

The usual advantage to using PL/Perl is that this allows use, within stored functions, of the manyfold "string munging" operators and functions available for Perl. Parsing complex strings may be easier using Perl than it is with the string functions and control structures provided in PL/pgSQL.

To install PL/Perl in a particular database, use `createlang plperl` *`dbname`*.

> **Tip:** If a language is installed into `template1`, all subsequently created databases will have the language installed automatically.

> **Note:** Users of source packages must specially enable the build of PL/Perl during the installation process. (Refer to Section 14.1 for more information.) Users of binary packages might find PL/Perl in a separate subpackage.

39.1. PL/Perl Functions and Arguments

To create a function in the PL/Perl language, use the standard *CREATE FUNCTION* syntax:

```
CREATE FUNCTION funcname (argument-types) RETURNS return-type AS $$
    # PL/Perl function body
$$ LANGUAGE plperl;
```

The body of the function is ordinary Perl code. In fact, the PL/Perl glue code wraps it inside a Perl subroutine. A PL/Perl function must always return a scalar value. You can return more complex structures (arrays, records, and sets) by returning a reference, as discussed below. Never return a list.

> **Note:** The use of named nested subroutines is dangerous in Perl, especially if they refer to lexical variables in the enclosing scope. Because a PL/Perl function is wrapped in a subroutine, any named subroutine you create will be nested. In general, it is far safer to create anonymous subroutines which you call via a coderef. See the `perldiag` man page for more details.

The syntax of the `CREATE FUNCTION` command requires the function body to be written as a string constant. It is usually most convenient to use dollar quoting (see Section 4.1.2.2) for the string constant. If you choose to use escape string syntax `E''`, you must double the single quote marks (`'`) and backslashes (`\`) used in the body of the function (see Section 4.1.2.1).

Arguments and results are handled as in any other Perl subroutine: arguments are passed in `@_`, and a result value is returned with `return` or as the last expression evaluated in the function.

For example, a function returning the greater of two integer values could be defined as:

1. http://www.perl.com

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    if ($_[0] > $_[1]) { return $_[0]; }
    return $_[1];
$$ LANGUAGE plperl;
```

If an SQL null value is passed to a function, the argument value will appear as "undefined" in Perl. The above function definition will not behave very nicely with null inputs (in fact, it will act as though they are zeroes). We could add `STRICT` to the function definition to make PostgreSQL do something more reasonable: if a null value is passed, the function will not be called at all, but will just return a null result automatically. Alternatively, we could check for undefined inputs in the function body. For example, suppose that we wanted `perl_max` with one null and one nonnull argument to return the nonnull argument, rather than a null value:

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    my ($x,$y) = @_;
    if (! defined $x) {
        if (! defined $y) { return undef; }
        return $y;
    }
    if (! defined $y) { return $x; }
    if ($x > $y) { return $x; }
    return $y;
$$ LANGUAGE plperl;
```

As shown above, to return an SQL null value from a PL/Perl function, return an undefined value. This can be done whether the function is strict or not.

Perl can return PostgreSQL arrays as references to Perl arrays. Here is an example:

```
CREATE OR REPLACE function returns_array()
RETURNS text[][] AS $$
    return [['a"b','c,d'],['e\\f','g']];
$$ LANGUAGE plperl;

select returns_array();
```

Composite-type arguments are passed to the function as references to hashes. The keys of the hash are the attribute names of the composite type. Here is an example:

```
CREATE TABLE employee (
    name text,
    basesalary integer,
    bonus integer
);

CREATE FUNCTION empcomp(employee) RETURNS integer AS $$
    my ($emp) = @_;
    return $emp->{basesalary} + $emp->{bonus};
$$ LANGUAGE plperl;
```

```
SELECT name, empcomp(employee.*) FROM employee;
```

A PL/Perl function can return a composite-type result using the same approach: return a reference to a hash that has the required attributes. For example,

```
CREATE TYPE testrowperl AS (f1 integer, f2 text, f3 text);

CREATE OR REPLACE FUNCTION perl_row() RETURNS testrowperl AS $$
    return {f2 => 'hello', f1 => 1, f3 => 'world'};
$$ LANGUAGE plperl;

SELECT * FROM perl_row();
```

Any columns in the declared result data type that are not present in the hash will be returned as null values.

PL/Perl functions can also return sets of either scalar or composite types. Usually you'll want to return rows one at a time, both to speed up startup time and to keep from queueing up the entire result set in memory. You can do this with `return_next` as illustrated below. Note that after the last `return_next`, you must put either `return` or (better) `return undef`.

```
CREATE OR REPLACE FUNCTION perl_set_int(int)
RETURNS SETOF INTEGER AS $$
    foreach (0..$_[0]) {
        return_next($_);
    }
    return undef;
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set()
RETURNS SETOF testrowperl AS $$
    return_next({ f1 => 1, f2 => 'Hello', f3 => 'World' });
    return_next({ f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' });
    return_next({ f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' });
    return undef;
$$ LANGUAGE plperl;
```

For small result sets, you can return a reference to an array that contains either scalars, references to arrays, or references to hashes for simple types, array types, and composite types, respectively. Here are some simple examples of returning the entire result set as an array reference:

```
CREATE OR REPLACE FUNCTION perl_set_int(int) RETURNS SETOF INTEGER AS $$
    return [0..$_[0]];
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set() RETURNS SETOF testrowperl AS $$
    return [
        { f1 => 1, f2 => 'Hello', f3 => 'World' },
        { f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' },
```

```
        { f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' }
    ];
$$ LANGUAGE plperl;

SELECT * FROM perl_set();
```

If you wish to use the `strict` pragma with your code, the easiest way to do so is to `SET plperl.use_strict` to true. This parameter affects subsequent compilations of PL/Perl functions, but not functions already compiled in the current session. To set the parameter before PL/Perl has been loaded, it is necessary to have added "`plperl`" to the custom_variable_classes list in `postgresql.conf`.

Another way to use the `strict` pragma is to put

```
use strict;
```

in the function body. But this only works in PL/PerlU functions, since `use` is not a trusted operation. In PL/Perl functions you can instead do

```
BEGIN { strict->import(); }
```

39.2. Database Access from PL/Perl

Access to the database itself from your Perl function can be done via the function `spi_exec_query` described below, or via an experimental module `DBD::PgSPI`[2] (also available at CPAN mirror sites[3]). This module makes available a DBI-compliant database-handle named `$pg_dbh` that can be used to perform queries with normal DBI syntax.

PL/Perl provides additional Perl commands:

```
spi_exec_query(query [, max-rows])
spi_query(command)
spi_fetchrow(cursor)
spi_prepare(command, argument types)
spi_exec_prepared(plan)
spi_query_prepared(plan [, attributes], arguments)
spi_cursor_close(cursor)
spi_freeplan(plan)
```

`spi_exec_query` executes an SQL command and returns the entire row set as a reference to an array of hash references. *You should only use this command when you know that the result set will be relatively small.* Here is an example of a query (`SELECT` command) with the optional maximum number of rows:

```
$rv = spi_exec_query('SELECT * FROM my_table', 5);
```

2. http://www.cpan.org/modules/by-module/DBD/APILOS/
3. http://www.cpan.org/SITES.html

This returns up to 5 rows from the table `my_table`. If `my_table` has a column `my_column`, you can get that value from row `$i` of the result like this:

```
$foo = $rv->{rows}[$i]->{my_column};
```

The total number of rows returned from a `SELECT` query can be accessed like this:

```
$nrows = $rv->{processed}
```

Here is an example using a different command type:

```
$query = "INSERT INTO my_table VALUES (1, 'test')";
$rv = spi_exec_query($query);
```

You can then access the command status (e.g., `SPI_OK_INSERT`) like this:

```
$res = $rv->{status};
```

To get the number of rows affected, do:

```
$nrows = $rv->{processed};
```

Here is a complete example:

```
CREATE TABLE test (
    i int,
    v varchar
);

INSERT INTO test (i, v) VALUES (1, 'first line');
INSERT INTO test (i, v) VALUES (2, 'second line');
INSERT INTO test (i, v) VALUES (3, 'third line');
INSERT INTO test (i, v) VALUES (4, 'immortal');

CREATE OR REPLACE FUNCTION test_munge() RETURNS SETOF test AS $$
    my $rv = spi_exec_query('select i, v from test;');
    my $status = $rv->{status};
    my $nrows = $rv->{processed};
    foreach my $rn (0 .. $nrows - 1) {
        my $row = $rv->{rows}[$rn];
        $row->{i} += 200 if defined($row->{i});
        $row->{v} =~ tr/A-Za-z/a-zA-Z/ if (defined($row->{v}));
        return_next($row);
    }
    return undef;
$$ LANGUAGE plperl;

SELECT * FROM test_munge();
```

`spi_query` and `spi_fetchrow` work together as a pair for row sets which may be large, or for cases where you wish to return rows as they arrive. `spi_fetchrow` works *only* with `spi_query`. The following example illustrates how you use them together:

```
CREATE TYPE foo_type AS (the_num INTEGER, the_text TEXT);

CREATE OR REPLACE FUNCTION lotsa_md5 (INTEGER) RETURNS SETOF foo_type AS $$
    use Digest::MD5 qw(md5_hex);
    my $file = '/usr/share/dict/words';
    my $t = localtime;
    elog(NOTICE, "opening file $file at $t" );
    open my $fh, '<', $file # ooh, it's a file access!
```

```
        or elog(ERROR, "can't open $file for reading: $!");
    my @words = <$fh>;
    close $fh;
    $t = localtime;
    elog(NOTICE, "closed file $file at $t");
    chomp(@words);
    my $row;
    my $sth = spi_query("SELECT * FROM generate_series(1,$_[0]) AS b(a)");
    while (defined ($row = spi_fetchrow($sth))) {
        return_next({
            the_num => $row->{a},
            the_text => md5_hex($words[rand @words])
        });
    }
    return;
$$ LANGUAGE plperlu;

SELECT * from lotsa_md5(500);
```

`spi_prepare`, `spi_query_prepared`, `spi_exec_prepared`, and `spi_freeplan` implement the same functionality but for prepared queries. Once a query plan is prepared by a call to `spi_prepare`, the plan can be used instead of the string query, either in `spi_exec_prepared`, where the result is the same as returned by `spi_exec_query`, or in `spi_query_prepared` which returns a cursor exactly as `spi_query` does, which can be later passed to `spi_fetchrow`.

The advantage of prepared queries is that is it possible to use one prepared plan for more than one query execution. After the plan is not needed anymore, it may be freed with `spi_freeplan`:

```
CREATE OR REPLACE FUNCTION init() RETURNS INTEGER AS $$
        $_SHARED{my_plan} = spi_prepare( 'SELECT (now() + $1)::date AS now', 'INTERVAL')
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION add_time( INTERVAL ) RETURNS TEXT AS $$
        return spi_exec_prepared(
                $_SHARED{my_plan},
                $_[0],
        )->{rows}->[0]->{now};
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION done() RETURNS INTEGER AS $$
        spi_freeplan( $_SHARED{my_plan});
        undef $_SHARED{my_plan};
$$ LANGUAGE plperl;

SELECT init();
SELECT add_time('1 day'), add_time('2 days'), add_time('3 days');
SELECT done();

  add_time  |  add_time  |  add_time
------------+------------+------------
 2005-12-10 | 2005-12-11 | 2005-12-12
```

Note that the parameter subscript in `spi_prepare` is defined via $1, $2, $3, etc, so avoid declaring query strings in double quotes that might easily lead to hard-to-catch bugs.

Normally, `spi_fetchrow` should be repeated until it returns `undef`, indicating that there are no more rows to read. The cursor is automatically freed when `spi_fetchrow` returns `undef`. If you do not wish to read all the rows, instead call `spi_cursor_close` to free the cursor. Failure to do so will result in memory leaks.

`elog(`*`level`*`, `*`msg`*`)`

Emit a log or error message. Possible levels are `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING`, and `ERROR`. `ERROR` raises an error condition; if this is not trapped by the surrounding Perl code, the error propagates out to the calling query, causing the current transaction or subtransaction to be aborted. This is effectively the same as the Perl `die` command. The other levels only generate messages of different priority levels. Whether messages of a particular priority are reported to the client, written to the server log, or both is controlled by the log_min_messages and client_min_messages configuration variables. See Chapter 17 for more information.

39.3. Data Values in PL/Perl

The argument values supplied to a PL/Perl function's code are simply the input arguments converted to text form (just as if they had been displayed by a `SELECT` statement). Conversely, the `return` command will accept any string that is acceptable input format for the function's declared return type. So, within the PL/Perl function, all values are just text strings.

39.4. Global Values in PL/Perl

You can use the global hash `%_SHARED` to store data, including code references, between function calls for the lifetime of the current session.

Here is a simple example for shared data:

```
CREATE OR REPLACE FUNCTION set_var(name text, val text) RETURNS text AS $$
    if ($_SHARED{$_[0]} = $_[1]) {
        return 'ok';
    } else {
        return "can't set shared variable $_[0] to $_[1]";
    }
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION get_var(name text) RETURNS text AS $$
    return $_SHARED{$_[0]};
$$ LANGUAGE plperl;

SELECT set_var('sample', 'Hello, PL/Perl!  How's tricks?');
SELECT get_var('sample');
```

Here is a slightly more complicated example using a code reference:

```
CREATE OR REPLACE FUNCTION myfuncs() RETURNS void AS $$
    $_SHARED{myquote} = sub {
        my $arg = shift;
        $arg =~ s/(['\\])/\\$1/g;
        return "'$arg'";
    };
$$ LANGUAGE plperl;

SELECT myfuncs(); /* initializes the function */

/* Set up a function that uses the quote function */

CREATE OR REPLACE FUNCTION use_quote(TEXT) RETURNS text AS $$
    my $text_to_quote = shift;
    my $qfunc = $_SHARED{myquote};
    return &$qfunc($text_to_quote);
$$ LANGUAGE plperl;
```

(You could have replaced the above with the one-liner `return $_SHARED{myquote}->($_[0]);` at the expense of readability.)

39.5. Trusted and Untrusted PL/Perl

Normally, PL/Perl is installed as a "trusted" programming language named `plperl`. In this setup, certain Perl operations are disabled to preserve security. In general, the operations that are restricted are those that interact with the environment. This includes file handle operations, `require`, and `use` (for external modules). There is no way to access internals of the database server process or to gain OS-level access with the permissions of the server process, as a C function can do. Thus, any unprivileged database user may be permitted to use this language.

Here is an example of a function that will not work because file system operations are not allowed for security reasons:

```
CREATE FUNCTION badfunc() RETURNS integer AS $$
    my $tmpfile = "/tmp/badfile";
    open my $fh, '>', $tmpfile
        or elog(ERROR, qq{could not open the file "$tmpfile": $!});
    print $fh "Testing writing to a file\n";
    close $fh or elog(ERROR, qq{could not close the file "$tmpfile": $!});
    return 1;
$$ LANGUAGE plperl;
```

The creation of this function will fail as its use of a forbidden operation will be caught by the validator.

Sometimes it is desirable to write Perl functions that are not restricted. For example, one might want a Perl function that sends mail. To handle these cases, PL/Perl can also be installed as an "untrusted" language (usually called PL/PerlU). In this case the full Perl language is available. If the `createlang` program is used to install the language, the language name `plperlu` will select the untrusted PL/Perl variant.

The writer of a PL/PerlU function must take care that the function cannot be used to do anything unwanted, since it will be able to do anything that could be done by a user logged in as the database administrator. Note that the database system allows only database superusers to create functions in untrusted languages.

If the above function was created by a superuser using the language `plperlu`, execution would succeed.

> **Note:** For security reasons, to stop a leak of privileged operations from PL/PerlU to PL/Perl, these two languages have to run in separate instances of the Perl interpreter. If your Perl installation has been appropriately compiled, this is not a problem. However, not all installations are compiled with the requisite flags. If PostgreSQL detects that this is the case then it will not start a second interpreter, but instead create an error. In consequence, in such an installation, you cannot use both PL/PerlU and PL/Perl in the same backend process. The remedy for this is to obtain a Perl installation created with the appropriate flags, namely either `usemultiplicity` or both `usethreads` and `useithreads`. For more details,see the `perlembed` manual page.

39.6. PL/Perl Triggers

PL/Perl can be used to write trigger functions. In a trigger function, the hash reference `$_TD` contains information about the current trigger event. `$_TD` is a global variable, which gets a separate local value for each invocation of the trigger. The fields of the `$_TD` hash reference are:

`$_TD->{new}{foo}`

`NEW` value of column `foo`

`$_TD->{old}{foo}`

`OLD` value of column `foo`

`$_TD->{name}`

Name of the trigger being called

`$_TD->{event}`

Trigger event: `INSERT`, `UPDATE`, `DELETE`, or `UNKNOWN`

`$_TD->{when}`

When the trigger was called: `BEFORE`, `AFTER`, or `UNKNOWN`

`$_TD->{level}`

The trigger level: `ROW`, `STATEMENT`, or `UNKNOWN`

`$_TD->{relid}`

OID of the table on which the trigger fired

`$_TD->{table_name}`

Name of the table on which the trigger fired

`$_TD->{relname}`

Name of the table on which the trigger fired. This has been deprecated, and could be removed in a future release. Please use $_TD->{table_name} instead.

`$_TD->{table_schema}`

Name of the schema in which the table on which the trigger fired, is

`$_TD->{argc}`

Number of arguments of the trigger function

`@{$_TD->{args}}`

Arguments of the trigger function. Does not exist if `$_TD->{argc}` is 0.

Triggers can return one of the following:

`return;`

Execute the statement

`"SKIP"`

Don't execute the statement

`"MODIFY"`

Indicates that the `NEW` row was modified by the trigger function

Here is an example of a trigger function, illustrating some of the above:

```
CREATE TABLE test (
    i int,
    v varchar
);

CREATE OR REPLACE FUNCTION valid_id() RETURNS trigger AS $$
    if (($_TD->{new}{i} >= 100) || ($_TD->{new}{i} <= 0)) {
        return "SKIP";    # skip INSERT/UPDATE command
    } elsif ($_TD->{new}{v} ne "immortal") {
        $_TD->{new}{v} .= "(modified by trigger)";
        return "MODIFY";  # modify row and execute INSERT/UPDATE command
    } else {
        return;           # execute INSERT/UPDATE command
    }
$$ LANGUAGE plperl;

CREATE TRIGGER test_valid_id_trig
    BEFORE INSERT OR UPDATE ON test
    FOR EACH ROW EXECUTE PROCEDURE valid_id();
```

39.7. Limitations and Missing Features

The following features are currently missing from PL/Perl, but they would make welcome contributions.

- PL/Perl functions cannot call each other directly (because they are anonymous subroutines inside Perl).
- SPI is not yet fully implemented.
- If you are fetching very large data sets using `spi_exec_query`, you should be aware that these will all go into memory. You can avoid this by using `spi_query`/`spi_fetchrow` as illustrated earlier.

  A similar problem occurs if a set-returning function passes a large set of rows back to PostgreSQL via `return`. You can avoid this problem too by instead using `return_next` for each row returned, as shown previously.

Chapter 40. PL/Python - Python Procedural Language

The PL/Python procedural language allows PostgreSQL functions to be written in the Python language[1].

To install PL/Python in a particular database, use `createlang plpythonu` *`dbname`*.

> **Tip:** If a language is installed into `template1`, all subsequently created databases will have the language installed automatically.

As of PostgreSQL 7.4, PL/Python is only available as an "untrusted" language (meaning it does not offer any way of restricting what users can do in it). It has therefore been renamed to `plpythonu`. The trusted variant `plpython` may become available again in future, if a new secure execution mechanism is developed in Python.

> **Note:** Users of source packages must specially enable the build of PL/Python during the installation process. (Refer to the installation instructions for more information.) Users of binary packages might find PL/Python in a separate subpackage.

40.1. PL/Python Functions

Functions in PL/Python are declared via the standard *CREATE FUNCTION* syntax:

```
CREATE FUNCTION funcname (argument-list)
  RETURNS return-type
AS $$
  # PL/Python function body
$$ LANGUAGE plpythonu;
```

The body of a function is simply a Python script. When the function is called, its arguments are passed as elements of the array `args[]`; named arguments are also passed as ordinary variables to the Python script. The result is returned from the Python code in the usual way, with `return` or `yield` (in case of a result-set statement).

For example, a function to return the greater of two integers can be defined as:

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if a > b:
    return a
  return b
$$ LANGUAGE plpythonu;
```

1. http://www.python.org

The Python code that is given as the body of the function definition is transformed into a Python function. For example, the above results in

```
def __plpython_procedure_pymax_23456():
  if a > b:
    return a
  return b
```

assuming that 23456 is the OID assigned to the function by PostgreSQL.

The PostgreSQL function parameters are available in the global `args` list. In the `pymax` example, `args[0]` contains whatever was passed in as the first argument and `args[1]` contains the second argument's value. Alternatively, one can use named parameters as shown in the example above. Use of named parameters is usually more readable.

If an SQL null value is passed to a function, the argument value will appear as `None` in Python. The above function definition will return the wrong answer for null inputs. We could add `STRICT` to the function definition to make PostgreSQL do something more reasonable: if a null value is passed, the function will not be called at all, but will just return a null result automatically. Alternatively, we could check for null inputs in the function body:

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if (a is None) or (b is None):
    return None
  if a > b:
    return a
  return b
$$ LANGUAGE plpythonu;
```

As shown above, to return an SQL null value from a PL/Python function, return the value `None`. This can be done whether the function is strict or not.

Composite-type arguments are passed to the function as Python mappings. The element names of the mapping are the attribute names of the composite type. If an attribute in the passed row has the null value, it has the value `None` in the mapping. Here is an example:

```
CREATE TABLE employee (
  name text,
  salary integer,
  age integer
);

CREATE FUNCTION overpaid (e employee)
  RETURNS boolean
AS $$
  if e["salary"] > 200000:
    return True
  if (e["age"] < 30) and (e["salary"] > 100000):
    return True
  return False
$$ LANGUAGE plpythonu;
```

There are multiple ways to return row or composite types from a Python function. The following examples assume we have:

```
CREATE TYPE named_value AS (
  name   text,
  value  integer
);
```

A composite result can be returned as a:

Sequence type (a tuple or list, but not a set because it is not indexable)

Returned sequence objects must have the same number of items as the composite result type has fields. The item with index 0 is assigned to the first field of the composite type, 1 to the second and so on. For example:

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  return [ name, value ]
  # or alternatively, as tuple: return ( name, value )
$$ LANGUAGE plpythonu;
```

To return a SQL null for any column, insert `None` at the corresponding position.

Mapping (dictionary)

The value for each result type column is retrieved from the mapping with the column name as key. Example:

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  return { "name": name, "value": value }
$$ LANGUAGE plpythonu;
```

Any extra dictionary key/value pairs are ignored. Missing keys are treated as errors. To return a SQL null value for any column, insert `None` with the corresponding column name as the key.

Object (any object providing method `__getattr__`)

This works the same as a mapping. Example:

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  class named_value:
    def __init__ (self, n, v):
      self.name = n
      self.value = v
  return named_value(name, value)

  # or simply
  class nv: pass
  nv.name = name
  nv.value = value
```

```
  return nv
$$ LANGUAGE plpythonu;
```

If you do not provide a return value, Python returns the default `None`. PL/Python translates Python's `None` into the SQL null value.

A PL/Python function can also return sets of scalar or composite types. There are several ways to achieve this because the returned object is internally turned into an iterator. The following examples assume we have composite type:

```
CREATE TYPE greeting AS (
  how text,
  who text
);
```

A set result can be returned from a:

Sequence type (tuple, list, set)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  # return tuple containing lists as composite types
  # all other combinations work also
  return ( [ how, "World" ], [ how, "PostgreSQL" ], [ how, "PL/Python" ] )
$$ LANGUAGE plpythonu;
```

Iterator (any object providing `__iter__` and `next` methods)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  class producer:
    def __init__ (self, how, who):
      self.how = how
      self.who = who
      self.ndx = -1

    def __iter__ (self):
      return self

    def next (self):
      self.ndx += 1
      if self.ndx == len(self.who):
        raise StopIteration
      return ( self.how, self.who[self.ndx] )

  return producer(how, [ "World", "PostgreSQL", "PL/Python" ])
$$ LANGUAGE plpythonu;
```

Generator (`yield`)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  for who in [ "World", "PostgreSQL", "PL/Python" ]:
    yield ( how, who )
$$ LANGUAGE plpythonu;
```

Warning

Currently, due to Python bug #1483133[2], some debug versions of Python 2.4 (configured and compiled with option `--with-pydebug`) are known to crash the PostgreSQL server when using an iterator to return a set result. Unpatched versions of Fedora 4 contain this bug. It does not happen in production versions of Python or on patched versions of Fedora 4.

The global dictionary `SD` is available to store data between function calls. This variable is private static data. The global dictionary `GD` is public data, available to all Python functions within a session. Use with care.

Each function gets its own execution environment in the Python interpreter, so that global data and function arguments from `myfunc` are not available to `myfunc2`. The exception is the data in the `GD` dictionary, as mentioned above.

40.2. Trigger Functions

When a function is used as a trigger, the dictionary `TD` contains trigger-related values. The trigger rows are in `TD["new"]` and/or `TD["old"]` depending on the trigger event. `TD["event"]` contains the event as a string (`INSERT`, `UPDATE`, `DELETE`, or `UNKNOWN`). `TD["when"]` contains one of `BEFORE`, `AFTER`, and `UNKNOWN`. `TD["level"]` contains one of `ROW`, `STATEMENT`, and `UNKNOWN`. `TD["name"]` contains the trigger name, `TD["table_name"]` contains the name of the table on which the trigger occurred, `TD["table_schema"]` contains the schema of the table on which the trigger occurred, `TD["name"]` contains the trigger name, and `TD["relid"]` contains the OID of the table on which the trigger occurred. If the `CREATE TRIGGER` command included arguments, they are available in `TD["args"][0]` to `TD["args"][(`*n*`-1)]`.

If `TD["when"]` is `BEFORE`, you may return `None` or `"OK"` from the Python function to indicate the row is unmodified, `"SKIP"` to abort the event, or `"MODIFY"` to indicate you've modified the row.

40.3. Database Access

The PL/Python language module automatically imports a Python module called `plpy`. The functions and constants in this module are available to you in the Python code as `plpy.`*`foo`*. At present `plpy` implements the functions `plpy.debug(`*`msg`*`)`, `plpy.log(`*`msg`*`)`,

`plpy.info(msg)`, `plpy.notice(msg)`, `plpy.warning(msg)`, `plpy.error(msg)`, and `plpy.fatal(msg)`. `plpy.error` and `plpy.fatal` actually raise a Python exception which, if uncaught, propagates out to the calling query, causing the current transaction or subtransaction to be aborted. `raise plpy.ERROR(msg)` and `raise plpy.FATAL(msg)` are equivalent to calling `plpy.error` and `plpy.fatal`, respectively. The other functions only generate messages of different priority levels. Whether messages of a particular priority are reported to the client, written to the server log, or both is controlled by the log_min_messages and client_min_messages configuration variables. See Chapter 17 for more information.

Additionally, the `plpy` module provides two functions called `execute` and `prepare`. Calling `plpy.execute` with a query string and an optional limit argument causes that query to be run and the result to be returned in a result object. The result object emulates a list or dictionary object. The result object can be accessed by row number and column name. It has these additional methods: `nrows` which returns the number of rows returned by the query, and `status` which is the `SPI_execute()` return value. The result object can be modified.

For example,

```
rv = plpy.execute("SELECT * FROM my_table", 5)
```

returns up to 5 rows from `my_table`. If `my_table` has a column `my_column`, it would be accessed as

```
foo = rv[i]["my_column"]
```

The second function, `plpy.prepare`, prepares the execution plan for a query. It is called with a query string and a list of parameter types, if you have parameter references in the query. For example:

```
plan = plpy.prepare("SELECT last_name FROM my_users WHERE first_name = $1", [ "text" ])
```

`text` is the type of the variable you will be passing for `$1`. After preparing a statement, you use the function `plpy.execute` to run it:

```
rv = plpy.execute(plan, [ "name" ], 5)
```

The third argument is the limit and is optional.

When you prepare a plan using the PL/Python module it is automatically saved. Read the SPI documentation (Chapter 41) for a description of what this means. In order to make effective use of this across function calls one needs to use one of the persistent storage dictionaries `SD` or `GD` (see Section 40.1). For example:

```
CREATE FUNCTION usesavedplan() RETURNS trigger AS $$
    if SD.has_key("plan"):
        plan = SD["plan"]
    else:
        plan = plpy.prepare("SELECT 1")
        SD["plan"] = plan
    # rest of function
$$ LANGUAGE plpythonu;
```

Chapter 41. Server Programming Interface

The *Server Programming Interface* (SPI) gives writers of user-defined C functions the ability to run SQL commands inside their functions. SPI is a set of interface functions to simplify access to the parser, planner, optimizer, and executor. SPI also does some memory management.

> **Note:** The available procedural languages provide various means to execute SQL commands from procedures. Most of these facilities are based on SPI, so this documentation might be of use for users of those languages as well.

To avoid misunderstanding we'll use the term "function" when we speak of SPI interface functions and "procedure" for a user-defined C-function that is using SPI.

Note that if a command invoked via SPI fails, then control will not be returned to your procedure. Rather, the transaction or subtransaction in which your procedure executes will be rolled back. (This may seem surprising given that the SPI functions mostly have documented error-return conventions. Those conventions only apply for errors detected within the SPI functions themselves, however.) It is possible to recover control after an error by establishing your own subtransaction surrounding SPI calls that might fail. This is not currently documented because the mechanisms required are still in flux.

SPI functions return a nonnegative result on success (either via a returned integer value or in the global variable `SPI_result`, as described below). On error, a negative result or `NULL` will be returned.

Source code files that use SPI must include the header file `executor/spi.h`.

41.1. Interface Functions

SPI_connect

Name

`SPI_connect` — connect a procedure to the SPI manager

Synopsis

```
int SPI_connect(void)
```

Description

`SPI_connect` opens a connection from a procedure invocation to the SPI manager. You must call this function if you want to execute commands through SPI. Some utility SPI functions may be called from unconnected procedures.

If your procedure is already connected, `SPI_connect` will return the error code SPI_ERROR_CONNECT. This could happen if a procedure that has called `SPI_connect` directly calls another procedure that calls `SPI_connect`. While recursive calls to the SPI manager are permitted when an SQL command called through SPI invokes another function that uses SPI, directly nested calls to `SPI_connect` and `SPI_finish` are forbidden. (But see `SPI_push` and `SPI_pop`.)

Return Value

`SPI_OK_CONNECT`

on success

`SPI_ERROR_CONNECT`

on error

SPI_finish

Name

`SPI_finish` — disconnect a procedure from the SPI manager

Synopsis

```
int SPI_finish(void)
```

Description

`SPI_finish` closes an existing connection to the SPI manager. You must call this function after completing the SPI operations needed during your procedure's current invocation. You do not need to worry about making this happen, however, if you abort the transaction via `elog(ERROR)`. In that case SPI will clean itself up automatically.

If `SPI_finish` is called without having a valid connection, it will return `SPI_ERROR_UNCONNECTED`. There is no fundamental problem with this; it means that the SPI manager has nothing to do.

Return Value

`SPI_OK_FINISH`

if properly disconnected

`SPI_ERROR_UNCONNECTED`

if called from an unconnected procedure

SPI_push

Name

SPI_push — push SPI stack to allow recursive SPI usage

Synopsis

```
void SPI_push(void)
```

Description

SPI_push should be called before executing another procedure that might itself wish to use SPI. After SPI_push, SPI is no longer in a "connected" state, and SPI function calls will be rejected unless a fresh SPI_connect is done. This ensures a clean separation between your procedure's SPI state and that of another procedure you call. After the other procedure returns, call SPI_pop to restore access to your own SPI state.

Note that SPI_execute and related functions automatically do the equivalent of SPI_push before passing control back to the SQL execution engine, so it is not necessary for you to worry about this when using those functions. Only when you are directly calling arbitrary code that might contain SPI_connect calls do you need to issue SPI_push and SPI_pop.

SPI_pop

Name

`SPI_pop` — pop SPI stack to return from recursive SPI usage

Synopsis

```
void SPI_pop(void)
```

Description

`SPI_pop` pops the previous environment from the SPI call stack. See `SPI_push`.

SPI_execute

Name

SPI_execute — execute a command

Synopsis

```
int SPI_execute(const char * command, bool read_only, long count)
```

Description

SPI_execute executes the specified SQL command for count rows. If read_only is true, the command must be read-only, and execution overhead is somewhat reduced.

This function may only be called from a connected procedure.

If count is zero then the command is executed for all rows that it applies to. If count is greater than 0, then the number of rows for which the command will be executed is restricted (much like a LIMIT clause). For example,

```
SPI_execute("INSERT INTO foo SELECT * FROM bar", false, 5);
```

will allow at most 5 rows to be inserted into the table.

You may pass multiple commands in one string. SPI_execute returns the result for the command executed last. The count limit applies to each command separately, but it is not applied to hidden commands generated by rules.

When read_only is false, SPI_execute increments the command counter and computes a new *snapshot* before executing each command in the string. The snapshot does not actually change if the current transaction isolation level is SERIALIZABLE, but in READ COMMITTED mode the snapshot update allows each command to see the results of newly committed transactions from other sessions. This is essential for consistent behavior when the commands are modifying the database.

When read_only is true, SPI_execute does not update either the snapshot or the command counter, and it allows only plain SELECT commands to appear in the command string. The commands are executed using the snapshot previously established for the surrounding query. This execution mode is somewhat faster than the read/write mode due to eliminating per-command overhead. It also allows genuinely *stable* functions to be built: since successive executions will all use the same snapshot, there will be no change in the results.

It is generally unwise to mix read-only and read-write commands within a single function using SPI; that could result in very confusing behavior, since the read-only queries would not see the results of any database updates done by the read-write queries.

The actual number of rows for which the (last) command was executed is returned in the global variable SPI_processed. If the return value of the function is SPI_OK_SELECT, SPI_OK_INSERT_RETURNING, SPI_OK_DELETE_RETURNING, or SPI_OK_UPDATE_RETURNING, then you may use the global pointer

`SPITupleTable *SPI_tuptable` to access the result rows. Some utility commands (such as `EXPLAIN`) also return row sets, and `SPI_tuptable` will contain the result in these cases too.

The structure `SPITupleTable` is defined thus:

```
typedef struct
{
    MemoryContext tuptabcxt;    /* memory context of result table */
    uint32      alloced;        /* number of alloced vals */
    uint32      free;           /* number of free vals */
    TupleDesc   tupdesc;        /* row descriptor */
    HeapTuple  *vals;           /* rows */
} SPITupleTable;
```

`vals` is an array of pointers to rows. (The number of valid entries is given by `SPI_processed`.) `tupdesc` is a row descriptor which you may pass to SPI functions dealing with rows. `tuptabcxt`, `alloced`, and `free` are internal fields not intended for use by SPI callers.

`SPI_finish` frees all `SPITupleTable`s allocated during the current procedure. You can free a particular result table earlier, if you are done with it, by calling `SPI_freetuptable`.

Arguments

`const char * command`

string containing command to execute

`bool read_only`

`true` for read-only execution

`long count`

maximum number of rows to process or return

Return Value

If the execution of the command was successful then one of the following (nonnegative) values will be returned:

`SPI_OK_SELECT`

if a `SELECT` (but not `SELECT INTO`) was executed

`SPI_OK_SELINTO`

if a `SELECT INTO` was executed

`SPI_OK_INSERT`

if an `INSERT` was executed

`SPI_OK_DELETE`

if a `DELETE` was executed

`SPI_OK_UPDATE`

if an `UPDATE` was executed

`SPI_OK_INSERT_RETURNING`

if an `INSERT RETURNING` was executed

`SPI_OK_DELETE_RETURNING`

if a `DELETE RETURNING` was executed

`SPI_OK_UPDATE_RETURNING`

if an `UPDATE RETURNING` was executed

`SPI_OK_UTILITY`

if a utility command (e.g., `CREATE TABLE`) was executed

On error, one of the following negative values is returned:

`SPI_ERROR_ARGUMENT`

if `command` is `NULL` or `count` is less than 0

`SPI_ERROR_COPY`

if `COPY TO stdout` or `COPY FROM stdin` was attempted

`SPI_ERROR_CURSOR`

if `DECLARE`, `CLOSE`, or `FETCH` was attempted

`SPI_ERROR_TRANSACTION`

if any command involving transaction manipulation was attempted (`BEGIN`, `COMMIT`, `ROLLBACK`, `SAVEPOINT`, `PREPARE TRANSACTION`, `COMMIT PREPARED`, `ROLLBACK PREPARED`, or any variant thereof)

`SPI_ERROR_OPUNKNOWN`

if the command type is unknown (shouldn't happen)

`SPI_ERROR_UNCONNECTED`

if called from an unconnected procedure

Notes

The functions `SPI_execute`, `SPI_exec`, `SPI_execute_plan`, and `SPI_execp` change both `SPI_processed` and `SPI_tuptable` (just the pointer, not the contents of the structure). Save these two global variables into local procedure variables if you need to access the result table of `SPI_execute` or a related function across later calls.

SPI_exec

Name

`SPI_exec` — execute a read/write command

Synopsis

```
int SPI_exec(const char * command, long count)
```

Description

`SPI_exec` is the same as `SPI_execute`, with the latter's `read_only` parameter always taken as `false`.

Arguments

`const char * command`

string containing command to execute

`long count`

maximum number of rows to process or return

Return Value

See `SPI_execute`.

SPI_prepare

Name

SPI_prepare — prepare a plan for a command, without executing it yet

Synopsis

```
void * SPI_prepare(const char * command, int nargs, Oid * argtypes)
```

Description

SPI_prepare creates and returns an execution plan for the specified command but doesn't execute the command. This function should only be called from a connected procedure.

When the same or a similar command is to be executed repeatedly, it may be advantageous to perform the planning only once. SPI_prepare converts a command string into an execution plan that can be executed repeatedly using SPI_execute_plan.

A prepared command can be generalized by writing parameters ($1, $2, etc.) in place of what would be constants in a normal command. The actual values of the parameters are then specified when SPI_execute_plan is called. This allows the prepared command to be used over a wider range of situations than would be possible without parameters.

The plan returned by SPI_prepare can be used only in the current invocation of the procedure, since SPI_finish frees memory allocated for a plan. But a plan can be saved for longer using the function SPI_saveplan.

Arguments

`const char * command`

command string

`int nargs`

number of input parameters ($1, $2, etc.)

`Oid * argtypes`

pointer to an array containing the OIDs of the data types of the parameters

Return Value

SPI_prepare returns a non-null pointer to an execution plan. On error, NULL will be returned, and SPI_result will be set to one of the same error codes used by SPI_execute, except that it is set to

`SPI_ERROR_ARGUMENT` if `command` is `NULL`, or if `nargs` is less than 0, or if `nargs` is greater than 0 and `argtypes` is `NULL`.

Notes

There is a disadvantage to using parameters: since the planner does not know the values that will be supplied for the parameters, it may make worse planning choices than it would make for a normal command with all constants visible.

SPI_getargcount

Name

`SPI_getargcount` — return the number of arguments needed by a plan prepared by `SPI_prepare`

Synopsis

```
int SPI_getargcount(void * plan)
```

Description

`SPI_getargcount` returns the number of arguments needed to execute a plan prepared by `SPI_prepare`.

Arguments

`void * plan`

execution plan (returned by `SPI_prepare`)

Return Value

The expected argument count for the `plan`, or `SPI_ERROR_ARGUMENT` if the `plan` is `NULL`

SPI_getargtypeid

Name

SPI_getargtypeid — return the data type OID for an argument of a plan prepared by SPI_prepare

Synopsis

```
Oid SPI_getargtypeid(void * plan, int argIndex)
```

Description

SPI_getargtypeid returns the OID representing the type id for the argIndex'th argument of a plan prepared by SPI_prepare. First argument is at index zero.

Arguments

`void * plan`

execution plan (returned by SPI_prepare)

`int argIndex`

zero based index of the argument

Return Value

The type id of the argument at the given index, or SPI_ERROR_ARGUMENT if the plan is NULL or argIndex is less than 0 or not less than the number of arguments declared for the plan

SPI_is_cursor_plan

Name

SPI_is_cursor_plan — return true if a plan prepared by SPI_prepare can be used with SPI_cursor_open

Synopsis

```
bool SPI_is_cursor_plan(void * plan)
```

Description

SPI_is_cursor_plan returns true if a plan prepared by SPI_prepare can be passed as an argument to SPI_cursor_open, or false if that is not the case. The criteria are that the plan represents one single command and that this command returns tuples to the caller; for example, SELECT is allowed unless it contains an INTO clause, and UPDATE is allowed only if it contains a RETURNING clause.

Arguments

void * plan

execution plan (returned by SPI_prepare)

Return Value

true or false to indicate if the plan can produce a cursor or not, or SPI_ERROR_ARGUMENT if the plan is NULL

SPI_execute_plan

Name

SPI_execute_plan — execute a plan prepared by SPI_prepare

Synopsis

```
int SPI_execute_plan(void * plan, Datum * values, const char * nulls,
                     bool read_only, long count)
```

Description

SPI_execute_plan executes a plan prepared by SPI_prepare. read_only and count have the same interpretation as in SPI_execute.

Arguments

void * plan

execution plan (returned by SPI_prepare)

Datum * values

An array of actual parameter values. Must have same length as the plan's number of arguments.

const char * nulls

An array describing which parameters are null. Must have same length as the plan's number of arguments. n indicates a null value (entry in values will be ignored); a space indicates a nonnull value (entry in values is valid).

If nulls is NULL then SPI_execute_plan assumes that no parameters are null.

bool read_only

true for read-only execution

long count

maximum number of rows to process or return

Return Value

The return value is the same as for `SPI_execute`, with the following additional possible error (negative) results:

`SPI_ERROR_ARGUMENT`

if `plan` is `NULL` or `count` is less than 0

`SPI_ERROR_PARAM`

if `values` is `NULL` and `plan` was prepared with some parameters

`SPI_processed` and `SPI_tuptable` are set as in `SPI_execute` if successful.

Notes

If one of the objects (a table, function, etc.) referenced by the prepared plan is dropped during the session then the result of `SPI_execute_plan` for this plan will be unpredictable.

SPI_execp

Name

SPI_execp — execute a plan in read/write mode

Synopsis

```
int SPI_execp(void * plan, Datum * values, const char * nulls, long count)
```

Description

SPI_execp is the same as SPI_execute_plan, with the latter's read_only parameter always taken as false.

Arguments

void * plan

execution plan (returned by SPI_prepare)

Datum * values

An array of actual parameter values. Must have same length as the plan's number of arguments.

const char * nulls

An array describing which parameters are null. Must have same length as the plan's number of arguments. n indicates a null value (entry in values will be ignored); a space indicates a nonnull value (entry in values is valid).

If nulls is NULL then SPI_execp assumes that no parameters are null.

long count

maximum number of rows to process or return

Return Value

See SPI_execute_plan.

SPI_processed and SPI_tuptable are set as in SPI_execute if successful.

SPI_cursor_open

Name

SPI_cursor_open — set up a cursor using a plan created with SPI_prepare

Synopsis

```
Portal SPI_cursor_open(const char * name, void * plan,
                       Datum * values, const char * nulls,
                       bool read_only)
```

Description

SPI_cursor_open sets up a cursor (internally, a portal) that will execute a plan prepared by SPI_prepare. The parameters have the same meanings as the corresponding parameters to SPI_execute_plan.

Using a cursor instead of executing the plan directly has two benefits. First, the result rows can be retrieved a few at a time, avoiding memory overrun for queries that return many rows. Second, a portal can outlive the current procedure (it can, in fact, live to the end of the current transaction). Returning the portal name to the procedure's caller provides a way of returning a row set as result.

Arguments

`const char * name`

name for portal, or NULL to let the system select a name

`void * plan`

execution plan (returned by SPI_prepare)

`Datum * values`

An array of actual parameter values. Must have same length as the plan's number of arguments.

`const char * nulls`

An array describing which parameters are null. Must have same length as the plan's number of arguments. n indicates a null value (entry in values will be ignored); a space indicates a nonnull value (entry in values is valid).

If nulls is NULL then SPI_cursor_open assumes that no parameters are null.

`bool read_only`

true for read-only execution

Return Value

pointer to portal containing the cursor, or `NULL` on error

SPI_cursor_find

Name

`SPI_cursor_find` — find an existing cursor by name

Synopsis

```
Portal SPI_cursor_find(const char * name)
```

Description

`SPI_cursor_find` finds an existing portal by name. This is primarily useful to resolve a cursor name returned as text by some other function.

Arguments

`const char * name`

name of the portal

Return Value

pointer to the portal with the specified name, or `NULL` if none was found

SPI_cursor_fetch

Name

`SPI_cursor_fetch` — fetch some rows from a cursor

Synopsis

```
void SPI_cursor_fetch(Portal portal, bool forward, long count)
```

Description

`SPI_cursor_fetch` fetches some rows from a cursor. This is equivalent to the SQL command `FETCH`.

Arguments

`Portal portal`

portal containing the cursor

`bool forward`

true for fetch forward, false for fetch backward

`long count`

maximum number of rows to fetch

Return Value

`SPI_processed` and `SPI_tuptable` are set as in `SPI_execute` if successful.

SPI_cursor_move

Name

`SPI_cursor_move` — move a cursor

Synopsis

```
void SPI_cursor_move(Portal portal, bool forward, long count)
```

Description

`SPI_cursor_move` skips over some number of rows in a cursor. This is equivalent to the SQL command `MOVE`.

Arguments

`Portal portal`

portal containing the cursor

`bool forward`

true for move forward, false for move backward

`long count`

maximum number of rows to move

SPI_cursor_close

Name

`SPI_cursor_close` — close a cursor

Synopsis

```
void SPI_cursor_close(Portal portal)
```

Description

`SPI_cursor_close` closes a previously created cursor and releases its portal storage.

All open cursors are closed automatically at the end of a transaction. `SPI_cursor_close` need only be invoked if it is desirable to release resources sooner.

Arguments

`Portal portal`

portal containing the cursor

SPI_saveplan

Name

SPI_saveplan — save a plan

Synopsis

```
void * SPI_saveplan(void * plan)
```

Description

SPI_saveplan saves a passed plan (prepared by SPI_prepare) in memory protected from freeing by SPI_finish and by the transaction manager and returns a pointer to the saved plan. This gives you the ability to reuse prepared plans in the subsequent invocations of your procedure in the current session.

Arguments

`void * plan`

the plan to be saved

Return Value

Pointer to the saved plan; NULL if unsuccessful. On error, SPI_result is set thus:

SPI_ERROR_ARGUMENT

if plan is NULL

SPI_ERROR_UNCONNECTED

if called from an unconnected procedure

Notes

If one of the objects (a table, function, etc.) referenced by the prepared plan is dropped during the session then the results of SPI_execute_plan for this plan will be unpredictable.

41.2. Interface Support Functions

The functions described here provide an interface for extracting information from result sets returned by `SPI_execute` and other SPI functions.

All functions described in this section may be used by both connected and unconnected procedures.

SPI_fname

Name

`SPI_fname` — determine the column name for the specified column number

Synopsis

```
char * SPI_fname(TupleDesc rowdesc, int colnumber)
```

Description

`SPI_fname` returns a copy of the column name of the specified column. (You can use `pfree` to release the copy of the name when you don't need it anymore.)

Arguments

`TupleDesc rowdesc`

input row description

`int colnumber`

column number (count starts at 1)

Return Value

The column name; `NULL` if `colnumber` is out of range. `SPI_result` set to `SPI_ERROR_NOATTRIBUTE` on error.

SPI_fnumber

Name

`SPI_fnumber` — determine the column number for the specified column name

Synopsis

```
int SPI_fnumber(TupleDesc rowdesc, const char * colname)
```

Description

`SPI_fnumber` returns the column number for the column with the specified name.

If `colname` refers to a system column (e.g., `oid`) then the appropriate negative column number will be returned. The caller should be careful to test the return value for exact equality to `SPI_ERROR_NOATTRIBUTE` to detect an error; testing the result for less than or equal to 0 is not correct unless system columns should be rejected.

Arguments

`TupleDesc rowdesc`

input row description

`const char * colname`

column name

Return Value

Column number (count starts at 1), or `SPI_ERROR_NOATTRIBUTE` if the named column was not found.

SPI_getvalue

Name

`SPI_getvalue` — return the string value of the specified column

Synopsis

```
char * SPI_getvalue(HeapTuple row, TupleDesc rowdesc, int colnumber)
```

Description

`SPI_getvalue` returns the string representation of the value of the specified column.

The result is returned in memory allocated using `palloc`. (You can use `pfree` to release the memory when you don't need it anymore.)

Arguments

`HeapTuple row`

input row to be examined

`TupleDesc rowdesc`

input row description

`int colnumber`

column number (count starts at 1)

Return Value

Column value, or `NULL` if the column is null, `colnumber` is out of range (`SPI_result` is set to `SPI_ERROR_NOATTRIBUTE`), or no output function is available (`SPI_result` is set to `SPI_ERROR_NOOUTFUNC`).

SPI_getbinval

Name

`SPI_getbinval` — return the binary value of the specified column

Synopsis

```
Datum SPI_getbinval(HeapTuple row, TupleDesc rowdesc, int colnumber, bool * isnull)
```

Description

`SPI_getbinval` returns the value of the specified column in the internal form (as type `Datum`).

This function does not allocate new space for the datum. In the case of a pass-by-reference data type, the return value will be a pointer into the passed row.

Arguments

`HeapTuple row`

input row to be examined

`TupleDesc rowdesc`

input row description

`int colnumber`

column number (count starts at 1)

`bool * isnull`

flag for a null value in the column

Return Value

The binary value of the column is returned. The variable pointed to by `isnull` is set to true if the column is null, else to false.

`SPI_result` is set to `SPI_ERROR_NOATTRIBUTE` on error.

SPI_gettype

Name

`SPI_gettype` — return the data type name of the specified column

Synopsis

```
char * SPI_gettype(TupleDesc rowdesc, int colnumber)
```

Description

`SPI_gettype` returns a copy of the data type name of the specified column. (You can use `pfree` to release the copy of the name when you don't need it anymore.)

Arguments

`TupleDesc rowdesc`

input row description

`int colnumber`

column number (count starts at 1)

Return Value

The data type name of the specified column, or `NULL` on error. `SPI_result` is set to `SPI_ERROR_NOATTRIBUTE` on error.

SPI_gettypeid

Name

`SPI_gettypeid` — return the data type OID of the specified column

Synopsis

```
Oid SPI_gettypeid(TupleDesc rowdesc, int colnumber)
```

Description

`SPI_gettypeid` returns the OID of the data type of the specified column.

Arguments

`TupleDesc rowdesc`

input row description

`int colnumber`

column number (count starts at 1)

Return Value

The OID of the data type of the specified column or `InvalidOid` on error. On error, `SPI_result` is set to `SPI_ERROR_NOATTRIBUTE`.

SPI_getrelname

Name

`SPI_getrelname` — return the name of the specified relation

Synopsis

```
char * SPI_getrelname(Relation rel)
```

Description

`SPI_getrelname` returns a copy of the name of the specified relation. (You can use `pfree` to release the copy of the name when you don't need it anymore.)

Arguments

`Relation rel`

input relation

Return Value

The name of the specified relation.

SPI_getnspname

Name

`SPI_getnspname` — return the namespace of the specified relation

Synopsis

```
char * SPI_getnspname(Relation rel)
```

Description

`SPI_getnspname` returns a copy of the name of the namespace that the specified `Relation` belongs to. This is equivalent to the relation's schema. You should `pfree` the return value of this function when you are finished with it.

Arguments

`Relation rel`

input relation

Return Value

The name of the specified relation's namespace.

41.3. Memory Management

PostgreSQL allocates memory within *memory contexts*, which provide a convenient method of managing allocations made in many different places that need to live for differing amounts of time. Destroying a context releases all the memory that was allocated in it. Thus, it is not necessary to keep track of individual objects to avoid memory leaks; instead only a relatively small number of contexts have to be managed. `palloc` and related functions allocate memory from the "current" context.

`SPI_connect` creates a new memory context and makes it current. `SPI_finish` restores the previous current memory context and destroys the context created by `SPI_connect`. These actions ensure that transient memory allocations made inside your procedure are reclaimed at procedure exit, avoiding memory leakage.

However, if your procedure needs to return an object in allocated memory (such as a value of a pass-by-reference data type), you cannot allocate that memory using `palloc`, at least not while you are connected to SPI. If you try, the object will be deallocated by `SPI_finish`, and your procedure will not work reliably. To solve this problem, use `SPI_palloc` to allocate memory for your return object. `SPI_palloc` allocates memory in the "upper executor context", that is, the memory context that was current when `SPI_connect` was called, which is precisely the right context for a value returned from your procedure.

If `SPI_palloc` is called while the procedure is not connected to SPI, then it acts the same as a normal `palloc`. Before a procedure connects to the SPI manager, the current memory context is the upper executor context, so all allocations made by the procedure via `palloc` or by SPI utility functions are made in this context.

When `SPI_connect` is called, the private context of the procedure, which is created by `SPI_connect`, is made the current context. All allocations made by `palloc`, `repalloc`, or SPI utility functions (except for `SPI_copytuple`, `SPI_returntuple`, `SPI_modifytuple`, and `SPI_palloc`) are made in this context. When a procedure disconnects from the SPI manager (via `SPI_finish`) the current context is restored to the upper executor context, and all allocations made in the procedure memory context are freed and cannot be used any more.

All functions described in this section may be used by both connected and unconnected procedures. In an unconnected procedure, they act the same as the underlying ordinary server functions (`palloc`, etc.).

SPI_palloc

Name

`SPI_palloc` — allocate memory in the upper executor context

Synopsis

```
void * SPI_palloc(Size size)
```

Description

`SPI_palloc` allocates memory in the upper executor context.

Arguments

`Size size`

size in bytes of storage to allocate

Return Value

pointer to new storage space of the specified size

SPI_repalloc

Name

`SPI_repalloc` — reallocate memory in the upper executor context

Synopsis

```
void * SPI_repalloc(void * pointer, Size size)
```

Description

`SPI_repalloc` changes the size of a memory segment previously allocated using `SPI_palloc`.

This function is no longer different from plain `repalloc`. It's kept just for backward compatibility of existing code.

Arguments

`void * pointer`

pointer to existing storage to change

`Size size`

size in bytes of storage to allocate

Return Value

pointer to new storage space of specified size with the contents copied from the existing area

SPI_pfree

Name

`SPI_pfree` — free memory in the upper executor context

Synopsis

```
void SPI_pfree(void * pointer)
```

Description

`SPI_pfree` frees memory previously allocated using `SPI_palloc` or `SPI_repalloc`.

This function is no longer different from plain `pfree`. It's kept just for backward compatibility of existing code.

Arguments

`void * pointer`

pointer to existing storage to free

SPI_copytuple

Name

`SPI_copytuple` — make a copy of a row in the upper executor context

Synopsis

```
HeapTuple SPI_copytuple(HeapTuple row)
```

Description

`SPI_copytuple` makes a copy of a row in the upper executor context. This is normally used to return a modified row from a trigger. In a function declared to return a composite type, use `SPI_returntuple` instead.

Arguments

`HeapTuple row`

row to be copied

Return Value

the copied row; `NULL` only if `tuple` is `NULL`

SPI_returntuple

Name

`SPI_returntuple` — prepare to return a tuple as a Datum

Synopsis

```
HeapTupleHeader SPI_returntuple(HeapTuple row, TupleDesc rowdesc)
```

Description

`SPI_returntuple` makes a copy of a row in the upper executor context, returning it in the form of a row type `Datum`. The returned pointer need only be converted to `Datum` via `PointerGetDatum` before returning.

Note that this should be used for functions that are declared to return composite types. It is not used for triggers; use `SPI_copytuple` for returning a modified row in a trigger.

Arguments

`HeapTuple row`

row to be copied

`TupleDesc rowdesc`

descriptor for row (pass the same descriptor each time for most effective caching)

Return Value

`HeapTupleHeader` pointing to copied row; `NULL` only if `row` or `rowdesc` is `NULL`

SPI_modifytuple

Name

SPI_modifytuple — create a row by replacing selected fields of a given row

Synopsis

```
HeapTuple SPI_modifytuple(Relation rel, HeapTuple row, ncols, colnum, Datum * values, const ch
```

Description

SPI_modifytuple creates a new row by substituting new values for selected columns, copying the original row's columns at other positions. The input row is not modified.

Arguments

Relation rel

Used only as the source of the row descriptor for the row. (Passing a relation rather than a row descriptor is a misfeature.)

HeapTuple row

row to be modified

int ncols

number of column numbers in the array colnum

int * colnum

array of the numbers of the columns that are to be changed (column numbers start at 1)

Datum * values

new values for the specified columns

const char * Nulls

which new values are null, if any (see SPI_execute_plan for the format)

Return Value

new row with modifications, allocated in the upper executor context; NULL only if row is NULL

On error, SPI_result is set as follows:

`SPI_ERROR_ARGUMENT`

if `rel` is `NULL`, or if `row` is `NULL`, or if `ncols` is less than or equal to 0, or if `colnum` is `NULL`, or if `values` is `NULL`.

`SPI_ERROR_NOATTRIBUTE`

if `colnum` contains an invalid column number (less than or equal to 0 or greater than the number of column in `row`)

SPI_freetuple

Name

`SPI_freetuple` — free a row allocated in the upper executor context

Synopsis

```
void SPI_freetuple(HeapTuple row)
```

Description

`SPI_freetuple` frees a row previously allocated in the upper executor context.

This function is no longer different from plain `heap_freetuple`. It's kept just for backward compatibility of existing code.

Arguments

`HeapTuple row`

row to free

SPI_freetuptable

Name

`SPI_freetuptable` — free a row set created by `SPI_execute` or a similar function

Synopsis

```
void SPI_freetuptable(SPITupleTable * tuptable)
```

Description

`SPI_freetuptable` frees a row set created by a prior SPI command execution function, such as `SPI_execute`. Therefore, this function is usually called with the global variable `SPI_tupletable` as argument.

This function is useful if a SPI procedure needs to execute multiple commands and does not want to keep the results of earlier commands around until it ends. Note that any unfreed row sets will be freed anyway at `SPI_finish`.

Arguments

`SPITupleTable * tuptable`

pointer to row set to free

SPI_freeplan

Name

SPI_freeplan — free a previously saved plan

Synopsis

```
int SPI_freeplan(void *plan)
```

Description

SPI_freeplan releases a command execution plan previously returned by SPI_prepare or saved by SPI_saveplan.

Arguments

void * plan

pointer to plan to free

Return Value

SPI_ERROR_ARGUMENT if plan is NULL.

41.4. Visibility of Data Changes

The following rules govern the visibility of data changes in functions that use SPI (or any other C function):

- During the execution of an SQL command, any data changes made by the command are invisible to the command itself. For example, in

  ```
  INSERT INTO a SELECT * FROM a;
  ```

  the inserted rows are invisible to the `SELECT` part.
- Changes made by a command C are visible to all commands that are started after C, no matter whether they are started inside C (during the execution of C) or after C is done.
- Commands executed via SPI inside a function called by an SQL command (either an ordinary function or a trigger) follow one or the other of the above rules depending on the read/write flag passed to SPI. Commands executed in read-only mode follow the first rule: they can't see changes of the calling command. Commands executed in read-write mode follow the second rule: they can see all changes made so far.
- All standard procedural languages set the SPI read-write mode depending on the volatility attribute of the function. Commands of `STABLE` and `IMMUTABLE` functions are done in read-only mode, while commands of `VOLATILE` functions are done in read-write mode. While authors of C functions are able to violate this convention, it's unlikely to be a good idea to do so.

The next section contains an example that illustrates the application of these rules.

41.5. Examples

This section contains a very simple example of SPI usage. The procedure `execq` takes an SQL command as its first argument and a row count as its second, executes the command using `SPI_exec` and returns the number of rows that were processed by the command. You can find more complex examples for SPI in the source tree in `src/test/regress/regress.c` and in `contrib/spi`.

```
#include "executor/spi.h"

int execq(text *sql, int cnt);

int
execq(text *sql, int cnt)
{
    char *command;
    int ret;
    int proc;

    /* Convert given text object to a C string */
    command = DatumGetCString(DirectFunctionCall1(textout,
                                                  PointerGetDatum(sql)));
```

```
    SPI_connect();

    ret = SPI_exec(command, cnt);

    proc = SPI_processed;
    /*
     * If some rows were fetched, print them via elog(INFO).
     */
    if (ret > 0 && SPI_tuptable != NULL)
    {
        TupleDesc tupdesc = SPI_tuptable->tupdesc;
        SPITupleTable *tuptable = SPI_tuptable;
        char buf[8192];
        int i, j;

        for (j = 0; j < proc; j++)
        {
            HeapTuple tuple = tuptable->vals[j];

            for (i = 1, buf[0] = 0; i <= tupdesc->natts; i++)
                snprintf(buf + strlen (buf), sizeof(buf) - strlen(buf), " %s%s",
                        SPI_getvalue(tuple, tupdesc, i),
                        (i == tupdesc->natts) ? " " : " |");
            elog(INFO, "EXECQ: %s", buf);
        }
    }

    SPI_finish();
    pfree(command);

    return (proc);
}
```

(This function uses call convention version 0, to make the example easier to understand. In real applications you should use the new version 1 interface.)

This is how you declare the function after having compiled it into a shared library:

```
CREATE FUNCTION execq(text, integer) RETURNS integer
    AS 'filename'
    LANGUAGE C;
```

Here is a sample session:

```
=> SELECT execq('CREATE TABLE a (x integer)', 0);
 execq
-------
     0
(1 row)

=> INSERT INTO a VALUES (execq('INSERT INTO a VALUES (0)', 0));
INSERT 0 1
```

```
=> SELECT execq('SELECT * FROM a', 0);
INFO:  EXECQ:  0    -- inserted by execq
INFO:  EXECQ:  1    -- returned by execq and inserted by upper INSERT

 execq
-------
     2
(1 row)

=> SELECT execq('INSERT INTO a SELECT x + 2 FROM a', 1);
 execq
-------
     1
(1 row)

=> SELECT execq('SELECT * FROM a', 10);
INFO:  EXECQ:  0
INFO:  EXECQ:  1
INFO:  EXECQ:  2    -- 0 + 2, only one row inserted - as specified

 execq
-------
     3              -- 10 is the max value only, 3 is the real number of rows
(1 row)

=> DELETE FROM a;
DELETE 3
=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INSERT 0 1
=> SELECT * FROM a;
 x
---
 1                  -- no rows in a (0) + 1
(1 row)

=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INFO:  EXECQ:  1
INSERT 0 1
=> SELECT * FROM a;
 x
---
 1
 2                  -- there was one row in a + 1
(2 rows)

-- This demonstrates the data changes visibility rule:

=> INSERT INTO a SELECT execq('SELECT * FROM a', 0) * x FROM a;
INFO:  EXECQ:  1
INFO:  EXECQ:  2
INFO:  EXECQ:  1
INFO:  EXECQ:  2
INFO:  EXECQ:  2
```

```
INSERT 0 2
=> SELECT * FROM a;
 x
---
 1
 2
 2                  -- 2 rows * 1 (x in first row)
 6                  -- 3 rows (2 + 1 just inserted) * 2 (x in second row)
(4 rows)               ^^^^^^
                       rows visible to execq() in different invocations
```

VI. Reference

The entries in this Reference are meant to provide in reasonable length an authoritative, complete, and formal summary about their respective subjects. More information about the use of PostgreSQL, in narrative, tutorial, or example form, may be found in other parts of this book. See the cross-references listed on each reference page.

The reference entries are also available as traditional “man” pages.

I. SQL Commands

This part contains reference information for the SQL commands supported by PostgreSQL. By “SQL” the language in general is meant; information about the standards conformance and compatibility of each command can be found on the respective reference page.

ABORT

Name

ABORT — abort the current transaction

Synopsis

```
ABORT [ WORK | TRANSACTION ]
```

Description

ABORT rolls back the current transaction and causes all the updates made by the transaction to be discarded. This command is identical in behavior to the standard SQL command *ROLLBACK*, and is present only for historical reasons.

Parameters

WORK
TRANSACTION

Optional key words. They have no effect.

Notes

Use *COMMIT* to successfully terminate a transaction.

Issuing ABORT when not inside a transaction does no harm, but it will provoke a warning message.

Examples

To abort all changes:

```
ABORT;
```

Compatibility

This command is a PostgreSQL extension present for historical reasons. `ROLLBACK` is the equivalent standard SQL command.

See Also

BEGIN, *COMMIT*, *ROLLBACK*

ALTER AGGREGATE

Name

ALTER AGGREGATE — change the definition of an aggregate function

Synopsis

```
ALTER AGGREGATE name ( type [ , ... ] ) RENAME TO new_name
ALTER AGGREGATE name ( type [ , ... ] ) OWNER TO new_owner
ALTER AGGREGATE name ( type [ , ... ] ) SET SCHEMA new_schema
```

Description

ALTER AGGREGATE changes the definition of an aggregate function.

You must own the aggregate function to use ALTER AGGREGATE. To change the schema of an aggregate function, you must also have CREATE privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the aggregate function's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the aggregate function. However, a superuser can alter ownership of any aggregate function anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing aggregate function.

type

An input data type on which the aggregate function operates. To reference a zero-argument aggregate function, write * in place of the list of input data types.

new_name

The new name of the aggregate function.

new_owner

The new owner of the aggregate function.

new_schema

The new schema for the aggregate function.

Examples

To rename the aggregate function `myavg` for type `integer` to `my_average`:

```
ALTER AGGREGATE myavg(integer) RENAME TO my_average;
```

To change the owner of the aggregate function `myavg` for type `integer` to `joe`:

```
ALTER AGGREGATE myavg(integer) OWNER TO joe;
```

To move the aggregate function `myavg` for type `integer` into schema `myschema`:

```
ALTER AGGREGATE myavg(integer) SET SCHEMA myschema;
```

Compatibility

There is no `ALTER AGGREGATE` statement in the SQL standard.

See Also

CREATE AGGREGATE, *DROP AGGREGATE*

ALTER CONVERSION

Name

ALTER CONVERSION — change the definition of a conversion

Synopsis

```
ALTER CONVERSION name RENAME TO newname
ALTER CONVERSION name OWNER TO newowner
```

Description

ALTER CONVERSION changes the definition of a conversion.

You must own the conversion to use ALTER CONVERSION. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the conversion's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the conversion. However, a superuser can alter ownership of any conversion anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing conversion.

newname

The new name of the conversion.

newowner

The new owner of the conversion.

Examples

To rename the conversion iso_8859_1_to_utf8 to latin1_to_unicode:

```
ALTER CONVERSION iso_8859_1_to_utf8 RENAME TO latin1_to_unicode;
```

To change the owner of the conversion iso_8859_1_to_utf8 to joe:

```
ALTER CONVERSION iso_8859_1_to_utf8 OWNER TO joe;
```

Compatibility

There is no `ALTER CONVERSION` statement in the SQL standard.

See Also

CREATE CONVERSION, *DROP CONVERSION*

ALTER DATABASE

Name

ALTER DATABASE — change a database

Synopsis

```
ALTER DATABASE name [ [ WITH ] option [ ... ] ]

where option can be:

    CONNECTION LIMIT connlimit

ALTER DATABASE name SET parameter { TO | = } { value | DEFAULT }
ALTER DATABASE name RESET parameter

ALTER DATABASE name RENAME TO newname

ALTER DATABASE name OWNER TO new_owner
```

Description

ALTER DATABASE changes the attributes of a database.

The first form changes certain per-database settings. (See below for details.) Only the database owner or a superuser can change these settings.

The second and third forms change the session default for a run-time configuration variable for a PostgreSQL database. Whenever a new session is subsequently started in that database, the specified value becomes the session default value. The database-specific default overrides whatever setting is present in postgresql.conf or has been received from the postgres command line. Only the database owner or a superuser can change the session defaults for a database. Certain variables cannot be set this way, or can only be set by a superuser.

The fourth form changes the name of the database. Only the database owner or a superuser can rename a database; non-superuser owners must also have the CREATEDB privilege. The current database cannot be renamed. (Connect to a different database if you need to do that.)

The fifth form changes the owner of the database. To alter the owner, you must own the database and also be a direct or indirect member of the new owning role, and you must have the CREATEDB privilege. (Note that superusers have all these privileges automatically.)

Parameters

name

The name of the database whose attributes are to be altered.

connlimit

How many concurrent connections can be made to this database. -1 means no limit.

parameter
value

Set this database's session default for the specified configuration parameter to the given value. If *value* is `DEFAULT` or, equivalently, `RESET` is used, the database-specific setting is removed, so the system-wide default setting will be inherited in new sessions. Use `RESET ALL` to clear all database-specific settings.

See *SET* and Chapter 17 for more information about allowed parameter names and values.

newname

The new name of the database.

new_owner

The new owner of the database.

Notes

It is also possible to tie a session default to a specific user rather than to a database; see *ALTER USER*. User-specific settings override database-specific ones if there is a conflict.

Examples

To disable index scans by default in the database `test`:

```
ALTER DATABASE test SET enable_indexscan TO off;
```

Compatibility

The `ALTER DATABASE` statement is a PostgreSQL extension.

See Also

CREATE DATABASE, *DROP DATABASE*, *SET*

ALTER DOMAIN

Name

ALTER DOMAIN — change the definition of a domain

Synopsis

```
ALTER DOMAIN name
    { SET DEFAULT expression | DROP DEFAULT }
ALTER DOMAIN name
    { SET | DROP } NOT NULL
ALTER DOMAIN name
    ADD domain_constraint
ALTER DOMAIN name
    DROP CONSTRAINT constraint_name [ RESTRICT | CASCADE ]
ALTER DOMAIN name
    OWNER TO new_owner
ALTER DOMAIN name
    SET SCHEMA new_schema
```

Description

ALTER DOMAIN changes the definition of an existing domain. There are several sub-forms:

SET/DROP DEFAULT

These forms set or remove the default value for a domain. Note that defaults only apply to subsequent INSERT commands; they do not affect rows already in a table using the domain.

SET/DROP NOT NULL

These forms change whether a domain is marked to allow NULL values or to reject NULL values. You may only SET NOT NULL when the columns using the domain contain no null values.

ADD *domain_constraint*

This form adds a new constraint to a domain using the same syntax as *CREATE DOMAIN*. This will only succeed if all columns using the domain satisfy the new constraint.

DROP CONSTRAINT

This form drops constraints on a domain.

OWNER

This form changes the owner of the domain to the specified user.

SET SCHEMA

This form changes the schema of the domain. Any constraints associated with the domain are moved into the new schema as well.

You must own the domain to use `ALTER DOMAIN`. To change the schema of a domain, you must also have `CREATE` privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the domain's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the domain. However, a superuser can alter ownership of any domain anyway.)

Parameters

name

The name (possibly schema-qualified) of an existing domain to alter.

domain_constraint

New domain constraint for the domain.

constraint_name

Name of an existing constraint to drop.

`CASCADE`

Automatically drop objects that depend on the constraint.

`RESTRICT`

Refuse to drop the constraint if there are any dependent objects. This is the default behavior.

new_owner

The user name of the new owner of the domain.

new_schema

The new schema for the domain.

Notes

Currently, `ALTER DOMAIN ADD CONSTRAINT` and `ALTER DOMAIN SET NOT NULL` will fail if the named domain or any derived domain is used within a composite-type column of any table in the database. They should eventually be improved to be able to verify the new constraint for such nested columns.

Examples

To add a `NOT NULL` constraint to a domain:

```
ALTER DOMAIN zipcode SET NOT NULL;
```

To remove a `NOT NULL` constraint from a domain:

```
ALTER DOMAIN zipcode DROP NOT NULL;
```

To add a check constraint to a domain:

```
ALTER DOMAIN zipcode ADD CONSTRAINT zipchk CHECK (char_length(VALUE) = 5);
```

To remove a check constraint from a domain:

```
ALTER DOMAIN zipcode DROP CONSTRAINT zipchk;
```

To move the domain into a different schema:

```
ALTER DOMAIN zipcode SET SCHEMA customers;
```

Compatibility

`ALTER DOMAIN` conforms to the SQL standard, except for the `OWNER` and `SET SCHEMA` variants, which are PostgreSQL extensions.

See Also

CREATE DOMAIN, *DROP DOMAIN*

ALTER FUNCTION

Name

ALTER FUNCTION — change the definition of a function

Synopsis

```
ALTER FUNCTION name ( [ [ argmode ] [ argname ] argtype [, ...] ] )
    action [, ... ] [ RESTRICT ]
ALTER FUNCTION name ( [ [ argmode ] [ argname ] argtype [, ...] ] )
    RENAME TO new_name
ALTER FUNCTION name ( [ [ argmode ] [ argname ] argtype [, ...] ] )
    OWNER TO new_owner
ALTER FUNCTION name ( [ [ argmode ] [ argname ] argtype [, ...] ] )
    SET SCHEMA new_schema

where action is one of:

    CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    IMMUTABLE | STABLE | VOLATILE
    [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
```

Description

ALTER FUNCTION changes the definition of a function.

You must own the function to use ALTER FUNCTION. To change a function's schema, you must also have CREATE privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the function's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the function. However, a superuser can alter ownership of any function anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing function.

argmode

The mode of an argument: either IN, OUT, or INOUT. If omitted, the default is IN. Note that ALTER FUNCTION does not actually pay any attention to OUT arguments, since only the input arguments are needed to determine the function's identity. So it is sufficient to list the IN and INOUT arguments.

argname

The name of an argument. Note that `ALTER FUNCTION` does not actually pay any attention to argument names, since only the argument data types are needed to determine the function's identity.

argtype

The data type(s) of the function's arguments (optionally schema-qualified), if any.

new_name

The new name of the function.

new_owner

The new owner of the function. Note that if the function is marked `SECURITY DEFINER`, it will subsequently execute as the new owner.

new_schema

The new schema for the function.

```
CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT
```

`CALLED ON NULL INPUT` changes the function so that it will be invoked when some or all of its arguments are null. `RETURNS NULL ON NULL INPUT` or `STRICT` changes the function so that it is not invoked if any of its arguments are null; instead, a null result is assumed automatically. See *CREATE FUNCTION* for more information.

```
IMMUTABLE
STABLE
VOLATILE
```

Change the volatility of the function to the specified setting. See *CREATE FUNCTION* for details.

```
[ EXTERNAL ] SECURITY INVOKER
[ EXTERNAL ] SECURITY DEFINER
```

Change whether the function is a security definer or not. The key word `EXTERNAL` is ignored for SQL conformance. See *CREATE FUNCTION* for more information about this capability.

```
RESTRICT
```

Ignored for conformance with the SQL standard.

Examples

To rename the function `sqrt` for type `integer` to `square_root`:

```
ALTER FUNCTION sqrt(integer) RENAME TO square_root;
```

To change the owner of the function `sqrt` for type `integer` to `joe`:

```
ALTER FUNCTION sqrt(integer) OWNER TO joe;
```

To change the schema of the function `sqrt` for type `integer` to `maths`:

```
ALTER FUNCTION sqrt(integer) SET SCHEMA maths;
```

Compatibility

This statement is partially compatible with the `ALTER FUNCTION` statement in the SQL standard. The standard allows more properties of a function to be modified, but does not provide the ability to rename a function, make a function a security definer, or change the owner, schema, or volatility of a function. The standard also requires the `RESTRICT` key word, which is optional in PostgreSQL.

See Also

CREATE FUNCTION, *DROP FUNCTION*

ALTER GROUP

Name

ALTER GROUP — change role name or membership

Synopsis

```
ALTER GROUP groupname ADD USER username [, ... ]
ALTER GROUP groupname DROP USER username [, ... ]

ALTER GROUP groupname RENAME TO newname
```

Description

ALTER GROUP changes the attributes of a user group. This is an obsolete command, though still accepted for backwards compatibility, because groups (and users too) have been superseded by the more general concept of roles.

The first two variants add users to a group or remove them from a group. (Any role can play the part of either a "user" or a "group" for this purpose.) These variants are effectively equivalent to granting or revoking membership in the role named as the "group"; so the preferred way to do this is to use *GRANT* or *REVOKE*.

The third variant changes the name of the group. This is exactly equivalent to renaming the role with *ALTER ROLE*.

Parameters

groupname

The name of the group (role) to modify.

username

Users (roles) that are to be added to or removed from the group. The users must already exist; ALTER GROUP does not create or drop users.

newname

The new name of the group.

Examples

Add users to a group:

```
ALTER GROUP staff ADD USER karl, john;
```

Remove a user from a group:

```
ALTER GROUP workers DROP USER beth;
```

Compatibility

There is no `ALTER GROUP` statement in the SQL standard.

See Also

GRANT, *REVOKE*, *ALTER ROLE*

ALTER INDEX

Name

ALTER INDEX — change the definition of an index

Synopsis

```
ALTER INDEX name RENAME TO new_name
ALTER INDEX name SET TABLESPACE tablespace_name
ALTER INDEX name SET ( storage_parameter = value [, ... ] )
ALTER INDEX name RESET ( storage_parameter [, ... ] )
```

Description

ALTER INDEX changes the definition of an existing index. There are several subforms:

RENAME

The RENAME form changes the name of the index. There is no effect on the stored data.

SET TABLESPACE

This form changes the index's tablespace to the specified tablespace and moves the data file(s) associated with the index to the new tablespace. See also *CREATE TABLESPACE*.

SET (*storage_parameter* = *value* [, ...])

This form changes one or more index-method-specific storage parameters for the index. See *CREATE INDEX* for details on the available parameters. Note that the index contents will not be modified immediately by this command; depending on the parameter you may need to rebuild the index with *REINDEX* to get the desired effects.

RESET (*storage_parameter* [, ...])

This form resets one or more index-method-specific storage parameters to their defaults. As with SET, a REINDEX may be needed to update the index entirely.

Parameters

name

The name (possibly schema-qualified) of an existing index to alter.

new_name

New name for the index.

tablespace_name

The tablespace to which the index will be moved.

storage_parameter

The name of an index-method-specific storage parameter.

value

The new value for an index-method-specific storage parameter. This might be a number or a word depending on the parameter.

Notes

These operations are also possible using *ALTER TABLE*. `ALTER INDEX` is in fact just an alias for the forms of `ALTER TABLE` that apply to indexes.

There was formerly an `ALTER INDEX OWNER` variant, but this is now ignored (with a warning). An index cannot have an owner different from its table's owner. Changing the table's owner automatically changes the index as well.

Changing any part of a system catalog index is not permitted.

Examples

To rename an existing index:

```
ALTER INDEX distributors RENAME TO suppliers;
```

To move an index to a different tablespace:

```
ALTER INDEX distributors SET TABLESPACE fasttablespace;
```

To change an index's fill factor (assuming that the index method supports it):

```
ALTER INDEX distributors SET (fillfactor = 75);
REINDEX INDEX distributors;
```

Compatibility

`ALTER INDEX` is a PostgreSQL extension.

See Also

CREATE INDEX, *REINDEX*

ALTER LANGUAGE

Name

`ALTER LANGUAGE` — change the definition of a procedural language

Synopsis

```
ALTER LANGUAGE name RENAME TO newname
```

Description

`ALTER LANGUAGE` changes the definition of a language. The only functionality is to rename the language. Only a superuser can rename languages.

Parameters

name

Name of a language

newname

The new name of the language

Compatibility

There is no `ALTER LANGUAGE` statement in the SQL standard.

See Also

CREATE LANGUAGE, *DROP LANGUAGE*

ALTER OPERATOR

Name

ALTER OPERATOR — change the definition of an operator

Synopsis

```
ALTER OPERATOR name ( { lefttype | NONE } , { righttype | NONE } ) OWNER TO newowner
```

Description

ALTER OPERATOR changes the definition of an operator. The only currently available functionality is to change the owner of the operator.

You must own the operator to use ALTER OPERATOR. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the operator's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the operator. However, a superuser can alter ownership of any operator anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing operator.

lefttype

The data type of the operator's left operand; write NONE if the operator has no left operand.

righttype

The data type of the operator's right operand; write NONE if the operator has no right operand.

newowner

The new owner of the operator.

Examples

Change the owner of a custom operator `a @@ b` for type `text`:

```
ALTER OPERATOR @@ (text, text) OWNER TO joe;
```

Compatibility

There is no `ALTER OPERATOR` statement in the SQL standard.

See Also

CREATE OPERATOR, *DROP OPERATOR*

ALTER OPERATOR CLASS

Name

ALTER OPERATOR CLASS — change the definition of an operator class

Synopsis

```
ALTER OPERATOR CLASS name USING index_method RENAME TO newname
ALTER OPERATOR CLASS name USING index_method OWNER TO newowner
```

Description

ALTER OPERATOR CLASS changes the definition of an operator class.

You must own the operator class to use ALTER OPERATOR CLASS. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the operator class's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the operator class. However, a superuser can alter ownership of any operator class anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing operator class.

index_method

The name of the index method this operator class is for.

newname

The new name of the operator class.

newowner

The new owner of the operator class.

Compatibility

There is no ALTER OPERATOR CLASS statement in the SQL standard.

See Also

CREATE OPERATOR CLASS, *DROP OPERATOR CLASS*

ALTER ROLE

Name

ALTER ROLE — change a database role

Synopsis

```
ALTER ROLE name [ [ WITH ] option [ ... ] ]

where option can be:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'

ALTER ROLE name RENAME TO newname

ALTER ROLE name SET configuration_parameter { TO | = } { value | DEFAULT }
ALTER ROLE name RESET configuration_parameter
```

Description

ALTER ROLE changes the attributes of a PostgreSQL role.

The first variant of this command listed in the synopsis can change many of the role attributes that can be specified in *CREATE ROLE*. (All the possible attributes are covered, except that there are no options for adding or removing memberships; use *GRANT* and *REVOKE* for that.) Attributes not mentioned in the command retain their previous settings. Database superusers can change any of these settings for any role. Roles having CREATEROLE privilege can change any of these settings, but only for non-superuser roles. Ordinary roles can only change their own password.

The second variant changes the name of the role. Database superusers can rename any role. Roles having CREATEROLE privilege can rename non-superuser roles. The current session user cannot be renamed. (Connect as a different user if you need to do that.) Because MD5-encrypted passwords use the role name as cryptographic salt, renaming a role clears its password if the password is MD5-encrypted.

The third and the fourth variant change a role's session default for a specified configuration variable. Whenever the role subsequently starts a new session, the specified value becomes the session default, overriding whatever setting is present in postgresql.conf or has been received from the postgres command line. (For a role without LOGIN privilege, session defaults have no effect.) Ordinary roles can change their own session defaults. Superusers can change anyone's session defaults. Roles having

CREATEROLE privilege can change defaults for non-superuser roles. Certain variables cannot be set this way, or can only be set if a superuser issues the command.

Parameters

name

The name of the role whose attributes are to be altered.

SUPERUSER
NOSUPERUSER
CREATEDB
NOCREATEDB
CREATEROLE
NOCREATEROLE
CREATEUSER
NOCREATEUSER
INHERIT
NOINHERIT
LOGIN
NOLOGIN
CONNECTION LIMIT *connlimit*
PASSWORD *password*
ENCRYPTED
UNENCRYPTED
VALID UNTIL '*timestamp*'

These clauses alter attributes originally set by *CREATE ROLE*. For more information, see the CREATE ROLE reference page.

newname

The new name of the role.

configuration_parameter
value

Set this role's session default for the specified configuration parameter to the given value. If *value* is DEFAULT or, equivalently, RESET is used, the role-specific variable setting is removed, so the role will inherit the system-wide default setting in new sessions. Use RESET ALL to clear all role-specific settings.

See *SET* and Chapter 17 for more information about allowed parameter names and values.

Notes

Use *CREATE ROLE* to add new roles, and *DROP ROLE* to remove a role.

ALTER ROLE cannot change a role's memberships. Use *GRANT* and *REVOKE* to do that.

Caution must be exercised when specifying an unencrypted password with this command. The password will be transmitted to the server in cleartext, and it might also be logged in the client's command history or the server log. *psql* contains a command `\password` that can be used to safely change a role's password.

It is also possible to tie a session default to a specific database rather than to a role; see *ALTER DATABASE*. Role-specific settings override database-specific ones if there is a conflict.

Examples

Change a role's password:

```
ALTER ROLE davide WITH PASSWORD 'hu8jmn3';
```

Change a password expiration date, specifying that the password should expire at midday on 4th May 2015 using the time zone which is one hour ahead of UTC:

```
ALTER ROLE chris VALID UNTIL 'May 4 12:00:00 2015 +1';
```

Make a password valid forever:

```
ALTER ROLE fred VALID UNTIL 'infinity';
```

Give a role the ability to create other roles and new databases:

```
ALTER ROLE miriam CREATEROLE CREATEDB;
```

Give a role a non-default setting of the maintenance_work_mem parameter:

```
ALTER ROLE worker_bee SET maintenance_work_mem = 100000;
```

Compatibility

The `ALTER ROLE` statement is a PostgreSQL extension.

See Also

CREATE ROLE, *DROP ROLE*, *SET*

ALTER SCHEMA

Name

ALTER SCHEMA — change the definition of a schema

Synopsis

```
ALTER SCHEMA name RENAME TO newname
ALTER SCHEMA name OWNER TO newowner
```

Description

ALTER SCHEMA changes the definition of a schema.

You must own the schema to use ALTER SCHEMA. To rename a schema you must also have the CREATE privilege for the database. To alter the owner, you must also be a direct or indirect member of the new owning role, and you must have the CREATE privilege for the database. (Note that superusers have all these privileges automatically.)

Parameters

name

The name of an existing schema.

newname

The new name of the schema. The new name cannot begin with pg_, as such names are reserved for system schemas.

newowner

The new owner of the schema.

Compatibility

There is no ALTER SCHEMA statement in the SQL standard.

See Also

CREATE SCHEMA, *DROP SCHEMA*

ALTER SEQUENCE

Name

ALTER SEQUENCE — change the definition of a sequence generator

Synopsis

```
ALTER SEQUENCE name [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ RESTART [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { table.column | NONE } ]
ALTER SEQUENCE name SET SCHEMA new_schema
```

Description

ALTER SEQUENCE changes the parameters of an existing sequence generator. Any parameters not specifically set in the ALTER SEQUENCE command retain their prior settings.

You must own the sequence to use ALTER SEQUENCE. To change a sequence's schema, you must also have CREATE privilege on the new schema.

Parameters

name

The name (optionally schema-qualified) of a sequence to be altered.

increment

The clause INCREMENT BY *increment* is optional. A positive value will make an ascending sequence, a negative one a descending sequence. If unspecified, the old increment value will be maintained.

minvalue
NO MINVALUE

The optional clause MINVALUE *minvalue* determines the minimum value a sequence can generate. If NO MINVALUE is specified, the defaults of 1 and $-2^{63}-1$ for ascending and descending sequences, respectively, will be used. If neither option is specified, the current minimum value will be maintained.

maxvalue
NO MAXVALUE

The optional clause MAXVALUE *maxvalue* determines the maximum value for the sequence. If NO MAXVALUE is specified, the defaults are 2^{63}-1 and -1 for ascending and descending sequences, respectively, will be used. If neither option is specified, the current maximum value will be maintained.

start

The optional clause RESTART WITH *start* changes the current value of the sequence.

cache

The clause CACHE *cache* enables sequence numbers to be preallocated and stored in memory for faster access. The minimum value is 1 (only one value can be generated at a time, i.e., no cache). If unspecified, the old cache value will be maintained.

CYCLE

The optional CYCLE key word may be used to enable the sequence to wrap around when the *maxvalue* or *minvalue* has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the *minvalue* or *maxvalue*, respectively.

NO CYCLE

If the optional NO CYCLE key word is specified, any calls to nextval after the sequence has reached its maximum value will return an error. If neither CYCLE or NO CYCLE are specified, the old cycle behavior will be maintained.

OWNED BY *table.column*
OWNED BY NONE

The OWNED BY option causes the sequence to be associated with a specific table column, such that if that column (or its whole table) is dropped, the sequence will be automatically dropped as well. If specified, this association replaces any previously specified association for the sequence. The specified table must have the same owner and be in the same schema as the sequence. Specifying OWNED BY NONE removes any existing association, making the sequence "free-standing".

new_schema

The new schema for the sequence.

Examples

Restart a sequence called serial, at 105:

```
ALTER SEQUENCE serial RESTART WITH 105;
```

Notes

To avoid blocking of concurrent transactions that obtain numbers from the same sequence, `ALTER SEQUENCE`'s effects on the sequence generation parameters are never rolled back; those changes take effect immediately and are not reversible. However, the `OWNED BY` and `SET SCHEMA` clauses are ordinary catalog updates and can be rolled back.

`ALTER SEQUENCE` will not immediately affect `nextval` results in backends, other than the current one, that have preallocated (cached) sequence values. They will use up all cached values prior to noticing the changed sequence generation parameters. The current backend will be affected immediately.

Some variants of `ALTER TABLE` can be used with sequences as well; for example, to rename a sequence use `ALTER TABLE RENAME`.

Compatibility

`ALTER SEQUENCE` conforms to the SQL standard, except for the `OWNED BY` and `SET SCHEMA` clauses, which are PostgreSQL extensions.

See Also

CREATE SEQUENCE, *DROP SEQUENCE*

ALTER TABLE

Name

ALTER TABLE — change the definition of a table

Synopsis

```
ALTER TABLE [ ONLY ] name [ * ]
    action [, ... ]
ALTER TABLE [ ONLY ] name [ * ]
    RENAME [ COLUMN ] column TO new_column
ALTER TABLE name
    RENAME TO new_name
ALTER TABLE name
    SET SCHEMA new_schema

where action is one of:

    ADD [ COLUMN ] column type [ column_constraint [ ... ] ]
    DROP [ COLUMN ] column [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] column TYPE type [ USING expression ]
    ALTER [ COLUMN ] column SET DEFAULT expression
    ALTER [ COLUMN ] column DROP DEFAULT
    ALTER [ COLUMN ] column { SET | DROP } NOT NULL
    ALTER [ COLUMN ] column SET STATISTICS integer
    ALTER [ COLUMN ] column SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
    ADD table_constraint
    DROP CONSTRAINT constraint_name [ RESTRICT | CASCADE ]
    DISABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE TRIGGER [ trigger_name | ALL | USER ]
    CLUSTER ON index_name
    SET WITHOUT CLUSTER
    SET WITHOUT OIDS
    SET ( storage_parameter = value [, ... ] )
    RESET ( storage_parameter [, ... ] )
    INHERIT parent_table
    NO INHERIT parent_table
    OWNER TO new_owner
    SET TABLESPACE new_tablespace
```

Description

ALTER TABLE changes the definition of an existing table. There are several subforms:

ADD COLUMN

This form adds a new column to the table, using the same syntax as *CREATE TABLE*.

`DROP COLUMN`

This form drops a column from a table. Indexes and table constraints involving the column will be automatically dropped as well. You will need to say `CASCADE` if anything outside the table depends on the column, for example, foreign key references or views.

`ALTER COLUMN TYPE`

This form changes the type of a column of a table. Indexes and simple table constraints involving the column will be automatically converted to use the new column type by reparsing the originally supplied expression. The optional `USING` clause specifies how to compute the new column value from the old; if omitted, the default conversion is the same as an assignment cast from old data type to new. A `USING` clause must be provided if there is no implicit or assignment cast from old to new type.

`SET/DROP DEFAULT`

These forms set or remove the default value for a column. The default values only apply to subsequent `INSERT` commands; they do not cause rows already in the table to change. Defaults may also be created for views, in which case they are inserted into `INSERT` statements on the view before the view's `ON INSERT` rule is applied.

`SET/DROP NOT NULL`

These forms change whether a column is marked to allow null values or to reject null values. You can only use `SET NOT NULL` when the column contains no null values.

`SET STATISTICS`

This form sets the per-column statistics-gathering target for subsequent *ANALYZE* operations. The target can be set in the range 0 to 1000; alternatively, set it to -1 to revert to using the system default statistics target (default_statistics_target). For more information on the use of statistics by the PostgreSQL query planner, refer to Section 13.2.

`SET STORAGE`

This form sets the storage mode for a column. This controls whether this column is held inline or in a supplementary table, and whether the data should be compressed or not. `PLAIN` must be used for fixed-length values such as `integer` and is inline, uncompressed. `MAIN` is for inline, compressible data. `EXTERNAL` is for external, uncompressed data, and `EXTENDED` is for external, compressed data. `EXTENDED` is the default for most data types that support non-`PLAIN` storage. Use of `EXTERNAL` will make substring operations on `text` and `bytea` columns faster, at the penalty of increased storage space. Note that `SET STORAGE` doesn't itself change anything in the table, it just sets the strategy to be pursued during future table updates. See Section 52.2 for more information.

`ADD` *`table_constraint`*

This form adds a new constraint to a table using the same syntax as *CREATE TABLE*.

`DROP CONSTRAINT`

This form drops the specified constraint on a table.

`DISABLE/ENABLE TRIGGER`

These forms disable or enable trigger(s) belonging to the table. A disabled trigger is still known to the system, but is not executed when its triggering event occurs. For a deferred trigger, the enable status is checked when the event occurs, not when the trigger function is actually executed. One

may disable or enable a single trigger specified by name, or all triggers on the table, or only user triggers (this option excludes triggers that are used to implement foreign key constraints). Disabling or enabling constraint triggers requires superuser privileges; it should be done with caution since of course the integrity of the constraint cannot be guaranteed if the triggers are not executed.

`CLUSTER`

This form selects the default index for future *CLUSTER* operations. It does not actually re-cluster the table.

`SET WITHOUT CLUSTER`

This form removes the most recently used *CLUSTER* index specification from the table. This affects future cluster operations that don't specify an index.

`SET WITHOUT OIDS`

This form removes the `oid` system column from the table. This is exactly equivalent to `DROP COLUMN oid RESTRICT`, except that it will not complain if there is already no `oid` column.

Note that there is no variant of `ALTER TABLE` that allows OIDs to be restored to a table once they have been removed.

`SET ( storage_parameter = value [, ... ] )`

This form changes one or more storage parameters for the table. See *CREATE TABLE* for details on the available parameters. Note that the table contents will not be modified immediately by this command; depending on the parameter you may need to rewrite the table to get the desired effects. That can be done with *CLUSTER* or one of the forms of `ALTER TABLE` that forces a table rewrite.

> **Note:** While `CREATE TABLE` allows `OIDS` to be specified in the `WITH (storage_parameter)` syntax, `ALTER TABLE` does not treat `OIDS` as a storage parameter.

`RESET ( storage_parameter [, ... ] )`

This form resets one or more storage parameters to their defaults. As with `SET`, a table rewrite may be needed to update the table entirely.

`INHERIT parent_table`

This form adds the target table as a new child of the specified parent table. Subsequently, queries against the parent will include records of the target table. To be added as a child, the target table must already contain all the same columns as the parent (it could have additional columns, too). The columns must have matching data types, and if they have `NOT NULL` constraints in the parent then they must also have `NOT NULL` constraints in the child.

There must also be matching child-table constraints for all `CHECK` constraints of the parent. Currently `UNIQUE`, `PRIMARY KEY`, and `FOREIGN KEY` constraints are not considered, but this may change in the future.

`NO INHERIT parent_table`

This form removes the target table from the list of children of the specified parent table. Queries against the parent table will no longer include records drawn from the target table.

`OWNER`

This form changes the owner of the table, sequence, or view to the specified user.

`SET TABLESPACE`

This form changes the table's tablespace to the specified tablespace and moves the data file(s) associated with the table to the new tablespace. Indexes on the table, if any, are not moved; but they can be moved separately with additional `SET TABLESPACE` commands. See also *CREATE TABLESPACE*.

`RENAME`

The `RENAME` forms change the name of a table (or an index, sequence, or view) or the name of an individual column in a table. There is no effect on the stored data.

`SET SCHEMA`

This form moves the table into another schema. Associated indexes, constraints, and sequences owned by table columns are moved as well.

All the actions except `RENAME` and `SET SCHEMA` can be combined into a list of multiple alterations to apply in parallel. For example, it is possible to add several columns and/or alter the type of several columns in a single command. This is particularly useful with large tables, since only one pass over the table need be made.

You must own the table to use `ALTER TABLE`. To change the schema of a table, you must also have `CREATE` privilege on the new schema. To add the table as a new child of a parent table, you must own the parent table as well. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the table's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the table. However, a superuser can alter ownership of any table anyway.)

Parameters

`name`

The name (possibly schema-qualified) of an existing table to alter. If `ONLY` is specified, only that table is altered. If `ONLY` is not specified, the table and all its descendant tables (if any) are updated. `*` can be appended to the table name to indicate that descendant tables are to be altered, but in the current version, this is the default behavior. (In releases before 7.1, `ONLY` was the default behavior. The default can be altered by changing the configuration parameter sql_inheritance.)

`column`

Name of a new or existing column.

`new_column`

New name for an existing column.

`new_name`

New name for the table.

type

Data type of the new column, or new data type for an existing column.

table_constraint

New table constraint for the table.

constraint_name

Name of an existing constraint to drop.

CASCADE

Automatically drop objects that depend on the dropped column or constraint (for example, views referencing the column).

RESTRICT

Refuse to drop the column or constraint if there are any dependent objects. This is the default behavior.

trigger_name

Name of a single trigger to disable or enable.

ALL

Disable or enable all triggers belonging to the table. (This requires superuser privilege if any of the triggers are for foreign key constraints.)

USER

Disable or enable all triggers belonging to the table except for foreign key constraint triggers.

index_name

The index name on which the table should be marked for clustering.

storage_parameter

The name of a table storage parameter.

value

The new value for a table storage parameter. This might be a number or a word depending on the parameter.

parent_table

A parent table to associate or de-associate with this table.

new_owner

The user name of the new owner of the table.

new_tablespace

The name of the tablespace to which the table will be moved.

new_schema

The name of the schema to which the table will be moved.

Notes

The key word COLUMN is noise and can be omitted.

When a column is added with ADD COLUMN, all existing rows in the table are initialized with the column's default value (NULL if no DEFAULT clause is specified).

Adding a column with a non-null default or changing the type of an existing column will require the entire table to be rewritten. This may take a significant amount of time for a large table; and it will temporarily require double the disk space.

Adding a CHECK or NOT NULL constraint requires scanning the table to verify that existing rows meet the constraint.

The main reason for providing the option to specify multiple changes in a single ALTER TABLE is that multiple table scans or rewrites can thereby be combined into a single pass over the table.

The DROP COLUMN form does not physically remove the column, but simply makes it invisible to SQL operations. Subsequent insert and update operations in the table will store a null value for the column. Thus, dropping a column is quick but it will not immediately reduce the on-disk size of your table, as the space occupied by the dropped column is not reclaimed. The space will be reclaimed over time as existing rows are updated.

The fact that ALTER TYPE requires rewriting the whole table is sometimes an advantage, because the rewriting process eliminates any dead space in the table. For example, to reclaim the space occupied by a dropped column immediately, the fastest way is

```
ALTER TABLE table ALTER COLUMN anycol TYPE anytype;
```

where anycol is any remaining table column and anytype is the same type that column already has. This results in no semantically-visible change in the table, but the command forces rewriting, which gets rid of no-longer-useful data.

The USING option of ALTER TYPE can actually specify any expression involving the old values of the row; that is, it can refer to other columns as well as the one being converted. This allows very general conversions to be done with the ALTER TYPE syntax. Because of this flexibility, the USING expression is not applied to the column's default value (if any); the result might not be a constant expression as required for a default. This means that when there is no implicit or assignment cast from old to new type, ALTER TYPE may fail to convert the default even though a USING clause is supplied. In such cases, drop the default with DROP DEFAULT, perform the ALTER TYPE, and then use SET DEFAULT to add a suitable new default. Similar considerations apply to indexes and constraints involving the column.

If a table has any descendant tables, it is not permitted to add, rename, or change the type of a column in the parent table without doing the same to the descendants. That is, ALTER TABLE ONLY will be rejected. This ensures that the descendants always have columns matching the parent.

A recursive DROP COLUMN operation will remove a descendant table's column only if the descendant does not inherit that column from any other parents and never had an independent definition of the column. A nonrecursive DROP COLUMN (i.e., ALTER TABLE ONLY ... DROP COLUMN) never removes any descendant columns, but instead marks them as independently defined rather than inherited.

The TRIGGER, CLUSTER, OWNER, and TABLESPACE actions never recurse to descendant tables; that is, they always act as though ONLY were specified. Adding a constraint can recurse only for CHECK constraints.

Changing any part of a system catalog table is not permitted.

Refer to *CREATE TABLE* for a further description of valid parameters. Chapter 5 has further information on inheritance.

Examples

To add a column of type `varchar` to a table:

```
ALTER TABLE distributors ADD COLUMN address varchar(30);
```

To drop a column from a table:

```
ALTER TABLE distributors DROP COLUMN address RESTRICT;
```

To change the types of two existing columns in one operation:

```
ALTER TABLE distributors
    ALTER COLUMN address TYPE varchar(80),
    ALTER COLUMN name TYPE varchar(100);
```

To change an integer column containing UNIX timestamps to `timestamp with time zone` via a `USING` clause:

```
ALTER TABLE foo
    ALTER COLUMN foo_timestamp TYPE timestamp with time zone
    USING
        timestamp with time zone 'epoch' + foo_timestamp * interval '1 second';
```

The same, when the column has a default expression that won't automatically cast to the new data type:

```
ALTER TABLE foo
    ALTER COLUMN foo_timestamp DROP DEFAULT,
    ALTER COLUMN foo_timestamp TYPE timestamp with time zone
    USING
        timestamp with time zone 'epoch' + foo_timestamp * interval '1 second',
    ALTER COLUMN foo_timestamp SET DEFAULT now();
```

To rename an existing column:

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

To rename an existing table:

```
ALTER TABLE distributors RENAME TO suppliers;
```

To add a not-null constraint to a column:

```
ALTER TABLE distributors ALTER COLUMN street SET NOT NULL;
```

To remove a not-null constraint from a column:

```
ALTER TABLE distributors ALTER COLUMN street DROP NOT NULL;
```

To add a check constraint to a table:

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5);
```

To remove a check constraint from a table and all its children:

```
ALTER TABLE distributors DROP CONSTRAINT zipchk;
```

To add a foreign key constraint to a table:

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address) REFERENCES addresses (
```

To add a (multicolumn) unique constraint to a table:

```
ALTER TABLE distributors ADD CONSTRAINT dist_id_zipcode_key UNIQUE (dist_id, zipcode);
```

To add an automatically named primary key constraint to a table, noting that a table can only ever have one primary key:

```
ALTER TABLE distributors ADD PRIMARY KEY (dist_id);
```

To move a table to a different tablespace:

```
ALTER TABLE distributors SET TABLESPACE fasttablespace;
```

To move a table to a different schema:

```
ALTER TABLE myschema.distributors SET SCHEMA yourschema;
```

Compatibility

The `ADD`, `DROP`, and `SET DEFAULT` forms conform with the SQL standard. The other forms are PostgreSQL extensions of the SQL standard. Also, the ability to specify more than one manipulation in a single `ALTER TABLE` command is an extension.

`ALTER TABLE DROP COLUMN` can be used to drop the only column of a table, leaving a zero-column table. This is an extension of SQL, which disallows zero-column tables.

ALTER TABLESPACE

Name

ALTER TABLESPACE — change the definition of a tablespace

Synopsis

```
ALTER TABLESPACE name RENAME TO newname
ALTER TABLESPACE name OWNER TO newowner
```

Description

ALTER TABLESPACE changes the definition of a tablespace.

You must own the tablespace to use ALTER TABLESPACE. To alter the owner, you must also be a direct or indirect member of the new owning role. (Note that superusers have these privileges automatically.)

Parameters

name

The name of an existing tablespace.

newname

The new name of the tablespace. The new name cannot begin with pg_, as such names are reserved for system tablespaces.

newowner

The new owner of the tablespace.

Examples

Rename tablespace index_space to fast_raid:

```
ALTER TABLESPACE index_space RENAME TO fast_raid;
```

Change the owner of tablespace index_space:

```
ALTER TABLESPACE index_space OWNER TO mary;
```

Compatibility

There is no `ALTER TABLESPACE` statement in the SQL standard.

See Also

CREATE TABLESPACE, *DROP TABLESPACE*

ALTER TRIGGER

Name

ALTER TRIGGER — change the definition of a trigger

Synopsis

```
ALTER TRIGGER name ON table RENAME TO newname
```

Description

ALTER TRIGGER changes properties of an existing trigger. The RENAME clause changes the name of the given trigger without otherwise changing the trigger definition.

You must own the table on which the trigger acts to be allowed to change its properties.

Parameters

name

The name of an existing trigger to alter.

table

The name of the table on which this trigger acts.

newname

The new name for the trigger.

Notes

The ability to temporarily enable or disable a trigger is provided by *ALTER TABLE*, not by ALTER TRIGGER, because ALTER TRIGGER has no convenient way to express the option of enabling or disabling all of a table's triggers at once.

Examples

To rename an existing trigger:

```
ALTER TRIGGER emp_stamp ON emp RENAME TO emp_track_chgs;
```

Compatibility

`ALTER TRIGGER` is a PostgreSQL extension of the SQL standard.

See Also

ALTER TABLE

ALTER TYPE

Name

ALTER TYPE — change the definition of a type

Synopsis

```
ALTER TYPE name OWNER TO new_owner
ALTER TYPE name SET SCHEMA new_schema
```

Description

ALTER TYPE changes the definition of an existing type. The only currently available capabilities are changing the owner and schema of a type.

You must own the type to use ALTER TYPE. To change the schema of a type, you must also have CREATE privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the type's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the type. However, a superuser can alter ownership of any type anyway.)

Parameters

name

The name (possibly schema-qualified) of an existing type to alter.

new_owner

The user name of the new owner of the type.

new_schema

The new schema for the type.

Examples

To change the owner of the user-defined type email to joe:

```
ALTER TYPE email OWNER TO joe;
```

To change the schema of the user-defined type `email` to `customers`:

```
ALTER TYPE email SET SCHEMA customers;
```

Compatibility

There is no `ALTER TYPE` statement in the SQL standard.

ALTER USER

Name

ALTER USER — change a database role

Synopsis

```
ALTER USER name [ [ WITH ] option [ ... ] ]

where option can be:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'

ALTER USER name RENAME TO newname

ALTER USER name SET configuration_parameter { TO | = } { value | DEFAULT }
ALTER USER name RESET configuration_parameter
```

Description

ALTER USER is now an alias for *ALTER ROLE*.

Compatibility

The ALTER USER statement is a PostgreSQL extension. The SQL standard leaves the definition of users to the implementation.

See Also

ALTER ROLE

ANALYZE

Name

ANALYZE — collect statistics about a database

Synopsis

```
ANALYZE [ VERBOSE ] [ table [ (column [, ...] ) ] ]
```

Description

ANALYZE collects statistics about the contents of tables in the database, and stores the results in the system table pg_statistic. Subsequently, the query planner uses these statistics to help determine the most efficient execution plans for queries.

With no parameter, ANALYZE examines every table in the current database. With a parameter, ANALYZE examines only that table. It is further possible to give a list of column names, in which case only the statistics for those columns are collected.

Parameters

VERBOSE

Enables display of progress messages.

table

The name (possibly schema-qualified) of a specific table to analyze. Defaults to all tables in the current database.

column

The name of a specific column to analyze. Defaults to all columns.

Outputs

When VERBOSE is specified, ANALYZE emits progress messages to indicate which table is currently being processed. Various statistics about the tables are printed as well.

Notes

It is a good idea to run ANALYZE periodically, or just after making major changes in the contents of a table. Accurate statistics will help the planner to choose the most appropriate query plan, and thereby improve

the speed of query processing. A common strategy is to run *VACUUM* and ANALYZE once a day during a low-usage time of day.

Unlike VACUUM FULL, ANALYZE requires only a read lock on the target table, so it can run in parallel with other activity on the table.

The statistics collected by ANALYZE usually include a list of some of the most common values in each column and a histogram showing the approximate data distribution in each column. One or both of these may be omitted if ANALYZE deems them uninteresting (for example, in a unique-key column, there are no common values) or if the column data type does not support the appropriate operators. There is more information about the statistics in Chapter 22.

For large tables, ANALYZE takes a random sample of the table contents, rather than examining every row. This allows even very large tables to be analyzed in a small amount of time. Note, however, that the statistics are only approximate, and will change slightly each time ANALYZE is run, even if the actual table contents did not change. This may result in small changes in the planner's estimated costs shown by EXPLAIN. In rare situations, this non-determinism will cause the query optimizer to choose a different query plan between runs of ANALYZE. To avoid this, raise the amount of statistics collected by ANALYZE, as described below.

The extent of analysis can be controlled by adjusting the default_statistics_target configuration variable, or on a column-by-column basis by setting the per-column statistics target with ALTER TABLE ... ALTER COLUMN ... SET STATISTICS (see *ALTER TABLE*). The target value sets the maximum number of entries in the most-common-value list and the maximum number of bins in the histogram. The default target value is 10, but this can be adjusted up or down to trade off accuracy of planner estimates against the time taken for ANALYZE and the amount of space occupied in pg_statistic. In particular, setting the statistics target to zero disables collection of statistics for that column. It may be useful to do that for columns that are never used as part of the WHERE, GROUP BY, or ORDER BY clauses of queries, since the planner will have no use for statistics on such columns.

The largest statistics target among the columns being analyzed determines the number of table rows sampled to prepare the statistics. Increasing the target causes a proportional increase in the time and space needed to do ANALYZE.

Compatibility

There is no ANALYZE statement in the SQL standard.

BEGIN

Name

BEGIN — start a transaction block

Synopsis

```
BEGIN [ WORK | TRANSACTION ] [ transaction_mode [, ...] ]

where transaction_mode is one of:

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
```

Description

BEGIN initiates a transaction block, that is, all statements after a BEGIN command will be executed in a single transaction until an explicit *COMMIT* or *ROLLBACK* is given. By default (without BEGIN), PostgreSQL executes transactions in "autocommit" mode, that is, each statement is executed in its own transaction and a commit is implicitly performed at the end of the statement (if execution was successful, otherwise a rollback is done).

Statements are executed more quickly in a transaction block, because transaction start/commit requires significant CPU and disk activity. Execution of multiple statements inside a transaction is also useful to ensure consistency when making several related changes: other sessions will be unable to see the intermediate states wherein not all the related updates have been done.

If the isolation level or read/write mode is specified, the new transaction has those characteristics, as if *SET TRANSACTION* was executed.

Parameters

WORK
TRANSACTION

Optional key words. They have no effect.

Refer to *SET TRANSACTION* for information on the meaning of the other parameters to this statement.

Notes

START TRANSACTION has the same functionality as BEGIN.

Use *COMMIT* or *ROLLBACK* to terminate a transaction block.

Issuing `BEGIN` when already inside a transaction block will provoke a warning message. The state of the transaction is not affected. To nest transactions within a transaction block, use savepoints (see *SAVEPOINT*).

For reasons of backwards compatibility, the commas between successive *`transaction_modes`* may be omitted.

Examples

To begin a transaction block:

```
BEGIN;
```

Compatibility

`BEGIN` is a PostgreSQL language extension. It is equivalent to the SQL-standard command *START TRANSACTION*, whose reference page contains additional compatibility information.

Incidentally, the `BEGIN` key word is used for a different purpose in embedded SQL. You are advised to be careful about the transaction semantics when porting database applications.

See Also

COMMIT, *ROLLBACK*, *START TRANSACTION*, *SAVEPOINT*

CHECKPOINT

Name

CHECKPOINT — force a transaction log checkpoint

Synopsis

```
CHECKPOINT
```

Description

Write-Ahead Logging (WAL) puts a checkpoint in the transaction log every so often. (To adjust the automatic checkpoint interval, see the run-time configuration options checkpoint_segments and checkpoint_timeout.) The CHECKPOINT command forces an immediate checkpoint when the command is issued, without waiting for a scheduled checkpoint.

A checkpoint is a point in the transaction log sequence at which all data files have been updated to reflect the information in the log. All data files will be flushed to disk. Refer to Chapter 27 for more information about the WAL system.

Only superusers may call CHECKPOINT. The command is not intended for use during normal operation.

Compatibility

The CHECKPOINT command is a PostgreSQL language extension.

CLOSE

Name

CLOSE — close a cursor

Synopsis

```
CLOSE name
```

Description

CLOSE frees the resources associated with an open cursor. After the cursor is closed, no subsequent operations are allowed on it. A cursor should be closed when it is no longer needed.

Every non-holdable open cursor is implicitly closed when a transaction is terminated by COMMIT or ROLLBACK. A holdable cursor is implicitly closed if the transaction that created it aborts via ROLLBACK. If the creating transaction successfully commits, the holdable cursor remains open until an explicit CLOSE is executed, or the client disconnects.

Parameters

name

The name of an open cursor to close.

Notes

PostgreSQL does not have an explicit OPEN cursor statement; a cursor is considered open when it is declared. Use the *DECLARE* statement to declare a cursor.

You can see all available cursors by querying the pg_cursors system view.

Examples

Close the cursor liahona:

```
CLOSE liahona;
```

Compatibility

`CLOSE` is fully conforming with the SQL standard.

See Also

DECLARE, *FETCH*, *MOVE*

CLUSTER

Name

CLUSTER — cluster a table according to an index

Synopsis

```
CLUSTER indexname ON tablename
CLUSTER tablename
CLUSTER
```

Description

CLUSTER instructs PostgreSQL to cluster the table specified by *tablename* based on the index specified by *indexname*. The index must already have been defined on *tablename*.

When a table is clustered, it is physically reordered based on the index information. Clustering is a one-time operation: when the table is subsequently updated, the changes are not clustered. That is, no attempt is made to store new or updated rows according to their index order. If one wishes, one can periodically recluster by issuing the command again.

When a table is clustered, PostgreSQL remembers on which index it was clustered. The form CLUSTER *tablename* reclusters the table on the same index that it was clustered before.

CLUSTER without any parameter reclusters all the tables in the current database that the calling user owns, or all tables if called by a superuser. (Never-clustered tables are not included.) This form of CLUSTER cannot be executed inside a transaction block.

When a table is being clustered, an ACCESS EXCLUSIVE lock is acquired on it. This prevents any other database operations (both reads and writes) from operating on the table until the CLUSTER is finished.

Parameters

indexname

The name of an index.

tablename

The name (possibly schema-qualified) of a table.

Notes

CLUSTER loses all visibility information of tuples, which makes the table look empty to any snapshot that was taken before the CLUSTER command finished. That makes CLUSTER unsuitable for applications where transactions that access the table being clustered are run concurrently with CLUSTER. This is most visible

with serializable transactions, because they take only one snapshot at the beginning of the transaction, but read-committed transactions are also affected.

In cases where you are accessing single rows randomly within a table, the actual order of the data in the table is unimportant. However, if you tend to access some data more than others, and there is an index that groups them together, you will benefit from using `CLUSTER`. If you are requesting a range of indexed values from a table, or a single indexed value that has multiple rows that match, `CLUSTER` will help because once the index identifies the table page for the first row that matches, all other rows that match are probably already on the same table page, and so you save disk accesses and speed up the query.

During the cluster operation, a temporary copy of the table is created that contains the table data in the index order. Temporary copies of each index on the table are created as well. Therefore, you need free space on disk at least equal to the sum of the table size and the index sizes.

Because `CLUSTER` remembers the clustering information, one can cluster the tables one wants clustered manually the first time, and setup a timed event similar to `VACUUM` so that the tables are periodically reclustered.

Because the planner records statistics about the ordering of tables, it is advisable to run *ANALYZE* on the newly clustered table. Otherwise, the planner may make poor choices of query plans.

There is another way to cluster data. The `CLUSTER` command reorders the original table by scanning it using the index you specify. This can be slow on large tables because the rows are fetched from the table in index order, and if the table is disordered, the entries are on random pages, so there is one disk page retrieved for every row moved. (PostgreSQL has a cache, but the majority of a big table will not fit in the cache.) The other way to cluster a table is to use

```
CREATE TABLE newtable AS
    SELECT * FROM table ORDER BY columnlist;
```

which uses the PostgreSQL sorting code to produce the desired order; this is usually much faster than an index scan for disordered data. Then you drop the old table, use `ALTER TABLE ... RENAME` to rename *`newtable`* to the old name, and recreate the table's indexes. The big disadvantage of this approach is that it does not preserve OIDs, constraints, foreign key relationships, granted privileges, and other ancillary properties of the table — all such items must be manually recreated. Another disadvantage is that this way requires a sort temporary file about the same size as the table itself, so peak disk usage is about three times the table size instead of twice the table size.

Examples

Cluster the table `employees` on the basis of its index `emp_ind`:

```
CLUSTER emp_ind ON emp;
```

Cluster the `employees` table using the same index that was used before:

```
CLUSTER emp;
```

Cluster all tables in the database that have previously been clustered:

```
CLUSTER;
```

Compatibility

There is no `CLUSTER` statement in the SQL standard.

See Also

clusterdb

COMMENT

Name

COMMENT — define or change the comment of an object

Synopsis

```
COMMENT ON
{
  TABLE object_name |
  COLUMN table_name.column_name |
  AGGREGATE agg_name (agg_type [, ...] ) |
  CAST (sourcetype AS targettype) |
  CONSTRAINT constraint_name ON table_name |
  CONVERSION object_name |
  DATABASE object_name |
  DOMAIN object_name |
  FUNCTION func_name ( [ [ argmode ] [ argname ] argtype [, ...] ] ) |
  INDEX object_name |
  LARGE OBJECT large_object_oid |
  OPERATOR op (leftoperand_type, rightoperand_type) |
  OPERATOR CLASS object_name USING index_method |
  [ PROCEDURAL ] LANGUAGE object_name |
  ROLE object_name |
  RULE rule_name ON table_name |
  SCHEMA object_name |
  SEQUENCE object_name |
  TABLESPACE object_name |
  TRIGGER trigger_name ON table_name |
  TYPE object_name |
  VIEW object_name
} IS 'text'
```

Description

COMMENT stores a comment about a database object.

To modify a comment, issue a new COMMENT command for the same object. Only one comment string is stored for each object. To remove a comment, write NULL in place of the text string. Comments are automatically dropped when the object is dropped.

Comments can be easily retrieved with the psql commands \dd, \d+, and \l+. Other user interfaces to retrieve comments can be built atop the same built-in functions that psql uses, namely obj_description, col_description, and shobj_description (see Table 9-44).

Parameters

object_name
table_name.column_name
agg_name
constraint_name
func_name
op
rule_name
trigger_name

The name of the object to be commented. Names of tables, aggregates, domains, functions, indexes, operators, operator classes, sequences, types, and views may be schema-qualified.

agg_type

An input data type on which the aggregate function operates. To reference a zero-argument aggregate function, write * in place of the list of input data types.

sourcetype

The name of the source data type of the cast.

targettype

The name of the target data type of the cast.

argmode

The mode of a function argument: either IN, OUT, or INOUT. If omitted, the default is IN. Note that COMMENT ON FUNCTION does not actually pay any attention to OUT arguments, since only the input arguments are needed to determine the function's identity. So it is sufficient to list the IN and INOUT arguments.

argname

The name of a function argument. Note that COMMENT ON FUNCTION does not actually pay any attention to argument names, since only the argument data types are needed to determine the function's identity.

argtype

The data type(s) of the function's arguments (optionally schema-qualified), if any.

large_object_oid

The OID of the large object.

PROCEDURAL

This is a noise word.

text

The new comment, written as a string literal; or NULL to drop the comment.

Notes

There is presently no security mechanism for comments: any user connected to a database can see all the comments for objects in that database (although only superusers can change comments for objects that they don't own). For shared objects such as databases, roles, and tablespaces comments are stored globally and any user connected to any database can see all the comments for shared objects. Therefore, don't put security-critical information in comments.

Examples

Attach a comment to the table `mytable`:

```
COMMENT ON TABLE mytable IS 'This is my table.';
```

Remove it again:

```
COMMENT ON TABLE mytable IS NULL;
```

Some more examples:

```
COMMENT ON AGGREGATE my_aggregate (double precision) IS 'Computes sample variance';
COMMENT ON CAST (text AS int4) IS 'Allow casts from text to int4';
COMMENT ON COLUMN my_table.my_column IS 'Employee ID number';
COMMENT ON CONVERSION my_conv IS 'Conversion to UTF8';
COMMENT ON DATABASE my_database IS 'Development Database';
COMMENT ON DOMAIN my_domain IS 'Email Address Domain';
COMMENT ON FUNCTION my_function (timestamp) IS 'Returns Roman Numeral';
COMMENT ON INDEX my_index IS 'Enforces uniqueness on employee ID';
COMMENT ON LANGUAGE plpython IS 'Python support for stored procedures';
COMMENT ON LARGE OBJECT 346344 IS 'Planning document';
COMMENT ON OPERATOR ^ (text, text) IS 'Performs intersection of two texts';
COMMENT ON OPERATOR - (NONE, text) IS 'This is a prefix operator on text';
COMMENT ON OPERATOR CLASS int4ops USING btree IS '4 byte integer operators for btrees';
COMMENT ON ROLE my_role IS 'Administration group for finance tables';
COMMENT ON RULE my_rule ON my_table IS 'Logs updates of employee records';
COMMENT ON SCHEMA my_schema IS 'Departmental data';
COMMENT ON SEQUENCE my_sequence IS 'Used to generate primary keys';
COMMENT ON TABLE my_schema.my_table IS 'Employee Information';
COMMENT ON TABLESPACE my_tablespace IS 'Tablespace for indexes';
COMMENT ON TRIGGER my_trigger ON my_table IS 'Used for RI';
COMMENT ON TYPE complex IS 'Complex number data type';
COMMENT ON VIEW my_view IS 'View of departmental costs';
```

Compatibility

There is no `COMMENT` command in the SQL standard.

COMMIT

Name

COMMIT — commit the current transaction

Synopsis

```
COMMIT [ WORK | TRANSACTION ]
```

Description

COMMIT commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs.

Parameters

WORK
TRANSACTION

Optional key words. They have no effect.

Notes

Use *ROLLBACK* to abort a transaction.

Issuing COMMIT when not inside a transaction does no harm, but it will provoke a warning message.

Examples

To commit the current transaction and make all changes permanent:

```
COMMIT;
```

Compatibility

The SQL standard only specifies the two forms COMMIT and COMMIT WORK. Otherwise, this command is fully conforming.

See Also

BEGIN, *ROLLBACK*

COMMIT PREPARED

Name

`COMMIT PREPARED` — commit a transaction that was earlier prepared for two-phase commit

Synopsis

```
COMMIT PREPARED transaction_id
```

Description

`COMMIT PREPARED` commits a transaction that is in prepared state.

Parameters

`transaction_id`

The transaction identifier of the transaction that is to be committed.

Notes

To commit a prepared transaction, you must be either the same user that executed the transaction originally, or a superuser. But you do not have to be in the same session that executed the transaction.

This command cannot be executed inside a transaction block. The prepared transaction is committed immediately.

All currently available prepared transactions are listed in the `pg_prepared_xacts` system view.

Examples

Commit the transaction identified by the transaction identifier `foobar`:

```
COMMIT PREPARED 'foobar';
```

See Also

PREPARE TRANSACTION, *ROLLBACK PREPARED*

COPY

Name

COPY — copy data between a file and a table

Synopsis

```
COPY tablename [ ( column [, ...] ) ]
    FROM { 'filename' | STDIN }
    [ [ WITH ]
          [ BINARY ]
          [ OIDS ]
          [ DELIMITER [ AS ] 'delimiter' ]
          [ NULL [ AS ] 'null string' ]
          [ CSV [ HEADER ]
                [ QUOTE [ AS ] 'quote' ]
                [ ESCAPE [ AS ] 'escape' ]
                [ FORCE NOT NULL column [, ...] ]

COPY { tablename [ ( column [, ...] ) ] | ( query ) }
    TO { 'filename' | STDOUT }
    [ [ WITH ]
          [ BINARY ]
          [ HEADER ]
          [ OIDS ]
          [ DELIMITER [ AS ] 'delimiter' ]
          [ NULL [ AS ] 'null string' ]
          [ CSV [ HEADER ]
                [ QUOTE [ AS ] 'quote' ]
                [ ESCAPE [ AS ] 'escape' ]
                [ FORCE QUOTE column [, ...] ]
```

Description

COPY moves data between PostgreSQL tables and standard file-system files. COPY TO copies the contents of a table *to* a file, while COPY FROM copies data *from* a file to a table (appending the data to whatever is in the table already). COPY TO can also copy the results of a SELECT query.

If a list of columns is specified, COPY will only copy the data in the specified columns to or from the file. If there are any columns in the table that are not in the column list, COPY FROM will insert the default values for those columns.

COPY with a file name instructs the PostgreSQL server to directly read from or write to a file. The file must be accessible to the server and the name must be specified from the viewpoint of the server. When STDIN or STDOUT is specified, data is transmitted via the connection between the client and the server.

Parameters

tablename

The name (optionally schema-qualified) of an existing table.

column

An optional list of columns to be copied. If no column list is specified, all columns of the table will be copied.

query

A *SELECT* or *VALUES* command whose results are to be copied. Note that parentheses are required around the query.

filename

The absolute path name of the input or output file. Windows users might need to use an `E''` string and double backslashes used as path separators.

`STDIN`

Specifies that input comes from the client application.

`STDOUT`

Specifies that output goes to the client application.

`BINARY`

Causes all data to be stored or read in binary format rather than as text. You cannot specify the `DELIMITER`, `NULL`, or `CSV` options in binary mode.

`OIDS`

Specifies copying the OID for each row. (An error is raised if `OIDS` is specified for a table that does not have OIDs, or in the case of copying a *query*.)

delimiter

The single ASCII character that separates columns within each row (line) of the file. The default is a tab character in text mode, a comma in `CSV` mode.

null string

The string that represents a null value. The default is `\N` (backslash-N) in text mode, and a empty value with no quotes in `CSV` mode. You might prefer an empty string even in text mode for cases where you don't want to distinguish nulls from empty strings.

> **Note:** When using `COPY FROM`, any data item that matches this string will be stored as a null value, so you should make sure that you use the same string as you used with `COPY TO`.

`CSV`

Selects Comma Separated Value (`CSV`) mode.

`HEADER`

Specifies the file contains a header line with the names of each column in the file. On output, the first line contains the column names from the table, and on input, the first line is ignored.

`quote`

Specifies the ASCII quotation character in `CSV` mode. The default is double-quote.

`escape`

Specifies the ASCII character that should appear before a `QUOTE` data character value in `CSV` mode. The default is the `QUOTE` value (usually double-quote).

`FORCE QUOTE`

In `CSV COPY TO` mode, forces quoting to be used for all non-`NULL` values in each specified column. `NULL` output is never quoted.

`FORCE NOT NULL`

In `CSV COPY FROM` mode, process each specified column as though it were quoted and hence not a `NULL` value. For the default null string in `CSV` mode (″), this causes missing values to be input as zero-length strings.

Outputs

On successful completion, a `COPY` command returns a command tag of the form

`COPY` *`count`*

The *`count`* is the number of rows copied.

Notes

`COPY` can only be used with plain tables, not with views. However, you can write `COPY (SELECT * FROM` *`viewname`*`) TO ...`.

The `BINARY` key word causes all data to be stored/read as binary format rather than as text. It is somewhat faster than the normal text mode, but a binary-format file is less portable across machine architectures and PostgreSQL versions.

You must have select privilege on the table whose values are read by `COPY TO`, and insert privilege on the table into which values are inserted by `COPY FROM`.

Files named in a `COPY` command are read or written directly by the server, not by the client application. Therefore, they must reside on or be accessible to the database server machine, not the client. They must be accessible to and readable or writable by the PostgreSQL user (the user ID the server runs as), not the client. `COPY` naming a file is only allowed to database superusers, since it allows reading or writing any file that the server has privileges to access.

Do not confuse `COPY` with the psql instruction `\copy`. `\copy` invokes `COPY FROM STDIN` or `COPY TO STDOUT`, and then fetches/stores the data in a file accessible to the psql client. Thus, file accessibility and access rights depend on the client rather than the server when `\copy` is used.

It is recommended that the file name used in COPY always be specified as an absolute path. This is enforced by the server in the case of COPY TO, but for COPY FROM you do have the option of reading from a file specified by a relative path. The path will be interpreted relative to the working directory of the server process (normally the cluster's data directory), not the client's working directory.

COPY FROM will invoke any triggers and check constraints on the destination table. However, it will not invoke rules.

COPY input and output is affected by DateStyle. To ensure portability to other PostgreSQL installations that might use non-default DateStyle settings, DateStyle should be set to ISO before using COPY TO.

COPY stops operation at the first error. This should not lead to problems in the event of a COPY TO, but the target table will already have received earlier rows in a COPY FROM. These rows will not be visible or accessible, but they still occupy disk space. This may amount to a considerable amount of wasted disk space if the failure happened well into a large copy operation. You may wish to invoke VACUUM to recover the wasted space.

File Formats

Text Format

When COPY is used without the BINARY or CSV options, the data read or written is a text file with one line per table row. Columns in a row are separated by the delimiter character. The column values themselves are strings generated by the output function, or acceptable to the input function, of each attribute's data type. The specified null string is used in place of columns that are null. COPY FROM will raise an error if any line of the input file contains more or fewer columns than are expected. If OIDS is specified, the OID is read or written as the first column, preceding the user data columns.

End of data can be represented by a single line containing just backslash-period (\.). An end-of-data marker is not necessary when reading from a file, since the end of file serves perfectly well; it is needed only when copying data to or from client applications using pre-3.0 client protocol.

Backslash characters (\) may be used in the COPY data to quote data characters that might otherwise be taken as row or column delimiters. In particular, the following characters *must* be preceded by a backslash if they appear as part of a column value: backslash itself, newline, carriage return, and the current delimiter character.

The specified null string is sent by COPY TO without adding any backslashes; conversely, COPY FROM matches the input against the null string before removing backslashes. Therefore, a null string such as \N cannot be confused with the actual data value \N (which would be represented as \\N).

The following special backslash sequences are recognized by COPY FROM:

Sequence	Represents
\b	Backspace (ASCII 8)
\f	Form feed (ASCII 12)
\n	Newline (ASCII 10)
\r	Carriage return (ASCII 13)
\t	Tab (ASCII 9)

Sequence	Represents
\v	Vertical tab (ASCII 11)
digits	Backslash followed by one to three octal digits specifies the character with that numeric code
\x*digits*	Backslash x followed by one or two hex digits specifies the character with that numeric code

Presently, COPY TO will never emit an octal or hex-digits backslash sequence, but it does use the other sequences listed above for those control characters.

Any other backslashed character that is not mentioned in the above table will be taken to represent itself. However, beware of adding backslashes unnecessarily, since that might accidentally produce a string matching the end-of-data marker (\.) or the null string (\N by default). These strings will be recognized before any other backslash processing is done.

It is strongly recommended that applications generating COPY data convert data newlines and carriage returns to the \n and \r sequences respectively. At present it is possible to represent a data carriage return by a backslash and carriage return, and to represent a data newline by a backslash and newline. However, these representations might not be accepted in future releases. They are also highly vulnerable to corruption if the COPY file is transferred across different machines (for example, from Unix to Windows or vice versa).

COPY TO will terminate each row with a Unix-style newline ("\n"). Servers running on Microsoft Windows instead output carriage return/newline ("\r\n"), but only for COPY to a server file; for consistency across platforms, COPY TO STDOUT always sends "\n" regardless of server platform. COPY FROM can handle lines ending with newlines, carriage returns, or carriage return/newlines. To reduce the risk of error due to un-backslashed newlines or carriage returns that were meant as data, COPY FROM will complain if the line endings in the input are not all alike.

CSV Format

This format is used for importing and exporting the Comma Separated Value (CSV) file format used by many other programs, such as spreadsheets. Instead of the escaping used by PostgreSQL's standard text mode, it produces and recognizes the common CSV escaping mechanism.

The values in each record are separated by the DELIMITER character. If the value contains the delimiter character, the QUOTE character, the NULL string, a carriage return, or line feed character, then the whole value is prefixed and suffixed by the QUOTE character, and any occurrence within the value of a QUOTE character or the ESCAPE character is preceded by the escape character. You can also use FORCE QUOTE to force quotes when outputting non-NULL values in specific columns.

The CSV format has no standard way to distinguish a NULL value from an empty string. PostgreSQL's COPY handles this by quoting. A NULL is output as the NULL string and is not quoted, while a data value matching the NULL string is quoted. Therefore, using the default settings, a NULL is written as an unquoted empty string, while an empty string is written with double quotes (""). Reading values follows similar rules. You can use FORCE NOT NULL to prevent NULL input comparisons for specific columns.

Because backslash is not a special character in the CSV format, \., the end-of-data marker, could also appear as a data value. To avoid any misinterpretation, a \. data value appearing as a lone entry on a line is automatically quoted on output, and on input, if quoted, is not interpreted as the end-of-data marker. If

you are loading a file created by another application that has a single unquoted column and might have a value of `\.`, you might need to quote that value in the input file.

> **Note:** In `CSV` mode, all characters are significant. A quoted value surrounded by white space, or any characters other than `DELIMITER`, will include those characters. This can cause errors if you import data from a system that pads `CSV` lines with white space out to some fixed width. If such a situation arises you might need to preprocess the `CSV` file to remove the trailing white space, before importing the data into PostgreSQL.

> **Note:** CSV mode will both recognize and produce CSV files with quoted values containing embedded carriage returns and line feeds. Thus the files are not strictly one line per table row like text-mode files.

> **Note:** Many programs produce strange and occasionally perverse CSV files, so the file format is more a convention than a standard. Thus you might encounter some files that cannot be imported using this mechanism, and `COPY` might produce files that other programs cannot process.

Binary Format

The file format used for `COPY BINARY` changed in PostgreSQL 7.4. The new format consists of a file header, zero or more tuples containing the row data, and a file trailer. Headers and data are now in network byte order.

File Header

The file header consists of 15 bytes of fixed fields, followed by a variable-length header extension area. The fixed fields are:

Signature

> 11-byte sequence `PGCOPY\n\377\r\n\0` — note that the zero byte is a required part of the signature. (The signature is designed to allow easy identification of files that have been munged by a non-8-bit-clean transfer. This signature will be changed by end-of-line-translation filters, dropped zero bytes, dropped high bits, or parity changes.)

Flags field

> 32-bit integer bit mask to denote important aspects of the file format. Bits are numbered from 0 (LSB) to 31 (MSB). Note that this field is stored in network byte order (most significant byte first), as are all the integer fields used in the file format. Bits 16-31 are reserved to denote critical file format issues; a reader should abort if it finds an unexpected bit set in this range. Bits 0-15 are reserved to signal backwards-compatible format issues; a reader should simply ignore any unexpected bits set in this range. Currently only one flag bit is defined, and the rest must be zero:
>
> Bit 16
>
> > if 1, OIDs are included in the data; if 0, not

Header extension area length

> 32-bit integer, length in bytes of remainder of header, not including self. Currently, this is zero, and the first tuple follows immediately. Future changes to the format might allow additional data to be present in the header. A reader should silently skip over any header extension data it does not know what to do with.

The header extension area is envisioned to contain a sequence of self-identifying chunks. The flags field is not intended to tell readers what is in the extension area. Specific design of header extension contents is left for a later release.

This design allows for both backwards-compatible header additions (add header extension chunks, or set low-order flag bits) and non-backwards-compatible changes (set high-order flag bits to signal such changes, and add supporting data to the extension area if needed).

Tuples

Each tuple begins with a 16-bit integer count of the number of fields in the tuple. (Presently, all tuples in a table will have the same count, but that might not always be true.) Then, repeated for each field in the tuple, there is a 32-bit length word followed by that many bytes of field data. (The length word does not include itself, and can be zero.) As a special case, -1 indicates a NULL field value. No value bytes follow in the NULL case.

There is no alignment padding or any other extra data between fields.

Presently, all data values in a `COPY BINARY` file are assumed to be in binary format (format code one). It is anticipated that a future extension may add a header field that allows per-column format codes to be specified.

To determine the appropriate binary format for the actual tuple data you should consult the PostgreSQL source, in particular the `*send` and `*recv` functions for each column's data type (typically these functions are found in the `src/backend/utils/adt/` directory of the source distribution).

If OIDs are included in the file, the OID field immediately follows the field-count word. It is a normal field except that it's not included in the field-count. In particular it has a length word — this will allow handling of 4-byte vs. 8-byte OIDs without too much pain, and will allow OIDs to be shown as null if that ever proves desirable.

File Trailer

The file trailer consists of a 16-bit integer word containing -1. This is easily distinguished from a tuple's field-count word.

A reader should report an error if a field-count word is neither -1 nor the expected number of columns. This provides an extra check against somehow getting out of sync with the data.

Examples

The following example copies a table to the client using the vertical bar (|) as the field delimiter:

```
COPY country TO STDOUT WITH DELIMITER '|';
```

To copy data from a file into the `country` table:

```
COPY country FROM '/usr1/proj/bray/sql/country_data';
```

To copy into a file just the countries whose names start with 'A':

```
COPY (SELECT * FROM country WHERE country_name LIKE 'A%') TO '/usr1/proj/bray/sql/a_list_co
```

Here is a sample of data suitable for copying into a table from `STDIN`:

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
ZM      ZAMBIA
ZW      ZIMBABWE
```

Note that the white space on each line is actually a tab character.

The following is the same data, output in binary format. The data is shown after filtering through the Unix utility `od -c`. The table has three columns; the first has type `char(2)`, the second has type `text`, and the third has type `integer`. All the rows have a null value in the third column.

```
0000000   P   G   C   O   P   Y  \n 377  \r  \n  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0 003  \0  \0  \0 002   A   F  \0  \0  \0 013   A
0000040   F   G   H   A   N   I   S   T   A   N 377 377 377 377  \0 003
0000060  \0  \0  \0 002   A   L  \0  \0  \0 007   A   L   B   A   N   I
0000100   A 377 377 377 377  \0 003  \0  \0  \0 002   D   Z  \0  \0  \0
0000120 007   A   L   G   E   R   I   A 377 377 377 377  \0 003  \0  \0
0000140  \0 002   Z   M  \0  \0  \0 006   Z   A   M   B   I   A 377 377
0000160 377 377  \0 003  \0  \0  \0 002   Z   W  \0  \0  \0  \b   Z   I
0000200   M   B   A   B   W   E 377 377 377 377 377 377
```

Compatibility

There is no `COPY` statement in the SQL standard.

The following syntax was used before PostgreSQL version 7.3 and is still supported:

```
COPY [ BINARY ] tablename [ WITH OIDS ]
    FROM { 'filename' | STDIN }
    [ [USING] DELIMITERS 'delimiter' ]
```

```
    [ WITH NULL AS 'null string' ]

COPY [ BINARY ] tablename [ WITH OIDS ]
    TO { 'filename' | STDOUT }
    [ [USING] DELIMITERS 'delimiter' ]
    [ WITH NULL AS 'null string' ]
```

CREATE AGGREGATE

Name

CREATE AGGREGATE — define a new aggregate function

Synopsis

```
CREATE AGGREGATE name ( input_data_type [ , ... ] ) (
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , FINALFUNC = ffunc ]
    [ , INITCOND = initial_condition ]
    [ , SORTOP = sort_operator ]
)

or the old syntax

CREATE AGGREGATE name (
    BASETYPE = base_type,
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , FINALFUNC = ffunc ]
    [ , INITCOND = initial_condition ]
    [ , SORTOP = sort_operator ]
)
```

Description

CREATE AGGREGATE defines a new aggregate function. Some basic and commonly-used aggregate functions are included with the distribution; they are documented in Section 9.15. If one defines new types or needs an aggregate function not already provided, then CREATE AGGREGATE can be used to provide the desired features.

If a schema name is given (for example, CREATE AGGREGATE myschema.myagg ...) then the aggregate function is created in the specified schema. Otherwise it is created in the current schema.

An aggregate function is identified by its name and input data type(s). Two aggregates in the same schema can have the same name if they operate on different input types. The name and input data type(s) of an aggregate must also be distinct from the name and input data type(s) of every ordinary function in the same schema.

An aggregate function is made from one or two ordinary functions: a state transition function *sfunc*, and an optional final calculation function *ffunc*. These are used as follows:

```
sfunc( internal-state, next-data-values ) ---> next-internal-state
ffunc( internal-state ) ---> aggregate-value
```

PostgreSQL creates a temporary variable of data type *stype* to hold the current internal state of the aggregate. At each input row, the aggregate argument value(s) are calculated and the state transition function is invoked with the current state value and the new argument value(s) to calculate a new internal state value. After all the rows have been processed, the final function is invoked once to calculate the aggregate's return value. If there is no final function then the ending state value is returned as-is.

An aggregate function may provide an initial condition, that is, an initial value for the internal state value. This is specified and stored in the database as a value of type `text`, but it must be a valid external representation of a constant of the state value data type. If it is not supplied then the state value starts out null.

If the state transition function is declared "strict", then it cannot be called with null inputs. With such a transition function, aggregate execution behaves as follows. Rows with any null input values are ignored (the function is not called and the previous state value is retained). If the initial state value is null, then at the first row with all-nonnull input values, the first argument value replaces the state value, and the transition function is invoked at subsequent rows with all-nonnull input values. This is handy for implementing aggregates like `max`. Note that this behavior is only available when *state_data_type* is the same as the first *input_data_type*. When these types are different, you must supply a nonnull initial condition or use a nonstrict transition function.

If the state transition function is not strict, then it will be called unconditionally at each input row, and must deal with null inputs and null transition values for itself. This allows the aggregate author to have full control over the aggregate's handling of null values.

If the final function is declared "strict", then it will not be called when the ending state value is null; instead a null result will be returned automatically. (Of course this is just the normal behavior of strict functions.) In any case the final function has the option of returning a null value. For example, the final function for `avg` returns null when it sees there were zero input rows.

Aggregates that behave like `MIN` or `MAX` can sometimes be optimized by looking into an index instead of scanning every input row. If this aggregate can be so optimized, indicate it by specifying a *sort operator*. The basic requirement is that the aggregate must yield the first element in the sort ordering induced by the operator; in other words

```
SELECT agg(col) FROM tab;
```

must be equivalent to

```
SELECT col FROM tab ORDER BY col USING sortop LIMIT 1;
```

Further assumptions are that the aggregate ignores null inputs, and that it delivers a null result if and only if there were no non-null inputs. Ordinarily, a data type's $<$ operator is the proper sort operator for `MIN`, and $>$ is the proper sort operator for `MAX`. Note that the optimization will never actually take effect unless the specified operator is the "less than" or "greater than" strategy member of a B-tree index operator class.

Parameters

name

The name (optionally schema-qualified) of the aggregate function to create.

input_data_type

An input data type on which this aggregate function operates. To create a zero-argument aggregate function, write * in place of the list of input data types. (An example of such an aggregate is count(*).)

base_type

In the old syntax for CREATE AGGREGATE, the input data type is specified by a basetype parameter rather than being written next to the aggregate name. Note that this syntax allows only one input parameter. To define a zero-argument aggregate function, specify the basetype as "ANY" (not *).

sfunc

The name of the state transition function to be called for each input row. For an *N*-argument aggregate function, the *sfunc* must take *N*+1 arguments, the first being of type *state_data_type* and the rest matching the declared input data type(s) of the aggregate. The function must return a value of type *state_data_type*. This function takes the current state value and the current input data value(s), and returns the next state value.

state_data_type

The data type for the aggregate's state value.

ffunc

The name of the final function called to compute the aggregate's result after all input rows have been traversed. The function must take a single argument of type *state_data_type*. The return data type of the aggregate is defined as the return type of this function. If *ffunc* is not specified, then the ending state value is used as the aggregate's result, and the return type is *state_data_type*.

initial_condition

The initial setting for the state value. This must be a string constant in the form accepted for the data type *state_data_type*. If not specified, the state value starts out null.

sort_operator

The associated sort operator for a MIN- or MAX-like aggregate. This is just an operator name (possibly schema-qualified). The operator is assumed to have the same input data types as the aggregate (which must be a single-argument aggregate).

The parameters of CREATE AGGREGATE can be written in any order, not just the order illustrated above.

Examples

See Section 33.10.

Compatibility

CREATE AGGREGATE is a PostgreSQL language extension. The SQL standard does not provide for user-defined aggregate functions.

See Also

ALTER AGGREGATE, *DROP AGGREGATE*

CREATE CAST

Name

CREATE CAST — define a new cast

Synopsis

```
CREATE CAST (sourcetype AS targettype)
    WITH FUNCTION funcname (argtypes)
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (sourcetype AS targettype)
    WITHOUT FUNCTION
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

Description

CREATE CAST defines a new cast. A cast specifies how to perform a conversion between two data types. For example,

```
SELECT CAST(42 AS text);
```

converts the integer constant 42 to type text by invoking a previously specified function, in this case text(int4). (If no suitable cast has been defined, the conversion fails.)

Two types may be *binary compatible*, which means that they can be converted into one another "for free" without invoking any function. This requires that corresponding values use the same internal representation. For instance, the types text and varchar are binary compatible.

By default, a cast can be invoked only by an explicit cast request, that is an explicit CAST(*x* AS *typename*) or *x*::*typename* construct.

If the cast is marked AS ASSIGNMENT then it can be invoked implicitly when assigning a value to a column of the target data type. For example, supposing that foo.f1 is a column of type text, then

```
INSERT INTO foo (f1) VALUES (42);
```

will be allowed if the cast from type integer to type text is marked AS ASSIGNMENT, otherwise not. (We generally use the term *assignment cast* to describe this kind of cast.)

If the cast is marked AS IMPLICIT then it can be invoked implicitly in any context, whether assignment or internally in an expression. For example, since || takes text operands,

```
SELECT 'The time is ' || now();
```

will be allowed only if the cast from type timestamp to text is marked AS IMPLICIT. Otherwise it will be necessary to write the cast explicitly, for example

```
SELECT 'The time is ' || CAST(now() AS text);
```

(We generally use the term *implicit cast* to describe this kind of cast.)

It is wise to be conservative about marking casts as implicit. An overabundance of implicit casting paths can cause PostgreSQL to choose surprising interpretations of commands, or to be unable to resolve commands at all because there are multiple possible interpretations. A good rule of thumb is to make a cast implicitly invokable only for information-preserving transformations between types in the same general type category. For example, the cast from `int2` to `int4` can reasonably be implicit, but the cast from `float8` to `int4` should probably be assignment-only. Cross-type-category casts, such as `text` to `int4`, are best made explicit-only.

To be able to create a cast, you must own the source or the target data type. To create a binary-compatible cast, you must be superuser. (This restriction is made because an erroneous binary-compatible cast conversion can easily crash the server.)

Parameters

sourcetype

The name of the source data type of the cast.

targettype

The name of the target data type of the cast.

funcname(*argtypes*)

The function used to perform the cast. The function name may be schema-qualified. If it is not, the function will be looked up in the schema search path. The function's result data type must match the target type of the cast. Its arguments are discussed below.

`WITHOUT FUNCTION`

Indicates that the source type and the target type are binary compatible, so no function is required to perform the cast.

`AS ASSIGNMENT`

Indicates that the cast may be invoked implicitly in assignment contexts.

`AS IMPLICIT`

Indicates that the cast may be invoked implicitly in any context.

Cast implementation functions may have one to three arguments. The first argument type must be identical to the cast's source type. The second argument, if present, must be type `integer`; it receives the type modifier associated with the destination type, or `-1` if there is none. The third argument, if present, must be type `boolean`; it receives `true` if the cast is an explicit cast, `false` otherwise. (Bizarrely, the SQL spec demands different behaviors for explicit and implicit casts in some cases. This argument is supplied for functions that must implement such casts. It is not recommended that you design your own data types so that this matters.)

Ordinarily a cast must have different source and target data types. However, it is allowed to declare a cast with identical source and target types if it has a cast implementation function with more than one argument. This is used to represent type-specific length coercion functions in the system catalogs. The named function is used to coerce a value of the type to the type modifier value given by its second

argument. (Since the grammar presently permits only certain built-in data types to have type modifiers, this feature is of no use for user-defined target types, but we mention it for completeness.)

When a cast has different source and target types and a function that takes more than one argument, it represents converting from one type to another and applying a length coercion in a single step. When no such entry is available, coercion to a type that uses a type modifier involves two steps, one to convert between data types and a second to apply the modifier.

Notes

Use *DROP CAST* to remove user-defined casts.

Remember that if you want to be able to convert types both ways you need to declare casts both ways explicitly.

Prior to PostgreSQL 7.3, every function that had the same name as a data type, returned that data type, and took one argument of a different type was automatically a cast function. This convention has been abandoned in face of the introduction of schemas and to be able to represent binary compatible casts in the system catalogs. The built-in cast functions still follow this naming scheme, but they have to be shown as casts in the system catalog `pg_cast` as well.

While not required, it is recommended that you continue to follow this old convention of naming cast implementation functions after the target data type. Many users are used to being able to cast data types using a function-style notation, that is *`typename`*(`x`). This notation is in fact nothing more nor less than a call of the cast implementation function; it is not specially treated as a cast. If your conversion functions are not named to support this convention then you will have surprised users. Since PostgreSQL allows overloading of the same function name with different argument types, there is no difficulty in having multiple conversion functions from different types that all use the target type's name.

> **Note:** There is one small lie in the preceding paragraph: there is still one case in which `pg_cast` will be used to resolve the meaning of an apparent function call. If a function call *`name`*(`x`) matches no actual function, but *`name`* is the name of a data type and `pg_cast` shows a binary-compatible cast to this type from the type of `x`, then the call will be construed as an explicit cast. This exception is made so that binary-compatible casts can be invoked using functional syntax, even though they lack any function.

Examples

To create a cast from type `text` to type `int4` using the function `int4(text)`:

```
CREATE CAST (text AS int4) WITH FUNCTION int4(text);
```

(This cast is already predefined in the system.)

Compatibility

The `CREATE CAST` command conforms to the SQL standard, except that SQL does not make provisions for binary-compatible types or extra arguments to implementation functions. `AS IMPLICIT` is a PostgreSQL extension, too.

See Also

CREATE FUNCTION, *CREATE TYPE*, *DROP CAST*

CREATE CONSTRAINT TRIGGER

Name

CREATE CONSTRAINT TRIGGER — define a new constraint trigger

Synopsis

```
CREATE CONSTRAINT TRIGGER name
    AFTER event [ OR ... ]
    ON table_name
    [ FROM referenced_table_name ]
    { NOT DEFERRABLE | [ DEFERRABLE ] { INITIALLY IMMEDIATE | INITIALLY DEFERRED } }
    FOR EACH ROW
    EXECUTE PROCEDURE funcname ( arguments )
```

Description

CREATE CONSTRAINT TRIGGER is used within CREATE TABLE/ALTER TABLE and by pg_dump to create the special triggers for referential integrity. It is not intended for general use.

Parameters

name

The name of the constraint trigger. The actual name of the created trigger will be of the form RI_ConstraintTrigger_0000 (where 0000 is some number assigned by the server). Use this assigned name when dropping the trigger.

event

One of INSERT, UPDATE, or DELETE; this specifies the event that will fire the trigger. Multiple events can be specified using OR.

table_name

The (possibly schema-qualified) name of the table in which the triggering events occur.

referenced_table_name

The (possibly schema-qualified) name of the table referenced by the constraint. Used by foreign key constraints triggers.

DEFERRABLE
NOT DEFERRABLE
INITIALLY IMMEDIATE
INITIALLY DEFERRED

See the *CREATE TABLE* documentation for details of these constraint options.

funcname(*args*)

The function to call as part of the trigger processing. See *CREATE TRIGGER* for details.

Compatibility

`CREATE CONTRAINT TRIGGER` is a PostgreSQL extension of the SQL standard.

CREATE CONVERSION

Name

CREATE CONVERSION — define a new encoding conversion

Synopsis

```
CREATE [ DEFAULT ] CONVERSION name
    FOR source_encoding TO dest_encoding FROM funcname
```

Description

CREATE CONVERSION defines a new conversion between character set encodings. Conversion names may be used in the convert function to specify a particular encoding conversion. Also, conversions that are marked DEFAULT can be used for automatic encoding conversion between client and server. For this purpose, two conversions, from encoding A to B *and* from encoding B to A, must be defined.

To be able to create a conversion, you must have EXECUTE privilege on the function and CREATE privilege on the destination schema.

Parameters

DEFAULT

The DEFAULT clause indicates that this conversion is the default for this particular source to destination encoding. There should be only one default encoding in a schema for the encoding pair.

name

The name of the conversion. The conversion name may be schema-qualified. If it is not, the conversion is defined in the current schema. The conversion name must be unique within a schema.

source_encoding

The source encoding name.

dest_encoding

The destination encoding name.

funcname

The function used to perform the conversion. The function name may be schema-qualified. If it is not, the function will be looked up in the path.

The function must have the following signature:

```
conv_proc(
    integer,  -- source encoding ID
    integer,  -- destination encoding ID
```

```
        cstring,  -- source string (null terminated C string)
        internal, -- destination (fill with a null terminated C string)
        integer   -- source string length
    ) RETURNS void;
```

Notes

Use `DROP CONVERSION` to remove user-defined conversions.

The privileges required to create a conversion may be changed in a future release.

Examples

To create a conversion from encoding `UTF8` to `LATIN1` using `myfunc`:

```
CREATE CONVERSION myconv FOR 'UTF8' TO 'LATIN1' FROM myfunc;
```

Compatibility

`CREATE CONVERSION` is a PostgreSQL extension. There is no `CREATE CONVERSION` statement in the SQL standard.

See Also

ALTER CONVERSION, *CREATE FUNCTION*, *DROP CONVERSION*

CREATE DATABASE

Name

CREATE DATABASE — create a new database

Synopsis

```
CREATE DATABASE name
    [ [ WITH ] [ OWNER [=] dbowner ]
           [ TEMPLATE [=] template ]
           [ ENCODING [=] encoding ]
           [ TABLESPACE [=] tablespace ]
           [ CONNECTION LIMIT [=] connlimit ] ]
```

Description

CREATE DATABASE creates a new PostgreSQL database.

To create a database, you must be a superuser or have the special CREATEDB privilege. See *CREATE USER*.

Normally, the creator becomes the owner of the new database. Superusers can create databases owned by other users, by using the OWNER clause. They can even create databases owned by users with no special privileges. Non-superusers with CREATEDB privilege can only create databases owned by themselves.

By default, the new database will be created by cloning the standard system database template1. A different template can be specified by writing TEMPLATE *name*. In particular, by writing TEMPLATE template0, you can create a virgin database containing only the standard objects predefined by your version of PostgreSQL. This is useful if you wish to avoid copying any installation-local objects that may have been added to template1.

Parameters

name

The name of a database to create.

dbowner

The name of the database user who will own the new database, or DEFAULT to use the default (namely, the user executing the command).

template

The name of the template from which to create the new database, or DEFAULT to use the default template (template1).

encoding

Character set encoding to use in the new database. Specify a string constant (e.g., `'SQL_ASCII'`), or an integer encoding number, or `DEFAULT` to use the default encoding (namely, the encoding of the template database). The character sets supported by the PostgreSQL server are described in Section 21.2.1.

tablespace

The name of the tablespace that will be associated with the new database, or `DEFAULT` to use the template database's tablespace. This tablespace will be the default tablespace used for objects created in this database. See *CREATE TABLESPACE* for more information.

connlimit

How many concurrent connections can be made to this database. -1 (the default) means no limit.

Optional parameters can be written in any order, not only the order illustrated above.

Notes

`CREATE DATABASE` cannot be executed inside a transaction block.

Errors along the line of "could not initialize database directory" are most likely related to insufficient permissions on the data directory, a full disk, or other file system problems.

Use *DROP DATABASE* to remove a database.

The program *createdb* is a wrapper program around this command, provided for convenience.

Although it is possible to copy a database other than `template1` by specifying its name as the template, this is not (yet) intended as a general-purpose "`COPY DATABASE`" facility. The principal limitation is that no other sessions can be connected to the template database while it is being copied. `CREATE DATABASE` will fail if any other connection exists when it starts; otherwise, new connections to the template database are locked out until `CREATE DATABASE` completes. See Section 19.3 for more information.

The `CONNECTION LIMIT` option is only enforced approximately; if two new sessions start at about the same time when just one connection "slot" remains for the database, it is possible that both will fail. Also, the limit is not enforced against superusers.

Examples

To create a new database:

```
CREATE DATABASE lusiadas;
```

To create a database `sales` owned by user `salesapp` with a default tablespace of `salesspace`:

```
CREATE DATABASE sales OWNER salesapp TABLESPACE salesspace;
```

To create a database `music` which supports the ISO-8859-1 character set:

```
CREATE DATABASE music ENCODING 'LATIN1';
```

Compatibility

There is no `CREATE DATABASE` statement in the SQL standard. Databases are equivalent to catalogs, whose creation is implementation-defined.

See Also

ALTER DATABASE, *DROP DATABASE*

CREATE DOMAIN

Name

CREATE DOMAIN — define a new domain

Synopsis

```
CREATE DOMAIN name [ AS ] data_type
    [ DEFAULT expression ]
    [ constraint [ ... ] ]

where constraint is:

[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | CHECK (expression) }
```

Description

CREATE DOMAIN creates a new domain. A domain is essentially a data type with optional constraints (restrictions on the allowed set of values). The user who defines a domain becomes its owner.

If a schema name is given (for example, CREATE DOMAIN myschema.mydomain ...) then the domain is created in the specified schema. Otherwise it is created in the current schema. The domain name must be unique among the types and domains existing in its schema.

Domains are useful for abstracting common constraints on fields into a single location for maintenance. For example, several tables might contain email address columns, all requiring the same CHECK constraint to verify the address syntax. Define a domain rather than setting up each table's constraint individually.

Parameters

name

The name (optionally schema-qualified) of a domain to be created.

data_type

The underlying data type of the domain. This may include array specifiers.

DEFAULT *expression*

The DEFAULT clause specifies a default value for columns of the domain data type. The value is any variable-free expression (but subqueries are not allowed). The data type of the default expression must match the data type of the domain. If no default value is specified, then the default value is the null value.

The default expression will be used in any insert operation that does not specify a value for the column. If a default value is defined for a particular column, it overrides any default associated with the domain. In turn, the domain default overrides any default value associated with the underlying data type.

`CONSTRAINT` *`constraint_name`*

An optional name for a constraint. If not specified, the system generates a name.

`NOT NULL`

Values of this domain are not allowed to be null.

`NULL`

Values of this domain are allowed to be null. This is the default.

This clause is only intended for compatibility with nonstandard SQL databases. Its use is discouraged in new applications.

`CHECK (`*`expression`*`)`

`CHECK` clauses specify integrity constraints or tests which values of the domain must satisfy. Each constraint must be an expression producing a Boolean result. It should use the key word `VALUE` to refer to the value being tested.

Currently, `CHECK` expressions cannot contain subqueries nor refer to variables other than `VALUE`.

Examples

This example creates the `us_postal_code` data type and then uses the type in a table definition. A regular expression test is used to verify that the value looks like a valid US postal code.

```
CREATE DOMAIN us_postal_code AS TEXT
CHECK(
   VALUE ~ '^\\d{5}$'
OR VALUE ~ '^\\d{5}-\\d{4}$'
);

CREATE TABLE us_snail_addy (
  address_id SERIAL PRIMARY KEY,
  street1 TEXT NOT NULL,
  street2 TEXT,
  street3 TEXT,
  city TEXT NOT NULL,
  postal us_postal_code NOT NULL
);
```

Compatibility

The command `CREATE DOMAIN` conforms to the SQL standard.

See Also

ALTER DOMAIN, *DROP DOMAIN*

CREATE FUNCTION

Name

CREATE FUNCTION — define a new function

Synopsis

```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [, ...] ] )
    [ RETURNS rettype ]
  { LANGUAGE langname
    | IMMUTABLE | STABLE | VOLATILE
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
    [ WITH ( attribute [, ...] ) ]
```

Description

CREATE FUNCTION defines a new function. CREATE OR REPLACE FUNCTION will either create a new function, or replace an existing definition.

If a schema name is included, then the function is created in the specified schema. Otherwise it is created in the current schema. The name of the new function must not match any existing function with the same argument types in the same schema. However, functions of different argument types may share a name (this is called *overloading*).

To update the definition of an existing function, use CREATE OR REPLACE FUNCTION. It is not possible to change the name or argument types of a function this way (if you tried, you would actually be creating a new, distinct function). Also, CREATE OR REPLACE FUNCTION will not let you change the return type of an existing function. To do that, you must drop and recreate the function. (When using OUT parameters, that means you can't change the names or types of any OUT parameters except by dropping the function.)

If you drop and then recreate a function, the new function is not the same entity as the old; you will have to drop existing rules, views, triggers, etc. that refer to the old function. Use CREATE OR REPLACE FUNCTION to change a function definition without breaking objects that refer to the function.

The user that creates the function becomes the owner of the function.

Parameters

name

The name (optionally schema-qualified) of the function to create.

argmode

The mode of an argument: either IN, OUT, or INOUT. If omitted, the default is IN.

argname

The name of an argument. Some languages (currently only PL/pgSQL) let you use the name in the function body. For other languages the name of an input argument is just extra documentation. But the name of an output argument is significant, since it defines the column name in the result row type. (If you omit the name for an output argument, the system will choose a default column name.)

argtype

The data type(s) of the function's arguments (optionally schema-qualified), if any. The argument types may be base, composite, or domain types, or may reference the type of a table column.

Depending on the implementation language it may also be allowed to specify "pseudotypes" such as cstring. Pseudotypes indicate that the actual argument type is either incompletely specified, or outside the set of ordinary SQL data types.

The type of a column is referenced by writing *tablename*.*columnname*%TYPE. Using this feature can sometimes help make a function independent of changes to the definition of a table.

rettype

The return data type (optionally schema-qualified). The return type may be a base, composite, or domain type, or may reference the type of a table column. Depending on the implementation language it may also be allowed to specify "pseudotypes" such as cstring. If the function is not supposed to return a value, specify void as the return type.

When there are OUT or INOUT parameters, the RETURNS clause may be omitted. If present, it must agree with the result type implied by the output parameters: RECORD if there are multiple output parameters, or the same type as the single output parameter.

The SETOF modifier indicates that the function will return a set of items, rather than a single item.

The type of a column is referenced by writing *tablename*.*columnname*%TYPE.

langname

The name of the language that the function is implemented in. May be SQL, C, internal, or the name of a user-defined procedural language. For backward compatibility, the name may be enclosed by single quotes.

IMMUTABLE
STABLE
VOLATILE

These attributes inform the query optimizer about the behavior of the function. At most one choice may be specified. If none of these appear, VOLATILE is the default assumption.

IMMUTABLE indicates that the function cannot modify the database and always returns the same result when given the same argument values; that is, it does not do database lookups or otherwise use information not directly present in its argument list. If this option is given, any call of the function with all-constant arguments can be immediately replaced with the function value.

STABLE indicates that the function cannot modify the database, and that within a single table scan it will consistently return the same result for the same argument values, but that its result could change across SQL statements. This is the appropriate selection for functions whose results depend

on database lookups, parameter variables (such as the current time zone), etc. Also note that the `current_timestamp` family of functions qualify as stable, since their values do not change within a transaction.

`VOLATILE` indicates that the function value can change even within a single table scan, so no optimizations can be made. Relatively few database functions are volatile in this sense; some examples are `random()`, `currval()`, `timeofday()`. But note that any function that has side-effects must be classified volatile, even if its result is quite predictable, to prevent calls from being optimized away; an example is `setval()`.

For additional details see Section 33.6.

`CALLED ON NULL INPUT`
`RETURNS NULL ON NULL INPUT`
`STRICT`

`CALLED ON NULL INPUT` (the default) indicates that the function will be called normally when some of its arguments are null. It is then the function author's responsibility to check for null values if necessary and respond appropriately.

`RETURNS NULL ON NULL INPUT` or `STRICT` indicates that the function always returns null whenever any of its arguments are null. If this parameter is specified, the function is not executed when there are null arguments; instead a null result is assumed automatically.

`[EXTERNAL] SECURITY INVOKER`
`[EXTERNAL] SECURITY DEFINER`

`SECURITY INVOKER` indicates that the function is to be executed with the privileges of the user that calls it. That is the default. `SECURITY DEFINER` specifies that the function is to be executed with the privileges of the user that created it.

The key word `EXTERNAL` is allowed for SQL conformance, but it is optional since, unlike in SQL, this feature applies to all functions not only external ones.

`definition`

A string constant defining the function; the meaning depends on the language. It may be an internal function name, the path to an object file, an SQL command, or text in a procedural language.

`obj_file`, *`link_symbol`*

This form of the `AS` clause is used for dynamically loadable C language functions when the function name in the C language source code is not the same as the name of the SQL function. The string *`obj_file`* is the name of the file containing the dynamically loadable object, and *`link_symbol`* is the function's link symbol, that is, the name of the function in the C language source code. If the link symbol is omitted, it is assumed to be the same as the name of the SQL function being defined.

`attribute`

The historical way to specify optional pieces of information about the function. The following attributes may appear here:

`isStrict`

Equivalent to `STRICT` or `RETURNS NULL ON NULL INPUT`.

`isCachable`

`isCachable` is an obsolete equivalent of `IMMUTABLE`; it's still accepted for backwards-compatibility reasons.

Attribute names are not case-sensitive.

Notes

Refer to Section 33.3 for further information on writing functions.

The full SQL type syntax is allowed for input arguments and return value. However, some details of the type specification (e.g., the precision field for type `numeric`) are the responsibility of the underlying function implementation and are silently swallowed (i.e., not recognized or enforced) by the `CREATE FUNCTION` command.

PostgreSQL allows function *overloading*; that is, the same name can be used for several different functions so long as they have distinct argument types. However, the C names of all functions must be different, so you must give overloaded C functions different C names (for example, use the argument types as part of the C names).

Two functions are considered the same if they have the same names and *input* argument types, ignoring any `OUT` parameters. Thus for example these declarations conflict:

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, out text) ...
```

When repeated `CREATE FUNCTION` calls refer to the same object file, the file is only loaded once. To unload and reload the file (perhaps during development), use the *LOAD* command.

Use *DROP FUNCTION* to remove user-defined functions.

It is often helpful to use dollar quoting (see Section 4.1.2.2) to write the function definition string, rather than the normal single quote syntax. Without dollar quoting, any single quotes or backslashes in the function definition must be escaped by doubling them.

To be able to define a function, the user must have the `USAGE` privilege on the language.

Examples

Here are some trivial examples to help you get started. For more information and examples, see Section 33.3.

```
CREATE FUNCTION add(integer, integer) RETURNS integer
    AS 'select $1 + $2;'
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;
```

Increment an integer, making use of an argument name, in PL/pgSQL:

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS integer AS $$
        BEGIN
                RETURN i + 1;
        END;
$$ LANGUAGE plpgsql;
```

Return a record containing multiple output parameters:

```
CREATE FUNCTION dup(in int, out f1 int, out f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

You can do the same thing more verbosely with an explicitly named composite type:

```
CREATE TYPE dup_result AS (f1 int, f2 text);

CREATE FUNCTION dup(int) RETURNS dup_result
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

Writing `SECURITY DEFINER` Functions Safely

Because a `SECURITY DEFINER` function is executed with the privileges of the user that created it, care is needed to ensure that the function cannot be misused. For security, search_path should be set to exclude any schemas writable by untrusted users. This prevents malicious users from creating objects that mask objects used by the function. Particularly important in this regard is the temporary-table schema, which is searched first by default, and is normally writable by anyone. A secure arrangement can be had by forcing the temporary schema to be searched last. To do this, write `pg_temp` as the last entry in `search_path`. This function illustrates safe usage:

```
CREATE FUNCTION check_password(uname TEXT, pass TEXT)
RETURNS BOOLEAN AS $$
DECLARE passed BOOLEAN;
        old_path TEXT;
BEGIN
        -- Save old search_path; notice we must qualify current_setting
        -- to ensure we invoke the right function
        old_path := pg_catalog.current_setting('search_path');

        -- Set a secure search_path: trusted schemas, then 'pg_temp'.
        -- We set is_local = true so that the old value will be restored
        -- in event of an error before we reach the function end.
        PERFORM pg_catalog.set_config('search_path', 'admin, pg_temp', true);
```

```
        -- Do whatever secure work we came for.
        SELECT  (pwd = $2) INTO passed
        FROM    pwds
        WHERE   username = $1;

        -- Restore caller's search_path
        PERFORM pg_catalog.set_config('search_path', old_path, true);

        RETURN passed;
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;
```

Compatibility

A `CREATE FUNCTION` command is defined in SQL:1999 and later. The PostgreSQL version is similar but not fully compatible. The attributes are not portable, neither are the different available languages.

For compatibility with some other database systems, *`argmode`* can be written either before or after *`argname`*. But only the first way is standard-compliant.

See Also

ALTER FUNCTION, *DROP FUNCTION*, *GRANT*, *LOAD*, *REVOKE*, *createlang*

CREATE GROUP

Name

`CREATE GROUP` — define a new database role

Synopsis

```
CREATE GROUP name [ [ WITH ] option [ ... ] ]

where option can be:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
    | IN ROLE rolename [, ...]
    | IN GROUP rolename [, ...]
    | ROLE rolename [, ...]
    | ADMIN rolename [, ...]
    | USER rolename [, ...]
    | SYSID uid
```

Description

`CREATE GROUP` is now an alias for *CREATE ROLE*.

Compatibility

There is no `CREATE GROUP` statement in the SQL standard.

See Also

CREATE ROLE

CREATE INDEX

Name

CREATE INDEX — define a new index

Synopsis

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] name ON table [ USING method ]
    ( { column | ( expression ) } [ opclass ] [, ...] )
    [ WITH ( storage_parameter = value [, ... ] ) ]
    [ TABLESPACE tablespace ]
    [ WHERE predicate ]
```

Description

CREATE INDEX constructs an index *index_name* on the specified table. Indexes are primarily used to enhance database performance (though inappropriate use can result in slower performance).

The key field(s) for the index are specified as column names, or alternatively as expressions written in parentheses. Multiple fields can be specified if the index method supports multicolumn indexes.

An index field can be an expression computed from the values of one or more columns of the table row. This feature can be used to obtain fast access to data based on some transformation of the basic data. For example, an index computed on `upper(col)` would allow the clause `WHERE upper(col) = 'JIM'` to use an index.

PostgreSQL provides the index methods B-tree, hash, GiST, and GIN. Users can also define their own index methods, but that is fairly complicated.

When the WHERE clause is present, a *partial index* is created. A partial index is an index that contains entries for only a portion of a table, usually a portion that is more useful for indexing than the rest of the table. For example, if you have a table that contains both billed and unbilled orders where the unbilled orders take up a small fraction of the total table and yet that is an often used section, you can improve performance by creating an index on just that portion. Another possible application is to use WHERE with UNIQUE to enforce uniqueness over a subset of a table. See Section 11.7 for more discussion.

The expression used in the WHERE clause may refer only to columns of the underlying table, but it can use all columns, not just the ones being indexed. Presently, subqueries and aggregate expressions are also forbidden in WHERE. The same restrictions apply to index fields that are expressions.

All functions and operators used in an index definition must be "immutable", that is, their results must depend only on their arguments and never on any outside influence (such as the contents of another table or the current time). This restriction ensures that the behavior of the index is well-defined. To use a user-defined function in an index expression or WHERE clause, remember to mark the function immutable when you create it.

Parameters

`UNIQUE`

Causes the system to check for duplicate values in the table when the index is created (if data already exist) and each time data is added. Attempts to insert or update data which would result in duplicate entries will generate an error.

`CONCURRENTLY`

When this option is used, PostgreSQL will build the index without taking any locks that prevent concurrent inserts, updates, or deletes on the table; whereas a standard index build locks out writes (but not reads) on the table until it's done. There are several caveats to be aware of when using this option — see *Building Indexes Concurrently*.

`name`

The name of the index to be created. No schema name can be included here; the index is always created in the same schema as its parent table.

`table`

The name (possibly schema-qualified) of the table to be indexed.

`method`

The name of the index method to be used. Choices are `btree`, `hash`, `gist`, and `gin`. The default method is `btree`.

`column`

The name of a column of the table.

`expression`

An expression based on one or more columns of the table. The expression usually must be written with surrounding parentheses, as shown in the syntax. However, the parentheses may be omitted if the expression has the form of a function call.

`opclass`

The name of an operator class. See below for details.

`storage_parameter`

The name of an index-method-specific storage parameter. See below for details.

`tablespace`

The tablespace in which to create the index. If not specified, default_tablespace is used, or the database's default tablespace if `default_tablespace` is an empty string.

`predicate`

The constraint expression for a partial index.

Index Storage Parameters

The `WITH` clause can specify *storage parameters* for indexes. Each index method can have its own set of allowed storage parameters. The built-in index methods all accept a single parameter:

FILLFACTOR

The fillfactor for an index is a percentage that determines how full the index method will try to pack index pages. For B-trees, leaf pages are filled to this percentage during initial index build, and also when extending the index at the right (largest key values). If pages subsequently become completely full, they will be split, leading to gradual degradation in the index's efficiency. B-trees use a default fillfactor of 90, but any value from 10 to 100 can be selected. If the table is static then fillfactor 100 is best to minimize the index's physical size, but for heavily updated tables a smaller fillfactor is better to minimize the need for page splits. The other index methods use fillfactor in different but roughly analogous ways; the default fillfactor varies between methods.

Building Indexes Concurrently

Creating an index can interfere with regular operation of a database. Normally PostgreSQL locks the table to be indexed against writes and performs the entire index build with a single scan of the table. Other transactions can still read the table, but if they try to insert, update, or delete rows in the table they will block until the index build is finished. This could have a severe effect if the system is a live production database. Large tables can take many hours to be indexed, and even for smaller tables, an index build can lock out writers for periods that are unacceptably long for a production system.

PostgreSQL supports building indexes without locking out writes. This method is invoked by specifying the CONCURRENTLY option of CREATE INDEX. When this option is used, PostgreSQL must perform two scans of the table, and in addition it must wait for all existing transactions to terminate. Thus this method requires more total work than a standard index build and takes significantly longer to complete. However, since it allows normal operations to continue while the index is built, this method is useful for adding new indexes in a production environment. Of course, the extra CPU and I/O load imposed by the index creation may slow other operations.

If a problem arises during the second scan of the table, such as a uniqueness violation in a unique index, the CREATE INDEX command will fail but leave behind an "invalid" index. This index will be ignored for querying purposes because it may be incomplete; however it will still consume update overhead. The recommended recovery method in such cases is to drop the index and try again to perform CREATE INDEX CONCURRENTLY. (Another possibility is to rebuild the index with REINDEX. However, since REINDEX does not support concurrent builds, this option is unlikely to seem attractive.)

Another caveat when building a unique index concurrently is that the uniqueness constraint is already being enforced against other transactions when the second table scan begins. This means that constraint violations could be reported in other queries prior to the index becoming available for use, or even in cases where the index build eventually fails. Also, if a failure does occur in the second scan, the "invalid" index continues to enforce its uniqueness constraint afterwards.

Concurrent builds of expression indexes and partial indexes are supported. Errors occurring in the evaluation of these expressions could cause behavior similar to that described above for unique constraint violations.

Regular index builds permit other regular index builds on the same table to occur in parallel, but only one concurrent index build can occur on a table at a time. In both cases, no other types of schema modification on the table are allowed meanwhile. Another difference is that a regular CREATE INDEX command can be performed within a transaction block, but CREATE INDEX CONCURRENTLY cannot.

Notes

See Chapter 11 for information about when indexes can be used, when they are not used, and in which particular situations they can be useful.

Currently, only the B-tree and GiST index methods support multicolumn indexes. Up to 32 fields may be specified by default. (This limit can be altered when building PostgreSQL.) Only B-tree currently supports unique indexes.

An *operator class* can be specified for each column of an index. The operator class identifies the operators to be used by the index for that column. For example, a B-tree index on four-byte integers would use the `int4_ops` class; this operator class includes comparison functions for four-byte integers. In practice the default operator class for the column's data type is usually sufficient. The main point of having operator classes is that for some data types, there could be more than one meaningful ordering. For example, we might want to sort a complex-number data type either by absolute value or by real part. We could do this by defining two operator classes for the data type and then selecting the proper class when making an index. More information about operator classes is in Section 11.8 and in Section 33.14.

Use *DROP INDEX* to remove an index.

Indexes are not used for `IS NULL` clauses by default. The best way to use indexes in such cases is to create a partial index using an `IS NULL` predicate.

Prior releases of PostgreSQL also had an R-tree index method. This method has been removed because it had no significant advantages over the GiST method. If `USING rtree` is specified, `CREATE INDEX` will interpret it as `USING gist`, to simplify conversion of old databases to GiST.

Examples

To create a B-tree index on the column `title` in the table `films`:

```
CREATE UNIQUE INDEX title_idx ON films (title);
```

To create an index on the expression `lower(title)`, allowing efficient case-insensitive searches:

```
CREATE INDEX lower_title_idx ON films ((lower(title)));
```

To create an index with non-default fill factor:

```
CREATE UNIQUE INDEX title_idx ON films (title) WITH (fillfactor = 70);
```

To create an index on the column `code` in the table `films` and have the index reside in the tablespace `indexspace`:

```
CREATE INDEX code_idx ON films(code) TABLESPACE indexspace;
```

To create an index without locking out writes to the table:

```
CREATE INDEX CONCURRENTLY sales_quantity_index ON sales_table (quantity);
```

Compatibility

`CREATE INDEX` is a PostgreSQL language extension. There are no provisions for indexes in the SQL standard.

See Also

ALTER INDEX, *DROP INDEX*

CREATE LANGUAGE

Name

CREATE LANGUAGE — define a new procedural language

Synopsis

```
CREATE [ PROCEDURAL ] LANGUAGE name
CREATE [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
    HANDLER call_handler [ VALIDATOR valfunction ]
```

Description

Using CREATE LANGUAGE, a PostgreSQL user can register a new procedural language with a PostgreSQL database. Subsequently, functions and trigger procedures can be defined in this new language. The user must have the PostgreSQL superuser privilege to register a new language.

CREATE LANGUAGE effectively associates the language name with a call handler that is responsible for executing functions written in the language. Refer to Chapter 36 for more information about language call handlers.

There are two forms of the CREATE LANGUAGE command. In the first form, the user supplies just the name of the desired language, and the PostgreSQL server consults the pg_pltemplate system catalog to determine the correct parameters. In the second form, the user supplies the language parameters along with the language name. The second form can be used to create a language that is not defined in pg_pltemplate, but this approach is considered obsolescent.

When the server finds an entry in the pg_pltemplate catalog for the given language name, it will use the catalog data even if the command includes language parameters. This behavior simplifies loading of old dump files, which are likely to contain out-of-date information about language support functions.

Parameters

TRUSTED

TRUSTED specifies that the call handler for the language is safe, that is, it does not offer an unprivileged user any functionality to bypass access restrictions. If this key word is omitted when registering the language, only users with the PostgreSQL superuser privilege can use this language to create new functions.

PROCEDURAL

This is a noise word.

name

The name of the new procedural language. The language name is case insensitive. The name must be unique among the languages in the database.

For backward compatibility, the name may be enclosed by single quotes.

`HANDLER` *`call_handler`*

`call_handler` is the name of a previously registered function that will be called to execute the procedural language functions. The call handler for a procedural language must be written in a compiled language such as C with version 1 call convention and registered with PostgreSQL as a function taking no arguments and returning the `language_handler` type, a placeholder type that is simply used to identify the function as a call handler.

`VALIDATOR` *`valfunction`*

`valfunction` is the name of a previously registered function that will be called when a new function in the language is created, to validate the new function. If no validator function is specified, then a new function will not be checked when it is created. The validator function must take one argument of type `oid`, which will be the OID of the to-be-created function, and will typically return `void`.

A validator function would typically inspect the function body for syntactical correctness, but it can also look at other properties of the function, for example if the language cannot handle certain argument types. To signal an error, the validator function should use the `ereport()` function. The return value of the function is ignored.

The `TRUSTED` option and the support function name(s) are ignored if the server has an entry for the specified language name in `pg_pltemplate`.

Notes

The createlang program is a simple wrapper around the `CREATE LANGUAGE` command. It eases installation of procedural languages from the shell command line.

Use *DROP LANGUAGE*, or better yet the droplang program, to drop procedural languages.

The system catalog `pg_language` (see Section 43.20) records information about the currently installed languages. Also, `createlang` has an option to list the installed languages.

To create functions in a procedural language, a user must have the `USAGE` privilege for the language. By default, `USAGE` is granted to `PUBLIC` (i.e., everyone) for trusted languages. This may be revoked if desired.

Procedural languages are local to individual databases. However, a language can be installed into the `template1` database, which will cause it to be available automatically in all subsequently-created databases.

The call handler function and the validator function (if any) must already exist if the server does not have an entry for the language in `pg_pltemplate`. But when there is an entry, the functions need not already exist; they will be automatically defined if not present in the database. (This can result in `CREATE LANGUAGE` failing, if the shared library that implements the language is not available in the installation.)

In PostgreSQL versions before 7.3, it was necessary to declare handler functions as returning the placeholder type `opaque`, rather than `language_handler`. To support loading of old dump files, `CREATE LANGUAGE` will accept a function declared as returning `opaque`, but it will issue a notice and change the function's declared return type to `language_handler`.

Examples

The preferred way of creating any of the standard procedural languages is just:

```
CREATE LANGUAGE plpgsql;
```

For a language not known in the `pg_pltemplate` catalog, a sequence such as this is needed:

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS '$libdir/plsample'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

Compatibility

`CREATE LANGUAGE` is a PostgreSQL extension.

See Also

ALTER LANGUAGE, *CREATE FUNCTION*, *DROP LANGUAGE*, *GRANT*, *REVOKE*, *createlang*, *droplang*

CREATE OPERATOR

Name

CREATE OPERATOR — define a new operator

Synopsis

```
CREATE OPERATOR name (
    PROCEDURE = funcname
    [, LEFTARG = lefttype ] [, RIGHTARG = righttype ]
    [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
    [, RESTRICT = res_proc ] [, JOIN = join_proc ]
    [, HASHES ] [, MERGES ]
    [, SORT1 = left_sort_op ] [, SORT2 = right_sort_op ]
    [, LTCMP = less_than_op ] [, GTCMP = greater_than_op ]
)
```

Description

CREATE OPERATOR defines a new operator, *name*. The user who defines an operator becomes its owner. If a schema name is given then the operator is created in the specified schema. Otherwise it is created in the current schema.

The operator name is a sequence of up to NAMEDATALEN-1 (63 by default) characters from the following list:

+ - * / < > = ~ ! @ # % ^ & | ` ?

There are a few restrictions on your choice of name:

- -- and /* cannot appear anywhere in an operator name, since they will be taken as the start of a comment.
- A multicharacter operator name cannot end in + or -, unless the name also contains at least one of these characters:

  ~ ! @ # % ^ & | ` ?

  For example, @- is an allowed operator name, but *- is not. This restriction allows PostgreSQL to parse SQL-compliant commands without requiring spaces between tokens.

The operator != is mapped to <> on input, so these two names are always equivalent.

At least one of LEFTARG and RIGHTARG must be defined. For binary operators, both must be defined. For right unary operators, only LEFTARG should be defined, while for left unary operators only RIGHTARG should be defined.

The *funcname* procedure must have been previously defined using `CREATE FUNCTION` and must be defined to accept the correct number of arguments (either one or two) of the indicated types.

The other clauses specify optional operator optimization clauses. Their meaning is detailed in Section 33.13.

Parameters

name

The name of the operator to be defined. See above for allowable characters. The name may be schema-qualified, for example `CREATE OPERATOR myschema.+ (...)`. If not, then the operator is created in the current schema. Two operators in the same schema can have the same name if they operate on different data types. This is called *overloading*.

funcname

The function used to implement this operator.

lefttype

The data type of the operator's left operand, if any. This option would be omitted for a left-unary operator.

righttype

The data type of the operator's right operand, if any. This option would be omitted for a right-unary operator.

com_op

The commutator of this operator.

neg_op

The negator of this operator.

res_proc

The restriction selectivity estimator function for this operator.

join_proc

The join selectivity estimator function for this operator.

HASHES

Indicates this operator can support a hash join.

MERGES

Indicates this operator can support a merge join.

left_sort_op

If this operator can support a merge join, the less-than operator that sorts the left-hand data type of this operator.

right_sort_op

If this operator can support a merge join, the less-than operator that sorts the right-hand data type of this operator.

less_than_op

If this operator can support a merge join, the less-than operator that compares the input data types of this operator.

greater_than_op

If this operator can support a merge join, the greater-than operator that compares the input data types of this operator.

To give a schema-qualified operator name in *com_op* or the other optional arguments, use the `OPERATOR()` syntax, for example

```
COMMUTATOR = OPERATOR(myschema.===) ,
```

Notes

Refer to Section 33.12 for further information.

Use *DROP OPERATOR* to delete user-defined operators from a database. Use *ALTER OPERATOR* to modify operators in a database.

Examples

The following command defines a new operator, area-equality, for the data type `box`:

```
CREATE OPERATOR === (
    LEFTARG = box,
    RIGHTARG = box,
    PROCEDURE = area_equal_procedure,
    COMMUTATOR = ===,
    NEGATOR = !==,
    RESTRICT = area_restriction_procedure,
    JOIN = area_join_procedure,
    HASHES,
    SORT1 = <<<,
    SORT2 = <<<
    -- Since sort operators were given, MERGES is implied.
    -- LTCMP and GTCMP are assumed to be < and > respectively
);
```

Compatibility

`CREATE OPERATOR` is a PostgreSQL extension. There are no provisions for user-defined operators in the SQL standard.

See Also

ALTER OPERATOR, *CREATE OPERATOR CLASS*, *DROP OPERATOR*

CREATE OPERATOR CLASS

Name

CREATE OPERATOR CLASS — define a new operator class

Synopsis

```
CREATE OPERATOR CLASS name [ DEFAULT ] FOR TYPE data_type USING index_method AS
  {  OPERATOR strategy_number operator_name [ ( op_type, op_type ) ] [ RECHECK ]
   | FUNCTION support_number funcname ( argument_type [, ...] )
   | STORAGE storage_type
  } [, ... ]
```

Description

CREATE OPERATOR CLASS creates a new operator class. An operator class defines how a particular data type can be used with an index. The operator class specifies that certain operators will fill particular roles or "strategies" for this data type and this index method. The operator class also specifies the support procedures to be used by the index method when the operator class is selected for an index column. All the operators and functions used by an operator class must be defined before the operator class is created.

If a schema name is given then the operator class is created in the specified schema. Otherwise it is created in the current schema. Two operator classes in the same schema can have the same name only if they are for different index methods.

The user who defines an operator class becomes its owner. Presently, the creating user must be a superuser. (This restriction is made because an erroneous operator class definition could confuse or even crash the server.)

CREATE OPERATOR CLASS does not presently check whether the operator class definition includes all the operators and functions required by the index method, nor whether the operators and functions form a self-consistent set. It is the user's responsibility to define a valid operator class.

Refer to Section 33.14 for further information.

Parameters

name

The name of the operator class to be created. The name may be schema-qualified.

DEFAULT

If present, the operator class will become the default operator class for its data type. At most one operator class can be the default for a specific data type and index method.

data_type

The column data type that this operator class is for.

index_method

The name of the index method this operator class is for.

strategy_number

The index method's strategy number for an operator associated with the operator class.

operator_name

The name (optionally schema-qualified) of an operator associated with the operator class.

op_type

The operand data type(s) of an operator, or NONE to signify a left-unary or right-unary operator. The operand data types may be omitted in the normal case where they are the same as the operator class's data type.

RECHECK

If present, the index is "lossy" for this operator, and so the rows retrieved using the index must be rechecked to verify that they actually satisfy the qualification clause involving this operator.

support_number

The index method's support procedure number for a function associated with the operator class.

funcname

The name (optionally schema-qualified) of a function that is an index method support procedure for the operator class.

argument_types

The parameter data type(s) of the function.

storage_type

The data type actually stored in the index. Normally this is the same as the column data type, but some index methods (GIN and GiST for now) allow it to be different. The STORAGE clause must be omitted unless the index method allows a different type to be used.

The OPERATOR, FUNCTION, and STORAGE clauses may appear in any order.

Notes

Because the index machinery does not check access permissions on functions before using them, including a function or operator in an operator class is tantamount to granting public execute permission on it. This is usually not an issue for the sorts of functions that are useful in an operator class.

The operators should not be defined by SQL functions. A SQL function is likely to be inlined into the calling query, which will prevent the optimizer from recognizing that the query matches an index.

Examples

The following example command defines a GiST index operator class for the data type `_int4` (array of `int4`). See `contrib/intarray/` for the complete example.

```
CREATE OPERATOR CLASS gist__int_ops
    DEFAULT FOR TYPE _int4 USING gist AS
        OPERATOR        3       &&,
        OPERATOR        6       =       RECHECK,
        OPERATOR        7       @>,
        OPERATOR        8       <@,
        OPERATOR        20      @@ (_int4, query_int),
        FUNCTION        1       g_int_consistent (internal, _int4, int4),
        FUNCTION        2       g_int_union (bytea, internal),
        FUNCTION        3       g_int_compress (internal),
        FUNCTION        4       g_int_decompress (internal),
        FUNCTION        5       g_int_penalty (internal, internal, internal),
        FUNCTION        6       g_int_picksplit (internal, internal),
        FUNCTION        7       g_int_same (_int4, _int4, internal);
```

Compatibility

`CREATE OPERATOR CLASS` is a PostgreSQL extension. There is no `CREATE OPERATOR CLASS` statement in the SQL standard.

See Also

ALTER OPERATOR CLASS, *DROP OPERATOR CLASS*

CREATE ROLE

Name

CREATE ROLE — define a new database role

Synopsis

```
CREATE ROLE name [ [ WITH ] option [ ... ] ]

where option can be:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
    | IN ROLE rolename [, ...]
    | IN GROUP rolename [, ...]
    | ROLE rolename [, ...]
    | ADMIN rolename [, ...]
    | USER rolename [, ...]
    | SYSID uid
```

Description

CREATE ROLE adds a new role to a PostgreSQL database cluster. A role is an entity that can own database objects and have database privileges; a role can be considered a “user”, a “group”, or both depending on how it is used. Refer to Chapter 18 and Chapter 20 for information about managing users and authentication. You must have CREATEROLE privilege or be a database superuser to use this command.

Note that roles are defined at the database cluster level, and so are valid in all databases in the cluster.

Parameters

name

The name of the new role.

SUPERUSER

NOSUPERUSER

These clauses determine whether the new role is a “superuser”, who can override all access restrictions within the database. Superuser status is dangerous and should be used only when really needed.

You must yourself be a superuser to create a new superuser. If not specified, `NOSUPERUSER` is the default.

`CREATEDB`
`NOCREATEDB`

These clauses define a role's ability to create databases. If `CREATEDB` is specified, the role being defined will be allowed to create new databases. Specifying `NOCREATEDB` will deny a role the ability to create databases. If not specified, `NOCREATEDB` is the default.

`CREATEROLE`
`NOCREATEROLE`

These clauses determine whether a role will be permitted to create new roles (that is, execute `CREATE ROLE`). A role with `CREATEROLE` privilege can also alter and drop other roles. If not specified, `NOCREATEROLE` is the default.

`CREATEUSER`
`NOCREATEUSER`

These clauses are an obsolete, but still accepted, spelling of `SUPERUSER` and `NOSUPERUSER`. Note that they are *not* equivalent to `CREATEROLE` as one might naively expect!

`INHERIT`
`NOINHERIT`

These clauses determine whether a role "inherits" the privileges of roles it is a member of. A role with the `INHERIT` attribute can automatically use whatever database privileges have been granted to all roles it is directly or indirectly a member of. Without `INHERIT`, membership in another role only grants the ability to `SET ROLE` to that other role; the privileges of the other role are only available after having done so. If not specified, `INHERIT` is the default.

`LOGIN`
`NOLOGIN`

These clauses determine whether a role is allowed to log in; that is, whether the role can be given as the initial session authorization name during client connection. A role having the `LOGIN` attribute can be thought of as a user. Roles without this attribute are useful for managing database privileges, but are not users in the usual sense of the word. If not specified, `NOLOGIN` is the default, except when `CREATE ROLE` is invoked through its alternate spelling `CREATE USER`.

`CONNECTION LIMIT` *`connlimit`*

If role can log in, this specifies how many concurrent connections the role can make. -1 (the default) means no limit.

`PASSWORD` *`password`*

Sets the role's password. (A password is only of use for roles having the `LOGIN` attribute, but you can nonetheless define one for roles without it.) If you do not plan to use password authentication you can omit this option. If no password is specified, the password will be set to null and password authentication will always fail for that user. A null password can optionally be written explicitly as `PASSWORD NULL`.

`ENCRYPTED`
`UNENCRYPTED`

These key words control whether the password is stored encrypted in the system catalogs. (If neither is specified, the default behavior is determined by the configuration parameter password_encryption.) If the presented password string is already in MD5-encrypted format, then it is stored encrypted as-is, regardless of whether `ENCRYPTED` or `UNENCRYPTED` is specified (since the system cannot decrypt the specified encrypted password string). This allows reloading of encrypted passwords during dump/restore.

Note that older clients may lack support for the MD5 authentication mechanism that is needed to work with passwords that are stored encrypted.

`VALID UNTIL '`*`timestamp`*`'`

The `VALID UNTIL` clause sets a date and time after which the role's password is no longer valid. If this clause is omitted the password will be valid for all time.

`IN ROLE` *`rolename`*

The `IN ROLE` clause lists one or more existing roles to which the new role will be immediately added as a new member. (Note that there is no option to add the new role as an administrator; use a separate `GRANT` command to do that.)

`IN GROUP` *`rolename`*

`IN GROUP` is an obsolete spelling of `IN ROLE`.

`ROLE` *`rolename`*

The `ROLE` clause lists one or more existing roles which are automatically added as members of the new role. (This in effect makes the new role a "group".)

`ADMIN` *`rolename`*

The `ADMIN` clause is like `ROLE`, but the named roles are added to the new role `WITH ADMIN OPTION`, giving them the right to grant membership in this role to others.

`USER` *`rolename`*

The `USER` clause is an obsolete spelling of the `ROLE` clause.

`SYSID` *`uid`*

The `SYSID` clause is ignored, but is accepted for backwards compatibility.

Notes

Use *ALTER ROLE* to change the attributes of a role, and *DROP ROLE* to remove a role. All the attributes specified by `CREATE ROLE` can be modified by later `ALTER ROLE` commands.

The preferred way to add and remove members of roles that are being used as groups is to use *GRANT* and *REVOKE*.

The `VALID UNTIL` clause defines an expiration time for a password only, not for the role *per se*. In particular, the expiration time is not enforced when logging in using a non-password-based authentication method.

The `INHERIT` attribute governs inheritance of grantable privileges (that is, access privileges for database objects and role memberships). It does not apply to the special role attributes set by `CREATE ROLE` and `ALTER ROLE`. For example, being a member of a role with `CREATEDB` privilege does not immediately grant the ability to create databases, even if `INHERIT` is set; it would be necessary to become that role via *SET ROLE* before creating a database.

The `INHERIT` attribute is the default for reasons of backwards compatibility: in prior releases of PostgreSQL, users always had access to all privileges of groups they were members of. However, `NOINHERIT` provides a closer match to the semantics specified in the SQL standard.

Be careful with the `CREATEROLE` privilege. There is no concept of inheritance for the privileges of a `CREATEROLE`-role. That means that even if a role does not have a certain privilege but is allowed to create other roles, it can easily create another role with different privileges than its own (except for creating roles with superuser privileges). For example, if the role "user" has the `CREATEROLE` privilege but not the `CREATEDB` privilege, nonetheless it can create a new role with the `CREATEDB` privilege. Therefore, regard roles that have the `CREATEROLE` privilege as almost-superuser-roles.

PostgreSQL includes a program *createuser* that has the same functionality as `CREATE ROLE` (in fact, it calls this command) but can be run from the command shell.

The `CONNECTION LIMIT` option is only enforced approximately; if two new sessions start at about the same time when just one connection "slot" remains for the role, it is possible that both will fail. Also, the limit is never enforced for superusers.

Caution must be exercised when specifying an unencrypted password with this command. The password will be transmitted to the server in cleartext, and it might also be logged in the client's command history or the server log. The command *createuser*, however, transmits the password encrypted. Also, *psql* contains a command `\password` that can be used to safely change the password later.

Examples

Create a role that can log in, but don't give it a password:

```
CREATE ROLE jonathan LOGIN;
```

Create a role with a password:

```
CREATE USER davide WITH PASSWORD 'jw8s0F4';
```

(`CREATE USER` is the same as `CREATE ROLE` except that it implies `LOGIN`.)

Create a role with a password that is valid until the end of 2004. After one second has ticked in 2005, the password is no longer valid.

```
CREATE ROLE miriam WITH LOGIN PASSWORD 'jw8s0F4' VALID UNTIL '2005-01-01';
```

Create a role that can create databases and manage roles:

```
CREATE ROLE admin WITH CREATEDB CREATEROLE;
```

Compatibility

The `CREATE ROLE` statement is in the SQL standard, but the standard only requires the syntax

```
CREATE ROLE name [ WITH ADMIN rolename ]
```

Multiple initial administrators, and all the other options of `CREATE ROLE`, are PostgreSQL extensions.

The SQL standard defines the concepts of users and roles, but it regards them as distinct concepts and leaves all commands defining users to be specified by each database implementation. In PostgreSQL we have chosen to unify users and roles into a single kind of entity. Roles therefore have many more optional attributes than they do in the standard.

The behavior specified by the SQL standard is most closely approximated by giving users the `NOINHERIT` attribute, while roles are given the `INHERIT` attribute.

See Also

SET ROLE, *ALTER ROLE*, *DROP ROLE*, *GRANT*, *REVOKE*, createuser

CREATE RULE

Name

CREATE RULE — define a new rewrite rule

Synopsis

```
CREATE [ OR REPLACE ] RULE name AS ON event
    TO table [ WHERE condition ]
    DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }
```

Description

CREATE RULE defines a new rule applying to a specified table or view. CREATE OR REPLACE RULE will either create a new rule, or replace an existing rule of the same name for the same table.

The PostgreSQL rule system allows one to define an alternate action to be performed on insertions, updates, or deletions in database tables. Roughly speaking, a rule causes additional commands to be executed when a given command on a given table is executed. Alternatively, an INSTEAD rule can replace a given command by another, or cause a command not to be executed at all. Rules are used to implement table views as well. It is important to realize that a rule is really a command transformation mechanism, or command macro. The transformation happens before the execution of the commands starts. If you actually want an operation that fires independently for each physical row, you probably want to use a trigger, not a rule. More information about the rules system is in Chapter 35.

Presently, ON SELECT rules must be unconditional INSTEAD rules and must have actions that consist of a single SELECT command. Thus, an ON SELECT rule effectively turns the table into a view, whose visible contents are the rows returned by the rule's SELECT command rather than whatever had been stored in the table (if anything). It is considered better style to write a CREATE VIEW command than to create a real table and define an ON SELECT rule for it.

You can create the illusion of an updatable view by defining ON INSERT, ON UPDATE, and ON DELETE rules (or any subset of those that's sufficient for your purposes) to replace update actions on the view with appropriate updates on other tables. If you want to support INSERT RETURNING and so on, then be sure to put a suitable RETURNING clause into each of these rules.

There is a catch if you try to use conditional rules for view updates: there *must* be an unconditional INSTEAD rule for each action you wish to allow on the view. If the rule is conditional, or is not INSTEAD, then the system will still reject attempts to perform the update action, because it thinks it might end up trying to perform the action on the dummy table of the view in some cases. If you want to handle all the useful cases in conditional rules, add an unconditional DO INSTEAD NOTHING rule to ensure that the system understands it will never be called on to update the dummy table. Then make the conditional rules non-INSTEAD; in the cases where they are applied, they add to the default INSTEAD NOTHING action. (This method does not currently work to support RETURNING queries, however.)

Parameters

name

The name of a rule to create. This must be distinct from the name of any other rule for the same table. Multiple rules on the same table and same event type are applied in alphabetical name order.

event

The event is one of SELECT, INSERT, UPDATE, or DELETE.

table

The name (optionally schema-qualified) of the table or view the rule applies to.

condition

Any SQL conditional expression (returning boolean). The condition expression may not refer to any tables except NEW and OLD, and may not contain aggregate functions.

INSTEAD

INSTEAD indicates that the commands should be executed *instead of* the original command.

ALSO

ALSO indicates that the commands should be executed *in addition to* the original command.

If neither ALSO nor INSTEAD is specified, ALSO is the default.

command

The command or commands that make up the rule action. Valid commands are SELECT, INSERT, UPDATE, DELETE, or NOTIFY.

Within *condition* and *command*, the special table names NEW and OLD may be used to refer to values in the referenced table. NEW is valid in ON INSERT and ON UPDATE rules to refer to the new row being inserted or updated. OLD is valid in ON UPDATE and ON DELETE rules to refer to the existing row being updated or deleted.

Notes

You must be the owner of a table to create or change rules for it.

In a rule for INSERT, UPDATE, or DELETE on a view, you can add a RETURNING clause that emits the view's columns. This clause will be used to compute the outputs if the rule is triggered by an INSERT RETURNING, UPDATE RETURNING, or DELETE RETURNING command respectively. When the rule is triggered by a command without RETURNING, the rule's RETURNING clause will be ignored. The current implementation allows only unconditional INSTEAD rules to contain RETURNING; furthermore there can be at most one RETURNING clause among all the rules for the same event. (This ensures that there is only one candidate RETURNING clause to be used to compute the results.) RETURNING queries on the view will be rejected if there is no RETURNING clause in any available rule.

It is very important to take care to avoid circular rules. For example, though each of the following two rule definitions are accepted by PostgreSQL, the SELECT command would cause PostgreSQL to report an error because of recursive expansion of a rule:

```
CREATE RULE "_RETURN" AS
    ON SELECT TO t1
    DO INSTEAD
        SELECT * FROM t2;

CREATE RULE "_RETURN" AS
    ON SELECT TO t2
    DO INSTEAD
        SELECT * FROM t1;

SELECT * FROM t1;
```

Presently, if a rule action contains a `NOTIFY` command, the `NOTIFY` command will be executed unconditionally, that is, the `NOTIFY` will be issued even if there are not any rows that the rule should apply to. For example, in

```
CREATE RULE notify_me AS ON UPDATE TO mytable DO ALSO NOTIFY mytable;

UPDATE mytable SET name = 'foo' WHERE id = 42;
```

one `NOTIFY` event will be sent during the `UPDATE`, whether or not there are any rows that match the condition `id = 42`. This is an implementation restriction that may be fixed in future releases.

Compatibility

`CREATE RULE` is a PostgreSQL language extension, as is the entire query rewrite system.

CREATE SCHEMA

Name

CREATE SCHEMA — define a new schema

Synopsis

```
CREATE SCHEMA schemaname [ AUTHORIZATION username ] [ schema_element [ ... ] ]
CREATE SCHEMA AUTHORIZATION username [ schema_element [ ... ] ]
```

Description

CREATE SCHEMA enters a new schema into the current database. The schema name must be distinct from the name of any existing schema in the current database.

A schema is essentially a namespace: it contains named objects (tables, data types, functions, and operators) whose names may duplicate those of other objects existing in other schemas. Named objects are accessed either by "qualifying" their names with the schema name as a prefix, or by setting a search path that includes the desired schema(s). A CREATE command specifying an unqualified object name creates the object in the current schema (the one at the front of the search path, which can be determined with the function current_schema).

Optionally, CREATE SCHEMA can include subcommands to create objects within the new schema. The subcommands are treated essentially the same as separate commands issued after creating the schema, except that if the AUTHORIZATION clause is used, all the created objects will be owned by that user.

Parameters

schemaname

The name of a schema to be created. If this is omitted, the user name is used as the schema name. The name cannot begin with pg_, as such names are reserved for system schemas.

username

The name of the user who will own the schema. If omitted, defaults to the user executing the command. Only superusers may create schemas owned by users other than themselves.

schema_element

An SQL statement defining an object to be created within the schema. Currently, only CREATE TABLE, CREATE VIEW, CREATE INDEX, CREATE SEQUENCE, CREATE TRIGGER and GRANT are accepted as clauses within CREATE SCHEMA. Other kinds of objects may be created in separate commands after the schema is created.

Notes

To create a schema, the invoking user must have the `CREATE` privilege for the current database. (Of course, superusers bypass this check.)

Examples

Create a schema:

```
CREATE SCHEMA myschema;
```

Create a schema for user `joe`; the schema will also be named `joe`:

```
CREATE SCHEMA AUTHORIZATION joe;
```

Create a schema and create a table and view within it:

```
CREATE SCHEMA hollywood
    CREATE TABLE films (title text, release date, awards text[])
    CREATE VIEW winners AS
        SELECT title, release FROM films WHERE awards IS NOT NULL;
```

Notice that the individual subcommands do not end with semicolons.

The following is an equivalent way of accomplishing the same result:

```
CREATE SCHEMA hollywood;
CREATE TABLE hollywood.films (title text, release date, awards text[]);
CREATE VIEW hollywood.winners AS
    SELECT title, release FROM hollywood.films WHERE awards IS NOT NULL;
```

Compatibility

The SQL standard allows a `DEFAULT CHARACTER SET` clause in `CREATE SCHEMA`, as well as more subcommand types than are presently accepted by PostgreSQL.

The SQL standard specifies that the subcommands in `CREATE SCHEMA` may appear in any order. The present PostgreSQL implementation does not handle all cases of forward references in subcommands; it may sometimes be necessary to reorder the subcommands in order to avoid forward references.

According to the SQL standard, the owner of a schema always owns all objects within it. PostgreSQL allows schemas to contain objects owned by users other than the schema owner. This can happen only if the schema owner grants the `CREATE` privilege on his schema to someone else.

See Also

ALTER SCHEMA, *DROP SCHEMA*

CREATE SEQUENCE

Name

CREATE SEQUENCE — define a new sequence generator

Synopsis

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE name [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { table.column | NONE } ]
```

Description

CREATE SEQUENCE creates a new sequence number generator. This involves creating and initializing a new special single-row table with the name *name*. The generator will be owned by the user issuing the command.

If a schema name is given then the sequence is created in the specified schema. Otherwise it is created in the current schema. Temporary sequences exist in a special schema, so a schema name may not be given when creating a temporary sequence. The sequence name must be distinct from the name of any other sequence, table, index, or view in the same schema.

After a sequence is created, you use the functions nextval, currval, and setval to operate on the sequence. These functions are documented in Section 9.12.

Although you cannot update a sequence directly, you can use a query like

```
SELECT * FROM name;
```

to examine the parameters and current state of a sequence. In particular, the last_value field of the sequence shows the last value allocated by any session. (Of course, this value may be obsolete by the time it's printed, if other sessions are actively doing nextval calls.)

Parameters

TEMPORARY or TEMP

If specified, the sequence object is created only for this session, and is automatically dropped on session exit. Existing permanent sequences with the same name are not visible (in this session) while the temporary sequence exists, unless they are referenced with schema-qualified names.

name

The name (optionally schema-qualified) of the sequence to be created.

increment

The optional clause `INCREMENT BY` *`increment`* specifies which value is added to the current sequence value to create a new value. A positive value will make an ascending sequence, a negative one a descending sequence. The default value is 1.

minvalue
`NO MINVALUE`

The optional clause `MINVALUE` *`minvalue`* determines the minimum value a sequence can generate. If this clause is not supplied or `NO MINVALUE` is specified, then defaults will be used. The defaults are 1 and $-2^{63}-1$ for ascending and descending sequences, respectively.

maxvalue
`NO MAXVALUE`

The optional clause `MAXVALUE` *`maxvalue`* determines the maximum value for the sequence. If this clause is not supplied or `NO MAXVALUE` is specified, then default values will be used. The defaults are $2^{63}-1$ and -1 for ascending and descending sequences, respectively.

start

The optional clause `START WITH` *`start`* allows the sequence to begin anywhere. The default starting value is *`minvalue`* for ascending sequences and *`maxvalue`* for descending ones.

cache

The optional clause `CACHE` *`cache`* specifies how many sequence numbers are to be preallocated and stored in memory for faster access. The minimum value is 1 (only one value can be generated at a time, i.e., no cache), and this is also the default.

`CYCLE`
`NO CYCLE`

The `CYCLE` option allows the sequence to wrap around when the *`maxvalue`* or *`minvalue`* has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the *`minvalue`* or *`maxvalue`*, respectively.

If `NO CYCLE` is specified, any calls to `nextval` after the sequence has reached its maximum value will return an error. If neither `CYCLE` or `NO CYCLE` are specified, `NO CYCLE` is the default.

`OWNED BY` *`table.column`*
`OWNED BY NONE`

The `OWNED BY` option causes the sequence to be associated with a specific table column, such that if that column (or its whole table) is dropped, the sequence will be automatically dropped as well. The specified table must have the same owner and be in the same schema as the sequence. `OWNED BY NONE`, the default, specifies that there is no such association.

Notes

Use `DROP SEQUENCE` to remove a sequence.

Sequences are based on `bigint` arithmetic, so the range cannot exceed the range of an eight-byte integer (-9223372036854775808 to 9223372036854775807). On some older platforms, there may be no

compiler support for eight-byte integers, in which case sequences use regular `integer` arithmetic (range -2147483648 to +2147483647).

Unexpected results may be obtained if a *`cache`* setting greater than one is used for a sequence object that will be used concurrently by multiple sessions. Each session will allocate and cache successive sequence values during one access to the sequence object and increase the sequence object's `last_value` accordingly. Then, the next *`cache`*-1 uses of `nextval` within that session simply return the preallocated values without touching the sequence object. So, any numbers allocated but not used within a session will be lost when that session ends, resulting in "holes" in the sequence.

Furthermore, although multiple sessions are guaranteed to allocate distinct sequence values, the values may be generated out of sequence when all the sessions are considered. For example, with a *`cache`* setting of 10, session A might reserve values 1..10 and return `nextval`=1, then session B might reserve values 11..20 and return `nextval`=11 before session A has generated `nextval`=2. Thus, with a *`cache`* setting of one it is safe to assume that `nextval` values are generated sequentially; with a *`cache`* setting greater than one you should only assume that the `nextval` values are all distinct, not that they are generated purely sequentially. Also, `last_value` will reflect the latest value reserved by any session, whether or not it has yet been returned by `nextval`.

Another consideration is that a `setval` executed on such a sequence will not be noticed by other sessions until they have used up any preallocated values they have cached.

Examples

Create an ascending sequence called `serial`, starting at 101:

```
CREATE SEQUENCE serial START 101;
```

Select the next number from this sequence:

```
SELECT nextval('serial');

 nextval
---------
     114
```

Use this sequence in an `INSERT` command:

```
INSERT INTO distributors VALUES (nextval('serial'), 'nothing');
```

Update the sequence value after a `COPY FROM`:

```
BEGIN;
COPY distributors FROM 'input_file';
SELECT setval('serial', max(id)) FROM distributors;
END;
```

Compatibility

`CREATE SEQUENCE` conforms to the SQL standard, with the following exceptions:

- The standard's `AS <data type>` expression is not supported.
- Obtaining the next value is done using the `nextval()` function instead of the standard's `NEXT VALUE FOR` expression.
- The `OWNED BY` clause is a PostgreSQL extension.

See Also

ALTER SEQUENCE, *DROP SEQUENCE*

CREATE TABLE

Name

CREATE TABLE — define a new table

Synopsis

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name ( [
  { column_name data_type [ DEFAULT default_expr ] [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE parent_table [ { INCLUDING | EXCLUDING } { DEFAULTS | CONSTRAINTS } ] ... }
    [, ... ]
] )
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace ]

where column_constraint is:

[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  UNIQUE index_parameters |
  PRIMARY KEY index_parameters |
  CHECK ( expression ) |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

and table_constraint is:

[ CONSTRAINT constraint_name ]
{ UNIQUE ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |
  CHECK ( expression ) |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE action ] [ ON UPDATE action ]
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

index_parameters in UNIQUE and PRIMARY KEY constraints are:

[ WITH ( storage_parameter [= value] [, ... ] ) ]
[ USING INDEX TABLESPACE tablespace ]
```

Description

CREATE TABLE will create a new, initially empty table in the current database. The table will be owned by the user issuing the command.

If a schema name is given (for example, CREATE TABLE myschema.mytable ...) then the table is created in the specified schema. Otherwise it is created in the current schema. Temporary tables exist in a special schema, so a schema name may not be given when creating a temporary table. The name of the table must be distinct from the name of any other table, sequence, index, or view in the same schema.

CREATE TABLE also automatically creates a data type that represents the composite type corresponding to one row of the table. Therefore, tables cannot have the same name as any existing data type in the same schema.

The optional constraint clauses specify constraints (tests) that new or updated rows must satisfy for an insert or update operation to succeed. A constraint is an SQL object that helps define the set of valid values in the table in various ways.

There are two ways to define constraints: table constraints and column constraints. A column constraint is defined as part of a column definition. A table constraint definition is not tied to a particular column, and it can encompass more than one column. Every column constraint can also be written as a table constraint; a column constraint is only a notational convenience for use when the constraint only affects one column.

Parameters

TEMPORARY or TEMP

If specified, the table is created as a temporary table. Temporary tables are automatically dropped at the end of a session, or optionally at the end of the current transaction (see ON COMMIT below). Existing permanent tables with the same name are not visible to the current session while the temporary table exists, unless they are referenced with schema-qualified names. Any indexes created on a temporary table are automatically temporary as well.

Optionally, GLOBAL or LOCAL can be written before TEMPORARY or TEMP. This makes no difference in PostgreSQL, but see *Compatibility*.

table_name

The name (optionally schema-qualified) of the table to be created.

column_name

The name of a column to be created in the new table.

data_type

The data type of the column. This may include array specifiers. For more information on the data types supported by PostgreSQL, refer to Chapter 8.

DEFAULT *default_expr*

The DEFAULT clause assigns a default data value for the column whose column definition it appears within. The value is any variable-free expression (subqueries and cross-references to other columns in the current table are not allowed). The data type of the default expression must match the data type of the column.

The default expression will be used in any insert operation that does not specify a value for the column. If there is no default for a column, then the default is null.

`INHERITS ( parent_table [, ... ] )`

The optional `INHERITS` clause specifies a list of tables from which the new table automatically inherits all columns.

Use of `INHERITS` creates a persistent relationship between the new child table and its parent table(s). Schema modifications to the parent(s) normally propagate to children as well, and by default the data of the child table is included in scans of the parent(s).

If the same column name exists in more than one parent table, an error is reported unless the data types of the columns match in each of the parent tables. If there is no conflict, then the duplicate columns are merged to form a single column in the new table. If the column name list of the new table contains a column name that is also inherited, the data type must likewise match the inherited column(s), and the column definitions are merged into one. However, inherited and new column declarations of the same name need not specify identical constraints: all constraints provided from any declaration are merged together and all are applied to the new table. If the new table explicitly specifies a default value for the column, this default overrides any defaults from inherited declarations of the column. Otherwise, any parents that specify default values for the column must all specify the same default, or an error will be reported.

`LIKE parent_table [ { INCLUDING | EXCLUDING } { DEFAULTS | CONSTRAINTS } ]`

The `LIKE` clause specifies a table from which the new table automatically copies all column names, their data types, and their not-null constraints.

Unlike `INHERITS`, the new table and original table are completely decoupled after creation is complete. Changes to the original table will not be applied to the new table, and it is not possible to include data of the new table in scans of the original table.

Default expressions for the copied column definitions will only be copied if `INCLUDING DEFAULTS` is specified. The default behavior is to exclude default expressions, resulting in the copied columns in the new table having null defaults.

Not-null constraints are always copied to the new table. `CHECK` constraints will only be copied if `INCLUDING CONSTRAINTS` is specified; other types of constraints will never be copied. Also, no distinction is made between column constraints and table constraints — when constraints are requested, all check constraints are copied.

Note also that unlike `INHERITS`, copied columns and constraints are not merged with similarly named columns and constraints. If the same name is specified explicitly or in another `LIKE` clause an error is signalled.

`CONSTRAINT constraint_name`

An optional name for a column or table constraint. If the constraint is violated, the constraint name is present in error messages, so constraint names like `col must be positive` can be used to communicate helpful constraint information to client applications. (Double-quotes are needed to specify constraint names that contain spaces.) If a constraint name is not specified, the system generates a name.

`NOT NULL`

The column is not allowed to contain null values.

`NULL`

The column is allowed to contain null values. This is the default.

This clause is only provided for compatibility with non-standard SQL databases. Its use is discouraged in new applications.

`UNIQUE` (column constraint)
`UNIQUE ( column_name [, ... ] )` (table constraint)

The `UNIQUE` constraint specifies that a group of one or more columns of a table may contain only unique values. The behavior of the unique table constraint is the same as that for column constraints, with the additional capability to span multiple columns.

For the purpose of a unique constraint, null values are not considered equal.

Each unique table constraint must name a set of columns that is different from the set of columns named by any other unique or primary key constraint defined for the table. (Otherwise it would just be the same constraint listed twice.)

`PRIMARY KEY` (column constraint)
`PRIMARY KEY ( column_name [, ... ] )` (table constraint)

The primary key constraint specifies that a column or columns of a table may contain only unique (non-duplicate), nonnull values. Technically, `PRIMARY KEY` is merely a combination of `UNIQUE` and `NOT NULL`, but identifying a set of columns as primary key also provides metadata about the design of the schema, as a primary key implies that other tables may rely on this set of columns as a unique identifier for rows.

Only one primary key can be specified for a table, whether as a column constraint or a table constraint.

The primary key constraint should name a set of columns that is different from other sets of columns named by any unique constraint defined for the same table.

`CHECK ( expression )`

The `CHECK` clause specifies an expression producing a Boolean result which new or updated rows must satisfy for an insert or update operation to succeed. Expressions evaluating to TRUE or UNKNOWN succeed. Should any row of an insert or update operation produce a FALSE result an error exception is raised and the insert or update does not alter the database. A check constraint specified as a column constraint should reference that column's value only, while an expression appearing in a table constraint may reference multiple columns.

Currently, `CHECK` expressions cannot contain subqueries nor refer to variables other than columns of the current row.

`REFERENCES reftable [ ( refcolumn ) ] [ MATCH matchtype ] [ ON DELETE action ] [ ON UPDATE action ]` (column constraint)
`FOREIGN KEY ( column [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ] [ MATCH matchtype ] [ ON DELETE action ] [ ON UPDATE action ]` (table constraint)

These clauses specify a foreign key constraint, which requires that a group of one or more columns of the new table must only contain values that match values in the referenced column(s) of some row of the referenced table. If *`refcolumn`* is omitted, the primary key of the *`reftable`* is used. The referenced columns must be the columns of a unique or primary key constraint in the referenced table. Note that foreign key constraints may not be defined between temporary tables and permanent tables.

A value inserted into the referencing column(s) is matched against the values of the referenced table and referenced columns using the given match type. There are three match types: `MATCH FULL`, `MATCH PARTIAL`, and `MATCH SIMPLE`, which is also the default. `MATCH FULL` will not allow one column of a multicolumn foreign key to be null unless all foreign key columns are null. `MATCH SIMPLE` allows some foreign key columns to be null while other parts of the foreign key are not null. `MATCH PARTIAL` is not yet implemented.

In addition, when the data in the referenced columns is changed, certain actions are performed on the data in this table's columns. The `ON DELETE` clause specifies the action to perform when a referenced row in the referenced table is being deleted. Likewise, the `ON UPDATE` clause specifies the action to perform when a referenced column in the referenced table is being updated to a new value. If the row is updated, but the referenced column is not actually changed, no action is done. Referential actions other than the `NO ACTION` check cannot be deferred, even if the constraint is declared deferrable. There are the following possible actions for each clause:

`NO ACTION`

Produce an error indicating that the deletion or update would create a foreign key constraint violation. If the constraint is deferred, this error will be produced at constraint check time if there still exist any referencing rows. This is the default action.

`RESTRICT`

Produce an error indicating that the deletion or update would create a foreign key constraint violation. This is the same as `NO ACTION` except that the check is not deferrable.

`CASCADE`

Delete any rows referencing the deleted row, or update the value of the referencing column to the new value of the referenced column, respectively.

`SET NULL`

Set the referencing column(s) to null.

`SET DEFAULT`

Set the referencing column(s) to their default values.

If the referenced column(s) are changed frequently, it may be wise to add an index to the foreign key column so that referential actions associated with the foreign key column can be performed more efficiently.

`DEFERRABLE`
`NOT DEFERRABLE`

This controls whether the constraint can be deferred. A constraint that is not deferrable will be checked immediately after every command. Checking of constraints that are deferrable may be postponed until the end of the transaction (using the *SET CONSTRAINTS* command). `NOT DEFERRABLE` is the default. Only foreign key constraints currently accept this clause. All other constraint types are not deferrable.

`INITIALLY IMMEDIATE`
`INITIALLY DEFERRED`

If a constraint is deferrable, this clause specifies the default time to check the constraint. If the constraint is `INITIALLY IMMEDIATE`, it is checked after each statement. This is the default. If the constraint is `INITIALLY DEFERRED`, it is checked only at the end of the transaction. The constraint check time can be altered with the *SET CONSTRAINTS* command.

`WITH ( storage_parameter [= value] [, ... ] )`

This clause specifies optional storage parameters for a table or index; see *Storage Parameters* for more information. The `WITH` clause for a table can also include `OIDS=TRUE` (or just `OIDS`) to specify that rows of the new table should have OIDs (object identifiers) assigned to them, or `OIDS=FALSE` to specify that the rows should not have OIDs. If `OIDS` is not specified, the default setting depends upon the default_with_oids configuration parameter. (If the new table inherits from any tables that have OIDs, then `OIDS=TRUE` is forced even if the command says `OIDS=FALSE`.)

If `OIDS=FALSE` is specified or implied, the new table does not store OIDs and no OID will be assigned for a row inserted into it. This is generally considered worthwhile, since it will reduce OID consumption and thereby postpone the wraparound of the 32-bit OID counter. Once the counter wraps around, OIDs can no longer be assumed to be unique, which makes them considerably less useful. In addition, excluding OIDs from a table reduces the space required to store the table on disk by 4 bytes per row (on most machines), slightly improving performance.

To remove OIDs from a table after it has been created, use *ALTER TABLE*.

`WITH OIDS`
`WITHOUT OIDS`

These are obsolescent syntaxes equivalent to `WITH (OIDS)` and `WITH (OIDS=FALSE)`, respectively. If you wish to give both an `OIDS` setting and storage parameters, you must use the `WITH ( ... )` syntax; see above.

`ON COMMIT`

The behavior of temporary tables at the end of a transaction block can be controlled using `ON COMMIT`. The three options are:

`PRESERVE ROWS`

No special action is taken at the ends of transactions. This is the default behavior.

`DELETE ROWS`

All rows in the temporary table will be deleted at the end of each transaction block. Essentially, an automatic *TRUNCATE* is done at each commit.

`DROP`

The temporary table will be dropped at the end of the current transaction block.

`TABLESPACE tablespace`

The *tablespace* is the name of the tablespace in which the new table is to be created. If not specified, default_tablespace is used, or the database's default tablespace if `default_tablespace` is an empty string.

`USING INDEX TABLESPACE` *tablespace*

This clause allows selection of the tablespace in which the index associated with a `UNIQUE` or `PRIMARY KEY` constraint will be created. If not specified, default_tablespace is used, or the database's default tablespace if `default_tablespace` is an empty string.

Storage Parameters

The `WITH` clause can specify *storage parameters* for tables, and for indexes associated with a `UNIQUE` or `PRIMARY KEY` constraint. Storage parameters for indexes are documented in *CREATE INDEX*. The only storage parameter currently available for tables is:

`FILLFACTOR`

The fillfactor for a table is a percentage between 10 and 100. 100 (complete packing) is the default. When a smaller fillfactor is specified, `INSERT` operations pack table pages only to the indicated percentage; the remaining space on each page is reserved for updating rows on that page. This gives `UPDATE` a chance to place the updated copy of a row on the same page as the original, which is more efficient than placing it on a different page. For a table whose entries are never updated, complete packing is the best choice, but in heavily updated tables smaller fillfactors are appropriate.

Notes

Using OIDs in new applications is not recommended: where possible, using a `SERIAL` or other sequence generator as the table's primary key is preferred. However, if your application does make use of OIDs to identify specific rows of a table, it is recommended to create a unique constraint on the `oid` column of that table, to ensure that OIDs in the table will indeed uniquely identify rows even after counter wraparound. Avoid assuming that OIDs are unique across tables; if you need a database-wide unique identifier, use the combination of `tableoid` and row OID for the purpose.

Tip: The use of `OIDS=FALSE` is not recommended for tables with no primary key, since without either an OID or a unique data key, it is difficult to identify specific rows.

PostgreSQL automatically creates an index for each unique constraint and primary key constraint to enforce uniqueness. Thus, it is not necessary to create an index explicitly for primary key columns. (See *CREATE INDEX* for more information.)

Unique constraints and primary keys are not inherited in the current implementation. This makes the combination of inheritance and unique constraints rather dysfunctional.

A table cannot have more than 1600 columns. (In practice, the effective limit is usually lower because of tuple-length constraints.)

Examples

Create table `films` and table `distributors`:

```
CREATE TABLE films (
    code        char(5) CONSTRAINT firstkey PRIMARY KEY,
    title       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute
);

CREATE TABLE distributors (
     did    integer PRIMARY KEY DEFAULT nextval('serial'),
     name   varchar(40) NOT NULL CHECK (name <> '')
);
```

Create a table with a 2-dimensional array:

```
CREATE TABLE array_int (
    vector  int[][]
);
```

Define a unique table constraint for the table `films`. Unique table constraints can be defined on one or more columns of the table.

```
CREATE TABLE films (
    code        char(5),
    title       varchar(40),
    did         integer,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute,
    CONSTRAINT production UNIQUE(date_prod)
);
```

Define a check column constraint:

```
CREATE TABLE distributors (
    did     integer CHECK (did > 100),
    name    varchar(40)
);
```

Define a check table constraint:

```
CREATE TABLE distributors (
    did     integer,
```

```
    name    varchar(40)
    CONSTRAINT con1 CHECK (did > 100 AND name <> '')
);
```

Define a primary key table constraint for the table `films`:

```
CREATE TABLE films (
    code        char(5),
    title       varchar(40),
    did         integer,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute,
    CONSTRAINT code_title PRIMARY KEY(code,title)
);
```

Define a primary key constraint for table `distributors`. The following two examples are equivalent, the first using the table constraint syntax, the second the column constraint syntax:

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    PRIMARY KEY(did)
);

CREATE TABLE distributors (
    did     integer PRIMARY KEY,
    name    varchar(40)
);
```

Assign a literal constant default value for the column `name`, arrange for the default value of column `did` to be generated by selecting the next value of a sequence object, and make the default value of `modtime` be the time at which the row is inserted:

```
CREATE TABLE distributors (
    name      varchar(40) DEFAULT 'Luso Films',
    did       integer DEFAULT nextval('distributors_serial'),
    modtime   timestamp DEFAULT current_timestamp
);
```

Define two `NOT NULL` column constraints on the table `distributors`, one of which is explicitly given a name:

```
CREATE TABLE distributors (
    did     integer CONSTRAINT no_null NOT NULL,
    name    varchar(40) NOT NULL
);
```

Define a unique constraint for the `name` column:

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40) UNIQUE
);
```

The same, specified as a table constraint:

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    UNIQUE(name)
);
```

Create the same table, specifying 70% fill factor for both the table and its unique index:

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    UNIQUE(name) WITH (fillfactor=70)
)
WITH (fillfactor=70);
```

Create table `cinemas` in tablespace `diskvol1`:

```
CREATE TABLE cinemas (
        id serial,
        name text,
        location text
) TABLESPACE diskvol1;
```

Compatibility

The `CREATE TABLE` command conforms to the SQL standard, with exceptions listed below.

Temporary Tables

Although the syntax of `CREATE TEMPORARY TABLE` resembles that of the SQL standard, the effect is not the same. In the standard, temporary tables are defined just once and automatically exist (starting with empty contents) in every session that needs them. PostgreSQL instead requires each session to issue its own `CREATE TEMPORARY TABLE` command for each temporary table to be used. This allows different sessions to use the same temporary table name for different purposes, whereas the standard's approach constrains all instances of a given temporary table name to have the same table structure.

The standard's definition of the behavior of temporary tables is widely ignored. PostgreSQL's behavior on this point is similar to that of several other SQL databases.

The standard's distinction between global and local temporary tables is not in PostgreSQL, since that distinction depends on the concept of modules, which PostgreSQL does not have. For compatibility's sake, PostgreSQL will accept the `GLOBAL` and `LOCAL` keywords in a temporary table declaration, but they have no effect.

The `ON COMMIT` clause for temporary tables also resembles the SQL standard, but has some differences. If the `ON COMMIT` clause is omitted, SQL specifies that the default behavior is `ON COMMIT DELETE ROWS`. However, the default behavior in PostgreSQL is `ON COMMIT PRESERVE ROWS`. The `ON COMMIT DROP` option does not exist in SQL.

Column Check Constraints

The SQL standard says that `CHECK` column constraints may only refer to the column they apply to; only `CHECK` table constraints may refer to multiple columns. PostgreSQL does not enforce this restriction; it treats column and table check constraints alike.

`NULL` "Constraint"

The `NULL` "constraint" (actually a non-constraint) is a PostgreSQL extension to the SQL standard that is included for compatibility with some other database systems (and for symmetry with the `NOT NULL` constraint). Since it is the default for any column, its presence is simply noise.

Inheritance

Multiple inheritance via the `INHERITS` clause is a PostgreSQL language extension. SQL:1999 and later define single inheritance using a different syntax and different semantics. SQL:1999-style inheritance is not yet supported by PostgreSQL.

Zero-column tables

PostgreSQL allows a table of no columns to be created (for example, `CREATE TABLE foo();`). This is an extension from the SQL standard, which does not allow zero-column tables. Zero-column tables are not in themselves very useful, but disallowing them creates odd special cases for `ALTER TABLE DROP COLUMN`, so it seems cleaner to ignore this spec restriction.

`WITH` clause

The `WITH` clause is a PostgreSQL extension; neither storage parameters nor OIDs are in the standard.

Tablespaces

The PostgreSQL concept of tablespaces is not part of the standard. Hence, the clauses `TABLESPACE` and `USING INDEX TABLESPACE` are extensions.

See Also

ALTER TABLE, *DROP TABLE*, *CREATE TABLESPACE*

CREATE TABLE AS

Name

CREATE TABLE AS — define a new table from the results of a query

Synopsis

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name
    [ (column_name [, ...] ) ]
    [ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
    [ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
    [ TABLESPACE tablespace ]
    AS query
```

Description

CREATE TABLE AS creates a table and fills it with data computed by a SELECT command. The table columns have the names and data types associated with the output columns of the SELECT (except that you can override the column names by giving an explicit list of new column names).

CREATE TABLE AS bears some resemblance to creating a view, but it is really quite different: it creates a new table and evaluates the query just once to fill the new table initially. The new table will not track subsequent changes to the source tables of the query. In contrast, a view re-evaluates its defining SELECT statement whenever it is queried.

Parameters

GLOBAL or LOCAL

Ignored for compatibility. Refer to *CREATE TABLE* for details.

TEMPORARY or TEMP

If specified, the table is created as a temporary table. Refer to *CREATE TABLE* for details.

table_name

The name (optionally schema-qualified) of the table to be created.

column_name

The name of a column in the new table. If column names are not provided, they are taken from the output column names of the query. If the table is created from an EXECUTE command, a column name list cannot be specified.

`WITH ( storage_parameter [= value] [, ... ] )`

This clause specifies optional storage parameters for the new table; see *Storage Parameters* for more information. The `WITH` clause can also include `OIDS=TRUE` (or just `OIDS`) to specify that rows of the new table should have OIDs (object identifiers) assigned to them, or `OIDS=FALSE` to specify that the rows should not have OIDs. See *CREATE TABLE* for more information.

`WITH OIDS`
`WITHOUT OIDS`

These are obsolescent syntaxes equivalent to `WITH (OIDS)` and `WITH (OIDS=FALSE)`, respectively. If you wish to give both an `OIDS` setting and storage parameters, you must use the `WITH ( ... )` syntax; see above.

`ON COMMIT`

The behavior of temporary tables at the end of a transaction block can be controlled using `ON COMMIT`. The three options are:

`PRESERVE ROWS`

No special action is taken at the ends of transactions. This is the default behavior.

`DELETE ROWS`

All rows in the temporary table will be deleted at the end of each transaction block. Essentially, an automatic *TRUNCATE* is done at each commit.

`DROP`

The temporary table will be dropped at the end of the current transaction block.

`TABLESPACE tablespace`

The *tablespace* is the name of the tablespace in which the new table is to be created. If not specified, default_tablespace is used, or the database's default tablespace if `default_tablespace` is an empty string.

query

A *SELECT* or *VALUES* command, or an *EXECUTE* command that runs a prepared `SELECT` or `VALUES` query.

Notes

This command is functionally similar to *SELECT INTO*, but it is preferred since it is less likely to be confused with other uses of the `SELECT INTO` syntax. Furthermore, `CREATE TABLE AS` offers a superset of the functionality offered by `SELECT INTO`.

Prior to PostgreSQL 8.0, `CREATE TABLE AS` always included OIDs in the table it created. As of PostgreSQL 8.0, the `CREATE TABLE AS` command allows the user to explicitly specify whether OIDs should be included. If the presence of OIDs is not explicitly specified, the default_with_oids configuration variable is used. As of PostgreSQL 8.1, this variable is false by default, so the default behavior is not identical to pre-8.0 releases. Applications that require OIDs in the table created by `CREATE TABLE AS` should explicitly specify `WITH (OIDS)` to ensure proper behavior.

Examples

Create a new table `films_recent` consisting of only recent entries from the table `films`:

```
CREATE TABLE films_recent AS
  SELECT * FROM films WHERE date_prod >= '2002-01-01';
```

Create a new temporary table `films_recent`, consisting of only recent entries from the table `films`, using a prepared statement. The new table has OIDs and will be dropped at commit:

```
PREPARE recentfilms(date) AS
  SELECT * FROM films WHERE date_prod > $1;
CREATE TEMP TABLE films_recent WITH (OIDS) ON COMMIT DROP AS
  EXECUTE recentfilms('2002-01-01');
```

Compatibility

`CREATE TABLE AS` conforms to the SQL standard, with the following exceptions:

- The standard requires parentheses around the subquery clause; in PostgreSQL, these parentheses are optional.
- The standard defines a `WITH [ NO ] DATA` clause; this is not currently implemented by PostgreSQL. The behavior provided by PostgreSQL is equivalent to the standard's `WITH DATA` case. `WITH NO DATA` can be simulated by appending `LIMIT 0` to the query.
- PostgreSQL handles temporary tables in a way rather different from the standard; see *CREATE TABLE* for details.
- The `WITH` clause is a PostgreSQL extension; neither storage parameters nor OIDs are in the standard.
- The PostgreSQL concept of tablespaces is not part of the standard. Hence, the clause `TABLESPACE` is an extension.

See Also

CREATE TABLE, *EXECUTE*, *SELECT*, *SELECT INTO*, *VALUES*

CREATE TABLESPACE

Name

CREATE TABLESPACE — define a new tablespace

Synopsis

```
CREATE TABLESPACE tablespacename [ OWNER username ] LOCATION 'directory'
```

Description

CREATE TABLESPACE registers a new cluster-wide tablespace. The tablespace name must be distinct from the name of any existing tablespace in the database cluster.

A tablespace allows superusers to define an alternative location on the file system where the data files containing database objects (such as tables and indexes) may reside.

A user with appropriate privileges can pass *tablespacename* to CREATE DATABASE, CREATE TABLE, CREATE INDEX or ADD CONSTRAINT to have the data files for these objects stored within the specified tablespace.

Parameters

tablespacename

The name of a tablespace to be created. The name cannot begin with pg_, as such names are reserved for system tablespaces.

username

The name of the user who will own the tablespace. If omitted, defaults to the user executing the command. Only superusers may create tablespaces, but they can assign ownership of tablespaces to non-superusers.

directory

The directory that will be used for the tablespace. The directory must be empty and must be owned by the PostgreSQL system user. The directory must be specified by an absolute path name.

Notes

Tablespaces are only supported on systems that support symbolic links.

CREATE TABLESPACE cannot be executed inside a transaction block.

Examples

Create a tablespace `dbspace` at `/data/dbs`:

```
CREATE TABLESPACE dbspace LOCATION '/data/dbs';
```

Create a tablespace `indexspace` at `/data/indexes` owned by user `genevieve`:

```
CREATE TABLESPACE indexspace OWNER genevieve LOCATION '/data/indexes';
```

Compatibility

`CREATE TABLESPACE` is a PostgreSQL extension.

See Also

CREATE DATABASE, *CREATE TABLE*, *CREATE INDEX*, *DROP TABLESPACE*, *ALTER TABLESPACE*

CREATE TRIGGER

Name

CREATE TRIGGER — define a new trigger

Synopsis

```
CREATE TRIGGER name { BEFORE | AFTER } { event [ OR ... ] }
    ON table [ FOR [ EACH ] { ROW | STATEMENT } ]
    EXECUTE PROCEDURE funcname ( arguments )
```

Description

CREATE TRIGGER creates a new trigger. The trigger will be associated with the specified table and will execute the specified function *funcname* when certain events occur.

The trigger can be specified to fire either before the operation is attempted on a row (before constraints are checked and the INSERT, UPDATE, or DELETE is attempted) or after the operation has completed (after constraints are checked and the INSERT, UPDATE, or DELETE has completed). If the trigger fires before the event, the trigger may skip the operation for the current row, or change the row being inserted (for INSERT and UPDATE operations only). If the trigger fires after the event, all changes, including the last insertion, update, or deletion, are "visible" to the trigger.

A trigger that is marked FOR EACH ROW is called once for every row that the operation modifies. For example, a DELETE that affects 10 rows will cause any ON DELETE triggers on the target relation to be called 10 separate times, once for each deleted row. In contrast, a trigger that is marked FOR EACH STATEMENT only executes once for any given operation, regardless of how many rows it modifies (in particular, an operation that modifies zero rows will still result in the execution of any applicable FOR EACH STATEMENT triggers).

If multiple triggers of the same kind are defined for the same event, they will be fired in alphabetical order by name.

SELECT does not modify any rows so you can not create SELECT triggers. Rules and views are more appropriate in such cases.

Refer to Chapter 34 for more information about triggers.

Parameters

name

The name to give the new trigger. This must be distinct from the name of any other trigger for the same table.

`BEFORE`
`AFTER`

Determines whether the function is called before or after the event.

`event`

One of `INSERT`, `UPDATE`, or `DELETE`; this specifies the event that will fire the trigger. Multiple events can be specified using `OR`.

`table`

The name (optionally schema-qualified) of the table the trigger is for.

`FOR EACH ROW`
`FOR EACH STATEMENT`

This specifies whether the trigger procedure should be fired once for every row affected by the trigger event, or just once per SQL statement. If neither is specified, `FOR EACH STATEMENT` is the default.

`funcname`

A user-supplied function that is declared as taking no arguments and returning type `trigger`, which is executed when the trigger fires.

`arguments`

An optional comma-separated list of arguments to be provided to the function when the trigger is executed. The arguments are literal string constants. Simple names and numeric constants may be written here, too, but they will all be converted to strings. Please check the description of the implementation language of the trigger function about how the trigger arguments are accessible within the function; it may be different from normal function arguments.

Notes

To create a trigger on a table, the user must have the `TRIGGER` privilege on the table.

In PostgreSQL versions before 7.3, it was necessary to declare trigger functions as returning the placeholder type `opaque`, rather than `trigger`. To support loading of old dump files, `CREATE TRIGGER` will accept a function declared as returning `opaque`, but it will issue a notice and change the function's declared return type to `trigger`.

Use *DROP TRIGGER* to remove a trigger.

Examples

Section 34.4 contains a complete example.

Compatibility

The `CREATE TRIGGER` statement in PostgreSQL implements a subset of the SQL standard. The following functionality is currently missing:

- SQL allows triggers to fire on updates to specific columns (e.g., `AFTER UPDATE OF col1, col2`).
- SQL allows you to define aliases for the "old" and "new" rows or tables for use in the definition of the triggered action (e.g., `CREATE TRIGGER ... ON tablename REFERENCING OLD ROW AS somename NEW ROW AS othername ...`). Since PostgreSQL allows trigger procedures to be written in any number of user-defined languages, access to the data is handled in a language-specific way.
- PostgreSQL only allows the execution of a user-defined function for the triggered action. The standard allows the execution of a number of other SQL commands, such as `CREATE TABLE` as the triggered action. This limitation is not hard to work around by creating a user-defined function that executes the desired commands.

SQL specifies that multiple triggers should be fired in time-of-creation order. PostgreSQL uses name order, which was judged to be more convenient.

SQL specifies that `BEFORE DELETE` triggers on cascaded deletes fire *after* the cascaded `DELETE` completes. The PostgreSQL behavior is for `BEFORE DELETE` to always fire before the delete action, even a cascading one. This is considered more consistent. There is also unpredictable behavior when `BEFORE` triggers modify rows that are later to be modified by referential actions. This can lead to constraint violations or stored data that does not honor the referential constraint.

The ability to specify multiple actions for a single trigger using `OR` is a PostgreSQL extension of the SQL standard.

See Also

CREATE FUNCTION, *ALTER TRIGGER*, *DROP TRIGGER*

CREATE TYPE

Name

CREATE TYPE — define a new data type

Synopsis

```
CREATE TYPE name AS
    ( attribute_name data_type [, ... ] )

CREATE TYPE name (
    INPUT = input_function,
    OUTPUT = output_function
    [ , RECEIVE = receive_function ]
    [ , SEND = send_function ]
    [ , ANALYZE = analyze_function ]
    [ , INTERNALLENGTH = { internallength | VARIABLE } ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alignment ]
    [ , STORAGE = storage ]
    [ , DEFAULT = default ]
    [ , ELEMENT = element ]
    [ , DELIMITER = delimiter ]
)

CREATE TYPE name
```

Description

CREATE TYPE registers a new data type for use in the current database. The user who defines a type becomes its owner.

If a schema name is given then the type is created in the specified schema. Otherwise it is created in the current schema. The type name must be distinct from the name of any existing type or domain in the same schema. (Because tables have associated data types, the type name must also be distinct from the name of any existing table in the same schema.)

Composite Types

The first form of CREATE TYPE creates a composite type. The composite type is specified by a list of attribute names and data types. This is essentially the same as the row type of a table, but using CREATE TYPE avoids the need to create an actual table when all that is wanted is to define a type. A stand-alone composite type is useful as the argument or return type of a function.

Base Types

The second form of `CREATE TYPE` creates a new base type (scalar type). The parameters may appear in any order, not only that illustrated above, and most are optional. You must register two or more functions (using `CREATE FUNCTION`) before defining the type. The support functions *`input_function`* and *`output_function`* are required, while the functions *`receive_function`*, *`send_function`* and *`analyze_function`* are optional. Generally these functions have to be coded in C or another low-level language.

The *`input_function`* converts the type's external textual representation to the internal representation used by the operators and functions defined for the type. *`output_function`* performs the reverse transformation. The input function may be declared as taking one argument of type `cstring`, or as taking three arguments of types `cstring`, `oid`, `integer`. The first argument is the input text as a C string, the second argument is the type's own OID (except for array types, which instead receive their element type's OID), and the third is the `typmod` of the destination column, if known (-1 will be passed if not). The input function must return a value of the data type itself. Usually, an input function should be declared STRICT; if it is not, it will be called with a NULL first parameter when reading a NULL input value. The function must still return NULL in this case, unless it raises an error. (This case is mainly meant to support domain input functions, which may need to reject NULL inputs.) The output function must be declared as taking one argument of the new data type. The output function must return type `cstring`. Output functions are not invoked for NULL values.

The optional *`receive_function`* converts the type's external binary representation to the internal representation. If this function is not supplied, the type cannot participate in binary input. The binary representation should be chosen to be cheap to convert to internal form, while being reasonably portable. (For example, the standard integer data types use network byte order as the external binary representation, while the internal representation is in the machine's native byte order.) The receive function should perform adequate checking to ensure that the value is valid. The receive function may be declared as taking one argument of type `internal`, or as taking three arguments of types `internal`, `oid`, `integer`. The first argument is a pointer to a `StringInfo` buffer holding the received byte string; the optional arguments are the same as for the text input function. The receive function must return a value of the data type itself. Usually, a receive function should be declared STRICT; if it is not, it will be called with a NULL first parameter when reading a NULL input value. The function must still return NULL in this case, unless it raises an error. (This case is mainly meant to support domain receive functions, which may need to reject NULL inputs.) Similarly, the optional *`send_function`* converts from the internal representation to the external binary representation. If this function is not supplied, the type cannot participate in binary output. The send function must be declared as taking one argument of the new data type. The send function must return type `bytea`. Send functions are not invoked for NULL values.

You should at this point be wondering how the input and output functions can be declared to have results or arguments of the new type, when they have to be created before the new type can be created. The answer is that the type should first be defined as a *shell type*, which is a placeholder type that has no properties except a name and an owner. This is done by issuing the command `CREATE TYPE` *`name`*, with no additional parameters. Then the I/O functions can be defined referencing the shell type. Finally, `CREATE TYPE` with a full definition replaces the shell entry with a complete, valid type definition, after which the new type can be used normally.

The optional *`analyze_function`* performs type-specific statistics collection for columns of the data type. By default, `ANALYZE` will attempt to gather statistics using the type's "equals" and "less-than" operators, if there is a default b-tree operator class for the type. For non-scalar types this behavior is likely to be unsuitable, so it can be overridden by specifying a custom analysis function. The analysis function

must be declared to take a single argument of type `internal`, and return a `boolean` result. The detailed API for analysis functions appears in `src/include/commands/vacuum.h`.

While the details of the new type's internal representation are only known to the I/O functions and other functions you create to work with the type, there are several properties of the internal representation that must be declared to PostgreSQL. Foremost of these is *`internallength`*. Base data types can be fixed-length, in which case *`internallength`* is a positive integer, or variable length, indicated by setting *`internallength`* to `VARIABLE`. (Internally, this is represented by setting `typlen` to -1.) The internal representation of all variable-length types must start with a 4-byte integer giving the total length of this value of the type.

The optional flag `PASSEDBYVALUE` indicates that values of this data type are passed by value, rather than by reference. You may not pass by value types whose internal representation is larger than the size of the `Datum` type (4 bytes on most machines, 8 bytes on a few).

The *`alignment`* parameter specifies the storage alignment required for the data type. The allowed values equate to alignment on 1, 2, 4, or 8 byte boundaries. Note that variable-length types must have an alignment of at least 4, since they necessarily contain an `int4` as their first component.

The *`storage`* parameter allows selection of storage strategies for variable-length data types. (Only `plain` is allowed for fixed-length types.) `plain` specifies that data of the type will always be stored in-line and not compressed. `extended` specifies that the system will first try to compress a long data value, and will move the value out of the main table row if it's still too long. `external` allows the value to be moved out of the main table, but the system will not try to compress it. `main` allows compression, but discourages moving the value out of the main table. (Data items with this storage strategy may still be moved out of the main table if there is no other way to make a row fit, but they will be kept in the main table preferentially over `extended` and `external` items.)

A default value may be specified, in case a user wants columns of the data type to default to something other than the null value. Specify the default with the `DEFAULT` key word. (Such a default may be overridden by an explicit `DEFAULT` clause attached to a particular column.)

To indicate that a type is an array, specify the type of the array elements using the `ELEMENT` key word. For example, to define an array of 4-byte integers (`int4`), specify `ELEMENT = int4`. More details about array types appear below.

To indicate the delimiter to be used between values in the external representation of arrays of this type, *`delimiter`* can be set to a specific character. The default delimiter is the comma (`,`). Note that the delimiter is associated with the array element type, not the array type itself.

Array Types

Whenever a user-defined base data type is created, PostgreSQL automatically creates an associated array type, whose name consists of the base type's name prepended with an underscore. The parser understands this naming convention, and translates requests for columns of type `foo[]` into requests for type `_foo`. The implicitly-created array type is variable length and uses the built-in input and output functions `array_in` and `array_out`.

You might reasonably ask why there is an `ELEMENT` option, if the system makes the correct array type automatically. The only case where it's useful to use `ELEMENT` is when you are making a fixed-length type that happens to be internally an array of a number of identical things, and you want to allow these things to be accessed directly by subscripting, in addition to whatever operations you plan to provide for the type

as a whole. For example, type `name` allows its constituent `char` elements to be accessed this way. A 2-D `point` type could allow its two component numbers to be accessed like `point[0]` and `point[1]`. Note that this facility only works for fixed-length types whose internal form is exactly a sequence of identical fixed-length fields. A subscriptable variable-length type must have the generalized internal representation used by `array_in` and `array_out`. For historical reasons (i.e., this is clearly wrong but it's far too late to change it), subscripting of fixed-length array types starts from zero, rather than from one as for variable-length arrays.

Parameters

name

The name (optionally schema-qualified) of a type to be created.

attribute_name

The name of an attribute (column) for the composite type.

data_type

The name of an existing data type to become a column of the composite type.

input_function

The name of a function that converts data from the type's external textual form to its internal form.

output_function

The name of a function that converts data from the type's internal form to its external textual form.

receive_function

The name of a function that converts data from the type's external binary form to its internal form.

send_function

The name of a function that converts data from the type's internal form to its external binary form.

analyze_function

The name of a function that performs statistical analysis for the data type.

internallength

A numeric constant that specifies the length in bytes of the new type's internal representation. The default assumption is that it is variable-length.

alignment

The storage alignment requirement of the data type. If specified, it must be `char`, `int2`, `int4`, or `double`; the default is `int4`.

storage

The storage strategy for the data type. If specified, must be `plain`, `external`, `extended`, or `main`; the default is `plain`.

default

The default value for the data type. If this is omitted, the default is null.

element

The type being created is an array; this specifies the type of the array elements.

delimiter

The delimiter character to be used between values in arrays made of this type.

Notes

User-defined type names cannot begin with the underscore character (_) and can only be 62 characters long (or in general NAMEDATALEN - 2, rather than the NAMEDATALEN - 1 characters allowed for other names). Type names beginning with underscore are reserved for internally-created array type names.

Because there are no restrictions on use of a data type once it's been created, creating a base type is tantamount to granting public execute permission on the functions mentioned in the type definition. (The creator of the type is therefore required to own these functions.) This is usually not an issue for the sorts of functions that are useful in a type definition. But you might want to think twice before designing a type in a way that would require "secret" information to be used while converting it to or from external form.

Before PostgreSQL version 8.2, the syntax CREATE TYPE *name* did not exist. The way to create a new base type was to create its input function first. In this approach, PostgreSQL will first see the name of the new data type as the return type of the input function. The shell type is implicitly created in this situation, and then it can be referenced in the definitions of the remaining I/O functions. This approach still works, but is deprecated and may be disallowed in some future release. Also, to avoid accidentally cluttering the catalogs with shell types as a result of simple typos in function definitions, a shell type will only be made this way when the input function is written in C.

In PostgreSQL versions before 7.3, it was customary to avoid creating a shell type at all, by replacing the functions' forward references to the type name with the placeholder pseudotype opaque. The cstring arguments and results also had to be declared as opaque before 7.3. To support loading of old dump files, CREATE TYPE will accept I/O functions declared using opaque, but it will issue a notice and change the function declarations to use the correct types.

Examples

This example creates a composite type and uses it in a function definition:

```
CREATE TYPE compfoo AS (f1 int, f2 text);

CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS $$
    SELECT fooid, fooname FROM foo
$$ LANGUAGE SQL;
```

This example creates the base data type box and then uses the type in a table definition:

```
CREATE TYPE box;

CREATE FUNCTION my_box_in_function(cstring) RETURNS box AS ... ;
CREATE FUNCTION my_box_out_function(box) RETURNS cstring AS ... ;
```

```
CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function
);

CREATE TABLE myboxes (
    id integer,
    description box
);
```

If the internal structure of `box` were an array of four `float4` elements, we might instead use

```
CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function,
    ELEMENT = float4
);
```

which would allow a box value's component numbers to be accessed by subscripting. Otherwise the type behaves the same as before.

This example creates a large object type and uses it in a table definition:

```
CREATE TYPE bigobj (
    INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE
);
CREATE TABLE big_objs (
    id integer,
    obj bigobj
);
```

More examples, including suitable input and output functions, are in Section 33.11.

Compatibility

This `CREATE TYPE` command is a PostgreSQL extension. There is a `CREATE TYPE` statement in the SQL standard that is rather different in detail.

See Also

CREATE FUNCTION, *DROP TYPE*, *ALTER TYPE*, *CREATE DOMAIN*

CREATE USER

Name

CREATE USER — define a new database role

Synopsis

```
CREATE USER name [ [ WITH ] option [ ... ] ]

where option can be:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
    | IN ROLE rolename [, ...]
    | IN GROUP rolename [, ...]
    | ROLE rolename [, ...]
    | ADMIN rolename [, ...]
    | USER rolename [, ...]
    | SYSID uid
```

Description

CREATE USER is now an alias for *CREATE ROLE*. The only difference is that when the command is spelled CREATE USER, LOGIN is assumed by default, whereas NOLOGIN is assumed when the command is spelled CREATE ROLE.

Compatibility

The CREATE USER statement is a PostgreSQL extension. The SQL standard leaves the definition of users to the implementation.

See Also

CREATE ROLE

CREATE VIEW

Name

`CREATE VIEW` — define a new view

Synopsis

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] VIEW name [ ( column_name [, ...] ) ]
    AS query
```

Description

`CREATE VIEW` defines a view of a query. The view is not physically materialized. Instead, the query is run every time the view is referenced in a query.

`CREATE OR REPLACE VIEW` is similar, but if a view of the same name already exists, it is replaced. You can only replace a view with a new query that generates the identical set of columns (i.e., same column names and data types).

If a schema name is given (for example, `CREATE VIEW myschema.myview ...`) then the view is created in the specified schema. Otherwise it is created in the current schema. Temporary views exist in a special schema, so a schema name may not be given when creating a temporary view. The name of the view must be distinct from the name of any other view, table, sequence, or index in the same schema.

Parameters

`TEMPORARY` or `TEMP`

If specified, the view is created as a temporary view. Temporary views are automatically dropped at the end of the current session. Existing permanent relations with the same name are not visible to the current session while the temporary view exists, unless they are referenced with schema-qualified names.

If any of the tables referenced by the view are temporary, the view is created as a temporary view (whether `TEMPORARY` is specified or not).

`name`

The name (optionally schema-qualified) of a view to be created.

`column_name`

An optional list of names to be used for columns of the view. If not given, the column names are deduced from the query.

`query`

A *SELECT* or *VALUES* command which will provide the columns and rows of the view.

Notes

Currently, views are read only: the system will not allow an insert, update, or delete on a view. You can get the effect of an updatable view by creating rules that rewrite inserts, etc. on the view into appropriate actions on other tables. For more information see *CREATE RULE*.

Use the *DROP VIEW* statement to drop views.

Be careful that the names and types of the view's columns will be assigned the way you want. For example,

```
CREATE VIEW vista AS SELECT 'Hello World';
```

is bad form in two ways: the column name defaults to `?column?`, and the column data type defaults to `unknown`. If you want a string literal in a view's result, use something like

```
CREATE VIEW vista AS SELECT text 'Hello World' AS hello;
```

Access to tables referenced in the view is determined by permissions of the view owner. However, functions called in the view are treated the same as if they had been called directly from the query using the view. Therefore the user of a view must have permissions to call all functions used by the view.

Examples

Create a view consisting of all comedy films:

```
CREATE VIEW comedies AS
    SELECT *
    FROM films
    WHERE kind = 'Comedy';
```

Compatibility

The SQL standard specifies some additional capabilities for the `CREATE VIEW` statement:

```
CREATE VIEW name [ ( column_name [, ...] ) ]
    AS query
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

The optional clauses for the full SQL command are:

`CHECK OPTION`

> This option has to do with updatable views. All `INSERT` and `UPDATE` commands on the view will be checked to ensure data satisfy the view-defining condition (that is, the new data would be visible through the view). If they do not, the update will be rejected.

`LOCAL`

Check for integrity on this view.

`CASCADED`

Check for integrity on this view and on any dependent view. `CASCADED` is assumed if neither `CASCADED` nor `LOCAL` is specified.

`CREATE OR REPLACE VIEW` is a PostgreSQL language extension. So is the concept of a temporary view.

See Also

DROP VIEW

DEALLOCATE

Name

`DEALLOCATE` — deallocate a prepared statement

Synopsis

```
DEALLOCATE [ PREPARE ] name
```

Description

`DEALLOCATE` is used to deallocate a previously prepared SQL statement. If you do not explicitly deallocate a prepared statement, it is deallocated when the session ends.

For more information on prepared statements, see *PREPARE*.

Parameters

`PREPARE`

This key word is ignored.

`name`

The name of the prepared statement to deallocate.

Compatibility

The SQL standard includes a `DEALLOCATE` statement, but it is only for use in embedded SQL.

See Also

EXECUTE, *PREPARE*

DECLARE

Name

DECLARE — define a cursor

Synopsis

```
DECLARE name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ]
    CURSOR [ { WITH | WITHOUT } HOLD ] FOR query
    [ FOR { READ ONLY | UPDATE [ OF column [, ...] ] } ]
```

Description

DECLARE allows a user to create cursors, which can be used to retrieve a small number of rows at a time out of a larger query. Cursors can return data either in text or in binary format using *FETCH*.

Normal cursors return data in text format, the same as a SELECT would produce. Since data is stored natively in binary format, the system must do a conversion to produce the text format. Once the information comes back in text form, the client application may need to convert it to a binary format to manipulate it. In addition, data in the text format is often larger in size than in the binary format. Binary cursors return the data in a binary representation that may be more easily manipulated. Nevertheless, if you intend to display the data as text anyway, retrieving it in text form will save you some effort on the client side.

As an example, if a query returns a value of one from an integer column, you would get a string of 1 with a default cursor whereas with a binary cursor you would get a 4-byte field containing the internal representation of the value (in big-endian byte order).

Binary cursors should be used carefully. Many applications, including psql, are not prepared to handle binary cursors and expect data to come back in the text format.

> **Note:** When the client application uses the "extended query" protocol to issue a FETCH command, the Bind protocol message specifies whether data is to be retrieved in text or binary format. This choice overrides the way that the cursor is defined. The concept of a binary cursor as such is thus obsolete when using extended query protocol — any cursor can be treated as either text or binary.

Parameters

name

The name of the cursor to be created.

BINARY

Causes the cursor to return data in binary rather than in text format.

`INSENSITIVE`

Indicates that data retrieved from the cursor should be unaffected by updates to the tables underlying the cursor while the cursor exists. In PostgreSQL, all cursors are insensitive; this key word currently has no effect and is present for compatibility with the SQL standard.

`SCROLL`
`NO SCROLL`

`SCROLL` specifies that the cursor may be used to retrieve rows in a nonsequential fashion (e.g., backward). Depending upon the complexity of the query's execution plan, specifying `SCROLL` may impose a performance penalty on the query's execution time. `NO SCROLL` specifies that the cursor cannot be used to retrieve rows in a nonsequential fashion. The default is to allow scrolling in some cases; this is not the same as specifying `SCROLL`. See *Notes* for details.

`WITH HOLD`
`WITHOUT HOLD`

`WITH HOLD` specifies that the cursor may continue to be used after the transaction that created it successfully commits. `WITHOUT HOLD` specifies that the cursor cannot be used outside of the transaction that created it. If neither `WITHOUT HOLD` nor `WITH HOLD` is specified, `WITHOUT HOLD` is the default.

`query`

A *SELECT* or *VALUES* command which will provide the rows to be returned by the cursor.

`FOR READ ONLY`
`FOR UPDATE`

`FOR READ ONLY` indicates that the cursor will be used in a read-only mode. `FOR UPDATE` indicates that the cursor will be used to update tables. Since cursor updates are not currently supported in PostgreSQL, specifying `FOR UPDATE` will cause an error message and specifying `FOR READ ONLY` has no effect.

`column`

Column(s) to be updated by the cursor. Since cursor updates are not currently supported in PostgreSQL, the `FOR UPDATE` clause provokes an error message.

The key words `BINARY`, `INSENSITIVE`, and `SCROLL` may appear in any order.

Notes

Unless `WITH HOLD` is specified, the cursor created by this command can only be used within the current transaction. Thus, `DECLARE` without `WITH HOLD` is useless outside a transaction block: the cursor would survive only to the completion of the statement. Therefore PostgreSQL reports an error if this command is used outside a transaction block. Use *BEGIN*, *COMMIT* and *ROLLBACK* to define a transaction block.

If `WITH HOLD` is specified and the transaction that created the cursor successfully commits, the cursor can continue to be accessed by subsequent transactions in the same session. (But if the creating transaction is aborted, the cursor is removed.) A cursor created with `WITH HOLD` is closed when an explicit `CLOSE` command is issued on it, or the session ends. In the current implementation, the rows represented by a held cursor are copied into a temporary file or memory area so that they remain available for subsequent transactions.

The `SCROLL` option should be specified when defining a cursor that will be used to fetch backwards. This is required by the SQL standard. However, for compatibility with earlier versions, PostgreSQL will allow backward fetches without `SCROLL`, if the cursor's query plan is simple enough that no extra overhead is needed to support it. However, application developers are advised not to rely on using backward fetches from a cursor that has not been created with `SCROLL`. If `NO SCROLL` is specified, then backward fetches are disallowed in any case.

The SQL standard only makes provisions for cursors in embedded SQL. The PostgreSQL server does not implement an `OPEN` statement for cursors; a cursor is considered to be open when it is declared. However, ECPG, the embedded SQL preprocessor for PostgreSQL, supports the standard SQL cursor conventions, including those involving `DECLARE` and `OPEN` statements.

You can see all available cursors by querying the `pg_cursors` system view.

Examples

To declare a cursor:

```
DECLARE liahona CURSOR FOR SELECT * FROM films;
```

See *FETCH* for more examples of cursor usage.

Compatibility

The SQL standard allows cursors only in embedded SQL and in modules. PostgreSQL permits cursors to be used interactively.

The SQL standard allows cursors to update table data. All PostgreSQL cursors are read only.

Binary cursors are a PostgreSQL extension.

See Also

CLOSE, *FETCH*, *MOVE*

DELETE

Name

DELETE — delete rows of a table

Synopsis

```
DELETE FROM [ ONLY ] table [ [ AS ] alias ]
    [ USING usinglist ]
    [ WHERE condition ]
    [ RETURNING * | output_expression [ AS output_name ] [, ...] ]
```

Description

DELETE deletes rows that satisfy the WHERE clause from the specified table. If the WHERE clause is absent, the effect is to delete all rows in the table. The result is a valid, but empty table.

> **Tip:** *TRUNCATE* is a PostgreSQL extension that provides a faster mechanism to remove all rows from a table.

By default, DELETE will delete rows in the specified table and all its child tables. If you wish to delete only from the specific table mentioned, you must use the ONLY clause.

There are two ways to delete rows in a table using information contained in other tables in the database: using sub-selects, or specifying additional tables in the USING clause. Which technique is more appropriate depends on the specific circumstances.

The optional RETURNING clause causes DELETE to compute and return value(s) based on each row actually deleted. Any expression using the table's columns, and/or columns of other tables mentioned in USING, can be computed. The syntax of the RETURNING list is identical to that of the output list of SELECT.

You must have the DELETE privilege on the table to delete from it, as well as the SELECT privilege for any table in the USING clause or whose values are read in the *condition*.

Parameters

ONLY

If specified, delete rows from the named table only. When not specified, any tables inheriting from the named table are also processed.

table

The name (optionally schema-qualified) of an existing table.

alias

A substitute name for the target table. When an alias is provided, it completely hides the actual name of the table. For example, given `DELETE FROM foo AS f`, the remainder of the `DELETE` statement must refer to this table as `f` not `foo`.

usinglist

A list of table expressions, allowing columns from other tables to appear in the `WHERE` condition. This is similar to the list of tables that can be specified in the *FROM Clause* of a `SELECT` statement; for example, an alias for the table name can be specified. Do not repeat the target table in the *usinglist*, unless you wish to set up a self-join.

condition

An expression returning a value of type `boolean`, which determines the rows that are to be deleted.

output_expression

An expression to be computed and returned by the `DELETE` command after each row is deleted. The expression may use any column names of the *table* or table(s) listed in `USING`. Write * to return all columns.

output_name

A name to use for a returned column.

Outputs

On successful completion, a `DELETE` command returns a command tag of the form

```
DELETE count
```

The *count* is the number of rows deleted. If *count* is 0, no rows matched the *condition* (this is not considered an error).

If the `DELETE` command contains a `RETURNING` clause, the result will be similar to that of a `SELECT` statement containing the columns and values defined in the `RETURNING` list, computed over the row(s) deleted by the command.

Notes

PostgreSQL lets you reference columns of other tables in the `WHERE` condition by specifying the other tables in the `USING` clause. For example, to delete all films produced by a given producer, one might do

```
DELETE FROM films USING producers
  WHERE producer_id = producers.id AND producers.name = 'foo';
```

What is essentially happening here is a join between `films` and `producers`, with all successfully joined `films` rows being marked for deletion. This syntax is not standard. A more standard way to do it is

```
DELETE FROM films
  WHERE producer_id IN (SELECT id FROM producers WHERE name = 'foo');
```

In some cases the join style is easier to write or faster to execute than the sub-select style.

Examples

Delete all films but musicals:

```
DELETE FROM films WHERE kind <> 'Musical';
```

Clear the table `films`:

```
DELETE FROM films;
```

Delete completed tasks, returning full details of the deleted rows:

```
DELETE FROM tasks WHERE status = 'DONE' RETURNING *;
```

Compatibility

This command conforms to the SQL standard, except that the `USING` and `RETURNING` clauses are PostgreSQL extensions.

DROP AGGREGATE

Name

`DROP AGGREGATE` — remove an aggregate function

Synopsis

```
DROP AGGREGATE [ IF EXISTS ] name ( type [ , ... ] ) [ CASCADE | RESTRICT ]
```

Description

`DROP AGGREGATE` will delete an existing aggregate function. To execute this command the current user must be the owner of the aggregate function.

Parameters

`IF EXISTS`

Do not throw an error if the aggregate does not exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of an existing aggregate function.

`type`

An input data type on which the aggregate function operates. To reference a zero-argument aggregate function, write `*` in place of the list of input data types.

`CASCADE`

Automatically drop objects that depend on the aggregate function.

`RESTRICT`

Refuse to drop the aggregate function if any objects depend on it. This is the default.

Examples

To remove the aggregate function `myavg` for type `integer`:

```
DROP AGGREGATE myavg(integer);
```

Compatibility

There is no `DROP AGGREGATE` statement in the SQL standard.

See Also

ALTER AGGREGATE, *CREATE AGGREGATE*

DROP CAST

Name

`DROP CAST` — remove a cast

Synopsis

```
DROP CAST [ IF EXISTS ] (sourcetype AS targettype) [ CASCADE | RESTRICT ]
```

Description

`DROP CAST` removes a previously defined cast.

To be able to drop a cast, you must own the source or the target data type. These are the same privileges that are required to create a cast.

Parameters

`IF EXISTS`

Do not throw an error if the cast does not exist. A notice is issued in this case.

`sourcetype`

The name of the source data type of the cast.

`targettype`

The name of the target data type of the cast.

`CASCADE`
`RESTRICT`

These key words do not have any effect, since there are no dependencies on casts.

Examples

To drop the cast from type `text` to type `int`:

```
DROP CAST (text AS int);
```

Compatibility

The `DROP CAST` command conforms to the SQL standard.

See Also

CREATE CAST

DROP CONVERSION

Name

DROP CONVERSION — remove a conversion

Synopsis

```
DROP CONVERSION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

Description

DROP CONVERSION removes a previously defined conversion. To be able to drop a conversion, you must own the conversion.

Parameters

IF EXISTS

Do not throw an error if the conversion does not exist. A notice is issued in this case.

name

The name of the conversion. The conversion name may be schema-qualified.

CASCADE
RESTRICT

These key words do not have any effect, since there are no dependencies on conversions.

Examples

To drop the conversion named myname:

```
DROP CONVERSION myname;
```

Compatibility

There is no DROP CONVERSION statement in the SQL standard.

See Also

ALTER CONVERSION, *CREATE CONVERSION*

DROP DATABASE

Name

`DROP DATABASE` — remove a database

Synopsis

```
DROP DATABASE [ IF EXISTS ] name
```

Description

`DROP DATABASE` drops a database. It removes the catalog entries for the database and deletes the directory containing the data. It can only be executed by the database owner. Also, it cannot be executed while you or anyone else are connected to the target database. (Connect to `postgres` or any other database to issue this command.)

`DROP DATABASE` cannot be undone. Use it with care!

Parameters

`IF EXISTS`

Do not throw an error if the database does not exist. A notice is issued in this case.

`name`

The name of the database to remove.

Notes

`DROP DATABASE` cannot be executed inside a transaction block.

This command cannot be executed while connected to the target database. Thus, it might be more convenient to use the program *dropdb* instead, which is a wrapper around this command.

Compatibility

There is no `DROP DATABASE` statement in the SQL standard.

See Also

CREATE DATABASE

DROP DOMAIN

Name

`DROP DOMAIN` — remove a domain

Synopsis

```
DROP DOMAIN [IF EXISTS ] name [, ...]  [ CASCADE | RESTRICT ]
```

Description

`DROP DOMAIN` will remove a domain. Only the owner of a domain can remove it.

Parameters

`IF EXISTS`

Do not throw an error if the domain does not exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of an existing domain.

`CASCADE`

Automatically drop objects that depend on the domain (such as table columns).

`RESTRICT`

Refuse to drop the domain if any objects depend on it. This is the default.

Examples

To remove the domain `box`:

```
DROP DOMAIN box;
```

Compatibility

This command conforms to the SQL standard, except for the `IF EXISTS` option, which is a PostgreSQL extension.

See Also

CREATE DOMAIN, *ALTER DOMAIN*

DROP FUNCTION

Name

DROP FUNCTION — remove a function

Synopsis

```
DROP FUNCTION [ IF EXISTS ] name ( [ [ argmode ] [ argname ] argtype [, ...] ] )
    [ CASCADE | RESTRICT ]
```

Description

DROP FUNCTION removes the definition of an existing function. To execute this command the user must be the owner of the function. The argument types to the function must be specified, since several different functions may exist with the same name and different argument lists.

Parameters

IF EXISTS

Do not throw an error if the function does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of an existing function.

argmode

The mode of an argument: either IN, OUT, or INOUT. If omitted, the default is IN. Note that DROP FUNCTION does not actually pay any attention to OUT arguments, since only the input arguments are needed to determine the function's identity. So it is sufficient to list the IN and INOUT arguments.

argname

The name of an argument. Note that DROP FUNCTION does not actually pay any attention to argument names, since only the argument data types are needed to determine the function's identity.

argtype

The data type(s) of the function's arguments (optionally schema-qualified), if any.

CASCADE

Automatically drop objects that depend on the function (such as operators or triggers).

RESTRICT

Refuse to drop the function if any objects depend on it. This is the default.

Examples

This command removes the square root function:

```
DROP FUNCTION sqrt(integer);
```

Compatibility

A `DROP FUNCTION` statement is defined in the SQL standard, but it is not compatible with this command.

See Also

CREATE FUNCTION, *ALTER FUNCTION*

DROP GROUP

Name

`DROP GROUP` — remove a database role

Synopsis

```
DROP GROUP [ IF EXISTS ] name [, ...]
```

Description

`DROP GROUP` is now an alias for *DROP ROLE*.

Compatibility

There is no `DROP GROUP` statement in the SQL standard.

See Also

DROP ROLE

DROP INDEX

Name

DROP INDEX — remove an index

Synopsis

```
DROP INDEX [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Description

`DROP INDEX` drops an existing index from the database system. To execute this command you must be the owner of the index.

Parameters

`IF EXISTS`

Do not throw an error if the index does not exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of an index to remove.

`CASCADE`

Automatically drop objects that depend on the index.

`RESTRICT`

Refuse to drop the index if any objects depend on it. This is the default.

Examples

This command will remove the index `title_idx`:

```
DROP INDEX title_idx;
```

Compatibility

`DROP INDEX` is a PostgreSQL language extension. There are no provisions for indexes in the SQL standard.

See Also

CREATE INDEX

DROP LANGUAGE

Name

DROP LANGUAGE — remove a procedural language

Synopsis

```
DROP [ PROCEDURAL ] LANGUAGE [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

Description

`DROP LANGUAGE` will remove the definition of the previously registered procedural language called *name*.

Parameters

`IF EXISTS`

Do not throw an error if the function does not exist. A notice is issued in this case.

name

The name of an existing procedural language. For backward compatibility, the name may be enclosed by single quotes.

`CASCADE`

Automatically drop objects that depend on the language (such as functions in the language).

`RESTRICT`

Refuse to drop the language if any objects depend on it. This is the default.

Examples

This command removes the procedural language `plsample`:

```
DROP LANGUAGE plsample;
```

Compatibility

There is no `DROP LANGUAGE` statement in the SQL standard.

See Also

ALTER LANGUAGE, *CREATE LANGUAGE*, droplang

DROP OPERATOR

Name

`DROP OPERATOR` — remove an operator

Synopsis

```
DROP OPERATOR [ IF EXISTS ] name ( { lefttype | NONE } , { righttype | NONE } ) [ CASCADE | R
```

Description

`DROP OPERATOR` drops an existing operator from the database system. To execute this command you must be the owner of the operator.

Parameters

`IF EXISTS`

Do not throw an error if the operator does not exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of an existing operator.

`lefttype`

The data type of the operator's left operand; write `NONE` if the operator has no left operand.

`righttype`

The data type of the operator's right operand; write `NONE` if the operator has no right operand.

`CASCADE`

Automatically drop objects that depend on the operator.

`RESTRICT`

Refuse to drop the operator if any objects depend on it. This is the default.

Examples

Remove the power operator `a^b` for type `integer`:

```
DROP OPERATOR ^ (integer, integer);
```

Remove the left unary bitwise complement operator `~b` for type `bit`:

```
DROP OPERATOR ~ (none, bit);
```

Remove the right unary factorial operator `x!` for type `bigint`:

```
DROP OPERATOR ! (bigint, none);
```

Compatibility

There is no `DROP OPERATOR` statement in the SQL standard.

See Also

CREATE OPERATOR, *ALTER OPERATOR*

DROP OPERATOR CLASS

Name

`DROP OPERATOR CLASS` — remove an operator class

Synopsis

```
DROP OPERATOR CLASS [ IF EXISTS ] name USING index_method [ CASCADE | RESTRICT ]
```

Description

`DROP OPERATOR CLASS` drops an existing operator class. To execute this command you must be the owner of the operator class.

Parameters

`IF EXISTS`

Do not throw an error if the operator class does not exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of an existing operator class.

`index_method`

The name of the index access method the operator class is for.

`CASCADE`

Automatically drop objects that depend on the operator class.

`RESTRICT`

Refuse to drop the operator class if any objects depend on it. This is the default.

Examples

Remove the B-tree operator class `widget_ops`:

```
DROP OPERATOR CLASS widget_ops USING btree;
```

This command will not succeed if there are any existing indexes that use the operator class. Add `CASCADE` to drop such indexes along with the operator class.

Compatibility

There is no `DROP OPERATOR CLASS` statement in the SQL standard.

See Also

ALTER OPERATOR CLASS, *CREATE OPERATOR CLASS*

DROP OWNED

Name

DROP OWNED — remove database objects owned by a database role

Synopsis

```
DROP OWNED BY name [, ...] [ CASCADE | RESTRICT ]
```

Description

DROP OWNED drops all the objects in the current database that are owned by one of the specified roles. Any privileges granted to the given roles on objects in the current database will also be revoked.

Parameters

name

The name of a role whose objects will be dropped, and whose privileges will be revoked.

CASCADE

Automatically drop objects that depend on the affected objects.

RESTRICT

Refuse to drop the objects owned by a role if any other database objects depend on one of the affected objects. This is the default.

Notes

DROP OWNED is often used to prepare for the removal of one or more roles. Because DROP OWNED only affects the objects in the current database, it is usually necessary to execute this command in each database that contains objects owned by a role that is to be removed.

Using the CASCADE option may make the command recurse to objects owned by other users.

The *REASSIGN OWNED* command is an alternative that reassigns the ownership of all the database objects owned by one or more roles.

Compatibility

The DROP OWNED statement is a PostgreSQL extension.

See Also

REASSIGN OWNED, *DROP ROLE*

DROP ROLE

Name

`DROP ROLE` — remove a database role

Synopsis

```
DROP ROLE [ IF EXISTS ] name [, ...]
```

Description

`DROP ROLE` removes the specified role(s). To drop a superuser role, you must be a superuser yourself; to drop non-superuser roles, you must have `CREATEROLE` privilege.

A role cannot be removed if it is still referenced in any database of the cluster; an error will be raised if so. Before dropping the role, you must drop all the objects it owns (or reassign their ownership) and revoke any privileges the role has been granted. The *REASSIGN OWNED* and *DROP OWNED* commands can be useful for this purpose.

However, it is not necessary to remove role memberships involving the role; `DROP ROLE` automatically revokes any memberships of the target role in other roles, and of other roles in the target role. The other roles are not dropped nor otherwise affected.

Parameters

`IF EXISTS`

Do not throw an error if the role does not exist. A notice is issued in this case.

`name`

The name of the role to remove.

Notes

PostgreSQL includes a program *dropuser* that has the same functionality as this command (in fact, it calls this command) but can be run from the command shell.

Examples

To drop a role:

```
DROP ROLE jonathan;
```

Compatibility

The SQL standard defines `DROP ROLE`, but it allows only one role to be dropped at a time, and it specifies different privilege requirements than PostgreSQL uses.

See Also

CREATE ROLE, *ALTER ROLE*, *SET ROLE*

DROP RULE

Name

`DROP RULE` — remove a rewrite rule

Synopsis

```
DROP RULE [ IF EXISTS ] name ON relation [ CASCADE | RESTRICT ]
```

Description

`DROP RULE` drops a rewrite rule.

Parameters

`IF EXISTS`

Do not throw an error if the rule does not exist. A notice is issued in this case.

`name`

The name of the rule to drop.

`relation`

The name (optionally schema-qualified) of the table or view that the rule applies to.

`CASCADE`

Automatically drop objects that depend on the rule.

`RESTRICT`

Refuse to drop the rule if any objects depend on it. This is the default.

Examples

To drop the rewrite rule `newrule`:

```
DROP RULE newrule ON mytable;
```

Compatibility

There is no `DROP RULE` statement in the SQL standard.

See Also

CREATE RULE

DROP SCHEMA

Name

`DROP SCHEMA` — remove a schema

Synopsis

```
DROP SCHEMA [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Description

`DROP SCHEMA` removes schemas from the database.

A schema can only be dropped by its owner or a superuser. Note that the owner can drop the schema (and thereby all contained objects) even if he does not own some of the objects within the schema.

Parameters

`IF EXISTS`

Do not throw an error if the schema does not exist. A notice is issued in this case.

`name`

The name of a schema.

`CASCADE`

Automatically drop objects (tables, functions, etc.) that are contained in the schema.

`RESTRICT`

Refuse to drop the schema if it contains any objects. This is the default.

Examples

To remove schema `mystuff` from the database, along with everything it contains:

```
DROP SCHEMA mystuff CASCADE;
```

Compatibility

`DROP SCHEMA` is fully conforming with the SQL standard, except that the standard only allows one schema to be dropped per command, and apart from the `IF EXISTS` option, which is a PostgreSQL extension.

See Also

ALTER SCHEMA, *CREATE SCHEMA*

DROP SEQUENCE

Name

DROP SEQUENCE — remove a sequence

Synopsis

```
DROP SEQUENCE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Description

DROP SEQUENCE removes sequence number generators.

Parameters

IF EXISTS

Do not throw an error if the sequence does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of a sequence.

CASCADE

Automatically drop objects that depend on the sequence.

RESTRICT

Refuse to drop the sequence if any objects depend on it. This is the default.

Examples

To remove the sequence serial:

```
DROP SEQUENCE serial;
```

Compatibility

DROP SEQUENCE conforms to the SQL standard, except that the standard only allows one sequence to be dropped per command, and apart from the IF EXISTS option, which is a PostgreSQL extension.

See Also

CREATE SEQUENCE, *ALTER SEQUENCE*

DROP TABLE

Name

`DROP TABLE` — remove a table

Synopsis

```
DROP TABLE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Description

`DROP TABLE` removes tables from the database. Only its owner may destroy a table. To empty a table of rows without destroying the table, use *DELETE* or *TRUNCATE*.

`DROP TABLE` always removes any indexes, rules, triggers, and constraints that exist for the target table. However, to drop a table that is referenced by a view or a foreign-key constraint of another table, `CASCADE` must be specified. (`CASCADE` will remove a dependent view entirely, but in the foreign-key case it will only remove the foreign-key constraint, not the other table entirely.)

Parameters

`IF EXISTS`

Do not throw an error if the table does not exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of the table to drop.

`CASCADE`

Automatically drop objects that depend on the table (such as views).

`RESTRICT`

Refuse to drop the table if any objects depend on it. This is the default.

Examples

To destroy two tables, `films` and `distributors`:

```
DROP TABLE films, distributors;
```

Compatibility

This command conforms to the SQL standard, except that the standard only allows one table to be dropped per command, and apart from the `IF EXISTS` option, which is a PostgreSQL extension.

See Also

ALTER TABLE, *CREATE TABLE*

DROP TABLESPACE

Name

DROP TABLESPACE — remove a tablespace

Synopsis

```
DROP TABLESPACE [ IF EXISTS ] tablespacename
```

Description

DROP TABLESPACE removes a tablespace from the system.

A tablespace can only be dropped by its owner or a superuser. The tablespace must be empty of all database objects before it can be dropped. It is possible that objects in other databases may still reside in the tablespace even if no objects in the current database are using the tablespace.

Parameters

IF EXISTS

Do not throw an error if the tablespace does not exist. A notice is issued in this case.

tablespacename

The name of a tablespace.

Notes

DROP TABLESPACE cannot be executed inside a transaction block.

Examples

To remove tablespace mystuff from the system:

```
DROP TABLESPACE mystuff;
```

Compatibility

DROP TABLESPACE is a PostgreSQL extension.

See Also

CREATE TABLESPACE, *ALTER TABLESPACE*

DROP TRIGGER

Name

`DROP TRIGGER` — remove a trigger

Synopsis

```
DROP TRIGGER [ IF EXISTS ] name ON table [ CASCADE | RESTRICT ]
```

Description

`DROP TRIGGER` will remove an existing trigger definition. To execute this command, the current user must be the owner of the table for which the trigger is defined.

Parameters

`IF EXISTS`

Do not throw an error if the trigger does not exist. A notice is issued in this case.

`name`

The name of the trigger to remove.

`table`

The name (optionally schema-qualified) of the table for which the trigger is defined.

`CASCADE`

Automatically drop objects that depend on the trigger.

`RESTRICT`

Refuse to drop the trigger if any objects depend on it. This is the default.

Examples

Destroy the trigger `if_dist_exists` on the table `films`:

```
DROP TRIGGER if_dist_exists ON films;
```

Compatibility

The `DROP TRIGGER` statement in PostgreSQL is incompatible with the SQL standard. In the SQL standard, trigger names are not local to tables, so the command is simply `DROP TRIGGER` *`name`*.

See Also

CREATE TRIGGER

DROP TYPE

Name

DROP TYPE — remove a data type

Synopsis

```
DROP TYPE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Description

DROP TYPE will remove a user-defined data type. Only the owner of a type can remove it.

Parameters

IF EXISTS

Do not throw an error if the type does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of the data type to remove.

CASCADE

Automatically drop objects that depend on the type (such as table columns, functions, operators).

RESTRICT

Refuse to drop the type if any objects depend on it. This is the default.

Examples

To remove the data type box:

```
DROP TYPE box;
```

Compatibility

This command is similar to the corresponding command in the SQL standard, apart from the IF EXISTS option, which is a PostgreSQL extension. But note that the CREATE TYPE command and the data type extension mechanisms in PostgreSQL differ from the SQL standard.

See Also

CREATE TYPE, *ALTER TYPE*

DROP USER

Name

`DROP USER` — remove a database role

Synopsis

```
DROP USER [ IF EXISTS ] name [, ...]
```

Description

`DROP USER` is now an alias for *DROP ROLE*.

Compatibility

The `DROP USER` statement is a PostgreSQL extension. The SQL standard leaves the definition of users to the implementation.

See Also

DROP ROLE

DROP VIEW

Name

DROP VIEW — remove a view

Synopsis

```
DROP VIEW [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Description

DROP VIEW drops an existing view. To execute this command you must be the owner of the view.

Parameters

IF EXISTS

Do not throw an error if the view does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of the view to remove.

CASCADE

Automatically drop objects that depend on the view (such as other views).

RESTRICT

Refuse to drop the view if any objects depend on it. This is the default.

Examples

This command will remove the view called kinds:

```
DROP VIEW kinds;
```

Compatibility

This command conforms to the SQL standard, except that the standard only allows one view to be dropped per command, and apart from the IF EXISTS option, which is a PostgreSQL extension.

See Also

CREATE VIEW

END

Name

END — commit the current transaction

Synopsis

```
END [ WORK | TRANSACTION ]
```

Description

END commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs. This command is a PostgreSQL extension that is equivalent to *COMMIT*.

Parameters

WORK
TRANSACTION

Optional key words. They have no effect.

Notes

Use *ROLLBACK* to abort a transaction.

Issuing END when not inside a transaction does no harm, but it will provoke a warning message.

Examples

To commit the current transaction and make all changes permanent:

```
END;
```

Compatibility

END is a PostgreSQL extension that provides functionality equivalent to *COMMIT*, which is specified in the SQL standard.

See Also

BEGIN, *COMMIT*, *ROLLBACK*

EXECUTE

Name

EXECUTE — execute a prepared statement

Synopsis

```
EXECUTE name [ (parameter [, ...] ) ]
```

Description

EXECUTE is used to execute a previously prepared statement. Since prepared statements only exist for the duration of a session, the prepared statement must have been created by a PREPARE statement executed earlier in the current session.

If the PREPARE statement that created the statement specified some parameters, a compatible set of parameters must be passed to the EXECUTE statement, or else an error is raised. Note that (unlike functions) prepared statements are not overloaded based on the type or number of their parameters; the name of a prepared statement must be unique within a database session.

For more information on the creation and usage of prepared statements, see *PREPARE*.

Parameters

name

The name of the prepared statement to execute.

parameter

The actual value of a parameter to the prepared statement. This must be an expression yielding a value that is compatible with the data type of this parameter, as was determined when the prepared statement was created.

Outputs

The command tag returned by EXECUTE is that of the prepared statement, and not EXECUTE.

Examples

Examples are given in the *Examples* section of the *PREPARE* documentation.

Compatibility

The SQL standard includes an `EXECUTE` statement, but it is only for use in embedded SQL. This version of the `EXECUTE` statement also uses a somewhat different syntax.

See Also

DEALLOCATE, *PREPARE*

EXPLAIN

Name

EXPLAIN — show the execution plan of a statement

Synopsis

```
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
```

Description

This command displays the execution plan that the PostgreSQL planner generates for the supplied statement. The execution plan shows how the table(s) referenced by the statement will be scanned — by plain sequential scan, index scan, etc. — and if multiple tables are referenced, what join algorithms will be used to bring together the required rows from each input table.

The most critical part of the display is the estimated statement execution cost, which is the planner's guess at how long it will take to run the statement (measured in units of disk page fetches). Actually two numbers are shown: the start-up time before the first row can be returned, and the total time to return all the rows. For most queries the total time is what matters, but in contexts such as a subquery in EXISTS, the planner will choose the smallest start-up time instead of the smallest total time (since the executor will stop after getting one row, anyway). Also, if you limit the number of rows to return with a LIMIT clause, the planner makes an appropriate interpolation between the endpoint costs to estimate which plan is really the cheapest.

The ANALYZE option causes the statement to be actually executed, not only planned. The total elapsed time expended within each plan node (in milliseconds) and total number of rows it actually returned are added to the display. This is useful for seeing whether the planner's estimates are close to reality.

> **Important:** Keep in mind that the statement is actually executed when ANALYZE is used. Although EXPLAIN will discard any output that a SELECT would return, other side effects of the statement will happen as usual. If you wish to use EXPLAIN ANALYZE on an INSERT, UPDATE, DELETE, or EXECUTE statement without letting the command affect your data, use this approach:
>
> ```
> BEGIN;
> EXPLAIN ANALYZE ...;
> ROLLBACK;
> ```

Parameters

ANALYZE

Carry out the command and show the actual run times.

VERBOSE

Show the full internal representation of the plan tree, rather than just a summary. Usually this option is only useful for specialized debugging purposes. The VERBOSE output is either pretty-printed or not, depending on the setting of the explain_pretty_print configuration parameter.

statement

Any SELECT, INSERT, UPDATE, DELETE, VALUES, EXECUTE, or DECLARE statement, whose execution plan you wish to see.

Notes

There is only sparse documentation on the optimizer's use of cost information in PostgreSQL. Refer to Section 13.1 for more information.

In order to allow the PostgreSQL query planner to make reasonably informed decisions when optimizing queries, the ANALYZE statement should be run to record statistics about the distribution of data within the table. If you have not done this (or if the statistical distribution of the data in the table has changed significantly since the last time ANALYZE was run), the estimated costs are unlikely to conform to the real properties of the query, and consequently an inferior query plan may be chosen.

Genetic query optimization (GEQO) randomly tests execution plans. Therefore, when the number of tables exceeds geqo_threshold causing genetic query optimization to be used, the execution plan is likely to change each time the statement is executed.

Examples

To show the plan for a simple query on a table with a single integer column and 10000 rows:

```
EXPLAIN SELECT * FROM foo;

                       QUERY PLAN
-------------------------------------------------------
 Seq Scan on foo  (cost=0.00..155.00 rows=10000 width=4)
(1 row)
```

If there is an index and we use a query with an indexable WHERE condition, EXPLAIN might show a different plan:

```
EXPLAIN SELECT * FROM foo WHERE i = 4;

                          QUERY PLAN
-------------------------------------------------------------
```

```
 Index Scan using fi on foo  (cost=0.00..5.98 rows=1 width=4)
   Index Cond: (i = 4)
(2 rows)
```

And here is an example of a query plan for a query using an aggregate function:

```
EXPLAIN SELECT sum(i) FROM foo WHERE i < 10;

                            QUERY PLAN
-------------------------------------------------------------------
 Aggregate  (cost=23.93..23.93 rows=1 width=4)
   ->  Index Scan using fi on foo  (cost=0.00..23.92 rows=6 width=4)
         Index Cond: (i < 10)
(3 rows)
```

Here is an example of using `EXPLAIN EXECUTE` to display the execution plan for a prepared query:

```
PREPARE query(int, int) AS SELECT sum(bar) FROM test
    WHERE id > $1 AND id < $2
    GROUP BY foo;

EXPLAIN ANALYZE EXECUTE query(100, 200);

                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------
 HashAggregate  (cost=39.53..39.53 rows=1 width=8) (actual time=0.661..0.672 rows=7 loops=1
   ->  Index Scan using test_pkey on test  (cost=0.00..32.97 rows=1311 width=8) (actual tim
         Index Cond: ((id > $1) AND (id < $2))
 Total runtime: 0.851 ms
(4 rows)
```

Of course, the specific numbers shown here depend on the actual contents of the tables involved. Also note that the numbers, and even the selected query strategy, may vary between PostgreSQL releases due to planner improvements. In addition, the `ANALYZE` command uses random sampling to estimate data statistics; therefore, it is possible for cost estimates to change after a fresh run of `ANALYZE`, even if the actual distribution of data in the table has not changed.

Compatibility

There is no `EXPLAIN` statement defined in the SQL standard.

See Also

ANALYZE

FETCH

Name

FETCH — retrieve rows from a query using a cursor

Synopsis

```
FETCH [ direction { FROM | IN } ] cursorname

where direction can be empty or one of:

    NEXT
    PRIOR
    FIRST
    LAST
    ABSOLUTE count
    RELATIVE count
    count
    ALL
    FORWARD
    FORWARD count
    FORWARD ALL
    BACKWARD
    BACKWARD count
    BACKWARD ALL
```

Description

FETCH retrieves rows using a previously-created cursor.

A cursor has an associated position, which is used by FETCH. The cursor position can be before the first row of the query result, on any particular row of the result, or after the last row of the result. When created, a cursor is positioned before the first row. After fetching some rows, the cursor is positioned on the row most recently retrieved. If FETCH runs off the end of the available rows then the cursor is left positioned after the last row, or before the first row if fetching backward. FETCH ALL or FETCH BACKWARD ALL will always leave the cursor positioned after the last row or before the first row.

The forms NEXT, PRIOR, FIRST, LAST, ABSOLUTE, RELATIVE fetch a single row after moving the cursor appropriately. If there is no such row, an empty result is returned, and the cursor is left positioned before the first row or after the last row as appropriate.

The forms using FORWARD and BACKWARD retrieve the indicated number of rows moving in the forward or backward direction, leaving the cursor positioned on the last-returned row (or after/before all rows, if the *count* exceeds the number of rows available).

RELATIVE 0, FORWARD 0, and BACKWARD 0 all request fetching the current row without moving the cursor, that is, re-fetching the most recently fetched row. This will succeed unless the cursor is positioned before the first row or after the last row; in which case, no row is returned.

Parameters

direction

direction defines the fetch direction and number of rows to fetch. It can be one of the following:

NEXT

Fetch the next row. This is the default if *direction* is omitted.

PRIOR

Fetch the prior row.

FIRST

Fetch the first row of the query (same as ABSOLUTE 1).

LAST

Fetch the last row of the query (same as ABSOLUTE -1).

ABSOLUTE *count*

Fetch the *count*'th row of the query, or the abs(*count*)'th row from the end if *count* is negative. Position before first row or after last row if *count* is out of range; in particular, ABSOLUTE 0 positions before the first row.

RELATIVE *count*

Fetch the *count*'th succeeding row, or the abs(*count*)'th prior row if *count* is negative. RELATIVE 0 re-fetches the current row, if any.

count

Fetch the next *count* rows (same as FORWARD *count*).

ALL

Fetch all remaining rows (same as FORWARD ALL).

FORWARD

Fetch the next row (same as NEXT).

FORWARD *count*

Fetch the next *count* rows. FORWARD 0 re-fetches the current row.

FORWARD ALL

Fetch all remaining rows.

BACKWARD

Fetch the prior row (same as PRIOR).

BACKWARD *count*

Fetch the prior *count* rows (scanning backwards). BACKWARD 0 re-fetches the current row.

BACKWARD ALL

Fetch all prior rows (scanning backwards).

count

count is a possibly-signed integer constant, determining the location or number of rows to fetch. For FORWARD and BACKWARD cases, specifying a negative *count* is equivalent to changing the sense of FORWARD and BACKWARD.

cursorname

An open cursor's name.

Outputs

On successful completion, a FETCH command returns a command tag of the form

```
FETCH count
```

The *count* is the number of rows fetched (possibly zero). Note that in psql, the command tag will not actually be displayed, since psql displays the fetched rows instead.

Notes

The cursor should be declared with the SCROLL option if one intends to use any variants of FETCH other than FETCH NEXT or FETCH FORWARD with a positive count. For simple queries PostgreSQL will allow backwards fetch from cursors not declared with SCROLL, but this behavior is best not relied on. If the cursor is declared with NO SCROLL, no backward fetches are allowed.

ABSOLUTE fetches are not any faster than navigating to the desired row with a relative move: the underlying implementation must traverse all the intermediate rows anyway. Negative absolute fetches are even worse: the query must be read to the end to find the last row, and then traversed backward from there. However, rewinding to the start of the query (as with FETCH ABSOLUTE 0) is fast.

Updating data via a cursor is currently not supported by PostgreSQL.

DECLARE is used to define a cursor. Use *MOVE* to change cursor position without retrieving data.

Examples

The following example traverses a table using a cursor.

```
BEGIN WORK;

-- Set up a cursor:
DECLARE liahona SCROLL CURSOR FOR SELECT * FROM films;

-- Fetch the first 5 rows in the cursor liahona:
FETCH FORWARD 5 FROM liahona;

 code  |          title          | did | date_prod  |   kind   |  len
-------+-------------------------+-----+------------+----------+-------
 BL101 | The Third Man           | 101 | 1949-12-23 | Drama    | 01:44
```

```
 BL102 | The African Queen       | 101 | 1951-08-11 | Romantic | 01:43
 JL201 | Une Femme est une Femme | 102 | 1961-03-12 | Romantic | 01:25
 P_301 | Vertigo                 | 103 | 1958-11-14 | Action   | 02:08
 P_302 | Becket                  | 103 | 1964-02-03 | Drama    | 02:28

-- Fetch the previous row:
FETCH PRIOR FROM liahona;

 code  |  title  | did | date_prod  |  kind  |  len
-------+---------+-----+------------+--------+-------
 P_301 | Vertigo | 103 | 1958-11-14 | Action | 02:08

-- Close the cursor and end the transaction:
CLOSE liahona;
COMMIT WORK;
```

Compatibility

The SQL standard defines `FETCH` for use in embedded SQL only. The variant of `FETCH` described here returns the data as if it were a `SELECT` result rather than placing it in host variables. Other than this point, `FETCH` is fully upward-compatible with the SQL standard.

The `FETCH` forms involving `FORWARD` and `BACKWARD`, as well as the forms `FETCH` *`count`* and `FETCH ALL`, in which `FORWARD` is implicit, are PostgreSQL extensions.

The SQL standard allows only `FROM` preceding the cursor name; the option to use `IN` is an extension.

See Also

CLOSE, *DECLARE*, *MOVE*

GRANT

Name

GRANT — define access privileges

Synopsis

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | REFERENCES | TRIGGER }
    [,...] | ALL [ PRIVILEGES ] }
    ON [ TABLE ] tablename [, ...]
    TO { username | GROUP groupname | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | SELECT | UPDATE }
    [,...] | ALL [ PRIVILEGES ] }
    ON SEQUENCE sequencename [, ...]
    TO { username | GROUP groupname | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [,...] | ALL [ PRIVILEGES ] }
    ON DATABASE dbname [, ...]
    TO { username | GROUP groupname | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTION funcname ( [ [ argmode ] [ argname ] argtype [, ...] ] ) [, ...]
    TO { username | GROUP groupname | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE langname [, ...]
    TO { username | GROUP groupname | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | USAGE } [,...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schemaname [, ...]
    TO { username | GROUP groupname | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE tablespacename [, ...]
    TO { username | GROUP groupname | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT role [, ...] TO username [, ...] [ WITH ADMIN OPTION ]
```

Description

The GRANT command has two basic variants: one that grants privileges on a database object (table, view, sequence, database, function, procedural language, schema, or tablespace), and one that grants membership in a role. These variants are similar in many ways, but they are different enough to be described separately.

As of PostgreSQL 8.1, the concepts of users and groups have been unified into a single kind of entity called a role. It is therefore no longer necessary to use the keyword GROUP to identify whether a grantee is a user or a group. GROUP is still allowed in the command, but it is a noise word.

GRANT on Database Objects

This variant of the GRANT command gives specific privileges on a database object to one or more roles. These privileges are added to those already granted, if any.

The key word PUBLIC indicates that the privileges are to be granted to all roles, including those that may be created later. PUBLIC may be thought of as an implicitly defined group that always includes all roles. Any particular role will have the sum of privileges granted directly to it, privileges granted to any role it is presently a member of, and privileges granted to PUBLIC.

If WITH GRANT OPTION is specified, the recipient of the privilege may in turn grant it to others. Without a grant option, the recipient cannot do that. Grant options cannot be granted to PUBLIC.

There is no need to grant privileges to the owner of an object (usually the user that created it), as the owner has all privileges by default. (The owner could, however, choose to revoke some of his own privileges for safety.) The right to drop an object, or to alter its definition in any way is not described by a grantable privilege; it is inherent in the owner, and cannot be granted or revoked. The owner implicitly has all grant options for the object, too.

Depending on the type of object, the initial default privileges may include granting some privileges to PUBLIC. The default is no public access for tables, schemas, and tablespaces; CONNECT privilege and TEMP table creation privilege for databases; EXECUTE privilege for functions; and USAGE privilege for languages. The object owner may of course revoke these privileges. (For maximum security, issue the REVOKE in the same transaction that creates the object; then there is no window in which another user may use the object.)

The possible privileges are:

SELECT

Allows *SELECT* from any column of the specified table, view, or sequence. Also allows the use of *COPY* TO. For sequences, this privilege also allows the use of the currval function.

INSERT

Allows *INSERT* of a new row into the specified table. Also allows *COPY* FROM.

UPDATE

Allows *UPDATE* of any column of the specified table. SELECT ... FOR UPDATE and SELECT ... FOR SHARE also require this privilege (besides the SELECT privilege). For sequences, this privilege allows the use of the nextval and setval functions.

DELETE

Allows *DELETE* of a row from the specified table.

REFERENCES

To create a foreign key constraint, it is necessary to have this privilege on both the referencing and referenced tables.

TRIGGER

Allows the creation of a trigger on the specified table. (See the *CREATE TRIGGER* statement.)

CREATE

For databases, allows new schemas to be created within the database.

For schemas, allows new objects to be created within the schema. To rename an existing object, you must own the object *and* have this privilege for the containing schema.

For tablespaces, allows tables and indexes to be created within the tablespace, and allows databases to be created that have the tablespace as their default tablespace. (Note that revoking this privilege will not alter the placement of existing objects.)

CONNECT

Allows the user to connect to the specified database. This privilege is checked at connection startup (in addition to checking any restrictions imposed by `pg_hba.conf`).

TEMPORARY
TEMP

Allows temporary tables to be created while using the database.

EXECUTE

Allows the use of the specified function and the use of any operators that are implemented on top of the function. This is the only type of privilege that is applicable to functions. (This syntax works for aggregate functions, as well.)

USAGE

For procedural languages, allows the use of the specified language for the creation of functions in that language. This is the only type of privilege that is applicable to procedural languages.

For schemas, allows access to objects contained in the specified schema (assuming that the objects' own privilege requirements are also met). Essentially this allows the grantee to "look up" objects within the schema. Without this permission, it is still possible to see the object names, e.g. by querying the system tables. Also, after revoking this permission, existing backends might have statements that have previously performed this lookup, so this is not a completely secure way to prevent object access.

For sequences, this privilege allows the use of the `currval` and `nextval` functions.

ALL PRIVILEGES

Grant all of the available privileges at once. The `PRIVILEGES` key word is optional in PostgreSQL, though it is required by strict SQL.

The privileges required by other commands are listed on the reference page of the respective command.

GRANT on Roles

This variant of the `GRANT` command grants membership in a role to one or more other roles. Membership in a role is significant because it conveys the privileges granted to a role to each of its members.

If `WITH ADMIN OPTION` is specified, the member may in turn grant membership in the role to others, and revoke membership in the role as well. Without the admin option, ordinary users cannot do that. However,

database superusers can grant or revoke membership in any role to anyone. Roles having `CREATEROLE` privilege can grant or revoke membership in any role that is not a superuser.

Unlike the case with privileges, membership in a role cannot be granted to `PUBLIC`. Note also that this form of the command does not allow the noise word `GROUP`.

Notes

The *REVOKE* command is used to revoke access privileges.

When a non-owner of an object attempts to `GRANT` privileges on the object, the command will fail outright if the user has no privileges whatsoever on the object. As long as some privilege is available, the command will proceed, but it will grant only those privileges for which the user has grant options. The `GRANT ALL PRIVILEGES` forms will issue a warning message if no grant options are held, while the other forms will issue a warning if grant options for any of the privileges specifically named in the command are not held. (In principle these statements apply to the object owner as well, but since the owner is always treated as holding all grant options, the cases can never occur.)

It should be noted that database superusers can access all objects regardless of object privilege settings. This is comparable to the rights of `root` in a Unix system. As with `root`, it's unwise to operate as a superuser except when absolutely necessary.

If a superuser chooses to issue a `GRANT` or `REVOKE` command, the command is performed as though it were issued by the owner of the affected object. In particular, privileges granted via such a command will appear to have been granted by the object owner. (For role membership, the membership appears to have been granted by the containing role itself.)

`GRANT` and `REVOKE` can also be done by a role that is not the owner of the affected object, but is a member of the role that owns the object, or is a member of a role that holds privileges `WITH GRANT OPTION` on the object. In this case the privileges will be recorded as having been granted by the role that actually owns the object or holds the privileges `WITH GRANT OPTION`. For example, if table `t1` is owned by role `g1`, of which role `u1` is a member, then `u1` can grant privileges on `t1` to `u2`, but those privileges will appear to have been granted directly by `g1`. Any other member of role `g1` could revoke them later.

If the role executing `GRANT` holds the required privileges indirectly via more than one role membership path, it is unspecified which containing role will be recorded as having done the grant. In such cases it is best practice to use `SET ROLE` to become the specific role you want to do the `GRANT` as.

Granting permission on a table does not automatically extend permissions to any sequences used by the table, including sequences tied to `SERIAL` columns. Permissions on sequence must be set separately.

Currently, PostgreSQL does not support granting or revoking privileges for individual columns of a table. One possible workaround is to create a view having just the desired columns and then grant privileges to that view.

Use psql's `\z` command to obtain information about existing privileges, for example:

```
=> \z mytable

                        Access privileges for database "lusitania"
 Schema |  Name   | Type  |                     Access privileges
--------+---------+-------+-----------------------------------------------------------
```

```
 public | mytable | table | {miriam=arwdxt/miriam,=r/miriam,"group todos=arw/miriam"}
(1 row)
```

The entries shown by `\z` are interpreted thus:

```
              =xxxx -- privileges granted to PUBLIC
         uname=xxxx -- privileges granted to a user
   group gname=xxxx -- privileges granted to a group

                 r -- SELECT ("read")
                 w -- UPDATE ("write")
                 a -- INSERT ("append")
                 d -- DELETE
                 x -- REFERENCES
                 t -- TRIGGER
                 X -- EXECUTE
                 U -- USAGE
                 C -- CREATE
                 c -- CONNECT
                 T -- TEMPORARY
            arwdxt -- ALL PRIVILEGES (for tables)
                 * -- grant option for preceding privilege

             /yyyy -- user who granted this privilege
```

The above example display would be seen by user `miriam` after creating table `mytable` and doing

```
GRANT SELECT ON mytable TO PUBLIC;
GRANT SELECT, UPDATE, INSERT ON mytable TO GROUP todos;
```

If the "Access privileges" column is empty for a given object, it means the object has default privileges (that is, its privileges column is null). Default privileges always include all privileges for the owner, and may include some privileges for `PUBLIC` depending on the object type, as explained above. The first `GRANT` or `REVOKE` on an object will instantiate the default privileges (producing, for example, `{miriam=arwdxt/miriam}`) and then modify them per the specified request.

Notice that the owner's implicit grant options are not marked in the access privileges display. A `*` will appear only when grant options have been explicitly granted to someone.

Examples

Grant insert privilege to all users on table `films`:

```
GRANT INSERT ON films TO PUBLIC;
```

Grant all available privileges to user `manuel` on view `kinds`:

```
GRANT ALL PRIVILEGES ON kinds TO manuel;
```

Note that while the above will indeed grant all privileges if executed by a superuser or the owner of `kinds`, when executed by someone else it will only grant those permissions for which the someone else has grant options.

Grant membership in role `admins` to user `joe`:

```
GRANT admins TO joe;
```

Compatibility

According to the SQL standard, the `PRIVILEGES` key word in `ALL PRIVILEGES` is required. The SQL standard does not support setting the privileges on more than one object per command.

PostgreSQL allows an object owner to revoke his own ordinary privileges: for example, a table owner can make the table read-only to himself by revoking his own INSERT, UPDATE, and DELETE privileges. This is not possible according to the SQL standard. The reason is that PostgreSQL treats the owner's privileges as having been granted by the owner to himself; therefore he can revoke them too. In the SQL standard, the owner's privileges are granted by an assumed entity "_SYSTEM". Not being "_SYSTEM", the owner cannot revoke these rights.

The SQL standard allows setting privileges for individual columns within a table:

```
GRANT privileges
    ON table [ ( column [, ...] ) ] [, ...]
    TO { PUBLIC | username [, ...] } [ WITH GRANT OPTION ]
```

The SQL standard provides for a `USAGE` privilege on other kinds of objects: character sets, collations, translations, domains.

Privileges on databases, tablespaces, schemas, and languages are PostgreSQL extensions.

See Also

REVOKE

INSERT

Name

INSERT — create new rows in a table

Synopsis

```
INSERT INTO table [ ( column [, ...] ) ]
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] | query }
    [ RETURNING * | output_expression [ AS output_name ] [, ...] ]
```

Description

INSERT inserts new rows into a table. One can insert one or more rows specified by value expressions, or zero or more rows resulting from a query.

The target column names may be listed in any order. If no list of column names is given at all, the default is all the columns of the table in their declared order; or the first *N* column names, if there are only *N* columns supplied by the VALUES clause or *query*. The values supplied by the VALUES clause or *query* are associated with the explicit or implicit column list left-to-right.

Each column not present in the explicit or implicit column list will be filled with a default value, either its declared default value or null if there is none.

If the expression for any column is not of the correct data type, automatic type conversion will be attempted.

The optional RETURNING clause causes INSERT to compute and return value(s) based on each row actually inserted. This is primarily useful for obtaining values that were supplied by defaults, such as a serial sequence number. However, any expression using the table's columns is allowed. The syntax of the RETURNING list is identical to that of the output list of SELECT.

You must have INSERT privilege on a table in order to insert into it, and SELECT privilege on it to use RETURNING. If you use the *query* clause to insert rows from a query, you also need to have SELECT privilege on any table used in the query.

Parameters

table

The name (optionally schema-qualified) of an existing table.

column

The name of a column in *table*. The column name can be qualified with a subfield name or array subscript, if needed. (Inserting into only some fields of a composite column leaves the other fields null.)

`DEFAULT VALUES`

All columns will be filled with their default values.

`expression`

An expression or value to assign to the corresponding *`column`*.

`DEFAULT`

The corresponding *`column`* will be filled with its default value.

`query`

A query (`SELECT` statement) that supplies the rows to be inserted. Refer to the *SELECT* statement for a description of the syntax.

`output_expression`

An expression to be computed and returned by the `INSERT` command after each row is inserted. The expression may use any column names of the *`table`*. Write `*` to return all columns of the inserted row(s).

`output_name`

A name to use for a returned column.

Outputs

On successful completion, an `INSERT` command returns a command tag of the form

```
INSERT oid count
```

The *`count`* is the number of rows inserted. If *`count`* is exactly one, and the target table has OIDs, then *`oid`* is the OID assigned to the inserted row. Otherwise *`oid`* is zero.

If the `INSERT` command contains a `RETURNING` clause, the result will be similar to that of a `SELECT` statement containing the columns and values defined in the `RETURNING` list, computed over the row(s) inserted by the command.

Examples

Insert a single row into table `films`:

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, '1971-07-13', 'Comedy', '82 minutes');
```

In this example, the `len` column is omitted and therefore it will have the default value:

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

This example uses the `DEFAULT` clause for the date columns rather than specifying a value:

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes');
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama');
```

To insert a row consisting entirely of default values:

```
INSERT INTO films DEFAULT VALUES;
```

To insert multiple rows using the multirow `VALUES` syntax:

```
INSERT INTO films (code, title, did, date_prod, kind) VALUES
    ('B6717', 'Tampopo', 110, '1985-02-10', 'Comedy'),
    ('HG120', 'The Dinner Game', 140, DEFAULT, 'Comedy');
```

This example inserts some rows into table `films` from a table `tmp_films` with the same column layout as `films`:

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod < '2004-05-07';
```

This example inserts into array columns:

```
-- Create an empty 3x3 gameboard for noughts-and-crosses
INSERT INTO tictactoe (game, board[1:3][1:3])
    VALUES (1, '{{" "," "," "},{" "," "," "},{" "," "," "}}');
-- The subscripts in the above example aren't really needed
INSERT INTO tictactoe (game, board)
    VALUES (2, '{{X," "," "},{" ",O," "},{" ",X," "}}');
```

Insert a single row into table `distributors`, returning the sequence number generated by the `DEFAULT` clause:

```
INSERT INTO distributors (did, dname) VALUES (DEFAULT, 'XYZ Widgets')
   RETURNING did;
```

Compatibility

`INSERT` conforms to the SQL standard, except that the `RETURNING` clause is a PostgreSQL extension. Also, the case in which a column name list is omitted, but not all the columns are filled from the `VALUES` clause or *`query`*, is disallowed by the standard.

Possible limitations of the `query` clause are documented under *SELECT*.

LISTEN

Name

LISTEN — listen for a notification

Synopsis

```
LISTEN name
```

Description

LISTEN registers the current session as a listener on the notification condition *name*. If the current session is already registered as a listener for this notification condition, nothing is done.

Whenever the command NOTIFY *name* is invoked, either by this session or another one connected to the same database, all the sessions currently listening on that notification condition are notified, and each will in turn notify its connected client application. See the discussion of NOTIFY for more information.

A session can be unregistered for a given notify condition with the UNLISTEN command. A session's listen registrations are automatically cleared when the session ends.

The method a client application must use to detect notification events depends on which PostgreSQL application programming interface it uses. With the libpq library, the application issues LISTEN as an ordinary SQL command, and then must periodically call the function PQnotifies to find out whether any notification events have been received. Other interfaces such as libpgtcl provide higher-level methods for handling notify events; indeed, with libpgtcl the application programmer should not even issue LISTEN or UNLISTEN directly. See the documentation for the interface you are using for more details.

NOTIFY contains a more extensive discussion of the use of LISTEN and NOTIFY.

Parameters

name

Name of a notify condition (any identifier).

Examples

Configure and execute a listen/notify sequence from psql:

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
```

Compatibility

There is no `LISTEN` statement in the SQL standard.

See Also

NOTIFY, *UNLISTEN*

LOAD

Name

LOAD — load or reload a shared library file

Synopsis

```
LOAD 'filename'
```

Description

This command loads a shared library file into the PostgreSQL server's address space. If the file had been loaded previously, it is first unloaded. This command is primarily useful to unload and reload a shared library file that has been changed since the server first loaded it. To make use of the shared library, function(s) in it need to be declared using the *CREATE FUNCTION* command.

The file name is specified in the same way as for shared library names in *CREATE FUNCTION*; in particular, one may rely on a search path and automatic addition of the system's standard shared library file name extension. See Section 33.9 for more information on this topic.

Non-superusers may only apply LOAD to library files located in `$libdir/plugins/` — the specified *filename* must begin with exactly that string. (It is the database administrator's responsibility to ensure that only "safe" libraries are installed there.)

Compatibility

LOAD is a PostgreSQL extension.

See Also

CREATE FUNCTION

LOCK

Name

LOCK — lock a table

Synopsis

```
LOCK [ TABLE ] name [, ...] [ IN lockmode MODE ] [ NOWAIT ]

where lockmode is one of:

    ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
    | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

Description

LOCK TABLE obtains a table-level lock, waiting if necessary for any conflicting locks to be released. If NOWAIT is specified, LOCK TABLE does not wait to acquire the desired lock: if it cannot be acquired immediately, the command is aborted and an error is emitted. Once obtained, the lock is held for the remainder of the current transaction. (There is no UNLOCK TABLE command; locks are always released at transaction end.)

When acquiring locks automatically for commands that reference tables, PostgreSQL always uses the least restrictive lock mode possible. LOCK TABLE provides for cases when you might need more restrictive locking. For example, suppose an application runs a transaction at the Read Committed isolation level and needs to ensure that data in a table remains stable for the duration of the transaction. To achieve this you could obtain SHARE lock mode over the table before querying. This will prevent concurrent data changes and ensure subsequent reads of the table see a stable view of committed data, because SHARE lock mode conflicts with the ROW EXCLUSIVE lock acquired by writers, and your LOCK TABLE *name* IN SHARE MODE statement will wait until any concurrent holders of ROW EXCLUSIVE mode locks commit or roll back. Thus, once you obtain the lock, there are no uncommitted writes outstanding; furthermore none can begin until you release the lock.

To achieve a similar effect when running a transaction at the Serializable isolation level, you have to execute the LOCK TABLE statement before executing any SELECT or data modification statement. A serializable transaction's view of data will be frozen when its first SELECT or data modification statement begins. A LOCK TABLE later in the transaction will still prevent concurrent writes — but it won't ensure that what the transaction reads corresponds to the latest committed values.

If a transaction of this sort is going to change the data in the table, then it should use SHARE ROW EXCLUSIVE lock mode instead of SHARE mode. This ensures that only one transaction of this type runs at a time. Without this, a deadlock is possible: two transactions might both acquire SHARE mode, and then be unable to also acquire ROW EXCLUSIVE mode to actually perform their updates. (Note that a transaction's own locks never conflict, so a transaction can acquire ROW EXCLUSIVE mode when it holds SHARE mode — but not if anyone else holds SHARE mode.) To avoid deadlocks, make sure all transactions acquire locks

on the same objects in the same order, and if multiple lock modes are involved for a single object, then transactions should always acquire the most restrictive mode first.

More information about the lock modes and locking strategies can be found in Section 12.3.

Parameters

name

The name (optionally schema-qualified) of an existing table to lock.

The command `LOCK TABLE a, b;` is equivalent to `LOCK TABLE a; LOCK TABLE b;`. The tables are locked one-by-one in the order specified in the `LOCK TABLE` command.

lockmode

The lock mode specifies which locks this lock conflicts with. Lock modes are described in Section 12.3.

If no lock mode is specified, then `ACCESS EXCLUSIVE`, the most restrictive mode, is used.

`NOWAIT`

Specifies that `LOCK TABLE` should not wait for any conflicting locks to be released: if the specified lock(s) cannot be acquired immediately without waiting, the transaction is aborted.

Notes

`LOCK TABLE ... IN ACCESS SHARE MODE` requires `SELECT` privileges on the target table. All other forms of `LOCK` require `UPDATE` and/or `DELETE` privileges.

`LOCK TABLE` is useful only inside a transaction block (`BEGIN`/`COMMIT` pair), since the lock is dropped as soon as the transaction ends. A `LOCK TABLE` command appearing outside any transaction block forms a self-contained transaction, so the lock will be dropped as soon as it is obtained.

`LOCK TABLE` only deals with table-level locks, and so the mode names involving `ROW` are all misnomers. These mode names should generally be read as indicating the intention of the user to acquire row-level locks within the locked table. Also, `ROW EXCLUSIVE` mode is a sharable table lock. Keep in mind that all the lock modes have identical semantics so far as `LOCK TABLE` is concerned, differing only in the rules about which modes conflict with which. For information on how to acquire an actual row-level lock, see Section 12.3.2 and the *FOR UPDATE/FOR SHARE Clause* in the `SELECT` reference documentation.

Examples

Obtain a `SHARE` lock on a primary key table when going to perform inserts into a foreign key table:

```
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
    WHERE name = 'Star Wars: Episode I - The Phantom Menace';
-- Do ROLLBACK if record was not returned
```

```
INSERT INTO films_user_comments VALUES
    (_id_, 'GREAT! I was waiting for it for so long!');
COMMIT WORK;
```

Take a `SHARE ROW EXCLUSIVE` lock on a primary key table when going to perform a delete operation:

```
BEGIN WORK;
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM films_user_comments WHERE id IN
    (SELECT id FROM films WHERE rating < 5);
DELETE FROM films WHERE rating < 5;
COMMIT WORK;
```

Compatibility

There is no `LOCK TABLE` in the SQL standard, which instead uses `SET TRANSACTION` to specify concurrency levels on transactions. PostgreSQL supports that too; see *SET TRANSACTION* for details.

Except for `ACCESS SHARE`, `ACCESS EXCLUSIVE`, and `SHARE UPDATE EXCLUSIVE` lock modes, the PostgreSQL lock modes and the `LOCK TABLE` syntax are compatible with those present in Oracle.

MOVE

Name

MOVE — position a cursor

Synopsis

```
MOVE [ direction { FROM | IN } ] cursorname
```

Description

MOVE repositions a cursor without retrieving any data. MOVE works exactly like the FETCH command, except it only positions the cursor and does not return rows.

Refer to *FETCH* for details on syntax and usage.

Outputs

On successful completion, a MOVE command returns a command tag of the form

```
MOVE count
```

The *count* is the number of rows that a FETCH command with the same parameters would have returned (possibly zero).

Examples

```
BEGIN WORK;
DECLARE liahona CURSOR FOR SELECT * FROM films;

-- Skip the first 5 rows:
MOVE FORWARD 5 IN liahona;
MOVE 5

-- Fetch the 6th row from the cursor liahona:
FETCH 1 FROM liahona;
 code  | title  | did | date_prod  |  kind  |  len
-------+--------+-----+------------+--------+-------
 P_303 | 48 Hrs | 103 | 1982-10-22 | Action | 01:37
(1 row)

-- Close the cursor liahona and end the transaction:
CLOSE liahona;
COMMIT WORK;
```

Compatibility

There is no `MOVE` statement in the SQL standard.

See Also

CLOSE, *DECLARE*, *FETCH*

NOTIFY

Name

NOTIFY — generate a notification

Synopsis

```
NOTIFY name
```

Description

The NOTIFY command sends a notification event to each client application that has previously executed LISTEN *name* for the specified notification name in the current database.

NOTIFY provides a simple form of signal or interprocess communication mechanism for a collection of processes accessing the same PostgreSQL database. Higher-level mechanisms can be built by using tables in the database to pass additional data (beyond a mere notification name) from notifier to listener(s).

The information passed to the client for a notification event includes the notification name and the notifying session's server process PID. It is up to the database designer to define the notification names that will be used in a given database and what each one means.

Commonly, the notification name is the same as the name of some table in the database, and the notify event essentially means, "I changed this table, take a look at it to see what's new". But no such association is enforced by the NOTIFY and LISTEN commands. For example, a database designer could use several different notification names to signal different sorts of changes to a single table.

When NOTIFY is used to signal the occurrence of changes to a particular table, a useful programming technique is to put the NOTIFY in a rule that is triggered by table updates. In this way, notification happens automatically when the table is changed, and the application programmer can't accidentally forget to do it.

NOTIFY interacts with SQL transactions in some important ways. Firstly, if a NOTIFY is executed inside a transaction, the notify events are not delivered until and unless the transaction is committed. This is appropriate, since if the transaction is aborted, all the commands within it have had no effect, including NOTIFY. But it can be disconcerting if one is expecting the notification events to be delivered immediately. Secondly, if a listening session receives a notification signal while it is within a transaction, the notification event will not be delivered to its connected client until just after the transaction is completed (either committed or aborted). Again, the reasoning is that if a notification were delivered within a transaction that was later aborted, one would want the notification to be undone somehow — but the server cannot "take back" a notification once it has sent it to the client. So notification events are only delivered between transactions. The upshot of this is that applications using NOTIFY for real-time signaling should try to keep their transactions short.

NOTIFY behaves like Unix signals in one important respect: if the same notification name is signaled multiple times in quick succession, recipients may get only one notification event for several executions of NOTIFY. So it is a bad idea to depend on the number of notifications received. Instead, use NOTIFY

to wake up applications that need to pay attention to something, and use a database object (such as a sequence) to keep track of what happened or how many times it happened.

It is common for a client that executes `NOTIFY` to be listening on the same notification name itself. In that case it will get back a notification event, just like all the other listening sessions. Depending on the application logic, this could result in useless work, for example, reading a database table to find the same updates that that session just wrote out. It is possible to avoid such extra work by noticing whether the notifying session's server process PID (supplied in the notification event message) is the same as one's own session's PID (available from libpq). When they are the same, the notification event is one's own work bouncing back, and can be ignored. (Despite what was said in the preceding paragraph, this is a safe technique. PostgreSQL keeps self-notifications separate from notifications arriving from other sessions, so you cannot miss an outside notification by ignoring your own notifications.)

Parameters

name

Name of the notification to be signaled (any identifier).

Examples

Configure and execute a listen/notify sequence from psql:

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
```

Compatibility

There is no `NOTIFY` statement in the SQL standard.

See Also

LISTEN, *UNLISTEN*

PREPARE

Name

PREPARE — prepare a statement for execution

Synopsis

```
PREPARE name [ (datatype [, ...] ) ] AS statement
```

Description

PREPARE creates a prepared statement. A prepared statement is a server-side object that can be used to optimize performance. When the PREPARE statement is executed, the specified statement is parsed, rewritten, and planned. When an EXECUTE command is subsequently issued, the prepared statement need only be executed. Thus, the parsing, rewriting, and planning stages are only performed once, instead of every time the statement is executed.

Prepared statements can take parameters: values that are substituted into the statement when it is executed. When creating the prepared statement, refer to parameters by position, using $1, $2, etc. A corresponding list of parameter data types can optionally be specified. When a parameter's data type is not specified or is declared as unknown, the type is inferred from the context in which the parameter is used (if possible). When executing the statement, specify the actual values for these parameters in the EXECUTE statement. Refer to *EXECUTE* for more information about that.

Prepared statements only last for the duration of the current database session. When the session ends, the prepared statement is forgotten, so it must be recreated before being used again. This also means that a single prepared statement cannot be used by multiple simultaneous database clients; however, each client can create their own prepared statement to use. The prepared statement can be manually cleaned up using the *DEALLOCATE* command.

Prepared statements have the largest performance advantage when a single session is being used to execute a large number of similar statements. The performance difference will be particularly significant if the statements are complex to plan or rewrite, for example, if the query involves a join of many tables or requires the application of several rules. If the statement is relatively simple to plan and rewrite but relatively expensive to execute, the performance advantage of prepared statements will be less noticeable.

Parameters

name

An arbitrary name given to this particular prepared statement. It must be unique within a single session and is subsequently used to execute or deallocate a previously prepared statement.

datatype

The data type of a parameter to the prepared statement. If the data type of a particular parameter is unspecified or is specified as `unknown`, it will be inferred from the context in which the parameter is used. To refer to the parameters in the prepared statement itself, use `$1`, `$2`, etc.

statement

Any `SELECT`, `INSERT`, `UPDATE`, `DELETE`, or `VALUES` statement.

Notes

In some situations, the query plan produced for a prepared statement will be inferior to the query plan that would have been chosen if the statement had been submitted and executed normally. This is because when the statement is planned and the planner attempts to determine the optimal query plan, the actual values of any parameters specified in the statement are unavailable. PostgreSQL collects statistics on the distribution of data in the table, and can use constant values in a statement to make guesses about the likely result of executing the statement. Since this data is unavailable when planning prepared statements with parameters, the chosen plan may be suboptimal. To examine the query plan PostgreSQL has chosen for a prepared statement, use *EXPLAIN*.

For more information on query planning and the statistics collected by PostgreSQL for that purpose, see the *ANALYZE* documentation.

You can see all available prepared statements of a session by querying the `pg_prepared_statements` system view.

Examples

Create a prepared statement for an `INSERT` statement, and then execute it:

```
PREPARE fooplan (int, text, bool, numeric) AS
    INSERT INTO foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

Create a prepared statement for a `SELECT` statement, and then execute it:

```
PREPARE usrrptplan (int) AS
    SELECT * FROM users u, logs l WHERE u.usrid=$1 AND u.usrid=l.usrid
    AND l.date = $2;
EXECUTE usrrptplan(1, current_date);
```

Note that the data type of the second parameter is not specified, so it is inferred from the context in which `$2` is used.

Compatibility

The SQL standard includes a `PREPARE` statement, but it is only for use in embedded SQL. This version of the `PREPARE` statement also uses a somewhat different syntax.

See Also

DEALLOCATE, *EXECUTE*

PREPARE TRANSACTION

Name

PREPARE TRANSACTION — prepare the current transaction for two-phase commit

Synopsis

```
PREPARE TRANSACTION transaction_id
```

Description

PREPARE TRANSACTION prepares the current transaction for two-phase commit. After this command, the transaction is no longer associated with the current session; instead, its state is fully stored on disk, and there is a very high probability that it can be committed successfully, even if a database crash occurs before the commit is requested.

Once prepared, a transaction can later be committed or rolled back with *COMMIT PREPARED* or *ROLLBACK PREPARED*, respectively. Those commands can be issued from any session, not only the one that executed the original transaction.

From the point of view of the issuing session, PREPARE TRANSACTION is not unlike a ROLLBACK command: after executing it, there is no active current transaction, and the effects of the prepared transaction are no longer visible. (The effects will become visible again if the transaction is committed.)

If the PREPARE TRANSACTION command fails for any reason, it becomes a ROLLBACK: the current transaction is canceled.

Parameters

transaction_id

An arbitrary identifier that later identifies this transaction for COMMIT PREPARED or ROLLBACK PREPARED. The identifier must be written as a string literal, and must be less than 200 bytes long. It must not be the same as the identifier used for any currently prepared transaction.

Notes

This command must be used inside a transaction block. Use *BEGIN* to start one.

It is not currently allowed to PREPARE a transaction that has executed any operations involving temporary tables or created any cursors WITH HOLD. Those features are too tightly tied to the current session to be useful in a transaction to be prepared.

If the transaction modified any run-time parameters with `SET`, those effects persist after `PREPARE TRANSACTION`, and will not be affected by any later `COMMIT PREPARED` or `ROLLBACK PREPARED`. Thus, in this one respect `PREPARE TRANSACTION` acts more like `COMMIT` than `ROLLBACK`.

All currently available prepared transactions are listed in the `pg_prepared_xacts` system view.

From a performance standpoint, it is unwise to leave transactions in the prepared state for a long time: this will for instance interfere with the ability of `VACUUM` to reclaim storage. Keep in mind also that the transaction continues to hold whatever locks it held. The intended usage of the feature is that a prepared transaction will normally be committed or rolled back as soon as an external transaction manager has verified that other databases are also prepared to commit.

If you make any serious use of prepared transactions, you will probably want to increase the value of max_prepared_transactions, as the default setting is quite small (to avoid wasting resources for those who don't use it). It is recommendable to make it at least equal to max_connections, so that every session can have a prepared transaction pending.

Examples

Prepare the current transaction for two-phase commit, using `foobar` as the transaction identifier:

```
PREPARE TRANSACTION 'foobar';
```

See Also

COMMIT PREPARED, *ROLLBACK PREPARED*

REASSIGN OWNED

Name

REASSIGN OWNED — change the ownership of database objects owned by a database role

Synopsis

```
REASSIGN OWNED BY old_role [, ...] TO new_role
```

Description

REASSIGN OWNED instructs the system to change the ownership of the database objects owned by one of the old_roles, to new_role.

Parameters

old_role

The name of a role. The ownership of all the objects in the current database owned by this role will be reassigned to *new_role*.

new_role

The name of the role that will be made the new owner of the affected objects.

Notes

REASSIGN OWNED is often used to prepare for the removal of one or more roles. Because REASSIGN OWNED only affects the objects in the current database, it is usually necessary to execute this command in each database that contains objects owned by a role that is to be removed.

The *DROP OWNED* command is an alternative that drops all the database objects owned by one or more roles.

The REASSIGN OWNED command does not affect the privileges granted to the old_roles in objects that are not owned by them. Use DROP OWNED to revoke those privileges.

Compatibility

The REASSIGN OWNED statement is a PostgreSQL extension.

See Also

DROP OWNED, *DROP ROLE*

REINDEX

Name

REINDEX — rebuild indexes

Synopsis

```
REINDEX { INDEX | TABLE | DATABASE | SYSTEM } name [ FORCE ]
```

Description

REINDEX rebuilds an index using the data stored in the index's table, replacing the old copy of the index. There are several scenarios in which to use REINDEX:

- An index has become corrupted, and no longer contains valid data. Although in theory this should never happen, in practice indexes may become corrupted due to software bugs or hardware failures. REINDEX provides a recovery method.
- An index has become "bloated", that it is contains many empty or nearly-empty pages. This can occur with B-tree indexes in PostgreSQL under certain uncommon access patterns. REINDEX provides a way to reduce the space consumption of the index by writing a new version of the index without the dead pages. See Section 22.2 for more information.
- You have altered a storage parameter (such as fillfactor) for an index, and wish to ensure that the change has taken full effect.
- An index build with the CONCURRENTLY option failed, leaving an "invalid" index. Such indexes are useless but it can be convenient to use REINDEX to rebuild them. Note that REINDEX will not perform a concurrent build. To build the index without interfering with production you should drop the index and reissue the CREATE INDEX CONCURRENTLY command.

Parameters

INDEX

Recreate the specified index.

TABLE

Recreate all indexes of the specified table. If the table has a secondary "TOAST" table, that is reindexed as well.

DATABASE

Recreate all indexes within the current database. Indexes on shared system catalogs are skipped except in stand-alone mode (see below). This form of REINDEX cannot be executed inside a transaction block.

SYSTEM

Recreate all indexes on system catalogs within the current database. Indexes on user tables are not processed. Also, indexes on shared system catalogs are skipped except in stand-alone mode (see below). This form of REINDEX cannot be executed inside a transaction block.

name

The name of the specific index, table, or database to be reindexed. Index and table names may be schema-qualified. Presently, REINDEX DATABASE and REINDEX SYSTEM can only reindex the current database, so their parameter must match the current database's name.

FORCE

This is an obsolete option; it is ignored if specified.

Notes

If you suspect corruption of an index on a user table, you can simply rebuild that index, or all indexes on the table, using REINDEX INDEX or REINDEX TABLE.

Things are more difficult if you need to recover from corruption of an index on a system table. In this case it's important for the system to not have used any of the suspect indexes itself. (Indeed, in this sort of scenario you may find that server processes are crashing immediately at start-up, due to reliance on the corrupted indexes.) To recover safely, the server must be started with the -P option, which prevents it from using indexes for system catalog lookups.

One way to do this is to shut down the server and start a single-user PostgreSQL server with the -P option included on its command line. Then, REINDEX DATABASE, REINDEX SYSTEM, REINDEX TABLE, or REINDEX INDEX can be issued, depending on how much you want to reconstruct. If in doubt, use REINDEX SYSTEM to select reconstruction of all system indexes in the database. Then quit the single-user server session and restart the regular server. See the postgres reference page for more information about how to interact with the single-user server interface.

Alternatively, a regular server session can be started with -P included in its command line options. The method for doing this varies across clients, but in all libpq-based clients, it is possible to set the PGOPTIONS environment variable to -P before starting the client. Note that while this method does not require locking out other clients, it may still be wise to prevent other users from connecting to the damaged database until repairs have been completed.

If corruption is suspected in the indexes of any of the shared system catalogs (which are pg_authid, pg_auth_members, pg_database, pg_pltemplate, pg_shdepend, pg_shdescription, and pg_tablespace), then a standalone server must be used to repair it. REINDEX will not process shared catalogs in multiuser mode.

For all indexes except the shared system catalogs, REINDEX is crash-safe and transaction-safe. REINDEX is not crash-safe for shared indexes, which is why this case is disallowed during normal operation. If a failure occurs while reindexing one of these catalogs in standalone mode, it will not be possible to restart

the regular server until the problem is rectified. (The typical symptom of a partially rebuilt shared index is "index is not a btree" errors.)

`REINDEX` is similar to a drop and recreate of the index in that the index contents are rebuilt from scratch. However, the locking considerations are rather different. `REINDEX` locks out writes but not reads of the index's parent table. It also takes an exclusive lock on the specific index being processed, which will block reads that attempt to use that index. In contrast, `DROP INDEX` momentarily takes exclusive lock on the parent table, blocking both writes and reads. The subsequent `CREATE INDEX` locks out writes but not reads; since the index is not there, no read will attempt to use it, meaning that there will be no blocking but reads may be forced into expensive sequential scans. Another important point is that the drop/create approach invalidates any cached query plans that use the index, while `REINDEX` does not.

Reindexing a single index or table requires being the owner of that index or table. Reindexing a database requires being the owner of the database (note that the owner can therefore rebuild indexes of tables owned by other users). Of course, superusers can always reindex anything.

Prior to PostgreSQL 8.1, `REINDEX DATABASE` processed only system indexes, not all indexes as one would expect from the name. This has been changed to reduce the surprise factor. The old behavior is available as `REINDEX SYSTEM`.

Prior to PostgreSQL 7.4, `REINDEX TABLE` did not automatically process TOAST tables, and so those had to be reindexed by separate commands. This is still possible, but redundant.

Examples

Rebuild a single index:

```
REINDEX INDEX my_index;
```

Rebuild all the indexes on the table `my_table`:

```
REINDEX TABLE my_table;
```

Rebuild all indexes in a particular database, without trusting the system indexes to be valid already:

```
$ export PGOPTIONS="-P"
$ psql broken_db
...
broken_db=> REINDEX DATABASE broken_db;
broken_db=> \q
```

Compatibility

There is no `REINDEX` command in the SQL standard.

RELEASE SAVEPOINT

Name

`RELEASE SAVEPOINT` — destroy a previously defined savepoint

Synopsis

```
RELEASE [ SAVEPOINT ] savepoint_name
```

Description

`RELEASE SAVEPOINT` destroys a savepoint previously defined in the current transaction.

Destroying a savepoint makes it unavailable as a rollback point, but it has no other user visible behavior. It does not undo the effects of commands executed after the savepoint was established. (To do that, see *ROLLBACK TO SAVEPOINT*.) Destroying a savepoint when it is no longer needed may allow the system to reclaim some resources earlier than transaction end.

`RELEASE SAVEPOINT` also destroys all savepoints that were established after the named savepoint was established.

Parameters

savepoint_name

The name of the savepoint to destroy.

Notes

Specifying a savepoint name that was not previously defined is an error.

It is not possible to release a savepoint when the transaction is in an aborted state.

If multiple savepoints have the same name, only the one that was most recently defined is released.

Examples

To establish and later destroy a savepoint:

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
```

```
COMMIT;
```

The above transaction will insert both 3 and 4.

Compatibility

This command conforms to the SQL standard. The standard specifies that the key word `SAVEPOINT` is mandatory, but PostgreSQL allows it to be omitted.

See Also

BEGIN, *COMMIT*, *ROLLBACK*, *ROLLBACK TO SAVEPOINT*, *SAVEPOINT*

RESET

Name

RESET — restore the value of a run-time parameter to the default value

Synopsis

```
RESET configuration_parameter
RESET ALL
```

Description

RESET restores run-time parameters to their default values. RESET is an alternative spelling for

```
SET configuration_parameter TO DEFAULT
```

Refer to *SET* for details.

The default value is defined as the value that the parameter would have had, had no SET ever been issued for it in the current session. The actual source of this value might be a compiled-in default, the configuration file, command-line options, or per-database or per-user default settings. See Chapter 17 for details.

See the SET reference page for details on the transaction behavior of RESET.

Parameters

configuration_parameter

The name of a run-time parameter. See *SET* for a list.

ALL

Resets all settable run-time parameters to default values.

Examples

Set the geqo configuration variable to its default value:

```
RESET geqo;
```

Compatibility

`RESET` is a PostgreSQL extension.

REVOKE

Name

REVOKE — remove access privileges

Synopsis

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | REFERENCES | TRIGGER }
    [,...] | ALL [ PRIVILEGES ] }
    ON [ TABLE ] tablename [, ...]
    FROM { username | GROUP groupname | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
    [,...] | ALL [ PRIVILEGES ] }
    ON SEQUENCE sequencename [, ...]
    FROM { username | GROUP groupname | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | CONNECT | TEMPORARY | TEMP } [,...] | ALL [ PRIVILEGES ] }
    ON DATABASE dbname [, ...]
    FROM { username | GROUP groupname | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTION funcname ( [ [ argmode ] [ argname ] argtype [, ...] ] ) [, ...]
    FROM { username | GROUP groupname | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE langname [, ...]
    FROM { username | GROUP groupname | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | USAGE } [,...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schemaname [, ...]
    FROM { username | GROUP groupname | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE tablespacename [, ...]
    FROM { username | GROUP groupname | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ ADMIN OPTION FOR ]
    role [, ...] FROM username [, ...]
    [ CASCADE | RESTRICT ]
```

Description

The `REVOKE` command revokes previously granted privileges from one or more roles. The key word `PUBLIC` refers to the implicitly defined group of all roles.

See the description of the *GRANT* command for the meaning of the privilege types.

Note that any particular role will have the sum of privileges granted directly to it, privileges granted to any role it is presently a member of, and privileges granted to `PUBLIC`. Thus, for example, revoking `SELECT` privilege from `PUBLIC` does not necessarily mean that all roles have lost `SELECT` privilege on the object: those who have it granted directly or via another role will still have it.

If `GRANT OPTION FOR` is specified, only the grant option for the privilege is revoked, not the privilege itself. Otherwise, both the privilege and the grant option are revoked.

If a user holds a privilege with grant option and has granted it to other users then the privileges held by those other users are called dependent privileges. If the privilege or the grant option held by the first user is being revoked and dependent privileges exist, those dependent privileges are also revoked if `CASCADE` is specified, else the revoke action will fail. This recursive revocation only affects privileges that were granted through a chain of users that is traceable to the user that is the subject of this `REVOKE` command. Thus, the affected users may effectively keep the privilege if it was also granted through other users.

When revoking membership in a role, `GRANT OPTION` is instead called `ADMIN OPTION`, but the behavior is similar. Note also that this form of the command does not allow the noise word `GROUP`.

Notes

Use psql's `\z` command to display the privileges granted on existing objects. See *GRANT* for information about the format.

A user can only revoke privileges that were granted directly by that user. If, for example, user A has granted a privilege with grant option to user B, and user B has in turned granted it to user C, then user A cannot revoke the privilege directly from C. Instead, user A could revoke the grant option from user B and use the `CASCADE` option so that the privilege is in turn revoked from user C. For another example, if both A and B have granted the same privilege to C, A can revoke his own grant but not B's grant, so C will still effectively have the privilege.

When a non-owner of an object attempts to `REVOKE` privileges on the object, the command will fail outright if the user has no privileges whatsoever on the object. As long as some privilege is available, the command will proceed, but it will revoke only those privileges for which the user has grant options. The `REVOKE ALL PRIVILEGES` forms will issue a warning message if no grant options are held, while the other forms will issue a warning if grant options for any of the privileges specifically named in the command are not held. (In principle these statements apply to the object owner as well, but since the owner is always treated as holding all grant options, the cases can never occur.)

If a superuser chooses to issue a `GRANT` or `REVOKE` command, the command is performed as though it were issued by the owner of the affected object. Since all privileges ultimately come from the object owner (possibly indirectly via chains of grant options), it is possible for a superuser to revoke all privileges, but this may require use of `CASCADE` as stated above.

`REVOKE` can also be done by a role that is not the owner of the affected object, but is a member of the role that owns the object, or is a member of a role that holds privileges `WITH GRANT OPTION` on the object. In this case the command is performed as though it were issued by the containing role that actually owns the object or holds the privileges `WITH GRANT OPTION`. For example, if table `t1` is owned by role `g1`, of which role `u1` is a member, then `u1` can revoke privileges on `t1` that are recorded as being granted by `g1`. This would include grants made by `u1` as well as by other members of role `g1`.

If the role executing `REVOKE` holds privileges indirectly via more than one role membership path, it is unspecified which containing role will be used to perform the command. In such cases it is best practice to use `SET ROLE` to become the specific role you want to do the `REVOKE` as. Failure to do so may lead to revoking privileges other than the ones you intended, or not revoking anything at all.

Examples

Revoke insert privilege for the public on table `films`:

```
REVOKE INSERT ON films FROM PUBLIC;
```

Revoke all privileges from user `manuel` on view `kinds`:

```
REVOKE ALL PRIVILEGES ON kinds FROM manuel;
```

Note that this actually means “revoke all privileges that I granted”.

Revoke membership in role `admins` from user `joe`:

```
REVOKE admins FROM joe;
```

Compatibility

The compatibility notes of the *GRANT* command apply analogously to `REVOKE`. The syntax summary is:

```
REVOKE [ GRANT OPTION FOR ] privileges
    ON object [ ( column [, ...] ) ]
    FROM { PUBLIC | username [, ...] }
    { RESTRICT | CASCADE }
```

One of `RESTRICT` or `CASCADE` is required according to the standard, but PostgreSQL assumes `RESTRICT` by default.

See Also

GRANT

ROLLBACK

Name

ROLLBACK — abort the current transaction

Synopsis

```
ROLLBACK [ WORK | TRANSACTION ]
```

Description

ROLLBACK rolls back the current transaction and causes all the updates made by the transaction to be discarded.

Parameters

WORK
TRANSACTION

Optional key words. They have no effect.

Notes

Use *COMMIT* to successfully terminate a transaction.

Issuing ROLLBACK when not inside a transaction does no harm, but it will provoke a warning message.

Examples

To abort all changes:

```
ROLLBACK;
```

Compatibility

The SQL standard only specifies the two forms ROLLBACK and ROLLBACK WORK. Otherwise, this command is fully conforming.

See Also

BEGIN, *COMMIT*, *ROLLBACK TO SAVEPOINT*

ROLLBACK PREPARED

Name

`ROLLBACK PREPARED` — cancel a transaction that was earlier prepared for two-phase commit

Synopsis

```
ROLLBACK PREPARED transaction_id
```

Description

`ROLLBACK PREPARED` rolls back a transaction that is in prepared state.

Parameters

`transaction_id`

The transaction identifier of the transaction that is to be rolled back.

Notes

To roll back a prepared transaction, you must be either the same user that executed the transaction originally, or a superuser. But you do not have to be in the same session that executed the transaction.

This command cannot be executed inside a transaction block. The prepared transaction is rolled back immediately.

All currently available prepared transactions are listed in the `pg_prepared_xacts` system view.

Examples

Roll back the transaction identified by the transaction identifier `foobar`:

```
ROLLBACK PREPARED 'foobar';
```

See Also

PREPARE TRANSACTION, *COMMIT PREPARED*

ROLLBACK TO SAVEPOINT

Name

`ROLLBACK TO SAVEPOINT` — roll back to a savepoint

Synopsis

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] savepoint_name
```

Description

Roll back all commands that were executed after the savepoint was established. The savepoint remains valid and can be rolled back to again later, if needed.

`ROLLBACK TO SAVEPOINT` implicitly destroys all savepoints that were established after the named savepoint.

Parameters

`savepoint_name`

The savepoint to roll back to.

Notes

Use *RELEASE SAVEPOINT* to destroy a savepoint without discarding the effects of commands executed after it was established.

Specifying a savepoint name that has not been established is an error.

Cursors have somewhat non-transactional behavior with respect to savepoints. Any cursor that is opened inside a savepoint will be closed when the savepoint is rolled back. If a previously opened cursor is affected by a `FETCH` command inside a savepoint that is later rolled back, the cursor position remains at the position that `FETCH` left it pointing to (that is, `FETCH` is not rolled back). Closing a cursor is not undone by rolling back, either. A cursor whose execution causes a transaction to abort is put in a can't-execute state, so while the transaction can be restored using `ROLLBACK TO SAVEPOINT`, the cursor can no longer be used.

Examples

To undo the effects of the commands executed after `my_savepoint` was established:

```
ROLLBACK TO SAVEPOINT my_savepoint;
```

Cursor positions are not affected by savepoint rollback:

```
BEGIN;

DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;

SAVEPOINT foo;

FETCH 1 FROM foo;
 ?column?
----------
        1

ROLLBACK TO SAVEPOINT foo;

FETCH 1 FROM foo;
 ?column?
----------
        2

COMMIT;
```

Compatibility

The SQL standard specifies that the key word `SAVEPOINT` is mandatory, but PostgreSQL and Oracle allow it to be omitted. SQL allows only `WORK`, not `TRANSACTION`, as a noise word after `ROLLBACK`. Also, SQL has an optional clause `AND [ NO ] CHAIN` which is not currently supported by PostgreSQL. Otherwise, this command conforms to the SQL standard.

See Also

BEGIN, *COMMIT*, *RELEASE SAVEPOINT*, *ROLLBACK*, *SAVEPOINT*

SAVEPOINT

Name

SAVEPOINT — define a new savepoint within the current transaction

Synopsis

```
SAVEPOINT savepoint_name
```

Description

SAVEPOINT establishes a new savepoint within the current transaction.

A savepoint is a special mark inside a transaction that allows all commands that are executed after it was established to be rolled back, restoring the transaction state to what it was at the time of the savepoint.

Parameters

savepoint_name

The name to give to the new savepoint.

Notes

Use *ROLLBACK TO SAVEPOINT* to rollback to a savepoint. Use *RELEASE SAVEPOINT* to destroy a savepoint, keeping the effects of commands executed after it was established.

Savepoints can only be established when inside a transaction block. There can be multiple savepoints defined within a transaction.

Examples

To establish a savepoint and later undo the effects of all commands executed after it was established:

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (2);
    ROLLBACK TO SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (3);
COMMIT;
```

The above transaction will insert the values 1 and 3, but not 2.

To establish and later destroy a savepoint:

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

The above transaction will insert both 3 and 4.

Compatibility

SQL requires a savepoint to be destroyed automatically when another savepoint with the same name is established. In PostgreSQL, the old savepoint is kept, though only the more recent one will be used when rolling back or releasing. (Releasing the newer savepoint will cause the older one to again become accessible to `ROLLBACK TO SAVEPOINT` and `RELEASE SAVEPOINT`.) Otherwise, `SAVEPOINT` is fully SQL conforming.

See Also

BEGIN, *COMMIT*, *RELEASE SAVEPOINT*, *ROLLBACK*, *ROLLBACK TO SAVEPOINT*

SELECT

Name

SELECT — retrieve rows from a table or view

Synopsis

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ AS output_name ] [, ...]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]
    [ FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT ] [...] ]

where from_item can be one of:

    [ ONLY ] table_name [ * ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    ( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
    function_name ( [ argument [, ...] ] ) [ AS ] alias [ ( column_alias [, ...] | column_defini
    function_name ( [ argument [, ...] ] ) AS ( column_definition [, ...] )
    from_item [ NATURAL ] join_type from_item [ ON join_condition | USING ( join_column [, ...]
```

Description

SELECT retrieves rows from zero or more tables. The general processing of SELECT is as follows:

1. All elements in the FROM list are computed. (Each element in the FROM list is a real or virtual table.) If more than one element is specified in the FROM list, they are cross-joined together. (See *FROM Clause* below.)
2. If the WHERE clause is specified, all rows that do not satisfy the condition are eliminated from the output. (See *WHERE Clause* below.)
3. If the GROUP BY clause is specified, the output is divided into groups of rows that match on one or more values. If the HAVING clause is present, it eliminates groups that do not satisfy the given condition. (See *GROUP BY Clause* and *HAVING Clause* below.)
4. The actual output rows are computed using the SELECT output expressions for each selected row. (See *SELECT List* below.)
5. Using the operators UNION, INTERSECT, and EXCEPT, the output of more than one SELECT statement can be combined to form a single result set. The UNION operator returns all rows that are in one or both of the result sets. The INTERSECT operator returns all rows that are strictly in both result sets.

The EXCEPT operator returns the rows that are in the first result set but not in the second. In all three cases, duplicate rows are eliminated unless ALL is specified. (See *UNION Clause*, *INTERSECT Clause*, and *EXCEPT Clause* below.)

6. If the ORDER BY clause is specified, the returned rows are sorted in the specified order. If ORDER BY is not given, the rows are returned in whatever order the system finds fastest to produce. (See *ORDER BY Clause* below.)
7. DISTINCT eliminates duplicate rows from the result. DISTINCT ON eliminates rows that match on all the specified expressions. ALL (the default) will return all candidate rows, including duplicates. (See *DISTINCT Clause* below.)
8. If the LIMIT or OFFSET clause is specified, the SELECT statement only returns a subset of the result rows. (See *LIMIT Clause* below.)
9. If FOR UPDATE or FOR SHARE is specified, the SELECT statement locks the selected rows against concurrent updates. (See *FOR UPDATE/FOR SHARE Clause* below.)

You must have SELECT privilege on a table to read its values. The use of FOR UPDATE or FOR SHARE requires UPDATE privilege as well.

Parameters

FROM Clause

The FROM clause specifies one or more source tables for the SELECT. If multiple sources are specified, the result is the Cartesian product (cross join) of all the sources. But usually qualification conditions are added to restrict the returned rows to a small subset of the Cartesian product.

The FROM clause can contain the following elements:

table_name

The name (optionally schema-qualified) of an existing table or view. If ONLY is specified, only that table is scanned. If ONLY is not specified, the table and all its descendant tables (if any) are scanned. * can be appended to the table name to indicate that descendant tables are to be scanned, but in the current version, this is the default behavior. (In releases before 7.1, ONLY was the default behavior.) The default behavior can be modified by changing the sql_inheritance configuration option.

alias

A substitute name for the FROM item containing the alias. An alias is used for brevity or to eliminate ambiguity for self-joins (where the same table is scanned multiple times). When an alias is provided, it completely hides the actual name of the table or function; for example given FROM foo AS f, the remainder of the SELECT must refer to this FROM item as f not foo. If an alias is written, a column alias list can also be written to provide substitute names for one or more columns of the table.

select

A sub-SELECT can appear in the FROM clause. This acts as though its output were created as a temporary table for the duration of this single SELECT command. Note that the sub-SELECT must be

surrounded by parentheses, and an alias *must* be provided for it. A *VALUES* command can also be used here.

function_name

Function calls can appear in the FROM clause. (This is especially useful for functions that return result sets, but any function can be used.) This acts as though its output were created as a temporary table for the duration of this single SELECT command. An alias may also be used. If an alias is written, a column alias list can also be written to provide substitute names for one or more attributes of the function's composite return type. If the function has been defined as returning the record data type, then an alias or the key word AS must be present, followed by a column definition list in the form (*column_name data_type* [, ...]). The column definition list must match the actual number and types of columns returned by the function.

join_type

One of

- [INNER] JOIN
- LEFT [OUTER] JOIN
- RIGHT [OUTER] JOIN
- FULL [OUTER] JOIN
- CROSS JOIN

For the INNER and OUTER join types, a join condition must be specified, namely exactly one of NATURAL, ON *join_condition*, or USING (*join_column* [, ...]). See below for the meaning. For CROSS JOIN, none of these clauses may appear.

A JOIN clause combines two FROM items. Use parentheses if necessary to determine the order of nesting. In the absence of parentheses, JOINs nest left-to-right. In any case JOIN binds more tightly than the commas separating FROM items.

CROSS JOIN and INNER JOIN produce a simple Cartesian product, the same result as you get from listing the two items at the top level of FROM, but restricted by the join condition (if any). CROSS JOIN is equivalent to INNER JOIN ON (TRUE), that is, no rows are removed by qualification. These join types are just a notational convenience, since they do nothing you couldn't do with plain FROM and WHERE.

LEFT OUTER JOIN returns all rows in the qualified Cartesian product (i.e., all combined rows that pass its join condition), plus one copy of each row in the left-hand table for which there was no right-hand row that passed the join condition. This left-hand row is extended to the full width of the joined table by inserting null values for the right-hand columns. Note that only the JOIN clause's own condition is considered while deciding which rows have matches. Outer conditions are applied afterwards.

Conversely, RIGHT OUTER JOIN returns all the joined rows, plus one row for each unmatched right-hand row (extended with nulls on the left). This is just a notational convenience, since you could convert it to a LEFT OUTER JOIN by switching the left and right inputs.

FULL OUTER JOIN returns all the joined rows, plus one row for each unmatched left-hand row (extended with nulls on the right), plus one row for each unmatched right-hand row (extended with nulls on the left).

`ON` *`join_condition`*

`join_condition` is an expression resulting in a value of type `boolean` (similar to a `WHERE` clause) that specifies which rows in a join are considered to match.

`USING (`*`join_column`* `[, ...])`

A clause of the form `USING ( a, b, ... )` is shorthand for `ON left_table.a = right_table.a AND left_table.b = right_table.b ...`. Also, `USING` implies that only one of each pair of equivalent columns will be included in the join output, not both.

`NATURAL`

`NATURAL` is shorthand for a `USING` list that mentions all columns in the two tables that have the same names.

`WHERE` Clause

The optional `WHERE` clause has the general form

`WHERE` *`condition`*

where *`condition`* is any expression that evaluates to a result of type `boolean`. Any row that does not satisfy this condition will be eliminated from the output. A row satisfies the condition if it returns true when the actual row values are substituted for any variable references.

`GROUP BY` Clause

The optional `GROUP BY` clause has the general form

`GROUP BY` *`expression`* `[, ...]`

`GROUP BY` will condense into a single row all selected rows that share the same values for the grouped expressions. *`expression`* can be an input column name, or the name or ordinal number of an output column (`SELECT` list item), or an arbitrary expression formed from input-column values. In case of ambiguity, a `GROUP BY` name will be interpreted as an input-column name rather than an output column name.

Aggregate functions, if any are used, are computed across all rows making up each group, producing a separate value for each group (whereas without `GROUP BY`, an aggregate produces a single value computed across all the selected rows). When `GROUP BY` is present, it is not valid for the `SELECT` list expressions to refer to ungrouped columns except within aggregate functions, since there would be more than one possible value to return for an ungrouped column.

`HAVING` Clause

The optional `HAVING` clause has the general form

`HAVING` *`condition`*

where *condition* is the same as specified for the WHERE clause.

HAVING eliminates group rows that do not satisfy the condition. HAVING is different from WHERE: WHERE filters individual rows before the application of GROUP BY, while HAVING filters group rows created by GROUP BY. Each column referenced in *condition* must unambiguously reference a grouping column, unless the reference appears within an aggregate function.

The presence of HAVING turns a query into a grouped query even if there is no GROUP BY clause. This is the same as what happens when the query contains aggregate functions but no GROUP BY clause. All the selected rows are considered to form a single group, and the SELECT list and HAVING clause can only reference table columns from within aggregate functions. Such a query will emit a single row if the HAVING condition is true, zero rows if it is not true.

SELECT List

The SELECT list (between the key words SELECT and FROM) specifies expressions that form the output rows of the SELECT statement. The expressions can (and usually do) refer to columns computed in the FROM clause. Using the clause AS *output_name*, another name can be specified for an output column. This name is primarily used to label the column for display. It can also be used to refer to the column's value in ORDER BY and GROUP BY clauses, but not in the WHERE or HAVING clauses; there you must write out the expression instead.

Instead of an expression, * can be written in the output list as a shorthand for all the columns of the selected rows. Also, one can write *table_name*.* as a shorthand for the columns coming from just that table.

UNION Clause

The UNION clause has this general form:

```
select_statement UNION [ ALL ] select_statement
```

select_statement is any SELECT statement without an ORDER BY, LIMIT, FOR UPDATE, or FOR SHARE clause. (ORDER BY and LIMIT can be attached to a subexpression if it is enclosed in parentheses. Without parentheses, these clauses will be taken to apply to the result of the UNION, not to its right-hand input expression.)

The UNION operator computes the set union of the rows returned by the involved SELECT statements. A row is in the set union of two result sets if it appears in at least one of the result sets. The two SELECT statements that represent the direct operands of the UNION must produce the same number of columns, and corresponding columns must be of compatible data types.

The result of UNION does not contain any duplicate rows unless the ALL option is specified. ALL prevents elimination of duplicates. (Therefore, UNION ALL is usually significantly quicker than UNION; use ALL when you can.)

Multiple UNION operators in the same SELECT statement are evaluated left to right, unless otherwise indicated by parentheses.

Currently, FOR UPDATE and FOR SHARE may not be specified either for a UNION result or for any input of a UNION.

`INTERSECT` Clause

The `INTERSECT` clause has this general form:

```
select_statement INTERSECT [ ALL ] select_statement
```

`select_statement` is any `SELECT` statement without an `ORDER BY`, `LIMIT`, `FOR UPDATE`, or `FOR SHARE` clause.

The `INTERSECT` operator computes the set intersection of the rows returned by the involved `SELECT` statements. A row is in the intersection of two result sets if it appears in both result sets.

The result of `INTERSECT` does not contain any duplicate rows unless the `ALL` option is specified. With `ALL`, a row that has *`m`* duplicates in the left table and *`n`* duplicates in the right table will appear min(*`m`*,*`n`*) times in the result set.

Multiple `INTERSECT` operators in the same `SELECT` statement are evaluated left to right, unless parentheses dictate otherwise. `INTERSECT` binds more tightly than `UNION`. That is, `A UNION B INTERSECT C` will be read as `A UNION (B INTERSECT C)`.

Currently, `FOR UPDATE` and `FOR SHARE` may not be specified either for an `INTERSECT` result or for any input of an `INTERSECT`.

`EXCEPT` Clause

The `EXCEPT` clause has this general form:

```
select_statement EXCEPT [ ALL ] select_statement
```

`select_statement` is any `SELECT` statement without an `ORDER BY`, `LIMIT`, `FOR UPDATE`, or `FOR SHARE` clause.

The `EXCEPT` operator computes the set of rows that are in the result of the left `SELECT` statement but not in the result of the right one.

The result of `EXCEPT` does not contain any duplicate rows unless the `ALL` option is specified. With `ALL`, a row that has *`m`* duplicates in the left table and *`n`* duplicates in the right table will appear max(*`m`*-*`n`*,0) times in the result set.

Multiple `EXCEPT` operators in the same `SELECT` statement are evaluated left to right, unless parentheses dictate otherwise. `EXCEPT` binds at the same level as `UNION`.

Currently, `FOR UPDATE` and `FOR SHARE` may not be specified either for an `EXCEPT` result or for any input of an `EXCEPT`.

`ORDER BY` Clause

The optional `ORDER BY` clause has this general form:

```
ORDER BY expression [ ASC | DESC | USING operator ] [, ...]
```

`expression` can be the name or ordinal number of an output column (`SELECT` list item), or it can be an arbitrary expression formed from input-column values.

The `ORDER BY` clause causes the result rows to be sorted according to the specified expressions. If two rows are equal according to the leftmost expression, the are compared according to the next expression and so on. If they are equal according to all specified expressions, they are returned in an implementation-dependent order.

The ordinal number refers to the ordinal (left-to-right) position of the result column. This feature makes it possible to define an ordering on the basis of a column that does not have a unique name. This is never absolutely necessary because it is always possible to assign a name to a result column using the `AS` clause.

It is also possible to use arbitrary expressions in the `ORDER BY` clause, including columns that do not appear in the `SELECT` result list. Thus the following statement is valid:

```
SELECT name FROM distributors ORDER BY code;
```

A limitation of this feature is that an `ORDER BY` clause applying to the result of a `UNION`, `INTERSECT`, or `EXCEPT` clause may only specify an output column name or number, not an expression.

If an `ORDER BY` expression is a simple name that matches both a result column name and an input column name, `ORDER BY` will interpret it as the result column name. This is the opposite of the choice that `GROUP BY` will make in the same situation. This inconsistency is made to be compatible with the SQL standard.

Optionally one may add the key word `ASC` (ascending) or `DESC` (descending) after any expression in the `ORDER BY` clause. If not specified, `ASC` is assumed by default. Alternatively, a specific ordering operator name may be specified in the `USING` clause. `ASC` is usually equivalent to `USING <` and `DESC` is usually equivalent to `USING >`. (But the creator of a user-defined data type can define exactly what the default sort ordering is, and it might correspond to operators with other names.)

The null value sorts higher than any other value. In other words, with ascending sort order, null values sort at the end, and with descending sort order, null values sort at the beginning.

Character-string data is sorted according to the locale-specific collation order that was established when the database cluster was initialized.

`DISTINCT` Clause

If `DISTINCT` is specified, all duplicate rows are removed from the result set (one row is kept from each group of duplicates). `ALL` specifies the opposite: all rows are kept; that is the default.

`DISTINCT ON ( `*`expression`*` [, ...] )` keeps only the first row of each set of rows where the given expressions evaluate to equal. The `DISTINCT ON` expressions are interpreted using the same rules as for `ORDER BY` (see above). Note that the "first row" of each set is unpredictable unless `ORDER BY` is used to ensure that the desired row appears first. For example,

```
SELECT DISTINCT ON (location) location, time, report
    FROM weather_reports
    ORDER BY location, time DESC;
```

retrieves the most recent weather report for each location. But if we had not used `ORDER BY` to force descending order of time values for each location, we'd have gotten a report from an unpredictable time for each location.

The DISTINCT ON expression(s) must match the leftmost ORDER BY expression(s). The ORDER BY clause will normally contain additional expression(s) that determine the desired precedence of rows within each DISTINCT ON group.

LIMIT Clause

The LIMIT clause consists of two independent sub-clauses:

```
LIMIT { count | ALL }
OFFSET start
```

count specifies the maximum number of rows to return, while *start* specifies the number of rows to skip before starting to return rows. When both are specified, *start* rows are skipped before starting to count the *count* rows to be returned.

When using LIMIT, it is a good idea to use an ORDER BY clause that constrains the result rows into a unique order. Otherwise you will get an unpredictable subset of the query's rows — you may be asking for the tenth through twentieth rows, but tenth through twentieth in what ordering? You don't know what ordering unless you specify ORDER BY.

The query planner takes LIMIT into account when generating a query plan, so you are very likely to get different plans (yielding different row orders) depending on what you use for LIMIT and OFFSET. Thus, using different LIMIT/OFFSET values to select different subsets of a query result *will give inconsistent results* unless you enforce a predictable result ordering with ORDER BY. This is not a bug; it is an inherent consequence of the fact that SQL does not promise to deliver the results of a query in any particular order unless ORDER BY is used to constrain the order.

FOR UPDATE/FOR SHARE Clause

The FOR UPDATE clause has this form:

```
FOR UPDATE [ OF table_name [, ...] ] [ NOWAIT ]
```

The closely related FOR SHARE clause has this form:

```
FOR SHARE [ OF table_name [, ...] ] [ NOWAIT ]
```

FOR UPDATE causes the rows retrieved by the SELECT statement to be locked as though for update. This prevents them from being modified or deleted by other transactions until the current transaction ends. That is, other transactions that attempt UPDATE, DELETE, or SELECT FOR UPDATE of these rows will be blocked until the current transaction ends. Also, if an UPDATE, DELETE, or SELECT FOR UPDATE from another transaction has already locked a selected row or rows, SELECT FOR UPDATE will wait for the other transaction to complete, and will then lock and return the updated row (or no row, if the row was deleted). For further discussion see Chapter 12.

To prevent the operation from waiting for other transactions to commit, use the NOWAIT option. SELECT FOR UPDATE NOWAIT reports an error, rather than waiting, if a selected row cannot be locked immedi-

ately. Note that NOWAIT applies only to the row-level lock(s) — the required ROW SHARE table-level lock is still taken in the ordinary way (see Chapter 12). You can use the NOWAIT option of *LOCK* if you need to acquire the table-level lock without waiting.

FOR SHARE behaves similarly, except that it acquires a shared rather than exclusive lock on each retrieved row. A shared lock blocks other transactions from performing UPDATE, DELETE, or SELECT FOR UPDATE on these rows, but it does not prevent them from performing SELECT FOR SHARE.

If specific tables are named in FOR UPDATE or FOR SHARE, then only rows coming from those tables are locked; any other tables used in the SELECT are simply read as usual. A FOR UPDATE or FOR SHARE clause without a table list affects all tables used in the command. If FOR UPDATE or FOR SHARE is applied to a view or sub-query, it affects all tables used in the view or sub-query.

Multiple FOR UPDATE and FOR SHARE clauses can be written if it is necessary to specify different locking behavior for different tables. If the same table is mentioned (or implicitly affected) by both FOR UPDATE and FOR SHARE clauses, then it is processed as FOR UPDATE. Similarly, a table is processed as NOWAIT if that is specified in any of the clauses affecting it.

FOR UPDATE and FOR SHARE cannot be used in contexts where returned rows can't be clearly identified with individual table rows; for example they can't be used with aggregation.

Caution

Avoid locking a row and then modifying it within a later savepoint or PL/pgSQL exception block. A subsequent rollback would cause the lock to be lost. For example,

```
BEGIN;
SELECT * FROM mytable WHERE key = 1 FOR UPDATE;
SAVEPOINT s;
UPDATE mytable SET ... WHERE key = 1;
ROLLBACK TO s;
```

After the ROLLBACK, the row is effectively unlocked, rather than returned to its pre-savepoint state of being locked but not modified. This hazard occurs if a row locked in the current transaction is updated or deleted, or if a shared lock is upgraded to exclusive: in all these cases, the former lock state is forgotten. If the transaction is then rolled back to a state between the original locking command and the subsequent change, the row will appear not to be locked at all. This is an implementation deficiency which will be addressed in a future release of PostgreSQL.

Caution

It is possible for a SELECT command using both LIMIT and FOR UPDATE/SHARE clauses to return fewer rows than specified by LIMIT. This is because LIMIT is applied first. The command selects the specified number of rows, but might then block trying to obtain lock on one or more of them. Once the SELECT unblocks, the row might have been deleted or updated so that it does not meet the query WHERE condition anymore, in which case it will not be returned.

Examples

To join the table `films` with the table `distributors`:

```
SELECT f.title, f.did, d.name, f.date_prod, f.kind
    FROM distributors d, films f
    WHERE f.did = d.did
```

```
       title       | did |     name     | date_prod  |   kind
-------------------+-----+--------------+------------+----------
 The Third Man     | 101 | British Lion | 1949-12-23 | Drama
 The African Queen | 101 | British Lion | 1951-08-11 | Romantic
 ...
```

To sum the column `len` of all films and group the results by `kind`:

```
SELECT kind, sum(len) AS total FROM films GROUP BY kind;
```

```
   kind   | total
----------+-------
 Action   | 07:34
 Comedy   | 02:58
 Drama    | 14:28
 Musical  | 06:42
 Romantic | 04:38
```

To sum the column `len` of all films, group the results by `kind` and show those group totals that are less than 5 hours:

```
SELECT kind, sum(len) AS total
    FROM films
    GROUP BY kind
    HAVING sum(len) < interval '5 hours';
```

```
   kind   | total
----------+-------
 Comedy   | 02:58
 Romantic | 04:38
```

The following two examples are identical ways of sorting the individual results according to the contents of the second column (`name`):

```
SELECT * FROM distributors ORDER BY name;
SELECT * FROM distributors ORDER BY 2;
```

```
 did |       name
-----+------------------
 109 | 20th Century Fox
 110 | Bavaria Atelier
```

```
 101 | British Lion
 107 | Columbia
 102 | Jean Luc Godard
 113 | Luso films
 104 | Mosfilm
 103 | Paramount
 106 | Toho
 105 | United Artists
 111 | Walt Disney
 112 | Warner Bros.
 108 | Westward
```

The next example shows how to obtain the union of the tables `distributors` and `actors`, restricting the results to those that begin with the letter W in each table. Only distinct rows are wanted, so the key word `ALL` is omitted.

```
distributors:               actors:
 did |     name             id |     name
-----+--------------       ----+----------------
 108 | Westward              1 | Woody Allen
 111 | Walt Disney           2 | Warren Beatty
 112 | Warner Bros.          3 | Walter Matthau
 ...                         ...
```

```
SELECT distributors.name
    FROM distributors
    WHERE distributors.name LIKE 'W%'
UNION
SELECT actors.name
    FROM actors
    WHERE actors.name LIKE 'W%';
```

```
      name
----------------
 Walt Disney
 Walter Matthau
 Warner Bros.
 Warren Beatty
 Westward
 Woody Allen
```

This example shows how to use a function in the `FROM` clause, both with and without a column definition list:

```
CREATE FUNCTION distributors(int) RETURNS SETOF distributors AS $$
    SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;

SELECT * FROM distributors(111);
 did |    name
```

```
-----+-------------
 111 | Walt Disney

CREATE FUNCTION distributors_2(int) RETURNS SETOF record AS $$
    SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;

SELECT * FROM distributors_2(111) AS (f1 int, f2 text);
 f1  |     f2
-----+-------------
 111 | Walt Disney
```

Compatibility

Of course, the SELECT statement is compatible with the SQL standard. But there are some extensions and some missing features.

Omitted FROM Clauses

PostgreSQL allows one to omit the FROM clause. It has a straightforward use to compute the results of simple expressions:

```
SELECT 2+2;

 ?column?
----------
        4
```

Some other SQL databases cannot do this except by introducing a dummy one-row table from which to do the SELECT.

Note that if a FROM clause is not specified, the query cannot reference any database tables. For example, the following query is invalid:

```
SELECT distributors.* WHERE distributors.name = 'Westward';
```

PostgreSQL releases prior to 8.1 would accept queries of this form, and add an implicit entry to the query's FROM clause for each table referenced by the query. This is no longer the default behavior, because it does not comply with the SQL standard, and is considered by many to be error-prone. For compatibility with applications that rely on this behavior the add_missing_from configuration variable can be enabled.

The AS Key Word

In the SQL standard, the optional key word AS is just noise and can be omitted without affecting the meaning. The PostgreSQL parser requires this key word when renaming output columns because the type extensibility features lead to parsing ambiguities without it. AS is optional in FROM items, however.

Namespace Available to `GROUP BY` and `ORDER BY`

In the SQL-92 standard, an `ORDER BY` clause may only use result column names or numbers, while a `GROUP BY` clause may only use expressions based on input column names. PostgreSQL extends each of these clauses to allow the other choice as well (but it uses the standard's interpretation if there is ambiguity). PostgreSQL also allows both clauses to specify arbitrary expressions. Note that names appearing in an expression will always be taken as input-column names, not as result-column names.

SQL:1999 and later use a slightly different definition which is not entirely upward compatible with SQL-92. In most cases, however, PostgreSQL will interpret an `ORDER BY` or `GROUP BY` expression the same way SQL:1999 does.

Nonstandard Clauses

The clauses `DISTINCT ON`, `LIMIT`, and `OFFSET` are not defined in the SQL standard.

SELECT INTO

Name

SELECT INTO — define a new table from the results of a query

Synopsis

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ AS output_name ] [, ...]
    INTO [ TEMPORARY | TEMP ] [ TABLE ] new_table
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]
    [ FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT ] [...] ]
```

Description

SELECT INTO creates a new table and fills it with data computed by a query. The data is not returned to the client, as it is with a normal SELECT. The new table's columns have the names and data types associated with the output columns of the SELECT.

Parameters

TEMPORARY or TEMP

If specified, the table is created as a temporary table. Refer to *CREATE TABLE* for details.

new_table

The name (optionally schema-qualified) of the table to be created.

All other parameters are described in detail under *SELECT*.

Notes

CREATE TABLE AS is functionally similar to SELECT INTO. CREATE TABLE AS is the recommended syntax, since this form of SELECT INTO is not available in ECPG or PL/pgSQL, because they interpret the INTO clause differently. Furthermore, CREATE TABLE AS offers a superset of the functionality provided by SELECT INTO.

Prior to PostgreSQL 8.1, the table created by `SELECT INTO` included OIDs by default. In PostgreSQL 8.1, this is not the case — to include OIDs in the new table, the default_with_oids configuration variable must be enabled. Alternatively, `CREATE TABLE AS` can be used with the `WITH OIDS` clause.

Examples

Create a new table `films_recent` consisting of only recent entries from the table `films`:

```
SELECT * INTO films_recent FROM films WHERE date_prod >= '2002-01-01';
```

Compatibility

The SQL standard uses `SELECT INTO` to represent selecting values into scalar variables of a host program, rather than creating a new table. This indeed is the usage found in ECPG (see Chapter 31) and PL/pgSQL (see Chapter 37). The PostgreSQL usage of `SELECT INTO` to represent table creation is historical. It is best to use `CREATE TABLE AS` for this purpose in new code.

See Also

CREATE TABLE AS

SET

Name

SET — change a run-time parameter

Synopsis

```
SET [ SESSION | LOCAL ] configuration_parameter { TO | = } { value | 'value' | DEFAULT }
SET [ SESSION | LOCAL ] TIME ZONE { timezone | LOCAL | DEFAULT }
```

Description

The SET command changes run-time configuration parameters. Many of the run-time parameters listed in Chapter 17 can be changed on-the-fly with SET. (But some require superuser privileges to change, and others cannot be changed after server or session start.) SET only affects the value used by the current session.

If SET or SET SESSION is issued within a transaction that is later aborted, the effects of the SET command disappear when the transaction is rolled back. (This behavior represents a change from PostgreSQL versions prior to 7.3, where the effects of SET would not roll back after a later error.) Once the surrounding transaction is committed, the effects will persist until the end of the session, unless overridden by another SET.

The effects of SET LOCAL last only till the end of the current transaction, whether committed or not. A special case is SET followed by SET LOCAL within a single transaction: the SET LOCAL value will be seen until the end of the transaction, but afterwards (if the transaction is committed) the SET value will take effect.

Parameters

SESSION

Specifies that the command takes effect for the current session. (This is the default if neither SESSION nor LOCAL appears.)

LOCAL

Specifies that the command takes effect for only the current transaction. After COMMIT or ROLLBACK, the session-level setting takes effect again. Note that SET LOCAL will appear to have no effect if it is executed outside a BEGIN block, since the transaction will end immediately.

configuration_parameter

Name of a settable run-time parameter. Available parameters are documented in Chapter 17 and below.

value

New value of parameter. Values can be specified as string constants, identifiers, numbers, or comma-separated lists of these. `DEFAULT` can be used to specify resetting the parameter to its default value.

Besides the configuration parameters documented in Chapter 17, there are a few that can only be adjusted using the `SET` command or that have a special syntax:

`NAMES`

`SET NAMES` *value* is an alias for `SET client_encoding TO` *value*.

`SEED`

Sets the internal seed for the random number generator (the function `random`). Allowed values are floating-point numbers between 0 and 1, which are then multiplied by 2^{31}-1.

The seed can also be set by invoking the function `setseed`:

```
SELECT setseed(value);
```

`TIME ZONE`

`SET TIME ZONE` *value* is an alias for `SET timezone TO` *value*. The syntax `SET TIME ZONE` allows special syntax for the time zone specification. Here are examples of valid values:

`'PST8PDT'`

The time zone for Berkeley, California.

`'Europe/Rome'`

The time zone for Italy.

`-7`

The time zone 7 hours west from UTC (equivalent to PDT). Positive values are east from UTC.

`INTERVAL '-08:00' HOUR TO MINUTE`

The time zone 8 hours west from UTC (equivalent to PST).

`LOCAL`
`DEFAULT`

Set the time zone to your local time zone (the one that the server's operating system defaults to).

See Section 8.5.3 for more information about time zones.

Notes

The function `set_config` provides equivalent functionality. See Section 9.20.

Examples

Set the schema search path:

```
SET search_path TO my_schema, public;
```

Set the style of date to traditional POSTGRES with “day before month” input convention:

```
SET datestyle TO postgres, dmy;
```

Set the time zone for Berkeley, California:

```
SET TIME ZONE 'PST8PDT';
```

Set the time zone for Italy:

```
SET TIME ZONE 'Europe/Rome';
```

Compatibility

`SET TIME ZONE` extends syntax defined in the SQL standard. The standard allows only numeric time zone offsets while PostgreSQL allows more flexible time-zone specifications. All other `SET` features are PostgreSQL extensions.

See Also

RESET, *SHOW*

SET CONSTRAINTS

Name

SET CONSTRAINTS — set constraint checking modes for the current transaction

Synopsis

```
SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }
```

Description

SET CONSTRAINTS sets the behavior of constraint checking within the current transaction. IMMEDIATE constraints are checked at the end of each statement. DEFERRED constraints are not checked until transaction commit. Each constraint has its own IMMEDIATE or DEFERRED mode.

Upon creation, a constraint is given one of three characteristics: DEFERRABLE INITIALLY DEFERRED, DEFERRABLE INITIALLY IMMEDIATE, or NOT DEFERRABLE. The third class is always IMMEDIATE and is not affected by the SET CONSTRAINTS command. The first two classes start every transaction in the indicated mode, but their behavior can be changed within a transaction by SET CONSTRAINTS.

SET CONSTRAINTS with a list of constraint names changes the mode of just those constraints (which must all be deferrable). The current schema search path is used to find the first matching name if no schema name is specified. SET CONSTRAINTS ALL changes the mode of all deferrable constraints.

When SET CONSTRAINTS changes the mode of a constraint from DEFERRED to IMMEDIATE, the new mode takes effect retroactively: any outstanding data modifications that would have been checked at the end of the transaction are instead checked during the execution of the SET CONSTRAINTS command. If any such constraint is violated, the SET CONSTRAINTS fails (and does not change the constraint mode). Thus, SET CONSTRAINTS can be used to force checking of constraints to occur at a specific point in a transaction.

Currently, only foreign key constraints are affected by this setting. Check and unique constraints are always effectively not deferrable.

Notes

This command only alters the behavior of constraints within the current transaction. Thus, if you execute this command outside of a transaction block (BEGIN/COMMIT pair), it will not appear to have any effect.

Compatibility

This command complies with the behavior defined in the SQL standard, except for the limitation that, in PostgreSQL, it only applies to foreign-key constraints.

SET ROLE

Name

SET ROLE — set the current user identifier of the current session

Synopsis

```
SET [ SESSION | LOCAL ] ROLE rolename
SET [ SESSION | LOCAL ] ROLE NONE
RESET ROLE
```

Description

This command sets the current user identifier of the current SQL session to be *rolename*. The role name may be written as either an identifier or a string literal. After SET ROLE, permissions checking for SQL commands is carried out as though the named role were the one that had logged in originally.

The specified *rolename* must be a role that the current session user is a member of. (If the session user is a superuser, any role can be selected.)

The SESSION and LOCAL modifiers act the same as for the regular *SET* command.

The NONE and RESET forms reset the current user identifier to be the current session user identifier. These forms may be executed by any user.

Notes

Using this command, it is possible to either add privileges or restrict one's privileges. If the session user role has the INHERITS attribute, then it automatically has all the privileges of every role that it could SET ROLE to; in this case SET ROLE effectively drops all the privileges assigned directly to the session user and to the other roles it is a member of, leaving only the privileges available to the named role. On the other hand, if the session user role has the NOINHERITS attribute, SET ROLE drops the privileges assigned directly to the session user and instead acquires the privileges available to the named role.

In particular, when a superuser chooses to SET ROLE to a non-superuser role, she loses her superuser privileges.

SET ROLE has effects comparable to *SET SESSION AUTHORIZATION*, but the privilege checks involved are quite different. Also, SET SESSION AUTHORIZATION determines which roles are allowable for later SET ROLE commands, whereas changing roles with SET ROLE does not change the set of roles allowed to a later SET ROLE.

SET ROLE cannot be used within a SECURITY DEFINER function.

Examples

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | peter

SET ROLE 'paul';

SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | paul
```

Compatibility

PostgreSQL allows identifier syntax (`"rolename"`), while the SQL standard requires the role name to be written as a string literal. SQL does not allow this command during a transaction; PostgreSQL does not make this restriction because there is no reason to. The `SESSION` and `LOCAL` modifiers are a PostgreSQL extension, as is the `RESET` syntax.

See Also

SET SESSION AUTHORIZATION

SET SESSION AUTHORIZATION

Name

SET SESSION AUTHORIZATION — set the session user identifier and the current user identifier of the current session

Synopsis

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION username
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION
```

Description

This command sets the session user identifier and the current user identifier of the current SQL session to be *username*. The user name may be written as either an identifier or a string literal. Using this command, it is possible, for example, to temporarily become an unprivileged user and later switch back to being a superuser.

The session user identifier is initially set to be the (possibly authenticated) user name provided by the client. The current user identifier is normally equal to the session user identifier, but might change temporarily in the context of SECURITY DEFINER functions and similar mechanisms; it can also be changed by *SET ROLE*. The current user identifier is relevant for permission checking.

The session user identifier may be changed only if the initial session user (the *authenticated user*) had the superuser privilege. Otherwise, the command is accepted only if it specifies the authenticated user name.

The SESSION and LOCAL modifiers act the same as for the regular *SET* command.

The DEFAULT and RESET forms reset the session and current user identifiers to be the originally authenticated user name. These forms may be executed by any user.

Notes

SET SESSION AUTHORIZATION cannot be used within a SECURITY DEFINER function.

Examples

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | peter

SET SESSION AUTHORIZATION 'paul';
```

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 paul         | paul
```

Compatibility

The SQL standard allows some other expressions to appear in place of the literal *username*, but these options are not important in practice. PostgreSQL allows identifier syntax (`"username"`), which SQL does not. SQL does not allow this command during a transaction; PostgreSQL does not make this restriction because there is no reason to. The `SESSION` and `LOCAL` modifiers are a PostgreSQL extension, as is the `RESET` syntax.

The privileges necessary to execute this command are left implementation-defined by the standard.

See Also

SET ROLE

SET TRANSACTION

Name

SET TRANSACTION — set the characteristics of the current transaction

Synopsis

```
SET TRANSACTION transaction_mode [, ...]
SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode [, ...]

where transaction_mode is one of:

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
```

Description

The SET TRANSACTION command sets the characteristics of the current transaction. It has no effect on any subsequent transactions. SET SESSION CHARACTERISTICS sets the default transaction characteristics for subsequent transactions of a session. These defaults can be overridden by SET TRANSACTION for an individual transaction.

The available transaction characteristics are the transaction isolation level and the transaction access mode (read/write or read-only).

The isolation level of a transaction determines what data the transaction can see when other transactions are running concurrently:

READ COMMITTED

A statement can only see rows committed before it began. This is the default.

SERIALIZABLE

All statements of the current transaction can only see rows committed before the first query or data-modification statement was executed in this transaction.

The SQL standard defines two additional levels, READ UNCOMMITTED and REPEATABLE READ. In PostgreSQL READ UNCOMMITTED is treated as READ COMMITTED, while REPEATABLE READ is treated as SERIALIZABLE.

The transaction isolation level cannot be changed after the first query or data-modification statement (SELECT, INSERT, DELETE, UPDATE, FETCH, or COPY) of a transaction has been executed. See Chapter 12 for more information about transaction isolation and concurrency control.

The transaction access mode determines whether the transaction is read/write or read-only. Read/write is the default. When a transaction is read-only, the following SQL commands are disallowed: INSERT, UPDATE, DELETE, and COPY FROM if the table they would write to is not a temporary table; all CREATE, ALTER, and DROP commands; COMMENT, GRANT, REVOKE, TRUNCATE; and EXPLAIN ANALYZE and

EXECUTE if the command they would execute is among those listed. This is a high-level notion of read-only that does not prevent all writes to disk.

Notes

If SET TRANSACTION is executed without a prior START TRANSACTION or BEGIN, it will appear to have no effect, since the transaction will immediately end.

It is possible to dispense with SET TRANSACTION by instead specifying the desired *transaction_modes* in BEGIN or START TRANSACTION.

The session default transaction modes can also be set by setting the configuration parameters default_transaction_isolation and default_transaction_read_only. (In fact SET SESSION CHARACTERISTICS is just a verbose equivalent for setting these variables with SET.) This means the defaults can be set in the configuration file, via ALTER DATABASE, etc. Consult Chapter 17 for more information.

Compatibility

Both commands are defined in the SQL standard. SERIALIZABLE is the default transaction isolation level in the standard. In PostgreSQL the default is ordinarily READ COMMITTED, but you can change it as mentioned above. Because of lack of predicate locking, the SERIALIZABLE level is not truly serializable. See Chapter 12 for details.

In the SQL standard, there is one other transaction characteristic that can be set with these commands: the size of the diagnostics area. This concept is specific to embedded SQL, and therefore is not implemented in the PostgreSQL server.

The SQL standard requires commas between successive *transaction_modes*, but for historical reasons PostgreSQL allows the commas to be omitted.

SHOW

Name

SHOW — show the value of a run-time parameter

Synopsis

```
SHOW name
SHOW ALL
```

Description

SHOW will display the current setting of run-time parameters. These variables can be set using the SET statement, by editing the postgresql.conf configuration file, through the PGOPTIONS environmental variable (when using libpq or a libpq-based application), or through command-line flags when starting the postgres. See Chapter 17 for details.

Parameters

name

The name of a run-time parameter. Available parameters are documented in Chapter 17 and on the *SET* reference page. In addition, there are a few parameters that can be shown but not set:

SERVER_VERSION

Shows the server's version number.

SERVER_ENCODING

Shows the server-side character set encoding. At present, this parameter can be shown but not set, because the encoding is determined at database creation time.

LC_COLLATE

Shows the database's locale setting for collation (text ordering). At present, this parameter can be shown but not set, because the setting is determined at initdb time.

LC_CTYPE

Shows the database's locale setting for character classification. At present, this parameter can be shown but not set, because the setting is determined at initdb time.

IS_SUPERUSER

True if the current role has superuser privileges.

ALL

Show the values of all configuration parameters, with descriptions.

Notes

The function current_setting produces equivalent output. See Section 9.20.

Examples

Show the current setting of the parameter DateStyle:

```
SHOW DateStyle;
 DateStyle
-----------
 ISO, MDY
(1 row)
```

Show the current setting of the parameter geqo:

```
SHOW geqo;
 geqo
------
 on
(1 row)
```

Show all settings:

```
SHOW ALL;
               name               |            setting              |
----------------------------------+---------------------------------+-------------------------
 add_missing_from                 | off                             | Automatically adds missi
 archive_command                  | unset                           | WAL archiving command.
    .
    .
    .
 work_mem                         | 1024                            | Sets the maximum memory
 zero_damaged_pages               | off                             | Continues processing pas
(146 rows)
```

Compatibility

The SHOW command is a PostgreSQL extension.

See Also

SET, *RESET*

START TRANSACTION

Name

START TRANSACTION — start a transaction block

Synopsis

```
START TRANSACTION [ transaction_mode [, ...] ]

where transaction_mode is one of:

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
```

Description

This command begins a new transaction block. If the isolation level or read/write mode is specified, the new transaction has those characteristics, as if *SET TRANSACTION* was executed. This is the same as the *BEGIN* command.

Parameters

Refer to *SET TRANSACTION* for information on the meaning of the parameters to this statement.

Compatibility

In the standard, it is not necessary to issue START TRANSACTION to start a transaction block: any SQL command implicitly begins a block. PostgreSQL's behavior can be seen as implicitly issuing a COMMIT after each command that does not follow START TRANSACTION (or BEGIN), and it is therefore often called "autocommit". Other relational database systems may offer an autocommit feature as a convenience.

The SQL standard requires commas between successive *transaction_modes*, but for historical reasons PostgreSQL allows the commas to be omitted.

See also the compatibility section of *SET TRANSACTION*.

See Also

BEGIN, *COMMIT*, *ROLLBACK*, *SAVEPOINT*, *SET TRANSACTION*

TRUNCATE

Name

TRUNCATE — empty a table or set of tables

Synopsis

```
TRUNCATE [ TABLE ] name [, ...] [ CASCADE | RESTRICT ]
```

Description

TRUNCATE quickly removes all rows from a set of tables. It has the same effect as an unqualified DELETE on each table, but since it does not actually scan the tables it is faster. This is most useful on large tables.

Parameters

name

The name (optionally schema-qualified) of a table to be truncated.

CASCADE

Automatically truncate all tables that have foreign-key references to any of the named tables, or to any tables added to the group due to CASCADE.

RESTRICT

Refuse to truncate if any of the tables have foreign-key references from tables that are not to be truncated. This is the default.

Notes

Only the owner of a table may TRUNCATE it.

TRUNCATE cannot be used on a table that has foreign-key references from other tables, unless all such tables are also truncated in the same command. Checking validity in such cases would require table scans, and the whole point is not to do one. The CASCADE option can be used to automatically include all dependent tables — but be very careful when using this option, else you might lose data you did not intend to!

TRUNCATE will not run any user-defined ON DELETE triggers that might exist for the tables.

Examples

Truncate the tables `bigtable` and `fattable`:

```
TRUNCATE TABLE bigtable, fattable;
```

Truncate the table `othertable`, and cascade to any tables that are referencing `othertable` via foreign-key constraints:

```
TRUNCATE othertable CASCADE;
```

Compatibility

There is no `TRUNCATE` command in the SQL standard.

UNLISTEN

Name

UNLISTEN — stop listening for a notification

Synopsis

```
UNLISTEN { name | * }
```

Description

UNLISTEN is used to remove an existing registration for NOTIFY events. UNLISTEN cancels any existing registration of the current PostgreSQL session as a listener on the notification *name*. The special wildcard * cancels all listener registrations for the current session.

NOTIFY contains a more extensive discussion of the use of LISTEN and NOTIFY.

Parameters

name

Name of a notification (any identifier).

*

All current listen registrations for this session are cleared.

Notes

You may unlisten something you were not listening for; no warning or error will appear.

At the end of each session, UNLISTEN * is automatically executed.

Examples

To make a registration:

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
```

Once UNLISTEN has been executed, further NOTIFY commands will be ignored:

```
UNLISTEN virtual;
NOTIFY virtual;
-- no NOTIFY event is received
```

Compatibility

There is no `UNLISTEN` command in the SQL standard.

See Also

LISTEN, *NOTIFY*

UPDATE

Name

UPDATE — update rows of a table

Synopsis

```
UPDATE [ ONLY ] table [ [ AS ] alias ]
    SET { column = { expression | DEFAULT } |
          ( column [, ...] ) = ( { expression | DEFAULT } [, ...] ) } [, ...]
    [ FROM fromlist ]
    [ WHERE condition ]
    [ RETURNING * | output_expression [ AS output_name ] [, ...] ]
```

Description

UPDATE changes the values of the specified columns in all rows that satisfy the condition. Only the columns to be modified need be mentioned in the SET clause; columns not explicitly modified retain their previous values.

By default, UPDATE will update rows in the specified table and all its subtables. If you wish to only update the specific table mentioned, you must use the ONLY clause.

There are two ways to modify a table using information contained in other tables in the database: using sub-selects, or specifying additional tables in the FROM clause. Which technique is more appropriate depends on the specific circumstances.

The optional RETURNING clause causes UPDATE to compute and return value(s) based on each row actually updated. Any expression using the table's columns, and/or columns of other tables mentioned in FROM, can be computed. The new (post-update) values of the table's columns are used. The syntax of the RETURNING list is identical to that of the output list of SELECT.

You must have the UPDATE privilege on the table to update it, as well as the SELECT privilege to any table whose values are read in the *expressions* or *condition*.

Parameters

table

The name (optionally schema-qualified) of the table to update.

alias

A substitute name for the target table. When an alias is provided, it completely hides the actual name of the table. For example, given UPDATE foo AS f, the remainder of the UPDATE statement must refer to this table as f not foo.

column

The name of a column in *table*. The column name can be qualified with a subfield name or array subscript, if needed. Do not include the table's name in the specification of a target column — for example, `UPDATE tab SET tab.col = 1` is invalid.

expression

An expression to assign to the column. The expression may use the old values of this and other columns in the table.

`DEFAULT`

Set the column to its default value (which will be NULL if no specific default expression has been assigned to it).

fromlist

A list of table expressions, allowing columns from other tables to appear in the `WHERE` condition and the update expressions. This is similar to the list of tables that can be specified in the *FROM Clause* of a `SELECT` statement. Note that the target table must not appear in the *fromlist*, unless you intend a self-join (in which case it must appear with an alias in the *fromlist*).

condition

An expression that returns a value of type `boolean`. Only rows for which this expression returns `true` will be updated.

output_expression

An expression to be computed and returned by the `UPDATE` command after each row is updated. The expression may use any column names of the *table* or table(s) listed in `FROM`. Write `*` to return all columns.

output_name

A name to use for a returned column.

Outputs

On successful completion, an `UPDATE` command returns a command tag of the form

`UPDATE` *count*

The *count* is the number of rows updated. If *count* is 0, no rows matched the *condition* (this is not considered an error).

If the `UPDATE` command contains a `RETURNING` clause, the result will be similar to that of a `SELECT` statement containing the columns and values defined in the `RETURNING` list, computed over the row(s) updated by the command.

Notes

When a `FROM` clause is present, what essentially happens is that the target table is joined to the tables mentioned in the *fromlist*, and each output row of the join represents an update operation for the target

table. When using `FROM` you should ensure that the join produces at most one output row for each row to be modified. In other words, a target row shouldn't join to more than one row from the other table(s). If it does, then only one of the join rows will be used to update the target row, but which one will be used is not readily predictable.

Because of this indeterminacy, referencing other tables only within sub-selects is safer, though often harder to read and slower than using a join.

Examples

Change the word `Drama` to `Dramatic` in the column `kind` of the table `films`:

```
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';
```

Adjust temperature entries and reset precipitation to its default value in one row of the table `weather`:

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
  WHERE city = 'San Francisco' AND date = '2003-07-03';
```

Perform the same operation and return the updated entries:

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
  WHERE city = 'San Francisco' AND date = '2003-07-03'
  RETURNING temp_lo, temp_hi, prcp;
```

Use the alternative column-list syntax to do the same update:

```
UPDATE weather SET (temp_lo, temp_hi, prcp) = (temp_lo+1, temp_lo+15, DEFAULT)
  WHERE city = 'San Francisco' AND date = '2003-07-03';
```

Increment the sales count of the salesperson who manages the account for Acme Corporation, using the `FROM` clause syntax:

```
UPDATE employees SET sales_count = sales_count + 1 FROM accounts
  WHERE accounts.name = 'Acme Corporation'
  AND employees.id = accounts.sales_person;
```

Perform the same operation, using a sub-select in the `WHERE` clause:

```
UPDATE employees SET sales_count = sales_count + 1 WHERE id =
  (SELECT sales_person FROM accounts WHERE name = 'Acme Corporation');
```

Attempt to insert a new stock item along with the quantity of stock. If the item already exists, instead update the stock count of the existing item. To do this without failing the entire transaction, use savepoints.

```
BEGIN;
-- other operations
SAVEPOINT sp1;
INSERT INTO wines VALUES('Chateau Lafite 2003', '24');
-- Assume the above fails because of a unique key violation,
-- so now we issue these commands:
ROLLBACK TO sp1;
UPDATE wines SET stock = stock + 24 WHERE winename = 'Chateau Lafite 2003';
-- continue with other operations, and eventually
COMMIT;
```

Compatibility

This command conforms to the SQL standard, except that the `FROM` and `RETURNING` clauses are PostgreSQL extensions.

According to the standard, the column-list syntax should allow a list of columns to be assigned from a single row-valued expression, such as a sub-select:

```
UPDATE accounts SET (contact_last_name, contact_first_name) =
    (SELECT last_name, first_name FROM salesmen
     WHERE salesmen.id = accounts.sales_id);
```

This is not currently implemented — the source must be a list of independent expressions.

Some other database systems offer a `FROM` option in which the target table is supposed to be listed again within `FROM`. That is not how PostgreSQL interprets `FROM`. Be careful when porting applications that use this extension.

VACUUM

Name

VACUUM — garbage-collect and optionally analyze a database

Synopsis

```
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ table ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] ANALYZE [ table [ (column [, ...] ) ] ]
```

Description

VACUUM reclaims storage occupied by deleted tuples. In normal PostgreSQL operation, tuples that are deleted or obsoleted by an update are not physically removed from their table; they remain present until a VACUUM is done. Therefore it's necessary to do VACUUM periodically, especially on frequently-updated tables.

With no parameter, VACUUM processes every table in the current database. With a parameter, VACUUM processes only that table.

VACUUM ANALYZE performs a VACUUM and then an ANALYZE for each selected table. This is a handy combination form for routine maintenance scripts. See *ANALYZE* for more details about its processing.

Plain VACUUM (without FULL) simply reclaims space and makes it available for re-use. This form of the command can operate in parallel with normal reading and writing of the table, as an exclusive lock is not obtained. VACUUM FULL does more extensive processing, including moving of tuples across blocks to try to compact the table to the minimum number of disk blocks. This form is much slower and requires an exclusive lock on each table while it is being processed.

Parameters

FULL

Selects "full" vacuum, which may reclaim more space, but takes much longer and exclusively locks the table.

FREEZE

Selects aggressive "freezing" of tuples. Specifying FREEZE is equivalent to performing VACUUM with the vacuum_freeze_min_age parameter set to zero. The FREEZE option is deprecated and will be removed in a future release; set the parameter instead.

VERBOSE

Prints a detailed vacuum activity report for each table.

ANALYZE

Updates statistics used by the planner to determine the most efficient way to execute a query.

table

The name (optionally schema-qualified) of a specific table to vacuum. Defaults to all tables in the current database.

column

The name of a specific column to analyze. Defaults to all columns.

Outputs

When VERBOSE is specified, VACUUM emits progress messages to indicate which table is currently being processed. Various statistics about the tables are printed as well.

Notes

VACUUM cannot be executed inside a transaction block.

We recommend that active production databases be vacuumed frequently (at least nightly), in order to remove expired rows. After adding or deleting a large number of rows, it may be a good idea to issue a VACUUM ANALYZE command for the affected table. This will update the system catalogs with the results of all recent changes, and allow the PostgreSQL query planner to make better choices in planning queries.

The FULL option is not recommended for routine use, but may be useful in special cases. An example is when you have deleted most of the rows in a table and would like the table to physically shrink to occupy less disk space. VACUUM FULL will usually shrink the table more than a plain VACUUM would. The FULL option does not shrink indexes; a periodic REINDEX is still recommended. In fact, it is often faster to drop all indexes, VACUUM FULL, and recreate the indexes.

VACUUM causes a substantial increase in I/O traffic, which can cause poor performance for other active sessions. Therefore, it is sometimes advisable to use the cost-based vacuum delay feature. See Section 17.4.4 for details.

PostgreSQL includes an "autovacuum" facility which can automate routine vacuum maintenance. For more information about automatic and manual vacuuming, see Section 22.1.

Examples

The following is an example from running VACUUM on a table in the regression database:

```
regression=# VACUUM VERBOSE ANALYZE onek;
INFO:  vacuuming "public.onek"
INFO:  index "onek_unique1" now contains 1000 tuples in 14 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.01s/0.08u sec elapsed 0.18 sec.
INFO:  index "onek_unique2" now contains 1000 tuples in 16 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.07u sec elapsed 0.23 sec.
```

```
INFO:  index "onek_hundred" now contains 1000 tuples in 13 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.01s/0.08u sec elapsed 0.17 sec.
INFO:  index "onek_stringu1" now contains 1000 tuples in 48 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.01s/0.09u sec elapsed 0.59 sec.
INFO:  "onek": removed 3000 tuples in 108 pages
DETAIL:  CPU 0.01s/0.06u sec elapsed 0.07 sec.
INFO:  "onek": found 3000 removable, 1000 nonremovable tuples in 143 pages
DETAIL:  0 dead tuples cannot be removed yet.
There were 0 unused item pointers.
0 pages are entirely empty.
CPU 0.07s/0.39u sec elapsed 1.56 sec.
INFO:  analyzing "public.onek"
INFO:  "onek": 36 pages, 1000 rows sampled, 1000 estimated total rows
VACUUM
```

Compatibility

There is no `VACUUM` statement in the SQL standard.

See Also

vacuumdb, *Cost-Based Vacuum Delay*

VALUES

Name

VALUES — compute a set of rows

Synopsis

```
VALUES ( expression [, ...] ) [, ...]
    [ ORDER BY sort_expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]
```

Description

VALUES computes a row value or set of row values specified by value expressions. It is most commonly used to generate a "constant table" within a larger command, but it can be used on its own.

When more than one row is specified, all the rows must have the same number of elements. The data types of the resulting table's columns are determined by combining the explicit or inferred types of the expressions appearing in that column, using the same rules as for UNION (see Section 10.5).

Within larger commands, VALUES is syntactically allowed anywhere that SELECT is. Because it is treated like a SELECT by the grammar, it is possible to use the ORDER BY, LIMIT, and OFFSET clauses with a VALUES command.

Parameters

expression

A constant or expression to compute and insert at the indicated place in the resulting table (set of rows). In a VALUES list appearing at the top level of an INSERT, an *expression* can be replaced by DEFAULT to indicate that the destination column's default value should be inserted. DEFAULT cannot be used when VALUES appears in other contexts.

sort_expression

An expression or integer constant indicating how to sort the result rows. This expression may refer to the columns of the VALUES result as column1, column2, etc. For more details see *ORDER BY Clause*.

operator

A sorting operator. For details see *ORDER BY Clause*.

count

The maximum number of rows to return. For details see *LIMIT Clause*.

start

The number of rows to skip before starting to return rows. For details see *LIMIT Clause*.

Notes

`VALUES` lists with very large numbers of rows should be avoided, as you may encounter out-of-memory failures or poor performance. `VALUES` appearing within `INSERT` is a special case (because the desired column types are known from the `INSERT`'s target table, and need not be inferred by scanning the `VALUES` list), so it can handle larger lists than are practical in other contexts.

Examples

A bare `VALUES` command:

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

This will return a table of two columns and three rows. It's effectively equivalent to

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

More usually, `VALUES` is used within a larger SQL command. The most common use is in `INSERT`:

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

In the context of `INSERT`, entries of a `VALUES` list can be `DEFAULT` to indicate that the column default should be used here instead of specifying a value:

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes'),
    ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama', DEFAULT);
```

`VALUES` can also be used where a sub-`SELECT` might be written, for example in a `FROM` clause:

```
SELECT f.*
  FROM films f, (VALUES('MGM', 'Horror'), ('UA', 'Sci-Fi')) AS t (studio, kind)
  WHERE f.studio = t.studio AND f.kind = t.kind;

UPDATE employees SET salary = salary * v.increase
  FROM (VALUES(1, 200000, 1.2), (2, 400000, 1.4)) AS v (depno, target, increase)
  WHERE employees.depno = v.depno AND employees.sales >= v.target;
```

Note that an `AS` clause is required when `VALUES` is used in a `FROM` clause, just as is true for `SELECT`. It is not required that the `AS` clause specify names for all the columns, but it's good practice to do so. (The default column names for `VALUES` are `column1`, `column2`, etc in PostgreSQL, but these names might be different in other database systems.)

When `VALUES` is used in `INSERT`, the values are all automatically coerced to the data type of the corresponding destination column. When it's used in other contexts, it may be necessary to specify the correct data type. If the entries are all quoted literal constants, coercing the first is sufficient to determine the assumed type for all:

```
SELECT * FROM machines
WHERE ip_address IN (VALUES('192.168.0.1'::inet), ('192.168.0.10'), ('192.168.1.43'));
```

> **Tip:** For simple `IN` tests, it's better to rely on the list-of-scalars form of `IN` than to write a `VALUES` query as shown above. The list of scalars method requires less writing and is often more efficient.

Compatibility

`VALUES` conforms to the SQL standard, except that `LIMIT` and `OFFSET` are PostgreSQL extensions.

See Also

INSERT, *SELECT*

II. PostgreSQL Client Applications

This part contains reference information for PostgreSQL client applications and utilities. Not all of these commands are of general utility, some may require special privileges. The common feature of these applications is that they can be run on any host, independent of where the database server resides.

clusterdb

Name

`clusterdb` — cluster a PostgreSQL database

Synopsis

`clusterdb` [*`connection-option`*...] [--table | -t *`table`*] [*`dbname`*]
`clusterdb` [*`connection-option`*...] [--all | -a]

Description

clusterdb is a utility for reclustering tables in a PostgreSQL database. It finds tables that have previously been clustered, and clusters them again on the same index that was last used. Tables that have never been clustered are not affected.

clusterdb is a wrapper around the SQL command *CLUSTER*. There is no effective difference between clustering databases via this utility and via other methods for accessing the server.

Options

clusterdb accepts the following command-line arguments:

`-a`
`--all`

Cluster all databases.

`[-d]` *`dbname`*
`[--dbname]` *`dbname`*

Specifies the name of the database to be clustered. If this is not specified and `-a` (or `--all`) is not used, the database name is read from the environment variable `PGDATABASE`. If that is not set, the user name specified for the connection is used.

`-e`
`--echo`

Echo the commands that clusterdb generates and sends to the server.

`-q`
`--quiet`

Do not display a response.

`-t` *table*
`--table` *table*

Cluster *table* only.

clusterdb also accepts the following command-line arguments for connection parameters:

`-h` *host*
`--host` *host*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *port*
`--port` *port*

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

`-U` *username*
`--username` *username*

User name to connect as.

`-W`
`--password`

Force password prompt.

Environment

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 29.12).

Diagnostics

In case of difficulty, see *CLUSTER* and psql for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Examples

To cluster the database `test`:

```
$ clusterdb test
```

To cluster a single table `foo` in a database named `xyzzy`:

```
$ clusterdb --table foo xyzzy
```

See Also

CLUSTER

createdb

Name

`createdb` — create a new PostgreSQL database

Synopsis

`createdb` [*option*...] [*dbname*] [*description*]

Description

createdb creates a new PostgreSQL database.

Normally, the database user who executes this command becomes the owner of the new database. However a different owner can be specified via the `-O` option, if the executing user has appropriate privileges.

createdb is a wrapper around the SQL command *CREATE DATABASE*. There is no effective difference between creating databases via this utility and via other methods for accessing the server.

Options

createdb accepts the following command-line arguments:

dbname

Specifies the name of the database to be created. The name must be unique among all PostgreSQL databases in this cluster. The default is to create a database with the same name as the current system user.

description

Specifies a comment to be associated with the newly created database.

`-D` *tablespace*
`--tablespace` *tablespace*

Specifies the default tablespace for the database.

`-e`
`--echo`

Echo the commands that createdb generates and sends to the server.

`-E` *encoding*
`--encoding` *encoding*

Specifies the character encoding scheme to be used in this database. The character sets supported by the PostgreSQL server are described in Section 21.2.1.

`-O` *owner*
`--owner` *owner*

Specifies the database user who will own the new database.

`-q`
`--quiet`

Do not display a response.

`-T` *template*
`--template` *template*

Specifies the template database from which to build this database.

The options `-D`, `-E`, `-O`, and `-T` correspond to options of the underlying SQL command *CREATE DATABASE*; see there for more information about them.

createdb also accepts the following command-line arguments for connection parameters:

`-h` *host*
`--host` *host*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *port*
`--port` *port*

Specifies the TCP port or the local Unix domain socket file extension on which the server is listening for connections.

`-U` *username*
`--username` *username*

User name to connect as

`-W`
`--password`

Force password prompt.

Environment

`PGDATABASE`

If set, the name of the database to create, unless overridden on the command line.

`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters. `PGUSER` also determines the name of the database to create, if it is not specified on the command line or by `PGDATABASE`.

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 29.12).

Diagnostics

In case of difficulty, see *CREATE DATABASE* and psql for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Examples

To create the database `demo` using the default database server:

```
$ createdb demo
CREATE DATABASE
```

The response is the same as you would have gotten from running the `CREATE DATABASE` SQL command.

To create the database `demo` using the server on host `eden`, port 5000, using the `LATIN1` encoding scheme with a look at the underlying command:

```
$ createdb -p 5000 -h eden -E LATIN1 -e demo
CREATE DATABASE "demo" WITH ENCODING = 'LATIN1'
CREATE DATABASE
```

See Also

dropdb, *CREATE DATABASE*

createlang

Name

`createlang` — define a new PostgreSQL procedural language

Synopsis

`createlang` [*connection-option*...] *langname* [*dbname*]
`createlang` [*connection-option*...] --list | -l *dbname*

Description

createlang is a utility for adding a new programming language to a PostgreSQL database. createlang is just a wrapper around the *CREATE LANGUAGE* command.

Options

createlang accepts the following command-line arguments:

langname

Specifies the name of the procedural programming language to be defined.

`[-d]` *dbname*
`[--dbname]` *dbname*

Specifies to which database the language should be added. The default is to use the database with the same name as the current system user.

`-e`
`--echo`

Display SQL commands as they are executed.

`-l`
`--list`

Show a list of already installed languages in the target database.

createlang also accepts the following command-line arguments for connection parameters:

`-h` *host*
`--host` *host*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *port*
`--port` *port*

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

`-U` *username*
`--username` *username*

User name to connect as.

`-W`
`--password`

Force password prompt.

Environment

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 29.12).

Diagnostics

Most error messages are self-explanatory. If not, run createlang with the `--echo` option and see under the respective SQL command for details. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Notes

Use droplang to remove a language.

Examples

To install the language `pltcl` into the database `template1`:

```
$ createlang pltcl template1
```

Note that installing the language into `template1` will cause it to be automatically installed into subsequently-created databases as well.

See Also

droplang, *CREATE LANGUAGE*

createuser

Name

`createuser` — define a new PostgreSQL user account

Synopsis

`createuser` [*option*...] [*username*]

Description

createuser creates a new PostgreSQL user (or more precisely, a role). Only superusers and users with `CREATEROLE` privilege can create new users, so createuser must be invoked by someone who can connect as a superuser or a user with `CREATEROLE` privilege.

If you wish to create a new superuser, you must connect as a superuser, not merely with `CREATEROLE` privilege. Being a superuser implies the ability to bypass all access permission checks within the database, so superuserdom should not be granted lightly.

createuser is a wrapper around the SQL command *CREATE ROLE*. There is no effective difference between creating users via this utility and via other methods for accessing the server.

Options

createuser accepts the following command-line arguments:

username

Specifies the name of the PostgreSQL user to be created. This name must be different from all existing roles in this PostgreSQL installation.

`-s`
`--superuser`

The new user will be a superuser.

`-S`
`--no-superuser`

The new user will not be a superuser. This is the default.

`-d`
`--createdb`

The new user will be allowed to create databases.

`-D`
`--no-createdb`

The new user will not be allowed to create databases. This is the default.

`-r`
`--createrole`

The new user will be allowed to create new roles (that is, this user will have `CREATEROLE` privilege).

`-R`
`--no-createrole`

The new user will not be allowed to create new roles. This is the default.

`-l`
`--login`

The new user will be allowed to log in (that is, the user name can be used as the initial session user identifier). This is the default.

`-L`
`--no-login`

The new user will not be allowed to log in. (A role without login privilege is still useful as a means of managing database permissions.)

`-i`
`--inherit`

The new role will automatically inherit privileges of roles it is a member of. This is the default.

`-I`
`--no-inherit`

The new role will not automatically inherit privileges of roles it is a member of.

`-c` *number*
`--connection-limit` *number*

Set a maximum number of connections for the new user. The default is to set no limit.

`-P`
`--pwprompt`

If given, createuser will issue a prompt for the password of the new user. This is not necessary if you do not plan on using password authentication.

`-E`
`--encrypted`

Encrypts the user's password stored in the database. If not specified, the default password behavior is used.

`-N`
`--unencrypted`

Does not encrypt the user's password stored in the database. If not specified, the default password behavior is used.

`-e`
`--echo`

Echo the commands that createuser generates and sends to the server.

`-q`
`--quiet`

Do not display a response.

You will be prompted for a name and other missing information if it is not specified on the command line.

createuser also accepts the following command-line arguments for connection parameters:

`-h` *host*
`--host` *host*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *port*
`--port` *port*

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

`-U` *username*
`--username` *username*

User name to connect as (not the user name to create).

`-W`
`--password`

Force password prompt (to connect to the server, not for the password of the new user).

Environment

`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 29.12).

Diagnostics

In case of difficulty, see *CREATE ROLE* and psql for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Examples

To create a user `joe` on the default database server:

```
$ createuser joe
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n) n
CREATE USER
```

To create the same user `joe` using the server on host `eden`, port 5000, avoiding the prompts and taking a look at the underlying command:

```
$ createuser -h eden -p 5000 -S -D -R -e joe
CREATE ROLE joe NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
CREATE ROLE
```

To create the user `joe` as a superuser, and assign a password immediately:

```
$ createuser -P -s -e joe
Enter password for new role: xyzzy
Enter it again: xyzzy
CREATE ROLE joe PASSWORD 'xyzzy' SUPERUSER CREATEDB CREATEROLE INHERIT LOGIN;
CREATE ROLE
```

In the above example, the new password isn't actually echoed when typed, but we show what was typed for clarity. However the password *will* appear in the echoed command, as illustrated — so you don't want to use `-e` when assigning a password, if anyone else can see your screen.

See Also

dropuser, *CREATE ROLE*

dropdb

Name

`dropdb` — remove a PostgreSQL database

Synopsis

`dropdb` [*option*...] *dbname*

Description

dropdb destroys an existing PostgreSQL database. The user who executes this command must be a database superuser or the owner of the database.

dropdb is a wrapper around the SQL command *DROP DATABASE*. There is no effective difference between dropping databases via this utility and via other methods for accessing the server.

Options

dropdb accepts the following command-line arguments:

dbname

Specifies the name of the database to be removed.

`-e`
`--echo`

Echo the commands that dropdb generates and sends to the server.

`-i`
`--interactive`

Issues a verification prompt before doing anything destructive.

`-q`
`--quiet`

Do not display a response.

dropdb also accepts the following command-line arguments for connection parameters:

`-h` *host*
`--host` *host*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *port*
`--port` *port*

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

`-U` *username*
`--username` *username*

User name to connect as

`-W`
`--password`

Force password prompt.

Environment

`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 29.12).

Diagnostics

In case of difficulty, see *DROP DATABASE* and psql for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Examples

To destroy the database `demo` on the default database server:

```
$ dropdb demo
DROP DATABASE
```

To destroy the database `demo` using the server on host `eden`, port 5000, with verification and a peek at the underlying command:

```
$ dropdb -p 5000 -h eden -i -e demo
Database "demo" will be permanently deleted.
Are you sure? (y/n) y
DROP DATABASE "demo"
```

```
DROP DATABASE
```

See Also

createdb, *DROP DATABASE*

droplang

Name

`droplang` — remove a PostgreSQL procedural language

Synopsis

`droplang` [*connection-option*...] *langname* [*dbname*]
`droplang` [*connection-option*...] --list | -l *dbname*

Description

droplang is a utility for removing an existing programming language from a PostgreSQL database. droplang can drop any procedural language, even those not supplied by the PostgreSQL distribution.

Although backend programming languages can be removed directly using several SQL commands, it is recommended to use droplang because it performs a number of checks and is much easier to use. See *DROP LANGUAGE* for more.

Options

droplang accepts the following command line arguments:

langname

Specifies the name of the backend programming language to be removed.

`[-d]` *dbname*
`[--dbname]` *dbname*

Specifies from which database the language should be removed. The default is to use the database with the same name as the current system user.

`-e`
`--echo`

Display SQL commands as they are executed.

`-l`
`--list`

Show a list of already installed languages in the target database.

droplang also accepts the following command line arguments for connection parameters:

`-h` *host*
`--host` *host*

Specifies the host name of the machine on which the server is running. If host begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *port*
`--port` *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the server is listening for connections.

`-U` *username*
`--username` *username*

User name to connect as

`-W`
`--password`

Force password prompt.

Environment

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 29.12).

Diagnostics

Most error messages are self-explanatory. If not, run droplang with the `--echo` option and see under the respective SQL command for details. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Notes

Use createlang to add a language.

Examples

To remove the language `pltcl`:

```
$ droplang pltcl dbname
```

See Also

createlang, *DROP LANGUAGE*

dropuser

Name

`dropuser` — remove a PostgreSQL user account

Synopsis

`dropuser` [*option*...] [*username*]

Description

dropuser removes an existing PostgreSQL user. Only superusers and users with the `CREATEROLE` privilege can remove PostgreSQL users. (To remove a superuser, you must yourself be a superuser.)

dropuser is a wrapper around the SQL command *DROP ROLE*. There is no effective difference between dropping users via this utility and via other methods for accessing the server.

Options

dropuser accepts the following command-line arguments:

username

Specifies the name of the PostgreSQL user to be removed. You will be prompted for a name if none is specified on the command line.

`-e`
`--echo`

Echo the commands that dropuser generates and sends to the server.

`-i`
`--interactive`

Prompt for confirmation before actually removing the user.

`-q`
`--quiet`

Do not display a response.

dropuser also accepts the following command-line arguments for connection parameters:

`-h` *host*
`--host` *host*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *port*
`--port` *port*

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

`-U` *username*
`--username` *username*

User name to connect as (not the user name to drop)

`-W`
`--password`

Force password prompt (to connect to the server, not for the password of the user to be dropped).

Environment

`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 29.12).

Diagnostics

In case of difficulty, see *DROP ROLE* and psql for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Examples

To remove user `joe` from the default database server:

```
$ dropuser joe
DROP ROLE
```

To remove user `joe` using the server on host `eden`, port 5000, with verification and a peek at the underlying command:

```
$ dropuser -p 5000 -h eden -i -e joe
Role "joe" will be permanently removed.
Are you sure? (y/n) y
DROP ROLE "joe"
DROP ROLE
```

See Also

createuser, *DROP ROLE*

ecpg

Name

ecpg — embedded SQL C preprocessor

Synopsis

ecpg [*option*...] *file*...

Description

ecpg is the embedded SQL preprocessor for C programs. It converts C programs with embedded SQL statements to normal C code by replacing the SQL invocations with special function calls. The output files can then be processed with any C compiler tool chain.

ecpg will convert each input file given on the command line to the corresponding C output file. Input files preferably have the extension .pgc, in which case the extension will be replaced by .c to determine the output file name. If the extension of the input file is not .pgc, then the output file name is computed by appending .c to the full file name. The output file name can also be overridden using the -o option.

This reference page does not describe the embedded SQL language. See Chapter 31 for more information on that topic.

Options

ecpg accepts the following command-line arguments:

-c

Automatically generate certain C code from SQL code. Currently, this works for EXEC SQL TYPE.

-C *mode*

Set a compatibility mode. *mode* may be INFORMIX or INFORMIX_SE.

-D *symbol*

Define a C preprocessor symbol.

-i

Parse system include files as well.

-I *directory*

Specify an additional include path, used to find files included via EXEC SQL INCLUDE. Defaults are . (current directory), /usr/local/include, the PostgreSQL include directory which is defined at compile time (default: /usr/local/pgsql/include), and /usr/include, in that order.

`-o` *`filename`*

Specifies that `ecpg` should write all its output to the given *`filename`*.

`-r` *`option`*

Selects a run-time behavior. Currently, *`option`* can only be `no_indicator`.

`-t`

Turn on autocommit of transactions. In this mode, each SQL command is automatically committed unless it is inside an explicit transaction block. In the default mode, commands are committed only when `EXEC SQL COMMIT` is issued.

`-v`

Print additional information including the version and the include path.

`--help`

Show a brief summary of the command usage, then exit.

`--version`

Output version information, then exit.

Notes

When compiling the preprocessed C code files, the compiler needs to be able to find the ECPG header files in the PostgreSQL include directory. Therefore, one might have to use the `-I` option when invoking the compiler (e.g., `-I/usr/local/pgsql/include`).

Programs using C code with embedded SQL have to be linked against the `libecpg` library, for example using the linker options `-L/usr/local/pgsql/lib -lecpg`.

The value of either of these directories that is appropriate for the installation can be found out using pg_config.

Examples

If you have an embedded SQL C source file named `prog1.pgc`, you can create an executable program using the following sequence of commands:

```
ecpg prog1.pgc
cc -I/usr/local/pgsql/include -c prog1.c
cc -o prog1 prog1.o -L/usr/local/pgsql/lib -lecpg
```

pg_config

Name

`pg_config` — retrieve information about the installed version of PostgreSQL

Synopsis

`pg_config` [*option*...]

Description

The pg_config utility prints configuration parameters of the currently installed version of PostgreSQL. It is intended, for example, to be used by software packages that want to interface to PostgreSQL to facilitate finding the required header files and libraries.

Options

To use pg_config, supply one or more of the following options:

`--bindir`

Print the location of user executables. Use this, for example, to find the `psql` program. This is normally also the location where the `pg_config` program resides.

`--docdir`

Print the location of documentation files. (This will be an empty string if `--without-docdir` was specified when PostgreSQL was built.)

`--includedir`

Print the location of C header files of the client interfaces.

`--pkgincludedir`

Print the location of other C header files.

`--includedir-server`

Print the location of C header files for server programming.

`--libdir`

Print the location of object code libraries.

`--pkglibdir`

Print the location of dynamically loadable modules, or where the server would search for them. (Other architecture-dependent data files may also be installed in this directory.)

`--localedir`

Print the location of locale support files. (This will be an empty string if locale support was not configured when PostgreSQL was built.)

`--mandir`

Print the location of manual pages.

`--sharedir`

Print the location of architecture-independent support files.

`--sysconfdir`

Print the location of system-wide configuration files.

`--pgxs`

Print the location of extension makefiles.

`--configure`

Print the options that were given to the `configure` script when PostgreSQL was configured for building. This can be used to reproduce the identical configuration, or to find out with what options a binary package was built. (Note however that binary packages often contain vendor-specific custom patches.) See also the examples below.

`--cc`

Print the value of the `CC` variable that was used for building PostgreSQL. This shows the C compiler used.

`--cppflags`

Print the value of the `CPPFLAGS` variable that was used for building PostgreSQL. This shows C compiler switches needed at preprocessing time (typically, `-I` switches).

`--cflags`

Print the value of the `CFLAGS` variable that was used for building PostgreSQL. This shows C compiler switches.

`--cflags_sl`

Print the value of the `CFLAGS_SL` variable that was used for building PostgreSQL. This shows extra C compiler switches used for building shared libraries.

`--ldflags`

Print the value of the `LDFLAGS` variable that was used for building PostgreSQL. This shows linker switches.

`--ldflags_sl`

Print the value of the `LDFLAGS_SL` variable that was used for building PostgreSQL. This shows linker switches used for building shared libraries.

`--libs`

Print the value of the `LIBS` variable that was used for building PostgreSQL. This normally contains `-l` switches for external libraries linked into PostgreSQL.

`--version`

Print the version of PostgreSQL.

If more than one option is given, the information is printed in that order, one item per line. If no options are given, all available information is printed, with labels.

Notes

The option `--includedir-server` was new in PostgreSQL 7.2. In prior releases, the server include files were installed in the same location as the client headers, which could be queried with the option `--includedir`. To make your package handle both cases, try the newer option first and test the exit status to see whether it succeeded.

The options `--docdir`, `--pkgincludedir`, `--localedir`, `--mandir`, `--sharedir`, `--sysconfdir`, `--cc`, `--cppflags`, `--cflags`, `--cflags_sl`, `--ldflags`, `--ldflags_sl`, and `--libs` are new in PostgreSQL 8.1.

In releases prior to PostgreSQL 7.1, before `pg_config` came to be, a method for finding the equivalent configuration information did not exist.

Example

To reproduce the build configuration of the current PostgreSQL installation, run the following command:

```
eval ./configure `pg_config --configure`
```

The output of `pg_config --configure` contains shell quotation marks so arguments with spaces are represented correctly. Therefore, using `eval` is required for proper results.

History

The `pg_config` utility first appeared in PostgreSQL 7.1.

pg_dump

Name

pg_dump — extract a PostgreSQL database into a script file or other archive file

Synopsis

pg_dump [*option*...] [*dbname*]

Description

pg_dump is a utility for backing up a PostgreSQL database. It makes consistent backups even if the database is being used concurrently. pg_dump does not block other users accessing the database (readers or writers).

Dumps can be output in script or archive file formats. Script dumps are plain-text files containing the SQL commands required to reconstruct the database to the state it was in at the time it was saved. To restore from such a script, feed it to psql. Script files can be used to reconstruct the database even on other machines and other architectures; with some modifications even on other SQL database products.

The alternative archive file formats must be used with pg_restore to rebuild the database. They allow pg_restore to be selective about what is restored, or even to reorder the items prior to being restored. The archive file formats are designed to be portable across architectures.

When used with one of the archive file formats and combined with pg_restore, pg_dump provides a flexible archival and transfer mechanism. pg_dump can be used to backup an entire database, then pg_restore can be used to examine the archive and/or select which parts of the database are to be restored. The most flexible output file format is the "custom" format (-Fc). It allows for selection and reordering of all archived items, and is compressed by default. The tar format (-Ft) is not compressed and it is not possible to reorder data when loading, but it is otherwise quite flexible; moreover, it can be manipulated with standard Unix tools such as tar.

While running pg_dump, one should examine the output for any warnings (printed on standard error), especially in light of the limitations listed below.

Options

The following command-line options control the content and format of the output.

dbname

Specifies the name of the database to be dumped. If this is not specified, the environment variable PGDATABASE is used. If that is not set, the user name specified for the connection is used.

`-a`
`--data-only`

Dump only the data, not the schema (data definitions).

This option is only meaningful for the plain-text format. For the archive formats, you may specify the option when you call `pg_restore`.

`-b`
`--blobs`

Include large objects in the dump. This is the default behavior except when `--schema`, `--table`, or `--schema-only` is specified, so the `-b` switch is only useful to add large objects to selective dumps.

`-c`
`--clean`

Output commands to clean (drop) database objects prior to (the commands for) creating them.

This option is only meaningful for the plain-text format. For the archive formats, you may specify the option when you call `pg_restore`.

`-C`
`--create`

Begin the output with a command to create the database itself and reconnect to the created database. (With a script of this form, it doesn't matter which database you connect to before running the script.)

This option is only meaningful for the plain-text format. For the archive formats, you may specify the option when you call `pg_restore`.

`-d`
`--inserts`

Dump data as `INSERT` commands (rather than `COPY`). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-PostgreSQL databases. Also, since this option generates a separate command for each row, an error in reloading a row causes only that row to be lost rather than the entire table contents. Note that the restore may fail altogether if you have rearranged column order. The `-D` option is safe against column order changes, though even slower.

`-D`
`--column-inserts`
`--attribute-inserts`

Dump data as `INSERT` commands with explicit column names (`INSERT INTO` *`table`* `(`*`column`*`, ...) VALUES ...`). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-PostgreSQL databases. Also, since this option generates a separate command for each row, an error in reloading a row causes only that row to be lost rather than the entire table contents.

`-E` *`encoding`*
`--encoding=`*`encoding`*

Create the dump in the specified character set encoding. By default, the dump is created in the database encoding. (Another way to get the same result is to set the `PGCLIENTENCODING` environment variable to the desired dump encoding.)

`-f` *file*
`--file=`*file*

Send output to the specified file. If this is omitted, the standard output is used.

`-F` *format*
`--format=`*format*

Selects the format of the output. *format* can be one of the following:

`p`
`plain`

Output a plain-text SQL script file (the default).

`c`
`custom`

Output a custom archive suitable for input into pg_restore. This is the most flexible format in that it allows reordering of loading data as well as object definitions. This format is also compressed by default.

`t`
`tar`

Output a `tar` archive suitable for input into pg_restore. Using this archive format allows reordering and/or exclusion of database objects at the time the database is restored. It is also possible to limit which data is reloaded at restore time.

`-i`
`--ignore-version`

Ignore version mismatch between pg_dump and the database server.

pg_dump can dump from servers running previous releases of PostgreSQL, but very old versions are not supported anymore (currently, those prior to 7.0). Dumping from a server newer than pg_dump is likely not to work at all. Use this option if you need to override the version check (and if pg_dump then fails, don't say you weren't warned).

`-n` *schema*
`--schema=`*schema*

Dump only schemas matching *schema*; this selects both the schema itself, and all its contained objects. When this option is not specified, all non-system schemas in the target database will be dumped. Multiple schemas can be selected by writing multiple `-n` switches. Also, the *schema* parameter is interpreted as a pattern according to the same rules used by psql's `\d` commands (see *Patterns*), so multiple schemas can also be selected by writing wildcard characters in the pattern. When using wildcards, be careful to quote the pattern if needed to prevent the shell from expanding the wildcards.

Note: When `-n` is specified, pg_dump makes no attempt to dump any other database objects that the selected schema(s) may depend upon. Therefore, there is no guarantee that the results of a specific-schema dump can be successfully restored by themselves into a clean database.

> **Note:** Non-schema objects such as blobs are not dumped when `-n` is specified. You can add blobs back to the dump with the `--blobs` switch.

`-N` *schema*
`--exclude-schema=`*schema*

Do not dump any schemas matching the *schema* pattern. The pattern is interpreted according to the same rules as for `-n`. `-N` can be given more than once to exclude schemas matching any of several patterns.

When both `-n` and `-N` are given, the behavior is to dump just the schemas that match at least one `-n` switch but no `-N` switches. If `-N` appears without `-n`, then schemas matching `-N` are excluded from what is otherwise a normal dump.

`-o`
`--oids`

Dump object identifiers (OIDs) as part of the data for every table. Use this option if your application references the OID columns in some way (e.g., in a foreign key constraint). Otherwise, this option should not be used.

`-O`
`--no-owner`

Do not output commands to set ownership of objects to match the original database. By default, pg_dump issues `ALTER OWNER` or `SET SESSION AUTHORIZATION` statements to set ownership of created database objects. These statements will fail when the script is run unless it is started by a superuser (or the same user that owns all of the objects in the script). To make a script that can be restored by any user, but will give that user ownership of all the objects, specify `-O`.

This option is only meaningful for the plain-text format. For the archive formats, you may specify the option when you call `pg_restore`.

`-R`
`--no-reconnect`

This option is obsolete but still accepted for backwards compatibility.

`-s`
`--schema-only`

Dump only the object definitions (schema), not data.

`-S` *username*
`--superuser=`*username*

Specify the superuser user name to use when disabling triggers. This is only relevant if `--disable-triggers` is used. (Usually, it's better to leave this out, and instead start the resulting script as superuser.)

`-t` *table*
`--table=`*table*

Dump only tables (or views or sequences) matching *table*. Multiple tables can be selected by writing multiple `-t` switches. Also, the *table* parameter is interpreted as a pattern according to the same rules used by psql's `\d` commands (see *Patterns*), so multiple tables can also be selected by writing

wildcard characters in the pattern. When using wildcards, be careful to quote the pattern if needed to prevent the shell from expanding the wildcards.

The `-n` and `-N` switches have no effect when `-t` is used, because tables selected by `-t` will be dumped regardless of those switches, and non-table objects will not be dumped.

> **Note:** When `-t` is specified, pg_dump makes no attempt to dump any other database objects that the selected table(s) may depend upon. Therefore, there is no guarantee that the results of a specific-table dump can be successfully restored by themselves into a clean database.

> **Note:** The behavior of the `-t` switch is not entirely upward compatible with pre-8.2 PostgreSQL versions. Formerly, writing `-t tab` would dump all tables named `tab`, but now it just dumps whichever one is visible in your default search path. To get the old behavior you can write `-t 'tab'`. Also, you must write something like `-t sch.tab` to select a table in a particular schema, rather than the old locution of `-n sch -t tab`.

`-T` *table*
`--exclude-table=`*table*

Do not dump any tables matching the *table* pattern. The pattern is interpreted according to the same rules as for `-t`. `-T` can be given more than once to exclude tables matching any of several patterns.

When both `-t` and `-T` are given, the behavior is to dump just the tables that match at least one `-t` switch but no `-T` switches. If `-T` appears without `-t`, then tables matching `-T` are excluded from what is otherwise a normal dump.

`-v`
`--verbose`

Specifies verbose mode. This will cause pg_dump to output detailed object comments and start/stop times to the dump file, and progress messages to standard error.

`-x`
`--no-privileges`
`--no-acl`

Prevent dumping of access privileges (grant/revoke commands).

`--disable-dollar-quoting`

This option disables the use of dollar quoting for function bodies, and forces them to be quoted using SQL standard string syntax.

`--disable-triggers`

This option is only relevant when creating a data-only dump. It instructs pg_dump to include commands to temporarily disable triggers on the target tables while the data is reloaded. Use this if you have referential integrity checks or other triggers on the tables that you do not want to invoke during data reload.

Presently, the commands emitted for `--disable-triggers` must be done as superuser. So, you should also specify a superuser name with `-S`, or preferably be careful to start the resulting script as a superuser.

This option is only meaningful for the plain-text format. For the archive formats, you may specify the option when you call `pg_restore`.

`--use-set-session-authorization`

Output SQL-standard `SET SESSION AUTHORIZATION` commands instead of `ALTER OWNER` commands to determine object ownership. This makes the dump more standards compatible, but depending on the history of the objects in the dump, may not restore properly. Also, a dump using `SET SESSION AUTHORIZATION` will certainly require superuser privileges to restore correctly, whereas `ALTER OWNER` requires lesser privileges.

`-Z` *`0..9`*
`--compress=`*`0..9`*

Specify the compression level to use in archive formats that support compression. (Currently only the custom archive format supports compression.)

The following command-line options control the database connection parameters.

`-h` *`host`*
`--host=`*`host`*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket. The default is taken from the `PGHOST` environment variable, if set, else a Unix domain socket connection is attempted.

`-p` *`port`*
`--port=`*`port`*

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections. Defaults to the `PGPORT` environment variable, if set, or a compiled-in default.

`-U` *`username`*

Connect as the given user

`-W`

Force a password prompt. This should happen automatically if the server requires password authentication.

Environment

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters.

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 29.12).

Diagnostics

pg_dump internally executes `SELECT` statements. If you have problems running pg_dump, make sure you are able to select information from the database using, for example, psql. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Notes

If your database cluster has any local additions to the `template1` database, be careful to restore the output of pg_dump into a truly empty database; otherwise you are likely to get errors due to duplicate definitions of the added objects. To make an empty database without any local additions, copy from `template0` not `template1`, for example:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

pg_dump has a few limitations:

- When a data-only dump is chosen and the option `--disable-triggers` is used, pg_dump emits commands to disable triggers on user tables before inserting the data and commands to re-enable them after the data has been inserted. If the restore is stopped in the middle, the system catalogs may be left in the wrong state.

Members of tar archives are limited to a size less than 8 GB. (This is an inherent limitation of the tar file format.) Therefore this format cannot be used if the textual representation of any one table exceeds that size. The total size of a tar archive and any of the other output formats is not limited, except possibly by the operating system.

The dump file produced by pg_dump does not contain the statistics used by the optimizer to make query planning decisions. Therefore, it is wise to run `ANALYZE` after restoring from a dump file to ensure good performance.

Because pg_dump is used to transfer data to newer versions of PostgreSQL, the output of pg_dump can be loaded into newer PostgreSQL databases. It also can read older PostgreSQL databases. However, it usually cannot read newer PostgreSQL databases or produce dump output that can be loaded into older database versions. To do this, manual editing of the dump file might be required.

Examples

To dump a database called `mydb` into a SQL-script file:

```
$ pg_dump mydb > db.sql
```

To reload such a script into a (freshly created) database named `newdb`:

```
$ psql -d newdb -f db.sql
```

To dump a database into a custom-format archive file:

```
$ pg_dump -Fc mydb > db.dump
```

To reload an archive file into a (freshly created) database named `newdb`:

```
$ pg_restore -d newdb db.dump
```

To dump a single table named `mytab`:

```
$ pg_dump -t mytab mydb > db.sql
```

To dump all tables whose names start with `emp` in the `detroit` schema, except for the table named `employee_log`:

```
$ pg_dump -t 'detroit.emp*' -T detroit.employee_log mydb > db.sql
```

To dump all schemas whose names start with `east` or `west` and end in `gsm`, excluding any schemas whose names contain the word `test`:

```
$ pg_dump -n 'east*gsm' -n 'west*gsm' -N '*test*' mydb > db.sql
```

The same, using regular expression notation to consolidate the switches:

```
$ pg_dump -n '(east|west)*gsm' -N '*test*' mydb > db.sql
```

To dump all database objects except for tables whose names begin with `ts_`:

```
$ pg_dump -T 'ts_*' mydb > db.sql
```

To specify an upper-case or mixed-case name in `-t` and related switches, you need to double-quote the name; else it will be folded to lower case (see *Patterns*). But double quotes are special to the shell, so in turn they must be quoted. Thus, to dump a single table with a mixed-case name, you need something like

```
$ pg_dump -t '"MixedCaseName"' mydb > mytab.sql
```

History

The pg_dump utility first appeared in Postgres95 release 0.02. The non-plain-text output formats were introduced in PostgreSQL release 7.1.

See Also

pg_dumpall, pg_restore, psql

pg_dumpall

Name

pg_dumpall — extract a PostgreSQL database cluster into a script file

Synopsis

pg_dumpall [*option*...]

Description

pg_dumpall is a utility for writing out ("dumping") all PostgreSQL databases of a cluster into one script file. The script file contains SQL commands that can be used as input to psql to restore the databases. It does this by calling pg_dump for each database in a cluster. pg_dumpall also dumps global objects that are common to all databases. (pg_dump does not save these objects.) This currently includes information about database users and groups, and access permissions that apply to databases as a whole.

Since pg_dumpall reads tables from all databases you will most likely have to connect as a database superuser in order to produce a complete dump. Also you will need superuser privileges to execute the saved script in order to be allowed to add users and groups, and to create databases.

The SQL script will be written to the standard output. Shell operators should be used to redirect it into a file.

pg_dumpall needs to connect several times to the PostgreSQL server (once per database). If you use password authentication it is likely to ask for a password each time. It is convenient to have a ~/.pgpass file in such cases. See Section 29.13 for more information.

Options

The following command-line options control the content and format of the output.

-a
--data-only

Dump only the data, not the schema (data definitions).

-c
--clean

Include SQL commands to clean (drop) databases before recreating them. DROP commands for roles and tablespaces are added as well.

`-d`
`--inserts`

Dump data as `INSERT` commands (rather than `COPY`). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-PostgreSQL databases. Note that the restore may fail altogether if you have rearranged column order. The `-D` option is safer, though even slower.

`-D`
`--column-inserts`
`--attribute-inserts`

Dump data as `INSERT` commands with explicit column names (`INSERT INTO` *`table`* `(`*`column`*`, ...) VALUES ...`). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-PostgreSQL databases.

`-g`
`--globals-only`

Dump only global objects (roles and tablespaces), no databases.

`-i`
`--ignore-version`

Ignore version mismatch between pg_dumpall and the database server.

pg_dumpall can handle databases from previous releases of PostgreSQL, but very old versions are not supported anymore (currently prior to 7.0). Use this option if you need to override the version check (and if pg_dumpall then fails, don't say you weren't warned).

`-o`
`--oids`

Dump object identifiers (OIDs) as part of the data for every table. Use this option if your application references the OID columns in some way (e.g., in a foreign key constraint). Otherwise, this option should not be used.

`-O`
`--no-owner`

Do not output commands to set ownership of objects to match the original database. By default, pg_dumpall issues `ALTER OWNER` or `SET SESSION AUTHORIZATION` statements to set ownership of created schema elements. These statements will fail when the script is run unless it is started by a superuser (or the same user that owns all of the objects in the script). To make a script that can be restored by any user, but will give that user ownership of all the objects, specify `-O`.

`-s`
`--schema-only`

Dump only the object definitions (schema), not data.

`-S` *`username`*
`--superuser=`*`username`*

Specify the superuser user name to use when disabling triggers. This is only relevant if `--disable-triggers` is used. (Usually, it's better to leave this out, and instead start the resulting script as superuser.)

`-v`
`--verbose`

Specifies verbose mode. This will cause pg_dumpall to output start/stop times to the dump file, and progress messages to standard error. It will also enable verbose output in pg_dump.

`-x`
`--no-privileges`
`--no-acl`

Prevent dumping of access privileges (grant/revoke commands).

`--disable-dollar-quoting`

This option disables the use of dollar quoting for function bodies, and forces them to be quoted using SQL standard string syntax.

`--disable-triggers`

This option is only relevant when creating a data-only dump. It instructs pg_dumpall to include commands to temporarily disable triggers on the target tables while the data is reloaded. Use this if you have referential integrity checks or other triggers on the tables that you do not want to invoke during data reload.

Presently, the commands emitted for `--disable-triggers` must be done as superuser. So, you should also specify a superuser name with `-S`, or preferably be careful to start the resulting script as a superuser.

`--use-set-session-authorization`

Output SQL-standard `SET SESSION AUTHORIZATION` commands instead of `ALTER OWNER` commands to determine object ownership. This makes the dump more standards compatible, but depending on the history of the objects in the dump, may not restore properly.

The following command-line options control the database connection parameters.

-h *host*

Specifies the host name of the machine on which the database server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket. The default is taken from the `PGHOST` environment variable, if set, else a Unix domain socket connection is attempted.

-p *port*

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections. Defaults to the `PGPORT` environment variable, if set, or a compiled-in default.

-U *username*

Connect as the given user.

-W

Force a password prompt. This should happen automatically if the server requires password authentication.

Environment

`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 29.12).

Notes

Since pg_dumpall calls pg_dump internally, some diagnostic messages will refer to pg_dump.

Once restored, it is wise to run `ANALYZE` on each database so the optimizer has useful statistics. You can also run `vacuumdb -a -z` to analyze all databases.

pg_dumpall requires all needed tablespace directories to exist before the restore or database creation will fail for databases in non-default locations.

Examples

To dump all databases:

```
$ pg_dumpall > db.out
```

To reload this database use, for example:

```
$ psql -f db.out postgres
```

(It is not important to which database you connect here since the script file created by pg_dumpall will contain the appropriate commands to create and connect to the saved databases.)

See Also

Check pg_dump for details on possible error conditions.

pg_restore

Name

pg_restore — restore a PostgreSQL database from an archive file created by pg_dump

Synopsis

pg_restore [*option*...] [*filename*]

Description

pg_restore is a utility for restoring a PostgreSQL database from an archive created by pg_dump in one of the non-plain-text formats. It will issue the commands necessary to reconstruct the database to the state it was in at the time it was saved. The archive files also allow pg_restore to be selective about what is restored, or even to reorder the items prior to being restored. The archive files are designed to be portable across architectures.

pg_restore can operate in two modes. If a database name is specified, the archive is restored directly into the database. Otherwise, a script containing the SQL commands necessary to rebuild the database is created and written to a file or standard output. The script output is equivalent to the plain text output format of pg_dump. Some of the options controlling the output are therefore analogous to pg_dump options.

Obviously, pg_restore cannot restore information that is not present in the archive file. For instance, if the archive was made using the "dump data as INSERT commands" option, pg_restore will not be able to load the data using COPY statements.

Options

pg_restore accepts the following command line arguments.

filename

Specifies the location of the archive file to be restored. If not specified, the standard input is used.

-a
--data-only

Restore only the data, not the schema (data definitions).

-c
--clean

Clean (drop) database objects before recreating them.

`-C`
`--create`

Create the database before restoring into it. (When this option is used, the database named with `-d` is used only to issue the initial `CREATE DATABASE` command. All data is restored into the database name that appears in the archive.)

`-d` *dbname*
`--dbname=`*dbname*

Connect to database *dbname* and restore directly into the database.

`-e`
`--exit-on-error`

Exit if an error is encountered while sending SQL commands to the database. The default is to continue and to display a count of errors at the end of the restoration.

`-f` *filename*
`--file=`*filename*

Specify output file for generated script, or for the listing when used with `-l`. Default is the standard output.

`-F` *format*
`--format=`*format*

Specify format of the archive. It is not necessary to specify the format, since pg_restore will determine the format automatically. If specified, it can be one of the following:

`t`
`tar`

The archive is a `tar` archive. Using this archive format allows reordering and/or exclusion of schema elements at the time the database is restored. It is also possible to limit which data is reloaded at restore time.

`c`
`custom`

The archive is in the custom format of pg_dump. This is the most flexible format in that it allows reordering of data load as well as schema elements. This format is also compressed by default.

`-i`
`--ignore-version`

Ignore database version checks.

`-I` *index*
`--index=`*index*

Restore definition of named index only.

`-l`
`--list`

List the contents of the archive. The output of this operation can be used with the `-L` option to restrict and reorder the items that are restored.

`-L` *list-file*
`--use-list=`*list-file*

Restore elements in *list-file* only, and in the order they appear in the file. Lines can be moved and may also be commented out by placing a `;` at the start of the line. (See below for examples.)

`-n` *namespace*
`--schema=`*schema*

Restore only objects that are in the named schema. This can be combined with the `-t` option to restore just a specific table.

`-O`
`--no-owner`

Do not output commands to set ownership of objects to match the original database. By default, pg_restore issues `ALTER OWNER` or `SET SESSION AUTHORIZATION` statements to set ownership of created schema elements. These statements will fail unless the initial connection to the database is made by a superuser (or the same user that owns all of the objects in the script). With `-O`, any user name can be used for the initial connection, and this user will own all the created objects.

`-P` *function-name(argtype [, ...])*
`--function=`*function-name(argtype [, ...])*

Restore the named function only. Be careful to spell the function name and arguments exactly as they appear in the dump file's table of contents.

`-R`
`--no-reconnect`

This option is obsolete but still accepted for backwards compatibility.

`-s`
`--schema-only`

Restore only the schema (data definitions), not the data (table contents). Sequence current values will not be restored, either. (Do not confuse this with the `--schema` option, which uses the word "schema" in a different meaning.)

`-S` *username*
`--superuser=`*username*

Specify the superuser user name to use when disabling triggers. This is only relevant if `--disable-triggers` is used.

`-t` *table*
`--table=`*table*

Restore definition and/or data of named table only.

`-T` *trigger*
`--trigger=`*trigger*

Restore named trigger only.

`-v`
`--verbose`

Specifies verbose mode.

`-x`
`--no-privileges`
`--no-acl`

Prevent restoration of access privileges (grant/revoke commands).

`--disable-triggers`

This option is only relevant when performing a data-only restore. It instructs pg_restore to execute commands to temporarily disable triggers on the target tables while the data is reloaded. Use this if you have referential integrity checks or other triggers on the tables that you do not want to invoke during data reload.

Presently, the commands emitted for `--disable-triggers` must be done as superuser. So, you should also specify a superuser name with `-S`, or preferably run pg_restore as a PostgreSQL superuser.

`--use-set-session-authorization`

Output SQL-standard `SET SESSION AUTHORIZATION` commands instead of `ALTER OWNER` commands to determine object ownership. This makes the dump more standards compatible, but depending on the history of the objects in the dump, may not restore properly.

`--no-data-for-failed-tables`

By default, table data is restored even if the creation command for the table failed (e.g., because it already exists). With this option, data for such a table is skipped. This behavior is useful when the target database may already contain the desired table contents. For example, auxiliary tables for PostgreSQL extensions such as PostGIS may already be loaded in the target database; specifying this option prevents duplicate or obsolete data from being loaded into them.

This option is effective only when restoring directly into a database, not when producing SQL script output.

pg_restore also accepts the following command line arguments for connection parameters:

`-h` *host*
`--host=`*host*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket. The default is taken from the `PGHOST` environment variable, if set, else a Unix domain socket connection is attempted.

`-p` *port*
`--port=`*port*

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections. Defaults to the `PGPORT` environment variable, if set, or a compiled-in default.

`-U` *username*

Connect as the given user

`-W`

Force a password prompt. This should happen automatically if the server requires password authentication.

`-1`
`--single-transaction`

Execute the restore as a single transaction (that is, wrap the emitted commands in `BEGIN`/`COMMIT`). This ensures that either all the commands complete successfully, or no changes are applied. This option implies `--exit-on-error`.

Environment

`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 29.12).

Diagnostics

When a direct database connection is specified using the `-d` option, pg_restore internally executes SQL statements. If you have problems running pg_restore, make sure you are able to select information from the database using, for example, psql. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Notes

If your installation has any local additions to the `template1` database, be careful to load the output of pg_restore into a truly empty database; otherwise you are likely to get errors due to duplicate definitions of the added objects. To make an empty database without any local additions, copy from `template0` not `template1`, for example:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

The limitations of pg_restore are detailed below.

- When restoring data to a pre-existing table and the option `--disable-triggers` is used, pg_restore emits commands to disable triggers on user tables before inserting the data then emits commands to re-enable them after the data has been inserted. If the restore is stopped in the middle, the system catalogs may be left in the wrong state.
- pg_restore will not restore large objects for a single table. If an archive contains large objects, then all large objects will be restored.

See also the pg_dump documentation for details on limitations of pg_dump.

Once restored, it is wise to run `ANALYZE` on each restored table so the optimizer has useful statistics.

Examples

Assume we have dumped a database called `mydb` into a custom-format dump file:

```
$ pg_dump -Fc mydb > db.dump
```

To drop the database and recreate it from the dump:

```
$ dropdb mydb
$ pg_restore -C -d postgres db.dump
```

The database named in the `-d` switch can be any database existing in the cluster; pg_restore only uses it to issue the `CREATE DATABASE` command for `mydb`. With `-C`, data is always restored into the database name that appears in the dump file.

To reload the dump into a new database called `newdb`:

```
$ createdb -T template0 newdb
$ pg_restore -d newdb db.dump
```

Notice we don't use `-C`, and instead connect directly to the database to be restored into. Also note that we clone the new database from `template0` not `template1`, to ensure it is initially empty.

To reorder database items, it is first necessary to dump the table of contents of the archive:

```
$ pg_restore -l db.dump > db.list
```

The listing file consists of a header and one line for each item, e.g.,

```
;
; Archive created at Fri Jul 28 22:28:36 2000
;     dbname: mydb
;     TOC Entries: 74
;     Compression: 0
;     Dump Version: 1.4-0
;     Format: CUSTOM
;
;
; Selected TOC Entries:
;
2; 145344 TABLE species postgres
3; 145344 ACL species
4; 145359 TABLE nt_header postgres
5; 145359 ACL nt_header
6; 145402 TABLE species_records postgres
7; 145402 ACL species_records
8; 145416 TABLE ss_old postgres
9; 145416 ACL ss_old
```

```
10; 145433 TABLE map_resolutions postgres
11; 145433 ACL map_resolutions
12; 145443 TABLE hs_old postgres
13; 145443 ACL hs_old
```

Semicolons start a comment, and the numbers at the start of lines refer to the internal archive ID assigned to each item.

Lines in the file can be commented out, deleted, and reordered. For example,

```
10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres
```

could be used as input to pg_restore and would only restore items 10 and 6, in that order:

```
$ pg_restore -L db.list db.dump
```

History

The pg_restore utility first appeared in PostgreSQL 7.1.

See Also

pg_dump, pg_dumpall, psql

psql

Name

psql — PostgreSQL interactive terminal

Synopsis

psql [*option*...] [*dbname* [*username*]]

Description

psql is a terminal-based front-end to PostgreSQL. It enables you to type in queries interactively, issue them to PostgreSQL, and see the query results. Alternatively, input can be from a file. In addition, it provides a number of meta-commands and various shell-like features to facilitate writing scripts and automating a wide variety of tasks.

Options

-a
--echo-all

Print all input lines to standard output as they are read. This is more useful for script processing rather than interactive mode. This is equivalent to setting the variable ECHO to all.

-A
--no-align

Switches to unaligned output mode. (The default output mode is otherwise aligned.)

-c *command*
--command *command*

Specifies that psql is to execute one command string, *command*, and then exit. This is useful in shell scripts.

command must be either a command string that is completely parsable by the server (i.e., it contains no psql specific features), or a single backslash command. Thus you cannot mix SQL and psql meta-commands with this option. To achieve that, you could pipe the string into psql, like this: echo '\x \\ SELECT * FROM foo;' | psql. (\\ is the separator meta-command.)

If the command string contains multiple SQL commands, they are processed in a single transaction, unless there are explicit BEGIN/COMMIT commands included in the string to divide it into multiple transactions. This is different from the behavior when the same string is fed to psql's standard input.

`-d` *dbname*
`--dbname` *dbname*

Specifies the name of the database to connect to. This is equivalent to specifying *dbname* as the first non-option argument on the command line.

`-e`
`--echo-queries`

Copy all SQL commands sent to the server to standard output as well. This is equivalent to setting the variable `ECHO` to `queries`.

`-E`
`--echo-hidden`

Echo the actual queries generated by `\d` and other backslash commands. You can use this to study psql's internal operations. This is equivalent to setting the variable `ECHO_HIDDEN` from within psql.

`-f` *filename*
`--file` *filename*

Use the file *filename* as the source of commands instead of reading commands interactively. After the file is processed, psql terminates. This is in many ways equivalent to the internal command `\i`.

If *filename* is `-` (hyphen), then standard input is read.

Using this option is subtly different from writing `psql < `*filename*. In general, both will do what you expect, but using `-f` enables some nice features such as error messages with line numbers. There is also a slight chance that using this option will reduce the start-up overhead. On the other hand, the variant using the shell's input redirection is (in theory) guaranteed to yield exactly the same output that you would have gotten had you entered everything by hand.

`-F` *separator*
`--field-separator` *separator*

Use *separator* as the field separator for unaligned output. This is equivalent to `\pset fieldsep` or `\f`.

`-h` *hostname*
`--host` *hostname*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix-domain socket.

`-H`
`--html`

Turn on HTML tabular output. This is equivalent to `\pset format html` or the `\H` command.

`-l`
`--list`

List all available databases, then exit. Other non-connection options are ignored. This is similar to the internal command `\list`.

`-L` *filename*
`--log-file` *filename*

Write all query output into file *filename*, in addition to the normal output destination.

`-o` *`filename`*
`--output` *`filename`*

Put all query output into file *`filename`*. This is equivalent to the command `\o`.

`-p` *`port`*
`--port` *`port`*

Specifies the TCP port or the local Unix-domain socket file extension on which the server is listening for connections. Defaults to the value of the `PGPORT` environment variable or, if not set, to the port specified at compile time, usually 5432.

`-P` *`assignment`*
`--pset` *`assignment`*

Allows you to specify printing options in the style of `\pset` on the command line. Note that here you have to separate name and value with an equal sign instead of a space. Thus to set the output format to LaTeX, you could write `-P format=latex`.

`-q`
`--quiet`

Specifies that psql should do its work quietly. By default, it prints welcome messages and various informational output. If this option is used, none of this happens. This is useful with the `-c` option. Within psql you can also set the `QUIET` variable to achieve the same effect.

`-R` *`separator`*
`--record-separator` *`separator`*

Use *`separator`* as the record separator for unaligned output. This is equivalent to the `\pset recordsep` command.

`-s`
`--single-step`

Run in single-step mode. That means the user is prompted before each command is sent to the server, with the option to cancel execution as well. Use this to debug scripts.

`-S`
`--single-line`

Runs in single-line mode where a newline terminates an SQL command, as a semicolon does.

> **Note:** This mode is provided for those who insist on it, but you are not necessarily encouraged to use it. In particular, if you mix SQL and meta-commands on a line the order of execution might not always be clear to the inexperienced user.

`-t`
`--tuples-only`

Turn off printing of column names and result row count footers, etc. This is equivalent to the `\t` command.

`-T` *`table_options`*
`--table-attr` *`table_options`*

Allows you to specify options to be placed within the HTML `table` tag. See `\pset` for details.

`-u`

Forces psql to prompt for the user name and password before connecting to the database.

This option is deprecated, as it is conceptually flawed. (Prompting for a non-default user name and prompting for a password because the server requires it are really two different things.) You are encouraged to look at the `-U` and `-W` options instead.

`-U` *`username`*
`--username` *`username`*

Connect to the database as the user *`username`* instead of the default. (You must have permission to do so, of course.)

`-v` *`assignment`*
`--set` *`assignment`*
`--variable` *`assignment`*

Perform a variable assignment, like the `\set` internal command. Note that you must separate name and value, if any, by an equal sign on the command line. To unset a variable, leave off the equal sign. To just set a variable without a value, use the equal sign but leave off the value. These assignments are done during a very early stage of start-up, so variables reserved for internal purposes might get overwritten later.

`-V`
`--version`

Print the psql version and exit.

`-W`
`--password`

Forces psql to prompt for a password before connecting to a database.

psql should automatically prompt for a password whenever the server requests password authentication. However, currently password request detection is not totally reliable, hence this option to force a prompt. If no password prompt is issued and the server requires password authentication, the connection attempt will fail.

This option will remain set for the entire session, even if you change the database connection with the meta-command `\connect`.

`-x`
`--expanded`

Turn on the expanded table formatting mode. This is equivalent to the `\x` command.

`-X,`
`--no-psqlrc`

Do not read the start-up file (neither the system-wide `psqlrc` file nor the user's `~/.psqlrc` file).

`-1`
`--single-transaction`

When psql executes a script with the `-f` option, adding this option wraps `BEGIN`/`COMMIT` around the script to execute it as a single transaction. This ensures that either all the commands complete successfully, or no changes are applied.

If the script itself uses `BEGIN`, `COMMIT`, or `ROLLBACK`, this option will not have the desired effects. Also, if the script contains any command that cannot be executed inside a transaction block, specifying this option will cause that command (and hence the whole transaction) to fail.

`-?`
`--help`

Show help about psql command line arguments, and exit.

Exit Status

psql returns 0 to the shell if it finished normally, 1 if a fatal error of its own (out of memory, file not found) occurs, 2 if the connection to the server went bad and the session was not interactive, and 3 if an error occurred in a script and the variable `ON_ERROR_STOP` was set.

Usage

Connecting To A Database

psql is a regular PostgreSQL client application. In order to connect to a database you need to know the name of your target database, the host name and port number of the server and what user name you want to connect as. psql can be told about those parameters via command line options, namely `-d`, `-h`, `-p`, and `-U` respectively. If an argument is found that does not belong to any option it will be interpreted as the database name (or the user name, if the database name is already given). Not all these options are required; there are useful defaults. If you omit the host name, psql will connect via a Unix-domain socket to a server on the local host, or via TCP/IP to `localhost` on machines that don't have Unix-domain sockets. The default port number is determined at compile time. Since the database server uses the same default, you will not have to specify the port in most cases. The default user name is your Unix user name, as is the default database name. Note that you can't just connect to any database under any user name. Your database administrator should have informed you about your access rights.

When the defaults aren't quite right, you can save yourself some typing by setting the environment variables `PGDATABASE`, `PGHOST`, `PGPORT` and/or `PGUSER` to appropriate values. (For additional environment variables, see Section 29.12.) It is also convenient to have a `~/.pgpass` file to avoid regularly having to type in passwords. See Section 29.13 for more information.

If the connection could not be made for any reason (e.g., insufficient privileges, server is not running on the targeted host, etc.), psql will return an error and terminate.

Entering SQL Commands

In normal operation, psql provides a prompt with the name of the database to which psql is currently connected, followed by the string =>. For example,

```
$ psql testdb
Welcome to psql 8.2.6, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
```

```
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

testdb=>
```

At the prompt, the user may type in SQL commands. Ordinarily, input lines are sent to the server when a command-terminating semicolon is reached. An end of line does not terminate a command. Thus commands can be spread over several lines for clarity. If the command was sent and executed without error, the results of the command are displayed on the screen.

Whenever a command is executed, psql also polls for asynchronous notification events generated by *LISTEN* and *NOTIFY*.

Meta-Commands

Anything you enter in psql that begins with an unquoted backslash is a psql meta-command that is processed by psql itself. These commands help make psql more useful for administration or scripting. Meta-commands are more commonly called slash or backslash commands.

The format of a psql command is the backslash, followed immediately by a command verb, then any arguments. The arguments are separated from the command verb and each other by any number of whitespace characters.

To include whitespace into an argument you may quote it with a single quote. To include a single quote into such an argument, use two single quotes. Anything contained in single quotes is furthermore subject to C-like substitutions for `\n` (new line), `\t` (tab), `\`*`digits`* (octal), and `\x`*`digits`* (hexadecimal).

If an unquoted argument begins with a colon (`:`), it is taken as a psql variable and the value of the variable is used as the argument instead.

Arguments that are enclosed in backquotes (`` ` ``) are taken as a command line that is passed to the shell. The output of the command (with any trailing newline removed) is taken as the argument value. The above escape sequences also apply in backquotes.

Some commands take an SQL identifier (such as a table name) as argument. These arguments follow the syntax rules of SQL: Unquoted letters are forced to lowercase, while double quotes (`"`) protect letters from case conversion and allow incorporation of whitespace into the identifier. Within double quotes, paired double quotes reduce to a single double quote in the resulting name. For example, `FOO"BAR"BAZ` is interpreted as `fooBARbaz`, and `"A weird"" name"` becomes `A weird" name`.

Parsing for arguments stops when another unquoted backslash occurs. This is taken as the beginning of a new meta-command. The special sequence `\\` (two backslashes) marks the end of arguments and continues parsing SQL commands, if any. That way SQL and psql commands can be freely mixed on a line. But in any case, the arguments of a meta-command cannot continue beyond the end of the line.

The following meta-commands are defined:

`\a`

If the current table output format is unaligned, it is switched to aligned. If it is not unaligned, it is set to unaligned. This command is kept for backwards compatibility. See `\pset` for a more general solution.

`\cd [ directory ]`

Changes the current working directory to *directory*. Without argument, changes to the current user's home directory.

> **Tip:** To print your current working directory, use `\!pwd`.

`\C [ title ]`

Sets the title of any tables being printed as the result of a query or unset any such title. This command is equivalent to `\pset title title`. (The name of this command derives from "caption", as it was previously only used to set the caption in an HTML table.)

`\connect` (or `\c`) `[ dbname [ username ] [ host ] [ port ] ]`

Establishes a new connection to a PostgreSQL server. If the new connection is successfully made, the previous connection is closed. If any of *dbname*, *username*, *host* or *port* are omitted or specified as `-`, the value of that parameter from the previous connection is used. If there is no previous connection, the libpq default for the parameter's value is used.

If the connection attempt failed (wrong user name, access denied, etc.), the previous connection will only be kept if psql is in interactive mode. When executing a non-interactive script, processing will immediately stop with an error. This distinction was chosen as a user convenience against typos on the one hand, and a safety mechanism that scripts are not accidentally acting on the wrong database on the other hand.

`\copy { table [(column_list)] | (query) } { from | to } { filename | stdin | stdout | pstdin | pstdout } [with] [binary] [oids] [delimiter [as] 'character'] [null [as] 'string'] [csv [header] [quote [as] 'character'] [escape [as] 'character'] [force quote column_list] [force not null column_list]]`

Performs a frontend (client) copy. This is an operation that runs an SQL *COPY* command, but instead of the server reading or writing the specified file, psql reads or writes the file and routes the data between the server and the local file system. This means that file accessibility and privileges are those of the local user, not the server, and no SQL superuser privileges are required.

The syntax of the command is similar to that of the SQL *COPY* command. Note that, because of this, special parsing rules apply to the `\copy` command. In particular, the variable substitution rules and backslash escapes do not apply.

`\copy ... from stdin | to stdout` reads/writes based on the command input and output respectively. All rows are read from the same source that issued the command, continuing until `\.` is read or the stream reaches EOF. Output is sent to the same place as command output. To read/write from psql's standard input or output, use `pstdin` or `pstdout`. This option is useful for populating tables in-line within a SQL script file.

> **Tip:** This operation is not as efficient as the SQL `COPY` command because all data must pass through the client/server connection. For large amounts of data the SQL command may be preferable.

`\copyright`

Shows the copyright and distribution terms of PostgreSQL.

`\d [ pattern ]`
`\d+ [ pattern ]`

For each relation (table, view, index, or sequence) matching the *pattern*, show all columns, their types, the tablespace (if not the default) and any special attributes such as `NOT NULL` or defaults, if any. Associated indexes, constraints, rules, and triggers are also shown, as is the view definition if the relation is a view. ("Matching the pattern" is defined below.)

The command form `\d+` is identical, except that more information is displayed: any comments associated with the columns of the table are shown, as is the presence of OIDs in the table.

> **Note:** If `\d` is used without a *pattern* argument, it is equivalent to `\dtvs` which will show a list of all tables, views, and sequences. This is purely a convenience measure.

`\da [ pattern ]`

Lists all available aggregate functions, together with the data types they operate on. If *pattern* is specified, only aggregates whose names match the pattern are shown.

`\db [ pattern ]`
`\db+ [ pattern ]`

Lists all available tablespaces. If *pattern* is specified, only tablespaces whose names match the pattern are shown. If + is appended to the command name, each object is listed with its associated permissions.

`\dc [ pattern ]`

Lists all available conversions between character-set encodings. If *pattern* is specified, only conversions whose names match the pattern are listed.

`\dC`

Lists all available type casts.

`\dd [ pattern ]`

Shows the descriptions of objects matching the *pattern*, or of all visible objects if no argument is given. But in either case, only objects that have a description are listed. ("Object" covers aggregates, functions, operators, types, relations (tables, views, indexes, sequences, large objects), rules, and triggers.) For example:

```
=> \dd version
                      Object descriptions
   Schema   |  Name   |  Object  |        Description
------------+---------+----------+---------------------------
 pg_catalog | version | function | PostgreSQL version string
```

```
(1 row)
```

Descriptions for objects can be created with the *COMMENT* SQL command.

`\dD [ pattern ]`

Lists all available domains. If *pattern* is specified, only matching domains are shown.

`\df [ pattern ]`
`\df+ [ pattern ]`

Lists available functions, together with their argument and return types. If *pattern* is specified, only functions whose names match the pattern are shown. If the form `\df+` is used, additional information about each function, including language and description, is shown.

> **Note:** To look up functions taking argument or returning values of a specific type, use your pager's search capability to scroll through the `\df` output.
>
> To reduce clutter, `\df` does not show data type I/O functions. This is implemented by ignoring functions that accept or return type `cstring`.

`\dg [ pattern ]`

Lists all database roles. If *pattern* is specified, only those roles whose names match the pattern are listed. (This command is now effectively the same as `\du`.)

`\distvS [ pattern ]`

This is not the actual command name: the letters `i`, `s`, `t`, `v`, `S` stand for index, sequence, table, view, and system table, respectively. You can specify any or all of these letters, in any order, to obtain a listing of all the matching objects. The letter `S` restricts the listing to system objects; without `S`, only non-system objects are shown. If `+` is appended to the command name, each object is listed with its associated description, if any.

If *pattern* is specified, only objects whose names match the pattern are listed.

`\dl`

This is an alias for `\lo_list`, which shows a list of large objects.

`\dn [ pattern ]`
`\dn+ [ pattern ]`

Lists all available schemas (namespaces). If *pattern* (a regular expression) is specified, only schemas whose names match the pattern are listed. Non-local temporary schemas are suppressed. If `+` is appended to the command name, each object is listed with its associated permissions and description, if any.

`\do [ pattern ]`

Lists available operators with their operand and return types. If *pattern* is specified, only operators whose names match the pattern are listed.

`\dp [ pattern ]`

Produces a list of all available tables, views and sequences with their associated access privileges. If *pattern* is specified, only tables, views and sequences whose names match the pattern are listed.

The *GRANT* and *REVOKE* commands are used to set access privileges.

`\dT [ pattern ]`
`\dT+ [ pattern ]`

Lists all data types or only those that match *pattern*. The command form `\dT+` shows extra information.

`\du [ pattern ]`

Lists all database roles, or only those that match *pattern*.

`\edit` (or `\e`) `[ filename ]`

If *filename* is specified, the file is edited; after the editor exits, its content is copied back to the query buffer. If no argument is given, the current query buffer is copied to a temporary file which is then edited in the same fashion.

The new query buffer is then re-parsed according to the normal rules of psql, where the whole buffer is treated as a single line. (Thus you cannot make scripts this way. Use `\i` for that.) This means also that if the query ends with (or rather contains) a semicolon, it is immediately executed. In other cases it will merely wait in the query buffer.

> **Tip:** psql searches the environment variables `PSQL_EDITOR`, `EDITOR`, and `VISUAL` (in that order) for an editor to use. If all of them are unset, `vi` is used on Unix systems, `notepad.exe` on Windows systems.

`\echo text [ ... ]`

Prints the arguments to the standard output, separated by one space and followed by a newline. This can be useful to intersperse information in the output of scripts. For example:

```
=> \echo `date`
Tue Oct 26 21:40:57 CEST 1999
```

If the first argument is an unquoted `-n` the trailing newline is not written.

> **Tip:** If you use the `\o` command to redirect your query output you may wish to use `\qecho` instead of this command.

`\encoding [ encoding ]`

Sets the client character set encoding. Without an argument, this command shows the current encoding.

`\f [ string ]`

Sets the field separator for unaligned query output. The default is the vertical bar (`|`). See also `\pset` for a generic way of setting output options.

`\g [ { filename | |command } ]`

Sends the current query input buffer to the server and optionally stores the query's output in *filename* or pipes the output into a separate Unix shell executing *command*. A bare `\g` is virtually equivalent to a semicolon. A `\g` with argument is a "one-shot" alternative to the `\o` command.

`\help` (or `\h`) `[ command ]`

Gives syntax help on the specified SQL command. If *command* is not specified, then psql will list all the commands for which syntax help is available. If *command* is an asterisk (`*`), then syntax help on all SQL commands is shown.

> **Note:** To simplify typing, commands that consists of several words do not have to be quoted. Thus it is fine to type **`\help alter table`**.

`\H`

Turns on HTML query output format. If the HTML format is already on, it is switched back to the default aligned text format. This command is for compatibility and convenience, but see `\pset` about setting other output options.

`\i` *filename*

Reads input from the file *filename* and executes it as though it had been typed on the keyboard.

> **Note:** If you want to see the lines on the screen as they are read you must set the variable `ECHO` to `all`.

`\l` (or `\list`)
`\l+` (or `\list+`)

List the names, owners, and character set encodings of all the databases in the server. If + is appended to the command name, database descriptions are also displayed.

`\lo_export` *loid filename*

Reads the large object with OID *loid* from the database and writes it to *filename*. Note that this is subtly different from the server function `lo_export`, which acts with the permissions of the user that the database server runs as and on the server's file system.

> **Tip:** Use `\lo_list` to find out the large object's OID.

`\lo_import` *filename* `[` *comment* `]`

Stores the file into a PostgreSQL large object. Optionally, it associates the given comment with the object. Example:

```
foo=> \lo_import '/home/peter/pictures/photo.xcf' 'a picture of me'
lo_import 152801
```

The response indicates that the large object received object ID 152801 which one ought to remember if one wants to access the object ever again. For that reason it is recommended to always associate a human-readable comment with every object. Those can then be seen with the `\lo_list` command.

Note that this command is subtly different from the server-side `lo_import` because it acts as the local user on the local file system, rather than the server's user and file system.

`\lo_list`

Shows a list of all PostgreSQL large objects currently stored in the database, along with any comments provided for them.

`\lo_unlink` *`loid`*

Deletes the large object with OID *`loid`* from the database.

> **Tip:** Use `\lo_list` to find out the large object's OID.

`\o` [{ *`filename`* | `|` *`command`* }]

Saves future query results to the file *`filename`* or pipes future results into a separate Unix shell to execute *`command`*. If no arguments are specified, the query output will be reset to the standard output.

"Query results" includes all tables, command responses, and notices obtained from the database server, as well as output of various backslash commands that query the database (such as `\d`), but not error messages.

> **Tip:** To intersperse text output in between query results, use `\qecho`.

`\p`

Print the current query buffer to the standard output.

`\password [` *`username`* `]`

Changes the password of the specified user (by default, the current user). This command prompts for the new password, encrypts it, and sends it to the server as an `ALTER ROLE` command. This makes sure that the new password does not appear in cleartext in the command history, the server log, or elsewhere.

`\pset` *`parameter`* `[` *`value`* `]`

This command sets options affecting the output of query result tables. *`parameter`* describes which option is to be set. The semantics of *`value`* depend thereon.

Adjustable printing options are:

`format`

Sets the output format to one of `unaligned`, `aligned`, `html`, `latex`, or `troff-ms`. Unique abbreviations are allowed. (That would mean one letter is enough.)

"Unaligned" writes all columns of a row on a line, separated by the currently active field separator. This is intended to create output that might be intended to be read in by other programs (tab-separated, comma-separated). "Aligned" mode is the standard, human-readable, nicely formatted text output that is default. The "HTML" and "LaTeX" modes put out tables that are intended to be included in documents using the respective mark-up language. They are not complete documents! (This might not be so dramatic in HTML, but in LaTeX you must have a complete document wrapper.)

`border`

The second argument must be a number. In general, the higher the number the more borders and lines the tables will have, but this depends on the particular format. In HTML mode, this will translate directly into the `border=...` attribute, in the others only values 0 (no border), 1 (internal dividing lines), and 2 (table frame) make sense.

`expanded` (or `x`)

Toggles between regular and expanded format. When expanded format is enabled, query results are displayed in two columns, with the column name on the left and the data on the right. This mode is useful if the data wouldn't fit on the screen in the normal "horizontal" mode.

Expanded mode is supported by all four output formats.

`null`

The second argument is a string that should be printed whenever a column is null. The default is not to print anything, which can easily be mistaken for, say, an empty string. Thus, one might choose to write `\pset null '(null)'`.

`fieldsep`

Specifies the field separator to be used in unaligned output mode. That way one can create, for example, tab- or comma-separated output, which other programs might prefer. To set a tab as field separator, type `\pset fieldsep '\t'`. The default field separator is `'|'` (a vertical bar).

`footer`

Toggles the display of the default footer `(x rows)`.

`numericlocale`

Toggles the display of a locale-aware character to separate groups of digits to the left of the decimal marker. It also enables a locale-aware decimal marker.

`recordsep`

Specifies the record (line) separator to use in unaligned output mode. The default is a newline character.

`tuples_only` (or `t`)

Toggles between tuples only and full display. Full display may show extra information such as column headers, titles, and various footers. In tuples only mode, only actual table data is shown.

`title [ text ]`

Sets the table title for any subsequently printed tables. This can be used to give your output descriptive tags. If no argument is given, the title is unset.

`tableattr` (or `T`) `[ text ]`

Allows you to specify any attributes to be placed inside the HTML `table` tag. This could for example be `cellpadding` or `bgcolor`. Note that you probably don't want to specify `border` here, as that is already taken care of by `\pset border`.

`pager`

Controls use of a pager for query and psql help output. If the environment variable `PAGER` is set, the output is piped to the specified program. Otherwise a platform-dependent default (such as `more`) is used.

When the pager is off, the pager is not used. When the pager is on, the pager is used only when appropriate, i.e. the output is to a terminal and will not fit on the screen. (psql does not do a perfect job of estimating when to use the pager.) `\pset pager` turns the pager on and off. Pager can also be set to `always`, which causes the pager to be always used.

Illustrations on how these different formats look can be seen in the *Examples* section.

Tip: There are various shortcut commands for `\pset`. See `\a`, `\C`, `\H`, `\t`, `\T`, and `\x`.

Note: It is an error to call `\pset` without arguments. In the future this call might show the current status of all printing options.

`\q`

Quits the psql program.

`\qecho` *`text`* `[ ... ]`

This command is identical to `\echo` except that the output will be written to the query output channel, as set by `\o`.

`\r`

Resets (clears) the query buffer.

`\s [` *`filename`* `]`

Print or save the command line history to *`filename`*. If *`filename`* is omitted, the history is written to the standard output. This option is only available if psql is configured to use the GNU Readline library.

`\set [` *`name`* `[` *`value`* `[ ... ] ] ]`

Sets the internal variable *`name`* to *`value`* or, if more than one value is given, to the concatenation of all of them. If no second argument is given, the variable is just set with no value. To unset a variable, use the `\unset` command.

Valid variable names can contain characters, digits, and underscores. See the section *Variables* below for details. Variable names are case-sensitive.

Although you are welcome to set any variable to anything you want, psql treats several variables as special. They are documented in the section about variables.

Note: This command is totally separate from the SQL command *SET*.

`\t`

Toggles the display of output column name headings and row count footer. This command is equivalent to `\pset tuples_only` and is provided for convenience.

`\T` *table_options*

Allows you to specify attributes to be placed within the `table` tag in HTML tabular output mode. This command is equivalent to `\pset tableattr` *table_options*.

`\timing`

Toggles a display of how long each SQL statement takes, in milliseconds.

`\w {`*filename*`| |`*command*`}`

Outputs the current query buffer to the file *filename* or pipes it to the Unix command *command*.

`\x`

Toggles expanded table formatting mode. As such it is equivalent to `\pset expanded`.

`\z [` *pattern* `]`

Produces a list of all available tables, views and sequences with their associated access privileges. If a *pattern* is specified, only tables,views and sequences whose names match the pattern are listed.

The *GRANT* and *REVOKE* commands are used to set access privileges.

This is an alias for `\dp` ("display privileges").

`\! [` *command* `]`

Escapes to a separate Unix shell or executes the Unix command *command*. The arguments are not further interpreted, the shell will see them as is.

`\?`

Shows help information about the backslash commands.

Patterns

The various `\d` commands accept a *pattern* parameter to specify the object name(s) to be displayed. In the simplest case, a pattern is just the exact name of the object. The characters within a pattern are normally folded to lower case, just as in SQL names; for example, `\dt FOO` will display the table named `foo`. As in SQL names, placing double quotes around a pattern stops folding to lower case. Should you need to include an actual double quote character in a pattern, write it as a pair of double quotes within a double-quote sequence; again this is in accord with the rules for SQL quoted identifiers. For example, `\dt "FOO""BAR"` will display the table named `FOO"BAR` (not `foo"bar`). Unlike the normal rules for SQL names, you can put double quotes around just part of a pattern, for instance `\dt FOO"FOO"BAR` will display the table named `fooFOObar`.

Within a pattern, `*` matches any sequence of characters (including no characters) and `?` matches any single character. (This notation is comparable to Unix shell file name patterns.) For example, `\dt int*` displays all tables whose names begin with `int`. But within double quotes, `*` and `?` lose these special meanings and are just matched literally.

A pattern that contains a dot (`.`) is interpreted as a schema name pattern followed by an object name pattern. For example, `\dt foo*.bar*` displays all tables whose table name starts with `bar` that are in schemas whose schema name starts with `foo`. When no dot appears, then the pattern matches only objects that are visible in the current schema search path. Again, a dot within double quotes loses its special meaning and is matched literally.

Advanced users can use regular-expression notations such as character classes, for example `[0-9]` to match any digit. All regular expression special characters work as specified in Section 9.7.3, except for `.` which is taken as a separator as mentioned above, `*` which is translated to the regular-expression notation `.*`, and `?` which is translated to `.`. You can emulate these pattern characters at need by writing `?` for `.`, `(R+|)` for *R*`*`, or `(R|)` for *R*`?`. Remember that the pattern must match the whole name, unlike the usual interpretation of regular expressions; write `*` at the beginning and/or end if you don't wish the pattern to be anchored. Note that within double quotes, all regular expression special characters lose their special meanings and are matched literally. Also, the regular expression special characters are matched literally in operator name patterns (i.e., the argument of `\do`).

Whenever the *pattern* parameter is omitted completely, the `\d` commands display all objects that are visible in the current schema search path — this is equivalent to using the pattern `*`. To see all objects in the database, use the pattern `*.*`.

Advanced features

Variables

psql provides variable substitution features similar to common Unix command shells. Variables are simply name/value pairs, where the value can be any string of any length. To set variables, use the psql meta-command `\set`:

```
testdb=> \set foo bar
```

sets the variable `foo` to the value `bar`. To retrieve the content of the variable, precede the name with a colon and use it as the argument of any slash command:

```
testdb=> \echo :foo
bar
```

> **Note:** The arguments of `\set` are subject to the same substitution rules as with other commands. Thus you can construct interesting references such as `\set :foo 'something'` and get "soft links" or "variable variables" of Perl or PHP fame, respectively. Unfortunately (or fortunately?), there is no way to do anything useful with these constructs. On the other hand, `\set bar :foo` is a perfectly valid way to copy a variable.

If you call `\set` without a second argument, the variable is set, with an empty string as value. To unset (or delete) a variable, use the command `\unset`.

psql's internal variable names can consist of letters, numbers, and underscores in any order and any number of them. A number of these variables are treated specially by psql. They indicate certain option settings that can be changed at run time by altering the value of the variable or represent some state of the application. Although you can use these variables for any other purpose, this is not recommended, as the program behavior might grow really strange really quickly. By convention, all specially treated variables consist of all upper-case letters (and possibly numbers and underscores). To ensure maximum compatibility in the future, avoid using such variable names for your own purposes. A list of all specially treated variables follows.

`AUTOCOMMIT`

When `on` (the default), each SQL command is automatically committed upon successful completion. To postpone commit in this mode, you must enter a `BEGIN` or `START TRANSACTION` SQL command. When `off` or unset, SQL commands are not committed until you explicitly issue `COMMIT` or `END`. The autocommit-off mode works by issuing an implicit `BEGIN` for you, just before any command that is not already in a transaction block and is not itself a `BEGIN` or other transaction-control command, nor a command that cannot be executed inside a transaction block (such as `VACUUM`).

Note: In autocommit-off mode, you must explicitly abandon any failed transaction by entering `ABORT` or `ROLLBACK`. Also keep in mind that if you exit the session without committing, your work will be lost.

Note: The autocommit-on mode is PostgreSQL's traditional behavior, but autocommit-off is closer to the SQL spec. If you prefer autocommit-off, you may wish to set it in the system-wide `psqlrc` file or your `~/.psqlrc` file.

`DBNAME`

The name of the database you are currently connected to. This is set every time you connect to a database (including program start-up), but can be unset.

`ECHO`

If set to `all`, all lines entered from the keyboard or from a script are written to the standard output before they are parsed or executed. To select this behavior on program start-up, use the switch `-a`. If set to `queries`, psql merely prints all queries as they are sent to the server. The switch for this is `-e`.

`ECHO_HIDDEN`

When this variable is set and a backslash command queries the database, the query is first shown. This way you can study the PostgreSQL internals and provide similar functionality in your own programs. (To select this behavior on program start-up, use the switch `-E`.) If you set the variable to the value `noexec`, the queries are just shown but are not actually sent to the server and executed.

`ENCODING`

The current client character set encoding.

`FETCH_COUNT`

If this variable is set to an integer value > 0, the results of `SELECT` queries are fetched and displayed in groups of that many rows, rather than the default behavior of collecting the entire result set before

display. Therefore only a limited amount of memory is used, regardless of the size of the result set. Settings of 100 to 1000 are commonly used when enabling this feature. Keep in mind that when using this feature, a query may fail after having already displayed some rows.

> **Tip:** Although you can use any output format with this feature, the default `aligned` format tends to look bad because each group of `FETCH_COUNT` rows will be formatted separately, leading to varying column widths across the row groups. The other output formats work better.

`HISTCONTROL`

If this variable is set to `ignorespace`, lines which begin with a space are not entered into the history list. If set to a value of `ignoredups`, lines matching the previous history line are not entered. A value of `ignoreboth` combines the two options. If unset, or if set to any other value than those above, all lines read in interactive mode are saved on the history list.

> **Note:** This feature was shamelessly plagiarized from Bash.

`HISTFILE`

The file name that will be used to store the history list. The default value is `~/.psql_history`. For example, putting

```
\set HISTFILE ~/.psql_history- :DBNAME
```

in `~/.psqlrc` will cause psql to maintain a separate history for each database.

> **Note:** This feature was shamelessly plagiarized from Bash.

`HISTSIZE`

The number of commands to store in the command history. The default value is 500.

> **Note:** This feature was shamelessly plagiarized from Bash.

`HOST`

The database server host you are currently connected to. This is set every time you connect to a database (including program start-up), but can be unset.

`IGNOREEOF`

If unset, sending an EOF character (usually **Control+D**) to an interactive session of psql will terminate the application. If set to a numeric value, that many EOF characters are ignored before the application terminates. If the variable is set but has no numeric value, the default is 10.

> **Note:** This feature was shamelessly plagiarized from Bash.

`LASTOID`

The value of the last affected OID, as returned from an `INSERT` or `lo_insert` command. This variable is only guaranteed to be valid until after the result of the next SQL command has been displayed.

`ON_ERROR_ROLLBACK`

When `on`, if a statement in a transaction block generates an error, the error is ignored and the transaction continues. When `interactive`, such errors are only ignored in interactive sessions, and not when reading script files. When `off` (the default), a statement in a transaction block that generates an error aborts the entire transaction. The on_error_rollback-on mode works by issuing an implicit `SAVEPOINT` for you, just before each command that is in a transaction block, and rolls back to the savepoint on error.

`ON_ERROR_STOP`

By default, if non-interactive scripts encounter an error, such as a malformed SQL command or internal meta-command, processing continues. This has been the traditional behavior of psql but it is sometimes not desirable. If this variable is set, script processing will immediately terminate. If the script was called from another script it will terminate in the same fashion. If the outermost script was not called from an interactive psql session but rather using the `-f` option, psql will return error code 3, to distinguish this case from fatal error conditions (error code 1).

`PORT`

The database server port to which you are currently connected. This is set every time you connect to a database (including program start-up), but can be unset.

`PROMPT1`
`PROMPT2`
`PROMPT3`

These specify what the prompts psql issues should look like. See *Prompting* below.

`QUIET`

This variable is equivalent to the command line option `-q`. It is probably not too useful in interactive mode.

`SINGLELINE`

This variable is equivalent to the command line option `-S`.

`SINGLESTEP`

This variable is equivalent to the command line option `-s`.

`USER`

The database user you are currently connected as. This is set every time you connect to a database (including program start-up), but can be unset.

`VERBOSITY`

This variable can be set to the values `default`, `verbose`, or `terse` to control the verbosity of error reports.

SQL Interpolation

An additional useful feature of psql variables is that you can substitute ("interpolate") them into regular SQL statements. The syntax for this is again to prepend the variable name with a colon (`:`).

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :foo;
```

would then query the table `my_table`. The value of the variable is copied literally, so it can even contain unbalanced quotes or backslash commands. You must make sure that it makes sense where you put it. Variable interpolation will not be performed into quoted SQL entities.

A popular application of this facility is to refer to the last inserted OID in subsequent statements to build a foreign key scenario. Another possible use of this mechanism is to copy the contents of a file into a table column. First load the file into a variable and then proceed as above.

```
testdb=> \set content '' `cat my_file.txt` ''
testdb=> INSERT INTO my_table VALUES (:content);
```

One problem with this approach is that `my_file.txt` might contain single quotes. These need to be escaped so that they don't cause a syntax error when the second line is processed. This could be done with the program `sed`:

```
testdb=> \set content '' `sed -e "s/'/''/g" < my_file.txt` ''
```

If you are using non-standard-conforming strings then you'll also need to double backslashes. This is a bit tricky:

```
testdb=> \set content '' `sed -e "s/'/''/g" -e 's/\\/\\\\/g' < my_file.txt` ''
```

Note the use of different shell quoting conventions so that neither the single quote marks nor the backslashes are special to the shell. Backslashes are still special to `sed`, however, so we need to double them. (Perhaps at one point you thought it was great that all Unix commands use the same escape character.)

Since colons may legally appear in SQL commands, the following rule applies: the character sequence ":name" is not changed unless "name" is the name of a variable that is currently set. In any case you can escape a colon with a backslash to protect it from substitution. (The colon syntax for variables is standard SQL for embedded query languages, such as ECPG. The colon syntax for array slices and type casts are PostgreSQL extensions, hence the conflict.)

Prompting

The prompts psql issues can be customized to your preference. The three variables `PROMPT1`, `PROMPT2`, and `PROMPT3` contain strings and special escape sequences that describe the appearance of the prompt. Prompt 1 is the normal prompt that is issued when psql requests a new command. Prompt 2 is issued when more input is expected during command input because the command was not terminated with a semicolon or a quote was not closed. Prompt 3 is issued when you run an SQL `COPY` command and you are expected to type in the row values on the terminal.

The value of the selected prompt variable is printed literally, except where a percent sign (`%`) is encountered. Depending on the next character, certain other text is substituted instead. Defined substitutions are:

`%M`

The full host name (with domain name) of the database server, or `[local]` if the connection is over a Unix domain socket, or `[local:`*`/dir/name`*`]`, if the Unix domain socket is not at the compiled in default location.

`%m`

The host name of the database server, truncated at the first dot, or `[local]` if the connection is over a Unix domain socket.

`%>`

The port number at which the database server is listening.

`%n`

The database session user name. (The expansion of this value might change during a database session as the result of the command `SET SESSION AUTHORIZATION`.)

`%/`

The name of the current database.

`%~`

Like `%/`, but the output is `~` (tilde) if the database is your default database.

`%#`

If the session user is a database superuser, then a `#`, otherwise a `>`. (The expansion of this value might change during a database session as the result of the command `SET SESSION AUTHORIZATION`.)

`%R`

In prompt 1 normally `=`, but `^` if in single-line mode, and `!` if the session is disconnected from the database (which can happen if `\connect` fails). In prompt 2 the sequence is replaced by `-`, `*`, a single quote, a double quote, or a dollar sign, depending on whether psql expects more input because the command wasn't terminated yet, because you are inside a `/* ... */` comment, or because you are inside a quoted or dollar-escaped string. In prompt 3 the sequence doesn't produce anything.

`%x`

Transaction status: an empty string when not in a transaction block, or `*` when in a transaction block, or `!` when in a failed transaction block, or `?` when the transaction state is indeterminate (for example, because there is no connection).

`%`*`digits`*

The character with the indicated octal code is substituted.

`%:`*`name`*`:`

The value of the psql variable *`name`*. See the section *Variables* for details.

`%`\`*`command`*\`

The output of *`command`*, similar to ordinary "back-tick" substitution.

`%[` ... `%]`

Prompts may contain terminal control characters which, for example, change the color, background, or style of the prompt text, or change the title of the terminal window. In order for the line editing

features of Readline to work properly, these non-printing control characters must be designated as invisible by surrounding them with `%[` and `%]`. Multiple pairs of these may occur within the prompt. For example,

```
testdb=> \set PROMPT1 '%[%033[1;33;40m%]%n@%/%R%[%033[0m%]%# '
```

results in a boldfaced (`1;`) yellow-on-black (`33;40`) prompt on VT100-compatible, color-capable terminals.

To insert a percent sign into your prompt, write `%%`. The default prompts are `'%/%R%# '` for prompts 1 and 2, and `'>> '` for prompt 3.

Note: This feature was shamelessly plagiarized from tcsh.

Command-Line Editing

psql supports the Readline library for convenient line editing and retrieval. The command history is automatically saved when psql exits and is reloaded when psql starts up. Tab-completion is also supported, although the completion logic makes no claim to be an SQL parser. If for some reason you do not like the tab completion, you can turn it off by putting this in a file named `.inputrc` in your home directory:

```
$if psql
set disable-completion on
$endif
```

(This is not a psql but a Readline feature. Read its documentation for further details.)

Environment

`PAGER`

If the query results do not fit on the screen, they are piped through this command. Typical values are `more` or `less`. The default is platform-dependent. The use of the pager can be disabled by using the `\pset` command.

`PGDATABASE`

Default connection database

`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters

`PSQL_EDITOR`
`EDITOR`
`VISUAL`

Editor used by the `\e` command. The variables are examined in the order listed; the first that is set is used.

`SHELL`

Command executed by the `\!` command.

`TMPDIR`

Directory for storing temporary files. The default is `/tmp`.

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 29.12).

Files

- Before starting up, psql attempts to read and execute commands from the system-wide `psqlrc` file and the user's `~/.psqlrc` file. (On Windows, the user's startup file is named `%APPDATA%\postgresql\psqlrc.conf`.) See *`PREFIX`*`/share/psqlrc.sample` for information on setting up the system-wide file. It could be used to set up the client or the server to taste (using the `\set` and `SET` commands).
- Both the system-wide `psqlrc` file and the user's `~/.psqlrc` file can be made version-specific by appending a dash and the PostgreSQL release number, for example `~/.psqlrc-8.2.6`. A matching version-specific file will be read in preference to a non-version-specific file.
- The command-line history is stored in the file `~/.psql_history`, or `%APPDATA%\postgresql\psql_history` on Windows.

Notes

- In an earlier life psql allowed the first argument of a single-letter backslash command to start directly after the command, without intervening whitespace. For compatibility this is still supported to some extent, but we are not going to explain the details here as this use is discouraged. If you get strange messages, keep this in mind. For example

  ```
  testdb=> \foo
  Field separator is "oo".
  ```
  which is perhaps not what one would expect.
- psql only works smoothly with servers of the same version. That does not mean other combinations will fail outright, but subtle and not-so-subtle problems might come up. Backslash commands are particularly likely to fail if the server is of a different version.

Notes for Windows users

psql is built as a "console application". Since the Windows console windows use a different encoding than the rest of the system, you must take special care when using 8-bit characters within psql. If psql detects a problematic console code page, it will warn you at startup. To change the console code page, two things are necessary:

- Set the code page by entering **`cmd.exe /c chcp 1252`**. (1252 is a code page that is appropriate for German; replace it with your value.) If you are using Cygwin, you can put this command in `/etc/profile`.
- Set the console font to `Lucida Console`, because the raster font does not work with the ANSI code page.

Examples

The first example shows how to spread a command over several lines of input. Notice the changing prompt:

```
testdb=> CREATE TABLE my_table (
testdb(>  first integer not null default 0,
testdb(>  second text)
testdb-> ;
CREATE TABLE
```

Now look at the table definition again:

```
testdb=> \d my_table
             Table "my_table"
 Attribute |  Type   |      Modifier
-----------+---------+--------------------
 first     | integer | not null default 0
 second    | text    |
```

Now we change the prompt to something more interesting:

```
testdb=> \set PROMPT1 '%n@%m %~%R%# '
peter@localhost testdb=>
```

Let's assume you have filled the table with data and want to take a look at it:

```
peter@localhost testdb=> SELECT * FROM my_table;
 first | second
-------+--------
     1 | one
     2 | two
     3 | three
     4 | four
(4 rows)
```

You can display tables in different ways by using the `\pset` command:

```
peter@localhost testdb=> \pset border 2
Border style is 2.
peter@localhost testdb=> SELECT * FROM my_table;
+-------+--------+
| first | second |
+-------+--------+
|     1 | one    |
|     2 | two    |
|     3 | three  |
|     4 | four   |
+-------+--------+
(4 rows)

peter@localhost testdb=> \pset border 0
Border style is 0.
peter@localhost testdb=> SELECT * FROM my_table;
first second
----- ------
    1 one
    2 two
    3 three
    4 four
(4 rows)

peter@localhost testdb=> \pset border 1
Border style is 1.
peter@localhost testdb=> \pset format unaligned
Output format is unaligned.
peter@localhost testdb=> \pset fieldsep ","
Field separator is ",".
peter@localhost testdb=> \pset tuples_only
Showing only tuples.
peter@localhost testdb=> SELECT second, first FROM my_table;
one,1
two,2
three,3
four,4
```

Alternatively, use the short commands:

```
peter@localhost testdb=> \a \t \x
Output format is aligned.
Tuples only is off.
Expanded display is on.
peter@localhost testdb=> SELECT * FROM my_table;
-[ RECORD 1 ]-
first  | 1
second | one
-[ RECORD 2 ]-
first  | 2
second | two
-[ RECORD 3 ]-
first  | 3
```

```
second | three
-[ RECORD 4 ]-
first  | 4
second | four
```

reindexdb

Name

`reindexdb` — reindex a PostgreSQL database

Synopsis

`reindexdb` [*connection-option*...] [--table | -t *table*] [--index | -i *index*] [*dbname*]
`reindexdb` [*connection-option*...] [--all | -a]
`reindexdb` [*connection-option*...] [--system | -s] [*dbname*]

Description

reindexdb is a utility for rebuilding indexes in a PostgreSQL database.

reindexdb is a wrapper around the SQL command *REINDEX*. There is no effective difference between reindexing databases via this utility and via other methods for accessing the server.

Options

reindexdb accepts the following command-line arguments:

`-a`
`--all`

Reindex all databases.

`-s`
`--system`

Reindex database's system catalogs.

`-t` *table*
`--table` *table*

Reindex *table* only.

`-i` *index*
`--index` *index*

Recreate *index* only.

`[-d]` *dbname*
`[--dbname]` *dbname*

Specifies the name of the database to be reindexed. If this is not specified and `-a` (or `--all`) is not used, the database name is read from the environment variable `PGDATABASE`. If that is not set, the user name specified for the connection is used.

`-e`
`--echo`

Echo the commands that reindexdb generates and sends to the server.

`-q`
`--quiet`

Do not display a response.

reindexdb also accepts the following command-line arguments for connection parameters:

`-h` *host*
`--host` *host*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *port*
`--port` *port*

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

`-U` *username*
`--username` *username*

User name to connect as.

`-W`
`--password`

Force password prompt.

Environment

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 29.12).

Diagnostics

In case of difficulty, see *REINDEX* and psql for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Notes

reindexdb might need to connect several times to the PostgreSQL server, asking for a password each time. It is convenient to have a `~/.pgpass` file in such cases. See Section 29.13 for more information.

Examples

To reindex the database `test`:

```
$ reindexdb test
```

To reindex the table `foo` and the index `bar` in a database named `abcd`:

```
$ reindexdb --table foo --index bar abcd
```

See Also

REINDEX

vacuumdb

Name

vacuumdb — garbage-collect and analyze a PostgreSQL database

Synopsis

vacuumdb [*connection-option*...] [--full | -f] [--verbose | -v] [--analyze | -z] [--table | -t *table* [(*column* [,...])]] [*dbname*]
vacuumdb [*connection-options*...] [--all | -a] [--full | -f] [--verbose | -v] [--analyze | -z]

Description

vacuumdb is a utility for cleaning a PostgreSQL database. vacuumdb will also generate internal statistics used by the PostgreSQL query optimizer.

vacuumdb is a wrapper around the SQL command *VACUUM*. There is no effective difference between vacuuming databases via this utility and via other methods for accessing the server.

Options

vacuumdb accepts the following command-line arguments:

`-a`
`--all`

Vacuum all databases.

`[-d] dbname`
`[--dbname] dbname`

Specifies the name of the database to be cleaned or analyzed. If this is not specified and `-a` (or `--all`) is not used, the database name is read from the environment variable `PGDATABASE`. If that is not set, the user name specified for the connection is used.

`-e`
`--echo`

Echo the commands that vacuumdb generates and sends to the server.

`-f`
`--full`

Perform “full” vacuuming.

`-q`
`--quiet`

Do not display a response.

`-t` *`table`* `[ (`*`column`* `[,...]) ]`
`--table` *`table`* `[ (`*`column`* `[,...]) ]`

Clean or analyze *`table`* only. Column names may be specified only in conjunction with the `--analyze` option.

> **Tip:** If you specify columns, you probably have to escape the parentheses from the shell. (See examples below.)

`-v`
`--verbose`

Print detailed information during processing.

`-z`
`--analyze`

Calculate statistics for use by the optimizer.

vacuumdb also accepts the following command-line arguments for connection parameters:

`-h` *`host`*
`--host` *`host`*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *`port`*
`--port` *`port`*

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

`-U` *`username`*
`--username` *`username`*

User name to connect as

`-W`
`--password`

Force password prompt.

Environment

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 29.12).

Diagnostics

In case of difficulty, see *VACUUM* and psql for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Notes

vacuumdb might need to connect several times to the PostgreSQL server, asking for a password each time. It is convenient to have a `~/.pgpass` file in such cases. See Section 29.13 for more information.

Examples

To clean the database `test`:

```
$ vacuumdb test
```

To clean and analyze for the optimizer a database named `bigdb`:

```
$ vacuumdb --analyze bigdb
```

To clean a single table `foo` in a database named `xyzzy`, and analyze a single column `bar` of the table for the optimizer:

```
$ vacuumdb --analyze --verbose --table 'foo(bar)' xyzzy
```

See Also

VACUUM

III. PostgreSQL Server Applications

This part contains reference information for PostgreSQL server applications and support utilities. These commands can only be run usefully on the host where the database server resides. Other utility programs are listed in Reference II, *PostgreSQL Client Applications*.

initdb

Name

initdb — create a new PostgreSQL database cluster

Synopsis

initdb [*option*...] --pgdata | -D *directory*

Description

initdb creates a new PostgreSQL database cluster. A database cluster is a collection of databases that are managed by a single server instance.

Creating a database cluster consists of creating the directories in which the database data will live, generating the shared catalog tables (tables that belong to the whole cluster rather than to any particular database), and creating the template1 and postgres databases. When you later create a new database, everything in the template1 database is copied. (Therefore, anything installed in template1 is automatically copied into each database created later.) The postgres database is a default database meant for use by users, utilities and third party applications.

Although initdb will attempt to create the specified data directory, it might not have permission if the parent directory of the desired data directory is root-owned. To initialize in such a setup, create an empty data directory as root, then use chown to assign ownership of that directory to the database user account, then su to become the database user to run initdb.

initdb must be run as the user that will own the server process, because the server needs to have access to the files and directories that initdb creates. Since the server may not be run as root, you must not run initdb as root either. (It will in fact refuse to do so.)

initdb initializes the database cluster's default locale and character set encoding. The collation order (LC_COLLATE) and character set classes (LC_CTYPE, e.g. upper, lower, digit) are fixed for all databases and can not be changed. Collation orders other than C or POSIX also have a performance penalty. For these reasons it is important to choose the right locale when running initdb. The remaining locale categories can be changed later when the server is started. All server locale values (lc_*) can be displayed via SHOW ALL. More details can be found in Section 21.1.

The character set encoding can be set separately for a database when it is created. initdb determines the encoding for the template1 database, which will serve as the default for all other databases. To alter the default encoding use the --encoding option. More details can be found in Section 21.2.

Options

`-A` *authmethod*
`--auth=`*authmethod*

This option specifies the authentication method for local users used in `pg_hba.conf`. Do not use `trust` unless you trust all local users on your system. `Trust` is the default for ease of installation.

`-D` *directory*
`--pgdata=`*directory*

This option specifies the directory where the database cluster should be stored. This is the only information required by `initdb`, but you can avoid writing it by setting the `PGDATA` environment variable, which can be convenient since the database server (`postgres`) can find the database directory later by the same variable.

`-E` *encoding*
`--encoding=`*encoding*

Selects the encoding of the template database. This will also be the default encoding of any database you create later, unless you override it there. The default is derived from the locale, or `SQL_ASCII` if that does not work. The character sets supported by the PostgreSQL server are described in Section 21.2.1.

`--locale=`*locale*

Sets the default locale for the database cluster. If this option is not specified, the locale is inherited from the environment that `initdb` runs in. Locale support is described in Section 21.1.

`--lc-collate=`*locale*
`--lc-ctype=`*locale*
`--lc-messages=`*locale*
`--lc-monetary=`*locale*
`--lc-numeric=`*locale*
`--lc-time=`*locale*

Like `--locale`, but only sets the locale in the specified category.

`-U` *username*
`--username=`*username*

Selects the user name of the database superuser. This defaults to the name of the effective user running `initdb`. It is really not important what the superuser's name is, but one might choose to keep the customary name postgres, even if the operating system user's name is different.

`-W`
`--pwprompt`

Makes `initdb` prompt for a password to give the database superuser. If you don't plan on using password authentication, this is not important. Otherwise you won't be able to use password authentication until you have a password set up.

`--pwfile=`*filename*

Makes `initdb` read the database superuser's password from a file. The first line of the file is taken as the password.

Other, less commonly used, parameters are also available:

`-d`
`--debug`

Print debugging output from the bootstrap backend and a few other messages of lesser interest for the general public. The bootstrap backend is the program `initdb` uses to create the catalog tables. This option generates a tremendous amount of extremely boring output.

`-L` *directory*

Specifies where `initdb` should find its input files to initialize the database cluster. This is normally not necessary. You will be told if you need to specify their location explicitly.

`-n`
`--noclean`

By default, when `initdb` determines that an error prevented it from completely creating the database cluster, it removes any files it may have created before discovering that it can't finish the job. This option inhibits tidying-up and is thus useful for debugging.

Environment

`PGDATA`

Specifies the directory where the database cluster is to be stored; may be overridden using the `-D` option.

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 29.12).

See Also

postgres

ipcclean

Name

`ipcclean` — remove shared memory and semaphores from a failed PostgreSQL server

Synopsis

```
ipcclean
```

Description

`ipcclean` removes all shared memory segments and semaphore sets owned by the current user. It is intended to be used for cleaning up after a crashed PostgreSQL server (postgres). Note that immediately restarting the server will also clean up shared memory and semaphores, so this command is of little real utility.

Only the database administrator should execute this program as it can cause bizarre behavior (i.e., crashes) if run during multiuser execution. If this command is executed while a server is running, the shared memory and semaphores allocated by that server will be deleted, which would have rather severe consequences for that server.

Notes

This script is a hack, but in the many years since it was written, no one has come up with an equally effective and portable solution. Since `postgres` can now clean up by itself, it is unlikely that `ipcclean` will be improved upon in the future.

The script makes assumptions about the output format of the `ipcs` utility which may not be true across different operating systems. Therefore, it may not work on your particular OS. It's wise to look at the script before trying it.

pg_controldata

Name

pg_controldata — display control information of a PostgreSQL database cluster

Synopsis

pg_controldata [*datadir*]

Description

pg_controldata prints information initialized during initdb, such as the catalog version and server locale. It also shows information about write-ahead logging and checkpoint processing. This information is cluster-wide, and not specific to any one database.

This utility may only be run by the user who initialized the cluster because it requires read access to the data directory. You can specify the data directory on the command line, or use the environment variable PGDATA.

Environment

PGDATA

Default data directory location

pg_ctl

Name

pg_ctl — start, stop, or restart a PostgreSQL server

Synopsis

pg_ctl start [-w] [-s] [-D *datadir*] [-l *filename*] [-o *options*] [-p *path*]
pg_ctl stop [-W] [-s] [-D *datadir*] [-m s[mart] | f[ast] | i[mmediate]]
pg_ctl restart [-w] [-s] [-D *datadir*] [-m s[mart] | f[ast] | i[mmediate]] [-o *options*]
pg_ctl reload [-s] [-D *datadir*]
pg_ctl status [-D *datadir*]
pg_ctl kill [*signal_name*] [*process_id*]
pg_ctl register [-N *servicename*] [-U *username*] [-P *password*] [-D *datadir*] [-w] [-o *options*]
pg_ctl unregister [-N *servicename*]

Description

pg_ctl is a utility for starting, stopping, or restarting the PostgreSQL backend server (postgres), or displaying the status of a running server. Although the server can be started manually, pg_ctl encapsulates tasks such as redirecting log output and properly detaching from the terminal and process group. It also provides convenient options for controlled shutdown.

In start mode, a new server is launched. The server is started in the background, and standard input is attached to /dev/null. The standard output and standard error are either appended to a log file (if the -l option is used), or redirected to pg_ctl's standard output (not standard error). If no log file is chosen, the standard output of pg_ctl should be redirected to a file or piped to another process such as a log rotating program like rotatelogs; otherwise postgres will write its output to the controlling terminal (from the background) and will not leave the shell's process group.

In stop mode, the server that is running in the specified data directory is shut down. Three different shutdown methods can be selected with the -m option: "Smart" mode waits for all the clients to disconnect. This is the default. "Fast" mode does not wait for clients to disconnect. All active transactions are rolled back and clients are forcibly disconnected, then the server is shut down. "Immediate" mode will abort all server processes without a clean shutdown. This will lead to a recovery run on restart.

restart mode effectively executes a stop followed by a start. This allows changing the postgres command-line options.

reload mode simply sends the postgres process a SIGHUP signal, causing it to reread its configuration files (postgresql.conf, pg_hba.conf, etc.). This allows changing of configuration-file options that do not require a complete restart to take effect.

status mode checks whether a server is running in the specified data directory. If it is, the PID and the command line options that were used to invoke it are displayed.

`kill` mode allows you to send a signal to a specified process. This is particularly valuable for Microsoft Windows which does not have a kill command. Use `--help` to see a list of supported signal names.

`register` mode allows you to register a system service on Microsoft Windows.

`unregister` mode allows you to unregister a system service on Microsoft Windows, previously registered with the `register` command.

Options

`-D` *`datadir`*

Specifies the file system location of the database files. If this is omitted, the environment variable `PGDATA` is used.

`-l` *`filename`*

Append the server log output to *`filename`*. If the file does not exist, it is created. The umask is set to 077, so access to the log file from other users is disallowed by default.

`-m` *`mode`*

Specifies the shutdown mode. *`mode`* may be `smart`, `fast`, or `immediate`, or the first letter of one of these three.

`-o` *`options`*

Specifies options to be passed directly to the `postgres` command.

The options are usually surrounded by single or double quotes to ensure that they are passed through as a group.

`-p` *`path`*

Specifies the location of the `postgres` executable. By default the `postgres` executable is taken from the same directory as `pg_ctl`, or failing that, the hard-wired installation directory. It is not necessary to use this option unless you are doing something unusual and get errors that the `postgres` executable was not found.

`-s`

Only print errors, no informational messages.

`-w`

Wait for the start or shutdown to complete. Times out after 60 seconds. This is the default for shutdowns. A successful shutdown is indicated by removal of the PID file. For starting up, a successful `psql -l` indicates success. `pg_ctl` will attempt to use the proper port for psql. If the environment variable `PGPORT` exists, that is used. Otherwise, it will see if a port has been set in the `postgresql.conf` file. If neither of those is used, it will use the default port that PostgreSQL was compiled with (5432 by default). When waiting, `pg_ctl` will return an accurate exit code based on the success of the startup or shutdown.

`-W`

Do not wait for start or shutdown to complete. This is the default for starts and restarts.

Options for Windows

`-N` *servicename*

Name of the system service to register. The name will be used as both the service name and the display name.

`-P` *password*

Password for the user to start the service.

`-U` *username*

User name for the user to start the service. For domain users, use the format `DOMAIN\username`.

Environment

`PGDATA`

Default data directory location.

`PGPORT`

Default port for psql (used by the -w option).

For additional server variables, see postgres. This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 29.12).

Files

`postmaster.pid`

The existence of this file in the data directory is used to help pg_ctl determine if the server is currently running or not.

`postmaster.opts.default`

If this file exists in the data directory, pg_ctl (in `start` mode) will pass the contents of the file as options to the `postgres` command, unless overridden by the `-o` option.

`postmaster.opts`

If this file exists in the data directory, pg_ctl (in `restart` mode) will pass the contents of the file as options to postgres, unless overridden by the `-o` option. The contents of this file are also displayed in `status` mode.

`postgresql.conf`

This file, located in the data directory, is parsed to find the proper port to use with psql when the `-w` is given in `start` mode.

Notes

Waiting for complete start is not a well-defined operation and may fail if access control is set up so that a local client cannot connect without manual interaction (e.g., password authentication).

Examples

Starting the Server

To start up a server:

```
$ pg_ctl start
```

An example of starting the server, blocking until the server has come up is:

```
$ pg_ctl -w start
```

For a server using port 5433, and running without `fsync`, use:

```
$ pg_ctl -o "-F -p 5433" start
```

Stopping the Server

```
$ pg_ctl stop
```

stops the server. Using the `-m` switch allows one to control *how* the backend shuts down.

Restarting the Server

Restarting the server is almost equivalent to stopping the server and starting it again except that `pg_ctl` saves and reuses the command line options that were passed to the previously running instance. To restart the server in the simplest form, use:

```
$ pg_ctl restart
```

To restart server, waiting for it to shut down and to come up:

```
$ pg_ctl -w restart
```

To restart using port 5433 and disabling `fsync` after restarting:

```
$ pg_ctl -o "-F -p 5433" restart
```

Showing the Server Status

Here is a sample status output from pg_ctl:

```
$ pg_ctl status
pg_ctl: server is running (pid: 13718)
Command line was:
/usr/local/pgsql/bin/postgres '-D' '/usr/local/pgsql/data' '-p' '5433' '-B' '128'
```

This is the command line that would be invoked in restart mode.

See Also

postgres

pg_resetxlog

Name

pg_resetxlog — reset the write-ahead log and other control information of a PostgreSQL database cluster

Synopsis

pg_resetxlog [-f] [-n] [-o*oid*] [-x *xid*] [-e *xid_epoch*] [-m *mxid*] [-O *mxoff*] [-l *timelineid*,*fileid*,*seg*] *datadir*

Description

pg_resetxlog clears the write-ahead log (WAL) and optionally resets some other control information stored in the pg_control file. This function is sometimes needed if these files have become corrupted. It should be used only as a last resort, when the server will not start due to such corruption.

After running this command, it should be possible to start the server, but bear in mind that the database may contain inconsistent data due to partially-committed transactions. You should immediately dump your data, run initdb, and reload. After reload, check for inconsistencies and repair as needed.

This utility can only be run by the user who installed the server, because it requires read/write access to the data directory. For safety reasons, you must specify the data directory on the command line. pg_resetxlog does not use the environment variable PGDATA.

If pg_resetxlog complains that it cannot determine valid data for pg_control, you can force it to proceed anyway by specifying the -f (force) switch. In this case plausible values will be substituted for the missing data. Most of the fields can be expected to match, but manual assistance may be needed for the next OID, next transaction ID and epoch, next multitransaction ID and offset, WAL starting address, and database locale fields. The first six of these can be set using the switches discussed below. pg_resetxlog's own environment is the source for its guess at the locale fields; take care that LANG and so forth match the environment that initdb was run in. If you are not able to determine correct values for all these fields, -f can still be used, but the recovered database must be treated with even more suspicion than usual: an immediate dump and reload is imperative. *Do not* execute any data-modifying operations in the database before you dump; as any such action is likely to make the corruption worse.

The -o, -x, -e, -m, -O, and -l switches allow the next OID, next transaction ID, next transaction ID's epoch, next multitransaction ID, next multitransaction offset, and WAL starting address values to be set manually. These are only needed when pg_resetxlog is unable to determine appropriate values by reading pg_control. Safe values may be determined as follows:

- A safe value for the next transaction ID (-x) may be determined by looking for the numerically largest file name in the directory pg_clog under the data directory, adding one, and then multiplying by 1048576. Note that the file names are in hexadecimal. It is usually easiest to specify the switch value in hexadecimal too. For example, if 0011 is the largest entry in pg_clog, -x 0x1200000 will work (five trailing zeroes provide the proper multiplier).

- A safe value for the next multitransaction ID (`-m`) may be determined by looking for the numerically largest file name in the directory `pg_multixact/offsets` under the data directory, adding one, and then multiplying by 65536. As above, the file names are in hexadecimal, so the easiest way to do this is to specify the switch value in hexadecimal and add four zeroes.
- A safe value for the next multitransaction offset (`-O`) may be determined by looking for the numerically largest file name in the directory `pg_multixact/members` under the data directory, adding one, and then multiplying by 65536. As above, the file names are in hexadecimal, so the easiest way to do this is to specify the switch value in hexadecimal and add four zeroes.
- The WAL starting address (`-l`) should be larger than any file name currently existing in the directory `pg_xlog` under the data directory. These names are also in hexadecimal and have three parts. The first part is the "timeline ID" and should usually be kept the same. Do not choose a value larger than 255 (`0xFF`) for the third part; instead increment the second part and reset the third part to 0. For example, if `000000010000003200000004A` is the largest entry in `pg_xlog`, `-l 0x1,0x32,0x4B` will work; but if the largest entry is `000000010000003A000000FF`, choose `-l 0x1,0x3B,0x0` or more.
- There is no comparably easy way to determine a next OID that's beyond the largest one in the database, but fortunately it is not critical to get the next-OID setting right.
- The transaction ID epoch is not actually stored anywhere in the database except in the field that is set by `pg_resetxlog`, so any value will work so far as the database itself is concerned. You might need to adjust this value to ensure that replication systems such as Slony-I work correctly — if so, an appropriate value should be obtainable from the state of the downstream replicated database.

The `-n` (no operation) switch instructs `pg_resetxlog` to print the values reconstructed from `pg_control` and then exit without modifying anything. This is mainly a debugging tool, but may be useful as a sanity check before allowing `pg_resetxlog` to proceed for real.

Notes

This command must not be used when the server is running. `pg_resetxlog` will refuse to start up if it finds a server lock file in the data directory. If the server crashed then a lock file may have been left behind; in that case you can remove the lock file to allow `pg_resetxlog` to run. But before you do so, make doubly certain that there is no server process still alive.

postgres

Name

postgres — PostgreSQL database server

Synopsis

postgres [*option*...]

Description

postgres is the PostgreSQL database server. In order for a client application to access a database it connects (over a network or locally) to a running postgres process. The postgres instance then starts a separate server process to handle the connection.

One postgres instance always manages the data from exactly one database cluster. A database cluster is a collection of databases that is stored at a common file system location (the "data area"). More than one postgres process can run on a system at one time, so long as they use different data areas and different communication ports (see below). When postgres starts it needs to know the location of the data area. The location must be specified by the -D option or the PGDATA environment variable; there is no default. Typically, -D or PGDATA points directly to the data area directory created by initdb. Other possible file layouts are discussed in Section 17.2.

By default postgres starts in the foreground and prints log messages to the standard error stream. In practical applications postgres should be started as a background process, perhaps at boot time.

The postgres command can also be called in single-user mode. The primary use for this mode is during bootstrapping by initdb. Sometimes it is used for debugging or disaster recovery (but note that running a single-user server is not truly suitable for debugging the server, since no realistic interprocess communication and locking will happen). When invoked in single-user mode from the shell, the user can enter queries and the results will be printed to the screen, but in a form that is more useful for developers than end users. In the single-user mode, the session user will be set to the user with ID 1, and implicit superuser powers are granted to this user. This user does not actually have to exist, so the single-user mode can be used to manually recover from certain kinds of accidental damage to the system catalogs.

Options

postgres accepts the following command-line arguments. For a detailed discussion of the options consult Chapter 17. You can save typing most of these options by setting up a configuration file. Some (safe) options can also be set from the connecting client in an application-dependent way to apply only for that session. For example, if the environment variable PGOPTIONS is set, then libpq-based clients will pass that string to the server, which will interpret it as postgres command-line options.

General Purpose

`-A 0|1`

Enables run-time assertion checks, which is a debugging aid to detect programming mistakes. This option is only available if assertions were enabled when PostgreSQL was compiled. If so, the default is on.

`-B` *nbuffers*

Sets the number of shared buffers for use by the server processes. The default value of this parameter is chosen automatically by initdb; refer to Section 17.4.1 for more information.

`-c` *name=value*

Sets a named run-time parameter. The configuration parameters supported by PostgreSQL are described in Chapter 17. Most of the other command line options are in fact short forms of such a parameter assignment. `-c` can appear multiple times to set multiple parameters.

`-d` *debug-level*

Sets the debug level. The higher this value is set, the more debugging output is written to the server log. Values are from 1 to 5. It is also possible to pass `-d 0` for a specific session, which will prevent the server log level of the parent `postgres` process from being propagated to this session.

`-D` *datadir*

Specifies the file system location of the data directory or configuration file(s). See Section 17.2 for details.

`-e`

Sets the default date style to "European", that is `DMY` ordering of input date fields. This also causes the day to be printed before the month in certain date output formats. See Section 8.5 for more information.

`-F`

Disables `fsync` calls for improved performance, at the risk of data corruption in the event of a system crash. Specifying this option is equivalent to disabling the fsync configuration parameter. Read the detailed documentation before using this!

`-h` *hostname*

Specifies the IP host name or address on which `postgres` is to listen for TCP/IP connections from client applications. The value can also be a comma-separated list of addresses, or `*` to specify listening on all available interfaces. An empty value specifies not listening on any IP addresses, in which case only Unix-domain sockets can be used to connect to the server. Defaults to listening only on localhost. Specifying this option is equivalent to setting the listen_addresses configuration parameter.

`-i`

Allows remote clients to connect via TCP/IP (Internet domain) connections. Without this option, only local connections are accepted. This option is equivalent to setting `listen_addresses` to `*` in `postgresql.conf` or via `-h`.

This option is deprecated since it does not allow access to the full functionality of listen_addresses. It's usually better to set `listen_addresses` directly.

`-k` *directory*

Specifies the directory of the Unix-domain socket on which `postgres` is to listen for connections from client applications. The default is normally `/tmp`, but can be changed at build time.

`-l`

Enables secure connections using SSL. PostgreSQL must have been compiled with support for SSL for this option to be available. For more information on using SSL, refer to Section 16.7.

`-N` *max-connections*

Sets the maximum number of client connections that this server will accept. By default, this value is 32, but it can be set as high as your system will support. (Note that `-B` is required to be at least twice `-N`. See Section 16.4 for a discussion of system resource requirements for large numbers of client connections.) Specifying this option is equivalent to setting the max_connections configuration parameter.

`-o` *extra-options*

The command line-style options specified in *extra-options* are passed to all server processes started by this `postgres` process. If the option string contains any spaces, the entire string must be quoted.

The use of this option is obsolete; all command-line options for server processes can be specified directly on the `postgres` command line.

`-p` *port*

Specifies the TCP/IP port or local Unix domain socket file extension on which `postgres` is to listen for connections from client applications. Defaults to the value of the `PGPORT` environment variable, or if `PGPORT` is not set, then defaults to the value established during compilation (normally 5432). If you specify a port other than the default port, then all client applications must specify the same port using either command-line options or `PGPORT`.

`-s`

Print time information and other statistics at the end of each command. This is useful for benchmarking or for use in tuning the number of buffers.

`-S` *work-mem*

Specifies the amount of memory to be used by internal sorts and hashes before resorting to temporary disk files. See the description of the `work_mem` configuration parameter in Section 17.4.1.

`--`*name=value*

Sets a named run-time parameter; a shorter form of `-c`.

`--describe-config`

This option dumps out the server's internal configuration variables, descriptions, and defaults in tab-delimited `COPY` format. It is designed primarily for use by administration tools.

Semi-internal Options

There are several other options that may be specified, used mainly for debugging purposes and in some cases to assist with recovery of severely damaged databases. There should be no reason to use them in

a production database setup. These are listed here only for the use by PostgreSQL system developers. Furthermore, any of these options may disappear or change in a future release without notice.

`-f { s | i | m | n | h }`

Forbids the use of particular scan and join methods: `s` and `i` disable sequential and index scans respectively, while `n`, `m`, and `h` disable nested-loop, merge and hash joins respectively.

Neither sequential scans nor nested-loop joins can be disabled completely; the `-fs` and `-fn` options simply discourage the optimizer from using those plan types if it has any other alternative.

`-n`

This option is for debugging problems that cause a server process to die abnormally. The ordinary strategy in this situation is to notify all other server processes that they must terminate and then reinitialize the shared memory and semaphores. This is because an errant server process could have corrupted some shared state before terminating. This option specifies that `postgres` will not reinitialize shared data structures. A knowledgeable system programmer can then use a debugger to examine shared memory and semaphore state.

`-O`

Allows the structure of system tables to be modified. This is used by `initdb`.

`-P`

Ignore system indexes when reading system tables (but still update the indexes when modifying the tables). This is useful when recovering from damaged system indexes.

`-t pa[rser] | pl[anner] | e[xecutor]`

Print timing statistics for each query relating to each of the major system modules. This option cannot be used together with the `-s` option.

`-T`

This option is for debugging problems that cause a server process to die abnormally. The ordinary strategy in this situation is to notify all other server processes that they must terminate and then reinitialize the shared memory and semaphores. This is because an errant server process could have corrupted some shared state before terminating. This option specifies that `postgres` will stop all other server processes by sending the signal `SIGSTOP`, but will not cause them to terminate. This permits system programmers to collect core dumps from all server processes by hand.

`-v` *`protocol`*

Specifies the version number of the frontend/backend protocol to be used for a particular session. This option is for internal use only.

`-W` *`seconds`*

A delay of this many seconds occurs when a new server process is started, after it conducts the authentication procedure. This is intended to give an opportunity to attach to the server process with a debugger.

`-y` *`database`*

Indicates that this is a subprocess started by a parent `postgres` process, and specifies the database to use. This option is for internal use only.

Options for single-user mode

The following options only apply to the single-user mode.

`--single`

Selects the single-user mode. This must be the first argument on the command line.

`database`

Specifies the name of the database to be accessed. If it is omitted it defaults to the user name.

`-E`

Echo all commands.

`-j`

Disables use of newline as a statement delimiter.

`-r` *`filename`*

Send all server log output to *`filename`*. In normal multiuser mode, this option is ignored, and stderr is used by all processes.

Environment

`PGCLIENTENCODING`

Default character encoding used by clients. (The clients may override this individually.) This value can also be set in the configuration file.

`PGDATA`

Default data directory location

`PGDATESTYLE`

Default value of the DateStyle run-time parameter. (The use of this environment variable is deprecated.)

`PGPORT`

Default port (preferably set in the configuration file)

`TZ`

Server time zone

Diagnostics

A failure message mentioning `semget` or `shmget` probably indicates you need to configure your kernel to provide adequate shared memory and semaphores. For more discussion see Section 16.4. You may be able to postpone reconfiguring your kernel by decreasing shared_buffers to reduce the shared memory consumption of PostgreSQL, and/or by reducing max_connections to reduce the semaphore consumption.

A failure message suggesting that another server is already running should be checked carefully, for example by using the command

```
$ ps ax | grep postgres
```

or

```
$ ps -ef | grep postgres
```

depending on your system. If you are certain that no conflicting server is running, you may remove the lock file mentioned in the message and try again.

A failure message indicating inability to bind to a port may indicate that that port is already in use by some non-PostgreSQL process. You may also get this error if you terminate `postgres` and immediately restart it using the same port; in this case, you must simply wait a few seconds until the operating system closes the port before trying again. Finally, you may get this error if you specify a port number that your operating system considers to be reserved. For example, many versions of Unix consider port numbers under 1024 to be "trusted" and only permit the Unix superuser to access them.

Notes

If at all possible, *do not* use `SIGKILL` to kill the main `postgres` server. Doing so will prevent `postgres` from freeing the system resources (e.g., shared memory and semaphores) that it holds before terminating. This may cause problems for starting a fresh `postgres` run.

To terminate the `postgres` server normally, the signals `SIGTERM`, `SIGINT`, or `SIGQUIT` can be used. The first will wait for all clients to terminate before quitting, the second will forcefully disconnect all clients, and the third will quit immediately without proper shutdown, resulting in a recovery run during restart. The `SIGHUP` signal will reload the server configuration files. It is also possible to send `SIGHUP` to an individual server process, but that is usually not sensible.

The utility command pg_ctl can be used to start and shut down the `postgres` server safely and comfortably.

To cancel a running query, send the `SIGINT` signal to the process running that command.

The `postgres` server uses `SIGTERM` to tell subordinate server processes to quit normally and `SIGQUIT` to terminate without the normal cleanup. These signals *should not* be used by users. It is also unwise to send `SIGKILL` to a server process — the main `postgres` process will interpret this as a crash and will force all the sibling processes to quit as part of its standard crash-recovery procedure.

Bugs

The `--` options will not work on FreeBSD or OpenBSD. Use `-c` instead. This is a bug in the affected operating systems; a future release of PostgreSQL will provide a workaround if this is not fixed.

Usage

To start a single-user mode server, use a command like

```
postgres --single -D /usr/local/pgsql/data other-options my_database
```

Provide the correct path to the database directory with `-D`, or make sure that the environment variable `PGDATA` is set. Also specify the name of the particular database you want to work in.

Normally, the single-user mode server treats newline as the command entry terminator; there is no intelligence about semicolons, as there is in psql. To continue a command across multiple lines, you must type backslash just before each newline except the last one.

But if you use the `-j` command line switch, then newline does not terminate command entry. In this case, the server will read the standard input until the end-of-file (EOF) marker, then process the input as a single command string. Backslash-newline is not treated specially in this case.

To quit the session, type EOF (**Control+D**, usually). If you've used `-j`, two consecutive EOFs are needed to exit.

Note that the single-user mode server does not provide sophisticated line-editing features (no command history, for example).

Examples

To start `postgres` in the background using default values, type:

```
$ nohup postgres >logfile 2>&1 </dev/null &
```

To start `postgres` with a specific port:

```
$ postgres -p 1234
```

This command will start up `postgres` communicating through the port 1234. In order to connect to this server using psql, you would need to run it as

```
$ psql -p 1234
```

or set the environment variable `PGPORT`:

```
$ export PGPORT=1234
$ psql
```

Named run-time parameters can be set in either of these styles:

```
$ postgres -c work_mem=1234
$ postgres --work-mem=1234
```

Either form overrides whatever setting might exist for `work_mem` in `postgresql.conf`. Notice that underscores in parameter names can be written as either underscore or dash on the command line. Except

for short-term experiments, it's probably better practice to edit the setting in `postgresql.conf` than to rely on a command-line switch to set a parameter.

See Also

initdb, pg_ctl

postmaster

Name

`postmaster` — PostgreSQL database server

Synopsis

`postmaster` [*`option`*...]

Description

`postmaster` is a deprecated alias of `postgres`.

See Also

postgres

VII. Internals

This part contains assorted information that can be of use to PostgreSQL developers.

Chapter 42. Overview of PostgreSQL Internals

Author: This chapter originated as part of *Enhancement of the ANSI SQL Implementation of PostgreSQL*, Stefan Simkovics' Master's Thesis prepared at Vienna University of Technology under the direction of O.Univ.Prof.Dr. Georg Gottlob and Univ.Ass. Mag. Katrin Seyr.

This chapter gives an overview of the internal structure of the backend of PostgreSQL. After having read the following sections you should have an idea of how a query is processed. This chapter does not aim to provide a detailed description of the internal operation of PostgreSQL, as such a document would be very extensive. Rather, this chapter is intended to help the reader understand the general sequence of operations that occur within the backend from the point at which a query is received, to the point at which the results are returned to the client.

42.1. The Path of a Query

Here we give a short overview of the stages a query has to pass in order to obtain a result.

1. A connection from an application program to the PostgreSQL server has to be established. The application program transmits a query to the server and waits to receive the results sent back by the server.
2. The *parser stage* checks the query transmitted by the application program for correct syntax and creates a *query tree*.
3. The *rewrite system* takes the query tree created by the parser stage and looks for any *rules* (stored in the *system catalogs*) to apply to the query tree. It performs the transformations given in the *rule bodies*.

   One application of the rewrite system is in the realization of *views*. Whenever a query against a view (i.e. a *virtual table*) is made, the rewrite system rewrites the user's query to a query that accesses the *base tables* given in the *view definition* instead.
4. The *planner/optimizer* takes the (rewritten) query tree and creates a *query plan* that will be the input to the *executor*.

   It does so by first creating all possible *paths* leading to the same result. For example if there is an index on a relation to be scanned, there are two paths for the scan. One possibility is a simple sequential scan and the other possibility is to use the index. Next the cost for the execution of each path is estimated and the cheapest path is chosen. The cheapest path is expanded into a complete plan that the executor can use.
5. The executor recursively steps through the *plan tree* and retrieves rows in the way represented by the plan. The executor makes use of the *storage system* while scanning relations, performs *sorts* and *joins*, evaluates *qualifications* and finally hands back the rows derived.

In the following sections we will cover each of the above listed items in more detail to give a better understanding of PostgreSQL's internal control and data structures.

42.2. How Connections are Established

PostgreSQL is implemented using a simple "process per user" client/server model. In this model there is one *client process* connected to exactly one *server process*. As we do not know ahead of time how many connections will be made, we have to use a *master process* that spawns a new server process every time a connection is requested. This master process is called `postgres` and listens at a specified TCP/IP port for incoming connections. Whenever a request for a connection is detected the `postgres` process spawns a new server process. The server tasks communicate with each other using *semaphores* and *shared memory* to ensure data integrity throughout concurrent data access.

The client process can be any program that understands the PostgreSQL protocol described in Chapter 44. Many clients are based on the C-language library libpq, but several independent implementations of the protocol exist, such as the Java JDBC driver.

Once a connection is established the client process can send a query to the *backend* (server). The query is transmitted using plain text, i.e. there is no parsing done in the *frontend* (client). The server parses the query, creates an *execution plan*, executes the plan and returns the retrieved rows to the client by transmitting them over the established connection.

42.3. The Parser Stage

The *parser stage* consists of two parts:

- The *parser* defined in `gram.y` and `scan.l` is built using the Unix tools yacc and lex.
- The *transformation process* does modifications and augmentations to the data structures returned by the parser.

42.3.1. Parser

The parser has to check the query string (which arrives as plain ASCII text) for valid syntax. If the syntax is correct a *parse tree* is built up and handed back; otherwise an error is returned. The parser and lexer are implemented using the well-known Unix tools yacc and lex.

The *lexer* is defined in the file `scan.l` and is responsible for recognizing *identifiers*, the *SQL key words* etc. For every key word or identifier that is found, a *token* is generated and handed to the parser.

The parser is defined in the file `gram.y` and consists of a set of *grammar rules* and *actions* that are executed whenever a rule is fired. The code of the actions (which is actually C code) is used to build up the parse tree.

The file `scan.l` is transformed to the C source file `scan.c` using the program lex and `gram.y` is transformed to `gram.c` using yacc. After these transformations have taken place a normal C compiler can be used to create the parser. Never make any changes to the generated C files as they will be overwritten the next time lex or yacc is called.

Note: The mentioned transformations and compilations are normally done automatically using the *makefiles* shipped with the PostgreSQL source distribution.

A detailed description of yacc or the grammar rules given in `gram.y` would be beyond the scope of this paper. There are many books and documents dealing with lex and yacc. You should be familiar with yacc before you start to study the grammar given in `gram.y` otherwise you won't understand what happens there.

42.3.2. Transformation Process

The parser stage creates a parse tree using only fixed rules about the syntactic structure of SQL. It does not make any lookups in the system catalogs, so there is no possibility to understand the detailed semantics of the requested operations. After the parser completes, the *transformation process* takes the tree handed back by the parser as input and does the semantic interpretation needed to understand which tables, functions, and operators are referenced by the query. The data structure that is built to represent this information is called the *query tree*.

The reason for separating raw parsing from semantic analysis is that system catalog lookups can only be done within a transaction, and we do not wish to start a transaction immediately upon receiving a query string. The raw parsing stage is sufficient to identify the transaction control commands (`BEGIN`, `ROLLBACK`, etc), and these can then be correctly executed without any further analysis. Once we know that we are dealing with an actual query (such as `SELECT` or `UPDATE`), it is okay to start a transaction if we're not already in one. Only then can the transformation process be invoked.

The query tree created by the transformation process is structurally similar to the raw parse tree in most places, but it has many differences in detail. For example, a `FuncCall` node in the parse tree represents something that looks syntactically like a function call. This may be transformed to either a `FuncExpr` or `Aggref` node depending on whether the referenced name turns out to be an ordinary function or an aggregate function. Also, information about the actual data types of columns and expression results is added to the query tree.

42.4. The PostgreSQL Rule System

PostgreSQL supports a powerful *rule system* for the specification of *views* and ambiguous *view updates*. Originally the PostgreSQL rule system consisted of two implementations:

- The first one worked using *row level* processing and was implemented deep in the *executor*. The rule system was called whenever an individual row had been accessed. This implementation was removed in 1995 when the last official release of the Berkeley Postgres project was transformed into Postgres95.
- The second implementation of the rule system is a technique called *query rewriting*. The *rewrite system* is a module that exists between the *parser stage* and the *planner/optimizer*. This technique is still implemented.

The query rewriter is discussed in some detail in Chapter 35, so there is no need to cover it here. We will only point out that both the input and the output of the rewriter are query trees, that is, there is no change in the representation or level of semantic detail in the trees. Rewriting can be thought of as a form of macro expansion.

42.5. Planner/Optimizer

The task of the *planner/optimizer* is to create an optimal execution plan. A given SQL query (and hence, a query tree) can be actually executed in a wide variety of different ways, each of which will produce the same set of results. If it is computationally feasible, the query optimizer will examine each of these possible execution plans, ultimately selecting the execution plan that is expected to run the fastest.

> **Note:** In some situations, examining each possible way in which a query may be executed would take an excessive amount of time and memory space. In particular, this occurs when executing queries involving large numbers of join operations. In order to determine a reasonable (not optimal) query plan in a reasonable amount of time, PostgreSQL uses a *Genetic Query Optimizer*.

The planner's search procedure actually works with data structures called *paths*, which are simply cut-down representations of plans containing only as much information as the planner needs to make its decisions. After the cheapest path is determined, a full-fledged *plan tree* is built to pass to the executor. This represents the desired execution plan in sufficient detail for the executor to run it. In the rest of this section we'll ignore the distinction between paths and plans.

42.5.1. Generating Possible Plans

The planner/optimizer starts by generating plans for scanning each individual relation (table) used in the query. The possible plans are determined by the available indexes on each relation. There is always the possibility of performing a sequential scan on a relation, so a sequential scan plan is always created. Assume an index is defined on a relation (for example a B-tree index) and a query contains the restriction `relation.attribute OPR constant`. If `relation.attribute` happens to match the key of the B-tree index and `OPR` is one of the operators listed in the index's *operator class*, another plan is created using the B-tree index to scan the relation. If there are further indexes present and the restrictions in the query happen to match a key of an index further plans will be considered.

After all feasible plans have been found for scanning single relations, plans for joining relations are created. The planner/optimizer preferentially considers joins between any two relations for which there exist a corresponding join clause in the `WHERE` qualification (i.e. for which a restriction like `where rel1.attr1=rel2.attr2` exists). Join pairs with no join clause are considered only when there is no other choice, that is, a particular relation has no available join clauses to any other relation. All possible plans are generated for every join pair considered by the planner/optimizer. The three possible join strategies are:

- *nested loop join*: The right relation is scanned once for every row found in the left relation. This strategy is easy to implement but can be very time consuming. (However, if the right relation can be scanned

with an index scan, this can be a good strategy. It is possible to use values from the current row of the left relation as keys for the index scan of the right.)

- *merge sort join*: Each relation is sorted on the join attributes before the join starts. Then the two relations are scanned in parallel, and matching rows are combined to form join rows. This kind of join is more attractive because each relation has to be scanned only once. The required sorting may be achieved either by an explicit sort step, or by scanning the relation in the proper order using an index on the join key.
- *hash join*: the right relation is first scanned and loaded into a hash table, using its join attributes as hash keys. Next the left relation is scanned and the appropriate values of every row found are used as hash keys to locate the matching rows in the table.

When the query involves more than two relations, the final result must be built up by a tree of join steps, each with two inputs. The planner examines different possible join sequences to find the cheapest one.

The finished plan tree consists of sequential or index scans of the base relations, plus nested-loop, merge, or hash join nodes as needed, plus any auxiliary steps needed, such as sort nodes or aggregate-function calculation nodes. Most of these plan node types have the additional ability to do *selection* (discarding rows that do not meet a specified boolean condition) and *projection* (computation of a derived column set based on given column values, that is, evaluation of scalar expressions where needed). One of the responsibilities of the planner is to attach selection conditions from the `WHERE` clause and computation of required output expressions to the most appropriate nodes of the plan tree.

42.6. Executor

The *executor* takes the plan handed back by the planner/optimizer and recursively processes it to extract the required set of rows. This is essentially a demand-pull pipeline mechanism. Each time a plan node is called, it must deliver one more row, or report that it is done delivering rows.

To provide a concrete example, assume that the top node is a `MergeJoin` node. Before any merge can be done two rows have to be fetched (one from each subplan). So the executor recursively calls itself to process the subplans (it starts with the subplan attached to `lefttree`). The new top node (the top node of the left subplan) is, let's say, a `Sort` node and again recursion is needed to obtain an input row. The child node of the `Sort` might be a `SeqScan` node, representing actual reading of a table. Execution of this node causes the executor to fetch a row from the table and return it up to the calling node. The `Sort` node will repeatedly call its child to obtain all the rows to be sorted. When the input is exhausted (as indicated by the child node returning a NULL instead of a row), the `Sort` code performs the sort, and finally is able to return its first output row, namely the first one in sorted order. It keeps the remaining rows stored so that it can deliver them in sorted order in response to later demands.

The `MergeJoin` node similarly demands the first row from its right subplan. Then it compares the two rows to see if they can be joined; if so, it returns a join row to its caller. On the next call, or immediately if it cannot join the current pair of inputs, it advances to the next row of one table or the other (depending on how the comparison came out), and again checks for a match. Eventually, one subplan or the other is exhausted, and the `MergeJoin` node returns NULL to indicate that no more join rows can be formed.

Complex queries may involve many levels of plan nodes, but the general approach is the same: each node computes and returns its next output row each time it is called. Each node is also responsible for applying any selection or projection expressions that were assigned to it by the planner.

The executor mechanism is used to evaluate all four basic SQL query types: `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. For `SELECT`, the top-level executor code only needs to send each row returned by the query plan tree off to the client. For `INSERT`, each returned row is inserted into the target table specified for the `INSERT`. (A simple `INSERT ... VALUES` command creates a trivial plan tree consisting of a single `Result` node, which computes just one result row. But `INSERT ... SELECT` may demand the full power of the executor mechanism.) For `UPDATE`, the planner arranges that each computed row includes all the updated column values, plus the *TID* (tuple ID, or row ID) of the original target row; the executor top level uses this information to create a new updated row and mark the old row deleted. For `DELETE`, the only column that is actually returned by the plan is the TID, and the executor top level simply uses the TID to visit each target row and mark it deleted.

Chapter 43. System Catalogs

The system catalogs are the place where a relational database management system stores schema metadata, such as information about tables and columns, and internal bookkeeping information. PostgreSQL's system catalogs are regular tables. You can drop and recreate the tables, add columns, insert and update values, and severely mess up your system that way. Normally, one should not change the system catalogs by hand, there are always SQL commands to do that. (For example, `CREATE DATABASE` inserts a row into the `pg_database` catalog — and actually creates the database on disk.) There are some exceptions for particularly esoteric operations, such as adding index access methods.

43.1. Overview

Table 43-1 lists the system catalogs. More detailed documentation of each catalog follows below.

Most system catalogs are copied from the template database during database creation and are thereafter database-specific. A few catalogs are physically shared across all databases in a cluster; these are noted in the descriptions of the individual catalogs.

Table 43-1. System Catalogs

Catalog Name	Purpose
`pg_aggregate`	aggregate functions
`pg_am`	index access methods
`pg_amop`	access method operators
`pg_amproc`	access method support procedures
`pg_attrdef`	column default values
`pg_attribute`	table columns ("attributes")
`pg_authid`	authorization identifiers (roles)
`pg_auth_members`	authorization identifier membership relationships
`pg_autovacuum`	per-relation autovacuum configuration parameters
`pg_cast`	casts (data type conversions)
`pg_class`	tables, indexes, sequences, views ("relations")
`pg_constraint`	check constraints, unique constraints, primary key constraints, foreign key constraints
`pg_conversion`	encoding conversion information
`pg_database`	databases within this database cluster
`pg_depend`	dependencies between database objects
`pg_description`	descriptions or comments on database objects
`pg_index`	additional index information
`pg_inherits`	table inheritance hierarchy

Catalog Name	Purpose
pg_language	languages for writing functions
pg_largeobject	large objects
pg_listener	asynchronous notification support
pg_namespace	schemas
pg_opclass	index access method operator classes
pg_operator	operators
pg_pltemplate	template data for procedural languages
pg_proc	functions and procedures
pg_rewrite	query rewrite rules
pg_shdepend	dependencies on shared objects
pg_shdescription	comments on shared objects
pg_statistic	planner statistics
pg_tablespace	tablespaces within this database cluster
pg_trigger	triggers
pg_type	data types

43.2. pg_aggregate

The catalog pg_aggregate stores information about aggregate functions. An aggregate function is a function that operates on a set of values (typically one column from each row that matches a query condition) and returns a single value computed from all these values. Typical aggregate functions are sum, count, and max. Each entry in pg_aggregate is an extension of an entry in pg_proc. The pg_proc entry carries the aggregate's name, input and output data types, and other information that is similar to ordinary functions.

Table 43-2. pg_aggregate Columns

Name	Type	References	Description
aggfnoid	regproc	pg_proc.oid	pg_proc OID of the aggregate function
aggtransfn	regproc	pg_proc.oid	Transition function
aggfinalfn	regproc	pg_proc.oid	Final function (zero if none)
aggsortop	oid	pg_operator.oid	Associated sort operator (zero if none)
aggtranstype	oid	pg_type.oid	Data type of the aggregate function's internal transition (state) data

Name	Type	References	Description
agginitval	text		The initial value of the transition state. This is a text field containing the initial value in its external string representation. If this field is NULL, the transition state value starts out NULL

New aggregate functions are registered with the *CREATE AGGREGATE* command. See Section 33.10 for more information about writing aggregate functions and the meaning of the transition functions, etc.

43.3. pg_am

The catalog pg_am stores information about index access methods. There is one row for each index access method supported by the system. The contents of this catalog are discussed in detail in Chapter 49.

Table 43-3. pg_am Columns

Name	Type	References	Description
amname	name		Name of the access method
amstrategies	int2		Number of operator strategies for this access method
amsupport	int2		Number of support routines for this access method
amorderstrategy	int2		Zero if the index offers no sort order, otherwise the strategy number of the strategy operator that describes the sort order
amcanunique	bool		Does the access method support unique indexes?
amcanmulticol	bool		Does the access method support multicolumn indexes?
amoptionalkey	bool		Does the access method support a scan without any constraint for the first index column?

Name	Type	References	Description
amindexnulls	bool		Does the access method support null index entries?
amstorage	bool		Can index storage data type differ from column data type?
amclusterable	bool		Can an index of this type be clustered on?
aminsert	regproc	pg_proc.oid	"Insert this tuple" function
ambeginscan	regproc	pg_proc.oid	"Start new scan" function
amgettuple	regproc	pg_proc.oid	"Next valid tuple" function
amgetmulti	regproc	pg_proc.oid	"Fetch multiple tuples" function
amrescan	regproc	pg_proc.oid	"Restart this scan" function
amendscan	regproc	pg_proc.oid	"End this scan" function
ammarkpos	regproc	pg_proc.oid	"Mark current scan position" function
amrestrpos	regproc	pg_proc.oid	"Restore marked scan position" function
ambuild	regproc	pg_proc.oid	"Build new index" function
ambulkdelete	regproc	pg_proc.oid	Bulk-delete function
amvacuumcleanup	regproc	pg_proc.oid	Post-VACUUM cleanup function
amcostestimate	regproc	pg_proc.oid	Function to estimate cost of an index scan
amoptions	regproc	pg_proc.oid	Function to parse and validate reloptions for an index

43.4. pg_amop

The catalog pg_amop stores information about operators associated with index access method operator classes. There is one row for each operator that is a member of an operator class.

Table 43-4. `pg_amop` Columns

Name	Type	References	Description
amopclaid	oid	pg_opclass.oid	The index operator class this entry is for
amopsubtype	oid	pg_type.oid	Subtype to distinguish multiple entries for one strategy; zero for default
amopstrategy	int2		Operator strategy number
amopreqcheck	bool		Index hit must be rechecked
amopopr	oid	pg_operator.oid	OID of the operator

43.5. pg_amproc

The catalog pg_amproc stores information about support procedures associated with index access method operator classes. There is one row for each support procedure belonging to an operator class.

Table 43-5. `pg_amproc` Columns

Name	Type	References	Description
amopclaid	oid	pg_opclass.oid	The index operator class this entry is for
amprocsubtype	oid	pg_type.oid	Subtype, if cross-type routine, else zero
amprocnum	int2		Support procedure number
amproc	regproc	pg_proc.oid	OID of the procedure

43.6. pg_attrdef

The catalog pg_attrdef stores column default values. The main information about columns is stored in pg_attribute (see below). Only columns that explicitly specify a default value (when the table is created or the column is added) will have an entry here.

Table 43-6. `pg_attrdef` Columns

Name	Type	References	Description
adrelid	oid	pg_class.oid	The table this column belongs to

Name	Type	References	Description
adnum	int2	pg_attribute.attnum	The number of the column
adbin	text		The internal representation of the column default value
adsrc	text		A human-readable representation of the default value

The adsrc field is historical, and is best not used, because it does not track outside changes that might affect the representation of the default value. Reverse-compiling the adbin field (with pg_get_expr for example) is a better way to display the default value.

43.7. pg_attribute

The catalog pg_attribute stores information about table columns. There will be exactly one pg_attribute row for every column in every table in the database. (There will also be attribute entries for indexes, and indeed all objects that have pg_class entries.)

The term attribute is equivalent to column and is used for historical reasons.

Table 43-7. pg_attribute Columns

Name	Type	References	Description
attrelid	oid	pg_class.oid	The table this column belongs to
attname	name		The column name
atttypid	oid	pg_type.oid	The data type of this column

Name	Type	References	Description
attstattarget	int4		attstattarget controls the level of detail of statistics accumulated for this column by *ANALYZE*. A zero value indicates that no statistics should be collected. A negative value says to use the system default statistics target. The exact meaning of positive values is data type-dependent. For scalar data types, attstattarget is both the target number of "most common values" to collect, and the target number of histogram bins to create
attlen	int2		A copy of pg_type.typlen of this column's type
attnum	int2		The number of the column. Ordinary columns are numbered from 1 up. System columns, such as oid, have (arbitrary) negative numbers
attndims	int4		Number of dimensions, if the column is an array type; otherwise 0. (Presently, the number of dimensions of an array is not enforced, so any nonzero value effectively means "it's an array")

Name	Type	References	Description
attcacheoff	int4		Always -1 in storage, but when loaded into a row descriptor in memory this may be updated to cache the offset of the attribute within the row
atttypmod	int4		atttypmod records type-specific data supplied at table creation time (for example, the maximum length of a varchar column). It is passed to type-specific input functions and length coercion functions. The value will generally be -1 for types that do not need atttypmod
attbyval	bool		A copy of pg_type.typbyval of this column's type
attstorage	char		Normally a copy of pg_type.typstorage of this column's type. For TOAST-able data types, this can be altered after column creation to control storage policy
attalign	char		A copy of pg_type.typalign of this column's type
attnotnull	bool		This represents a not-null constraint. It is possible to change this column to enable or disable the constraint
atthasdef	bool		This column has a default value, in which case there will be a corresponding entry in the pg_attrdef catalog that actually defines the value

Name	Type	References	Description
attisdropped	bool		This column has been dropped and is no longer valid. A dropped column is still physically present in the table, but is ignored by the parser and so cannot be accessed via SQL
attislocal	bool		This column is defined locally in the relation. Note that a column may be locally defined and inherited simultaneously
attinhcount	int4		The number of direct ancestors this column has. A column with a nonzero number of ancestors cannot be dropped nor renamed

In a dropped column's pg_attribute entry, atttypid is reset to zero, but attlen and the other fields copied from pg_type are still valid. This arrangement is needed to cope with the situation where the dropped column's data type was later dropped, and so there is no pg_type row anymore. attlen and the other fields can be used to interpret the contents of a row of the table.

43.8. pg_authid

The catalog pg_authid contains information about database authorization identifiers (roles). A role subsumes the concepts of "users" and "groups". A user is essentially just a role with the rolcanlogin flag set. Any role (with or without rolcanlogin) may have other roles as members; see pg_auth_members.

Since this catalog contains passwords, it must not be publicly readable. pg_roles is a publicly readable view on pg_authid that blanks out the password field.

Chapter 18 contains detailed information about user and privilege management.

Because user identities are cluster-wide, pg_authid is shared across all databases of a cluster: there is only one copy of pg_authid per cluster, not one per database.

Table 43-8. pg_authid Columns

Name	Type	Description	
rolname	name	Role name	
rolsuper	bool	Role has superuser privileges	

Name	Type	Description	
rolinherit	bool	Role automatically inherits privileges of roles it is a member of	
rolcreaterole	bool	Role may create more roles	
rolcreatedb	bool	Role may create databases	
rolcatupdate	bool	Role may update system catalogs directly. (Even a superuser may not do this unless this column is true)	
rolcanlogin	bool	Role may log in. That is, this role can be given as the initial session authorization identifier	
rolconnlimit	int4	For roles that can log in, this sets maximum number of concurrent connections this role can make. -1 means no limit	
rolpassword	text	Password (possibly encrypted); NULL if none	
rolvaliduntil	timestamptz	Password expiry time (only used for password authentication); NULL if no expiration	
rolconfig	text[]	Session defaults for run-time configuration variables	

43.9. pg_auth_members

The catalog pg_auth_members shows the membership relations between roles. Any non-circular set of relationships is allowed.

Because user identities are cluster-wide, pg_auth_members is shared across all databases of a cluster: there is only one copy of pg_auth_members per cluster, not one per database.

Table 43-9. pg_auth_members Columns

Name	Type	References	Description
roleid	oid	pg_authid.oid	ID of a role that has a member
member	oid	pg_authid.oid	ID of a role that is a member of roleid
grantor	oid	pg_authid.oid	ID of the role that granted this membership
admin_option	bool		True if member may grant membership in roleid to others

43.10. pg_autovacuum

The catalog pg_autovacuum stores optional per-relation configuration parameters for the autovacuum daemon. If there is an entry here for a particular relation, the given parameters will be used for autovacuuming that table. If no entry is present, the system-wide defaults will be used. For more information about the autovacuum daemon, see Section 22.1.4.

Table 43-10. pg_autovacuum Columns

Name	Type	References	Description
vacrelid	oid	pg_class.oid	The table this entry is for
enabled	bool		If false, this table is never autovacuumed
vac_base_thresh	integer		Minimum number of modified tuples before vacuum
vac_scale_factor	float4		Multiplier for reltuples to add to vac_base_thresh
anl_base_thresh	integer		Minimum number of modified tuples before analyze
anl_scale_factor	float4		Multiplier for reltuples to add to anl_base_thresh
vac_cost_delay	integer		Custom vacuum_cost_delay parameter

Name	Type	References	Description
vac_cost_limit	integer		Custom vacuum_cost_limit parameter
freeze_min_age	integer		Custom vacuum_freeze_min_age parameter
freeze_max_age	integer		Custom autovacuum_freeze_max_age parameter

The autovacuum daemon will initiate a VACUUM operation on a particular table when the number of updated or deleted tuples exceeds vac_base_thresh plus vac_scale_factor times the number of live tuples currently estimated to be in the relation. Similarly, it will initiate an ANALYZE operation when the number of inserted, updated or deleted tuples exceeds anl_base_thresh plus anl_scale_factor times the number of live tuples currently estimated to be in the relation.

Also, the autovacuum daemon will perform a VACUUM operation to prevent transaction ID wraparound if the table's pg_class.relfrozenxid field attains an age of more than freeze_max_age transactions, whether the table has been changed or not. The system will launch autovacuum to perform such VACUUMs even if autovacuum is otherwise disabled. See Section 22.1.3 for more about wraparound prevention.

Any of the numerical fields can contain -1 (or indeed any negative value) to indicate that the system-wide default should be used for this particular value. Observe that the vac_cost_delay variable inherits its default value from the autovacuum_vacuum_cost_delay configuration parameter, or from vacuum_cost_delay if the former is set to a negative value. The same applies to vac_cost_limit. Also, autovacuum will ignore attempts to set a per-table freeze_max_age larger than the system-wide setting (it can only be set smaller), and the freeze_min_age value will be limited to half the system-wide autovacuum_freeze_max_age setting.

43.11. pg_cast

The catalog pg_cast stores data type conversion paths, both built-in paths and those defined with *CREATE CAST*.

Table 43-11. pg_cast Columns

Name	Type	References	Description
castsource	oid	pg_type.oid	OID of the source data type
casttarget	oid	pg_type.oid	OID of the target data type

Name	Type	References	Description
castfunc	oid	pg_proc.oid	The OID of the function to use to perform this cast. Zero is stored if the data types are binary compatible (that is, no run-time operation is needed to perform the cast)
castcontext	char		Indicates what contexts the cast may be invoked in. e means only as an explicit cast (using CAST or :: syntax). a means implicitly in assignment to a target column, as well as explicitly. i means implicitly in expressions, as well as the other cases

The cast functions listed in pg_cast must always take the cast source type as their first argument type, and return the cast destination type as their result type. A cast function can have up to three arguments. The second argument, if present, must be type integer; it receives the type modifier associated with the destination type, or -1 if there is none. The third argument, if present, must be type boolean; it receives true if the cast is an explicit cast, false otherwise.

It is legitimate to create a pg_cast entry in which the source and target types are the same, if the associated function takes more than one argument. Such entries represent "length coercion functions" that coerce values of the type to be legal for a particular type modifier value. Note however that at present there is no support for associating non-default type modifiers with user-created data types, and so this facility is only of use for the small number of built-in types that have type modifier syntax built into the grammar.

When a pg_cast entry has different source and target types and a function that takes more than one argument, it represents converting from one type to another and applying a length coercion in a single step. When no such entry is available, coercion to a type that uses a type modifier involves two steps, one to convert between data types and a second to apply the modifier.

43.12. pg_class

The catalog pg_class catalogs tables and most everything else that has columns or is otherwise similar to a table. This includes indexes (but see also pg_index), sequences, views, composite types, and TOAST tables; see relkind. Below, when we mean all of these kinds of objects we speak of "relations". Not all columns are meaningful for all relation types.

Table 43-12. pg_class Columns

Name	Type	References	Description
relname	name		Name of the table, index, view, etc.
relnamespace	oid	pg_namespace.oid	The OID of the namespace that contains this relation
reltype	oid	pg_type.oid	The OID of the data type that corresponds to this table's row type, if any (zero for indexes, which have no pg_type entry)
relowner	oid	pg_authid.oid	Owner of the relation
relam	oid	pg_am.oid	If this is an index, the access method used (B-tree, hash, etc.)
relfilenode	oid		Name of the on-disk file of this relation; 0 if none
reltablespace	oid	pg_tablespace.oid	The tablespace in which this relation is stored. If zero, the database's default tablespace is implied. (Not meaningful if the relation has no on-disk file.)
relpages	int4		Size of the on-disk representation of this table in pages (of size BLCKSZ). This is only an estimate used by the planner. It is updated by VACUUM, ANALYZE, and a few DDL commands such as CREATE INDEX
reltuples	float4		Number of rows in the table. This is only an estimate used by the planner. It is updated by VACUUM, ANALYZE, and a few DDL commands such as CREATE INDEX

Name	Type	References	Description
reltoastrelid	oid	pg_class.oid	OID of the TOAST table associated with this table, 0 if none. The TOAST table stores large attributes "out of line" in a secondary table
reltoastidxid	oid	pg_class.oid	For a TOAST table, the OID of its index. 0 if not a TOAST table
relhasindex	bool		True if this is a table and it has (or recently had) any indexes. This is set by CREATE INDEX, but not cleared immediately by DROP INDEX. VACUUM clears relhasindex if it finds the table has no indexes
relisshared	bool		True if this table is shared across all databases in the cluster. Only certain system catalogs (such as pg_database) are shared
relkind	char		r = ordinary table, i = index, S = sequence, v = view, c = composite type, t = TOAST table
relnatts	int2		Number of user columns in the relation (system columns not counted). There must be this many corresponding entries in pg_attribute. See also pg_attribute.attnum
relchecks	int2		Number of check constraints on the table; see pg_constraint catalog

Name	Type	References	Description
reltriggers	int2		Number of triggers on the table; see pg_trigger catalog
relukeys	int2		Unused (*not* the number of unique keys)
relfkeys	int2		Unused (*not* the number of foreign keys on the table)
relrefs	int2		Unused
relhasoids	bool		True if we generate an OID for each row of the relation
relhaspkey	bool		True if the table has (or once had) a primary key
relhasrules	bool		True if table has rules; see pg_rewrite catalog
relhassubclass	bool		True if table has (or once had) any inheritance children
relfrozenxid	xid		All transaction IDs before this one have been replaced with a permanent ("frozen") transaction ID in this table. This is used to track whether the table needs to be vacuumed in order to prevent transaction ID wraparound or to allow pg_clog to be shrunk. Zero (InvalidTransactionId) if the relation is not a table
relacl	aclitem[]		Access privileges; see *GRANT* and *REVOKE* for details
reloptions	text[]		Access-method-specific options, as "keyword=value" strings

43.13. pg_constraint

The catalog pg_constraint stores check, primary key, unique, and foreign key constraints on tables. (Column constraints are not treated specially. Every column constraint is equivalent to some table constraint.) Not-null constraints are represented in the pg_attribute catalog.

Check constraints on domains are stored here, too.

Table 43-13. pg_constraint Columns

Name	Type	References	Description
conname	name		Constraint name (not necessarily unique!)
connamespace	oid	pg_namespace.oid	The OID of the namespace that contains this constraint
contype	char		c = check constraint, f = foreign key constraint, p = primary key constraint, u = unique constraint
condeferrable	bool		Is the constraint deferrable?
condeferred	bool		Is the constraint deferred by default?
conrelid	oid	pg_class.oid	The table this constraint is on; 0 if not a table constraint
contypid	oid	pg_type.oid	The domain this constraint is on; 0 if not a domain constraint
confrelid	oid	pg_class.oid	If a foreign key, the referenced table; else 0
confupdtype	char		Foreign key update action code
confdeltype	char		Foreign key deletion action code
confmatchtype	char		Foreign key match type
conkey	int2[]	pg_attribute.attnum	If a table constraint, list of columns which the constraint constrains
confkey	int2[]	pg_attribute.attnum	If a foreign key, list of the referenced columns

Name	Type	References	Description
conbin	text		If a check constraint, an internal representation of the expression
consrc	text		If a check constraint, a human-readable representation of the expression

Note: consrc is not updated when referenced objects change; for example, it won't track renaming of columns. Rather than relying on this field, it's best to use pg_get_constraintdef() to extract the definition of a check constraint.

Note: pg_class.relchecks needs to agree with the number of check-constraint entries found in this table for the given relation.

43.14. pg_conversion

The catalog pg_conversion describes the available encoding conversion procedures. See *CREATE CONVERSION* for more information.

Table 43-14. pg_conversion Columns

Name	Type	References	Description
conname	name		Conversion name (unique within a namespace)
connamespace	oid	pg_namespace.oid	The OID of the namespace that contains this conversion
conowner	oid	pg_authid.oid	Owner of the conversion
conforencoding	int4		Source encoding ID
contoencoding	int4		Destination encoding ID
conproc	regproc	pg_proc.oid	Conversion procedure
condefault	bool		True if this is the default conversion

43.15. pg_database

The catalog pg_database stores information about the available databases. Databases are created with the *CREATE DATABASE* command. Consult Chapter 19 for details about the meaning of some of the parameters.

Unlike most system catalogs, pg_database is shared across all databases of a cluster: there is only one copy of pg_database per cluster, not one per database.

Table 43-15. pg_database Columns

Name	Type	References	Description
datname	name		Database name
datdba	oid	pg_authid.oid	Owner of the database, usually the user who created it
encoding	int4		Character encoding for this database (pg_encoding_to_char() can translate this number to the encoding name)
datistemplate	bool		If true then this database can be used in the TEMPLATE clause of CREATE DATABASE to create a new database as a clone of this one
datallowconn	bool		If false then no one can connect to this database. This is used to protect the template0 database from being altered
datconnlimit	int4		Sets maximum number of concurrent connections that can be made to this database. -1 means no limit
datlastsysoid	oid		Last system OID in the database; useful particularly to pg_dump

Name	Type	References	Description
datfrozenxid	xid		All transaction IDs before this one have been replaced with a permanent ("frozen") transaction ID in this database. This is used to track whether the database needs to be vacuumed in order to prevent transaction ID wraparound or to allow pg_clog to be shrunk. It is the minimum of the per-table pg_class.relfrozenxid values
dattablespace	oid	pg_tablespace.oid	The default tablespace for the database. Within this database, all tables for which pg_class.reltablespace is zero will be stored in this tablespace; in particular, all the non-shared system catalogs will be there
datconfig	text[]		Session defaults for run-time configuration variables
datacl	aclitem[]		Access privileges; see *GRANT* and *REVOKE* for details

43.16. pg_depend

The catalog pg_depend records the dependency relationships between database objects. This information allows DROP commands to find which other objects must be dropped by DROP CASCADE or prevent dropping in the DROP RESTRICT case.

See also pg_shdepend, which performs a similar function for dependencies involving objects that are shared across a database cluster.

Table 43-16. pg_depend Columns

Name	Type	References	Description
classid	oid	pg_class.oid	The OID of the system catalog the dependent object is in
objid	oid	any OID column	The OID of the specific dependent object
objsubid	int4		For a table column, this is the column number (the objid and classid refer to the table itself). For all other object types, this column is zero
refclassid	oid	pg_class.oid	The OID of the system catalog the referenced object is in
refobjid	oid	any OID column	The OID of the specific referenced object
refobjsubid	int4		For a table column, this is the column number (the refobjid and refclassid refer to the table itself). For all other object types, this column is zero
deptype	char		A code defining the specific semantics of this dependency relationship; see text

In all cases, a pg_depend entry indicates that the referenced object may not be dropped without also dropping the dependent object. However, there are several subflavors identified by deptype:

DEPENDENCY_NORMAL (n)

A normal relationship between separately-created objects. The dependent object may be dropped without affecting the referenced object. The referenced object may only be dropped by specifying CASCADE, in which case the dependent object is dropped, too. Example: a table column has a normal dependency on its data type.

DEPENDENCY_AUTO (a)

The dependent object can be dropped separately from the referenced object, and should be automatically dropped (regardless of RESTRICT or CASCADE mode) if the referenced object is dropped. Example: a named constraint on a table is made autodependent on the table, so that it will go away if the table is dropped.

DEPENDENCY_INTERNAL (i)

The dependent object was created as part of creation of the referenced object, and is really just a part of its internal implementation. A DROP of the dependent object will be disallowed outright (we'll tell the user to issue a DROP against the referenced object, instead). A DROP of the referenced object will be propagated through to drop the dependent object whether CASCADE is specified or not. Example: a trigger that's created to enforce a foreign-key constraint is made internally dependent on the constraint's pg_constraint entry.

DEPENDENCY_PIN (p)

There is no dependent object; this type of entry is a signal that the system itself depends on the referenced object, and so that object must never be deleted. Entries of this type are created only by initdb. The columns for the dependent object contain zeroes.

Other dependency flavors may be needed in future.

43.17. pg_description

The catalog pg_description stores optional descriptions (comments) for each database object. Descriptions can be manipulated with the *COMMENT* command and viewed with psql's \d commands. Descriptions of many built-in system objects are provided in the initial contents of pg_description.

See also pg_shdescription, which performs a similar function for descriptions involving objects that are shared across a database cluster.

Table 43-17. pg_description Columns

Name	Type	References	Description
objoid	oid	any OID column	The OID of the object this description pertains to
classoid	oid	pg_class.oid	The OID of the system catalog this object appears in
objsubid	int4		For a comment on a table column, this is the column number (the objoid and classoid refer to the table itself). For all other object types, this column is zero
description	text		Arbitrary text that serves as the description of this object

43.18. pg_index

The catalog pg_index contains part of the information about indexes. The rest is mostly in pg_class.

Table 43-18. pg_index Columns

Name	Type	References	Description
indexrelid	oid	pg_class.oid	The OID of the pg_class entry for this index
indrelid	oid	pg_class.oid	The OID of the pg_class entry for the table this index is for
indnatts	int2		The number of columns in the index (duplicates pg_class.relnatts)
indisunique	bool		If true, this is a unique index
indisprimary	bool		If true, this index represents the primary key of the table. (indisunique should always be true when this is true.)
indisclustered	bool		If true, the table was last clustered on this index
indisvalid	bool		If true, the index is currently valid for queries. False means the index is possibly incomplete: it must still be modified by INSERT/UPDATE operations, but it cannot safely be used for queries. If it is unique, the uniqueness property is not true either

Name	Type	References	Description
indkey	int2vector	pg_attribute.attnum	This is an array of indnatts values that indicate which table columns this index indexes. For example a value of 1 3 would mean that the first and the third table columns make up the index key. A zero in this array indicates that the corresponding index attribute is an expression over the table columns, rather than a simple column reference.
indclass	oidvector	pg_opclass.oid	For each column in the index key this contains the OID of the operator class to use. See pg_opclass for details
indexprs	text		Expression trees (in nodeToString() representation) for index attributes that are not simple column references. This is a list with one element for each zero entry in indkey. NULL if all index attributes are simple references
indpred	text		Expression tree (in nodeToString() representation) for partial index predicate. NULL if not a partial index

43.19. pg_inherits

The catalog pg_inherits records information about table inheritance hierarchies. There is one entry for each direct child table in the database. (Indirect inheritance can be determined by following chains of

entries.)

Table 43-19. `pg_inherits` Columns

Name	Type	References	Description
`inhrelid`	`oid`	`pg_class.oid`	The OID of the child table
`inhparent`	`oid`	`pg_class.oid`	The OID of the parent table
`inhseqno`	`int4`		If there is more than one direct parent for a child table (multiple inheritance), this number tells the order in which the inherited columns are to be arranged. The count starts at 1

43.20. `pg_language`

The catalog `pg_language` registers languages in which you can write functions or stored procedures. See *CREATE LANGUAGE* and Chapter 36 for more information about language handlers.

Table 43-20. `pg_language` Columns

Name	Type	References	Description
`lanname`	`name`		Name of the language
`lanispl`	`bool`		This is false for internal languages (such as SQL) and true for user-defined languages. Currently, pg_dump still uses this to determine which languages need to be dumped, but this may be replaced by a different mechanism in the future

Name	Type	References	Description
lanpltrusted	bool		True if this is a trusted language, which means that it is believed not to grant access to anything outside the normal SQL execution environment. Only superusers may create functions in untrusted languages
lanplcallfoid	oid	pg_proc.oid	For noninternal languages this references the language handler, which is a special function that is responsible for executing all functions that are written in the particular language
lanvalidator	oid	pg_proc.oid	This references a language validator function that is responsible for checking the syntax and validity of new functions when they are created. Zero if no validator is provided
lanacl	aclitem[]		Access privileges; see *GRANT* and *REVOKE* for details

43.21. pg_largeobject

The catalog pg_largeobject holds the data making up "large objects". A large object is identified by an OID assigned when it is created. Each large object is broken into segments or "pages" small enough to be conveniently stored as rows in pg_largeobject. The amount of data per page is defined to be LOBLKSIZE (which is currently BLCKSZ/4, or typically 2 kB).

Table 43-21. pg_largeobject Columns

Name	Type	Description	
loid	oid	Identifier of the large object that includes this page	

Name	Type	Description	
pageno	int4	Page number of this page within its large object (counting from zero)	
data	bytea	Actual data stored in the large object. This will never be more than LOBLKSIZE bytes and may be less	

Each row of pg_largeobject holds data for one page of a large object, beginning at byte offset (pageno * LOBLKSIZE) within the object. The implementation allows sparse storage: pages may be missing, and may be shorter than LOBLKSIZE bytes even if they are not the last page of the object. Missing regions within a large object read as zeroes.

43.22. pg_listener

The catalog pg_listener supports the *LISTEN* and *NOTIFY* commands. A listener creates an entry in pg_listener for each notification name it is listening for. A notifier scans pg_listener and updates each matching entry to show that a notification has occurred. The notifier also sends a signal (using the PID recorded in the table) to awaken the listener from sleep.

Table 43-22. pg_listener Columns

Name	Type	References	Description
relname	name	Notify condition name. (The name need not match any actual relation in the database; the name relname is historical.)	
listenerpid	int4	PID of the server process that created this entry	
notification	int4	Zero if no event is pending for this listener. If an event is pending, the PID of the server process that sent the notification	

43.23. pg_namespace

The catalog pg_namespace stores namespaces. A namespace is the structure underlying SQL schemas: each namespace can have a separate collection of relations, types, etc. without name conflicts.

Table 43-23. pg_namespace Columns

Name	Type	References	Description
nspname	name		Name of the namespace
nspowner	oid	pg_authid.oid	Owner of the namespace
nspacl	aclitem[]		Access privileges; see *GRANT* and *REVOKE* for details

43.24. pg_opclass

The catalog pg_opclass defines index access method operator classes. Each operator class defines semantics for index columns of a particular data type and a particular index access method. Note that there can be multiple operator classes for a given data type/access method combination, thus supporting multiple behaviors.

Operator classes are described at length in Section 33.14.

Table 43-24. pg_opclass Columns

Name	Type	References	Description
opcamid	oid	pg_am.oid	Index access method operator class is for
opcname	name		Name of this operator class
opcnamespace	oid	pg_namespace.oid	Namespace of this operator class
opcowner	oid	pg_authid.oid	Owner of the operator class
opcintype	oid	pg_type.oid	Data type that the operator class indexes
opcdefault	bool		True if this operator class is the default for opcintype
opckeytype	oid	pg_type.oid	Type of data stored in index, or zero if same as opcintype

The majority of the information defining an operator class is actually not in its pg_opclass row, but in

the associated rows in pg_amop and pg_amproc. Those rows are considered to be part of the operator class definition — this is not unlike the way that a relation is defined by a single pg_class row plus associated rows in pg_attribute and other tables.

43.25. pg_operator

The catalog pg_operator stores information about operators. See *CREATE OPERATOR* and Section 33.12 for more information.

Table 43-25. pg_operator Columns

Name	Type	References	Description
oprname	name		Name of the operator
oprnamespace	oid	pg_namespace.oid	The OID of the namespace that contains this operator
oprowner	oid	pg_authid.oid	Owner of the operator
oprkind	char		b = infix ("both"), l = prefix ("left"), r = postfix ("right")
oprcanhash	bool		This operator supports hash joins
oprleft	oid	pg_type.oid	Type of the left operand
oprright	oid	pg_type.oid	Type of the right operand
oprresult	oid	pg_type.oid	Type of the result
oprcom	oid	pg_operator.oid	Commutator of this operator, if any
oprnegate	oid	pg_operator.oid	Negator of this operator, if any
oprlsortop	oid	pg_operator.oid	If this operator supports merge joins, the operator that sorts the type of the left-hand operand (L<L)
oprrsortop	oid	pg_operator.oid	If this operator supports merge joins, the operator that sorts the type of the right-hand operand (R<R)

Name	Type	References	Description
oprltcmpop	oid	pg_operator.oid	If this operator supports merge joins, the less-than operator that compares the left and right operand types (L<R)
oprgtcmpop	oid	pg_operator.oid	If this operator supports merge joins, the greater-than operator that compares the left and right operand types (L>R)
oprcode	regproc	pg_proc.oid	Function that implements this operator
oprrest	regproc	pg_proc.oid	Restriction selectivity estimation function for this operator
oprjoin	regproc	pg_proc.oid	Join selectivity estimation function for this operator

Unused column contain zeroes. For example, oprleft is zero for a prefix operator.

43.26. pg_pltemplate

The catalog pg_pltemplate stores "template" information for procedural languages. A template for a language allows the language to be created in a particular database by a simple CREATE LANGUAGE command, with no need to specify implementation details.

Unlike most system catalogs, pg_pltemplate is shared across all databases of a cluster: there is only one copy of pg_pltemplate per cluster, not one per database. This allows the information to be accessible in each database as it is needed.

Table 43-26. pg_pltemplate Columns

Name	Type	Description	
tmplname	name	Name of the language this template is for	
tmpltrusted	boolean	True if language is considered trusted	
tmplhandler	text	Name of call handler function	

Name	Type	Description	
tmplvalidator	text	Name of validator function, or NULL if none	
tmpllibrary	text	Path of shared library that implements language	
tmplacl	aclitem[]	Access privileges for template (not yet used)	

There are not currently any commands that manipulate procedural language templates; to change the built-in information, a superuser must modify the table using ordinary INSERT, DELETE, or UPDATE commands. It is likely that a future release of PostgreSQL will offer commands to change the entries in a cleaner fashion.

When implemented, the tmplacl field will provide access control for the template itself (i.e., the right to create a language using it), not for the languages created from the template.

43.27. pg_proc

The catalog pg_proc stores information about functions (or procedures). See *CREATE FUNCTION* and Section 33.3 for more information.

The table contains data for aggregate functions as well as plain functions. If proisagg is true, there should be a matching row in pg_aggregate.

Table 43-27. pg_proc Columns

Name	Type	References	Description
proname	name		Name of the function
pronamespace	oid	pg_namespace.oid	The OID of the namespace that contains this function
proowner	oid	pg_authid.oid	Owner of the function
prolang	oid	pg_language.oid	Implementation language or call interface of this function
proisagg	bool		Function is an aggregate function
prosecdef	bool		Function is a security definer (i.e., a “setuid” function)

Name	Type	References	Description
proisstrict	bool		Function returns null if any call argument is null. In that case the function won't actually be called at all. Functions that are not "strict" must be prepared to handle null inputs
proretset	bool		Function returns a set (i.e., multiple values of the specified data type)
provolatile	char		provolatile tells whether the function's result depends only on its input arguments, or is affected by outside factors. It is i for "immutable" functions, which always deliver the same result for the same inputs. It is s for "stable" functions, whose results (for fixed inputs) do not change within a scan. It is v for "volatile" functions, whose results may change at any time. (Use v also for functions with side-effects, so that calls to them cannot get optimized away.)
pronargs	int2		Number of arguments
prorettype	oid	pg_type.oid	Data type of the return value
proargtypes	oidvector	pg_type.oid	An array with the data types of the function arguments. This includes only input arguments (including INOUT arguments), and thus represents the call signature of the function

Name	Type	References	Description
proallargtypes	oid[]	pg_type.oid	An array with the data types of the function arguments. This includes all arguments (including OUT and INOUT arguments); however, if all the arguments are IN arguments, this field will be null. Note that subscripting is 1-based, whereas for historical reasons proargtypes is subscripted from 0
proargmodes	char[]		An array with the modes of the function arguments, encoded as i for IN arguments, o for OUT arguments, b for INOUT arguments. If all the arguments are IN arguments, this field will be null. Note that subscripts correspond to positions of proallargtypes not proargtypes
proargnames	text[]		An array with the names of the function arguments. Arguments without a name are set to empty strings in the array. If none of the arguments have a name, this field will be null. Note that subscripts correspond to positions of proallargtypes not proargtypes

Name	Type	References	Description
prosrc	text		This tells the function handler how to invoke the function. It might be the actual source code of the function for interpreted languages, a link symbol, a file name, or just about anything else, depending on the implementation language/call convention
probin	bytea		Additional information about how to invoke the function. Again, the interpretation is language-specific
proacl	aclitem[]		Access privileges; see *GRANT* and *REVOKE* for details

For compiled functions, both built-in and dynamically loaded, prosrc contains the function's C-language name (link symbol). For all other currently-known language types, prosrc contains the function's source text. probin is unused except for dynamically-loaded C functions, for which it gives the name of the shared library file containing the function.

43.28. pg_rewrite

The catalog pg_rewrite stores rewrite rules for tables and views.

Table 43-28. pg_rewrite Columns

Name	Type	References	Description
rulename	name		Rule name
ev_class	oid	pg_class.oid	The table this rule is for
ev_attr	int2		The column this rule is for (currently, always zero to indicate the whole table)
ev_type	char		Event type that the rule is for: 1 = SELECT, 2 = UPDATE, 3 = INSERT, 4 = DELETE

Name	Type	References	Description
is_instead	bool		True if the rule is an INSTEAD rule
ev_qual	text		Expression tree (in the form of a nodeToString() representation) for the rule's qualifying condition
ev_action	text		Query tree (in the form of a nodeToString() representation) for the rule's action

Note: pg_class.relhasrules must be true if a table has any rules in this catalog.

43.29. pg_shdepend

The catalog pg_shdepend records the dependency relationships between database objects and shared objects, such as roles. This information allows PostgreSQL to ensure that those objects are unreferenced before attempting to delete them.

See also pg_depend, which performs a similar function for dependencies involving objects within a single database.

Unlike most system catalogs, pg_shdepend is shared across all databases of a cluster: there is only one copy of pg_shdepend per cluster, not one per database.

Table 43-29. pg_shdepend Columns

Name	Type	References	Description
dbid	oid	pg_database.oid	The OID of the database the dependent object is in, or zero for a shared object
classid	oid	pg_class.oid	The OID of the system catalog the dependent object is in
objid	oid	any OID column	The OID of the specific dependent object

Name	Type	References	Description
refclassid	oid	pg_class.oid	The OID of the system catalog the referenced object is in (must be a shared catalog)
refobjid	oid	any OID column	The OID of the specific referenced object
deptype	char		A code defining the specific semantics of this dependency relationship; see text

In all cases, a pg_shdepend entry indicates that the referenced object may not be dropped without also dropping the dependent object. However, there are several subflavors identified by deptype:

SHARED_DEPENDENCY_OWNER (o)

The referenced object (which must be a role) is the owner of the dependent object.

SHARED_DEPENDENCY_ACL (a)

The referenced object (which must be a role) is mentioned in the ACL (access control list, i.e., privileges list) of the dependent object. (A SHARED_DEPENDENCY_ACL entry is not made for the owner of the object, since the owner will have a SHARED_DEPENDENCY_OWNER entry anyway.)

SHARED_DEPENDENCY_PIN (p)

There is no dependent object; this type of entry is a signal that the system itself depends on the referenced object, and so that object must never be deleted. Entries of this type are created only by initdb. The columns for the dependent object contain zeroes.

Other dependency flavors may be needed in future. Note in particular that the current definition only supports roles as referenced objects.

43.30. pg_shdescription

The catalog pg_shdescription stores optional descriptions (comments) for shared database objects. Descriptions can be manipulated with the *COMMENT* command and viewed with psql's \d commands.

See also pg_description, which performs a similar function for descriptions involving objects within a single database.

Unlike most system catalogs, pg_shdescription is shared across all databases of a cluster: there is only one copy of pg_shdescription per cluster, not one per database.

Table 43-30. pg_shdescription Columns

Name	Type	References	Description

Name	Type	References	Description
objoid	oid	any OID column	The OID of the object this description pertains to
classoid	oid	pg_class.oid	The OID of the system catalog this object appears in
description	text		Arbitrary text that serves as the description of this object

43.31. pg_statistic

The catalog pg_statistic stores statistical data about the contents of the database. Entries are created by ANALYZE and subsequently used by the query planner. There is one entry for each table column that has been analyzed. Note that all the statistical data is inherently approximate, even assuming that it is up-to-date.

pg_statistic also stores statistical data about the values of index expressions. These are described as if they were actual data columns; in particular, starelid references the index. No entry is made for an ordinary non-expression index column, however, since it would be redundant with the entry for the underlying table column.

Since different kinds of statistics may be appropriate for different kinds of data, pg_statistic is designed not to assume very much about what sort of statistics it stores. Only extremely general statistics (such as nullness) are given dedicated columns in pg_statistic. Everything else is stored in "slots", which are groups of associated columns whose content is identified by a code number in one of the slot's columns. For more information see src/include/catalog/pg_statistic.h.

pg_statistic should not be readable by the public, since even statistical information about a table's contents may be considered sensitive. (Example: minimum and maximum values of a salary column might be quite interesting.) pg_stats is a publicly readable view on pg_statistic that only exposes information about those tables that are readable by the current user.

Table 43-31. pg_statistic Columns

Name	Type	References	Description
starelid	oid	pg_class.oid	The table or index that the described column belongs to
staattnum	int2	pg_attribute.attnum	The number of the described column
stanullfrac	float4		The fraction of the column's entries that are null

Name	Type	References	Description
stawidth	int4		The average stored width, in bytes, of nonnull entries
stadistinct	float4		The number of distinct nonnull data values in the column. A value greater than zero is the actual number of distinct values. A value less than zero is the negative of a fraction of the number of rows in the table (for example, a column in which values appear about twice on the average could be represented by stadistinct = -0.5). A zero value means the number of distinct values is unknown
stakind*N*	int2		A code number indicating the kind of statistics stored in the *N*th "slot" of the pg_statistic row
staop*N*	oid	pg_operator.oid	An operator used to derive the statistics stored in the *N*th "slot". For example, a histogram slot would show the < operator that defines the sort order of the data
stanumbers*N*	float4[]		Numerical statistics of the appropriate kind for the *N*th "slot", or NULL if the slot kind does not involve numerical values

Name	Type	References	Description
stavalues*N*	anyarray		Column data values of the appropriate kind for the *N*th "slot", or NULL if the slot kind does not store any data values. Each array's element values are actually of the specific column's data type, so there is no way to define these columns' type more specifically than anyarray

43.32. pg_tablespace

The catalog pg_tablespace stores information about the available tablespaces. Tables can be placed in particular tablespaces to aid administration of disk layout.

Unlike most system catalogs, pg_tablespace is shared across all databases of a cluster: there is only one copy of pg_tablespace per cluster, not one per database.

Table 43-32. pg_tablespace Columns

Name	Type	References	Description
spcname	name		Tablespace name
spcowner	oid	pg_authid.oid	Owner of the tablespace, usually the user who created it
spclocation	text		Location (directory path) of the tablespace
spcacl	aclitem[]		Access privileges; see *GRANT* and *REVOKE* for details

43.33. pg_trigger

The catalog pg_trigger stores triggers on tables. See *CREATE TRIGGER* for more information.

Table 43-33. pg_trigger Columns

Name	Type	References	Description

Name	Type	References	Description
tgrelid	oid	pg_class.oid	The table this trigger is on
tgname	name		Trigger name (must be unique among triggers of same table)
tgfoid	oid	pg_proc.oid	The function to be called
tgtype	int2		Bit mask identifying trigger conditions
tgenabled	bool		True if trigger is enabled
tgisconstraint	bool		True if trigger implements a referential integrity constraint
tgconstrname	name		Referential integrity constraint name
tgconstrrelid	oid	pg_class.oid	The table referenced by an referential integrity constraint
tgdeferrable	bool		True if deferrable
tginitdeferred	bool		True if initially deferred
tgnargs	int2		Number of argument strings passed to trigger function
tgattr	int2vector		Currently unused
tgargs	bytea		Argument strings to pass to trigger, each NULL-terminated

Note: pg_class.reltriggers needs to agree with the number of triggers found in this table for the given relation.

43.34. pg_type

The catalog pg_type stores information about data types. Base types (scalar types) are created with *CREATE TYPE*, and domains with *CREATE DOMAIN*. A composite type is automatically created for each table in the database, to represent the row structure of the table. It is also possible to create composite types with CREATE TYPE AS.

Table 43-34. `pg_type` Columns

Name	Type	References	Description
`typname`	`name`		Data type name
`typnamespace`	`oid`	`pg_namespace.oid`	The OID of the namespace that contains this type
`typowner`	`oid`	`pg_authid.oid`	Owner of the type
`typlen`	`int2`		For a fixed-size type, `typlen` is the number of bytes in the internal representation of the type. But for a variable-length type, `typlen` is negative. -1 indicates a "varlena" type (one that has a length word), -2 indicates a null-terminated C string.
`typbyval`	`bool`		`typbyval` determines whether internal routines pass a value of this type by value or by reference. `typbyval` had better be false if `typlen` is not 1, 2, or 4 (or 8 on machines where Datum is 8 bytes). Variable-length types are always passed by reference. Note that `typbyval` can be false even if the length would allow pass-by-value; this is currently true for type `float4`, for example
`typtype`	`char`		`typtype` is `b` for a base type, `c` for a composite type (e.g., a table's row type), `d` for a domain, or `p` for a pseudo-type. See also `typrelid` and `typbasetype`

Name	Type	References	Description
typisdefined	bool		True if the type is defined, false if this is a placeholder entry for a not-yet-defined type. When typisdefined is false, nothing except the type name, namespace, and OID can be relied on
typdelim	char		Character that separates two values of this type when parsing array input. Note that the delimiter is associated with the array element data type, not the array data type
typrelid	oid	pg_class.oid	If this is a composite type (see typtype), then this column points to the pg_class entry that defines the corresponding table. (For a free-standing composite type, the pg_class entry doesn't really represent a table, but it is needed anyway for the type's pg_attribute entries to link to.) Zero for non-composite types

Name	Type	References	Description
typelem	oid	pg_type.oid	If typelem is not 0 then it identifies another row in pg_type. The current type can then be subscripted like an array yielding values of type typelem. A "true" array type is variable length (typlen = -1), but some fixed-length (typlen > 0) types also have nonzero typelem, for example name and point. If a fixed-length type has a typelem then its internal representation must be some number of values of the typelem data type with no other data. Variable-length array types have a header defined by the array subroutines
typinput	regproc	pg_proc.oid	Input conversion function (text format)
typoutput	regproc	pg_proc.oid	Output conversion function (text format)
typreceive	regproc	pg_proc.oid	Input conversion function (binary format), or 0 if none
typsend	regproc	pg_proc.oid	Output conversion function (binary format), or 0 if none
typanalyze	regproc	pg_proc.oid	Custom ANALYZE function, or 0 to use the standard function

Name	Type	References	Description
typalign	char		typalign is the alignment required when storing a value of this type. It applies to storage on disk as well as most representations of the value inside PostgreSQL. When multiple values are stored consecutively, such as in the representation of a complete row on disk, padding is inserted before a datum of this type so that it begins on the specified boundary. The alignment reference is the beginning of the first datum in the sequence. Possible values are: • c = char alignment, i.e., no alignment needed. • s = short alignment (2 bytes on most machines). • i = int alignment (4 bytes on most machines). • d = double alignment (8 bytes on many machines, but by no means all). **Note:** For types used in system tables, it is critical that the size and alignment defined in pg_type agree with the way that the compiler will lay out the column in a structure representing a table row.

Name	Type	References	Description
typstorage	char		typstorage tells for varlena types (those with typlen = -1) if the type is prepared for toasting and what the default strategy for attributes of this type should be. Possible values are • p: Value must always be stored plain. • e: Value can be stored in a "secondary" relation (if relation has one, see pg_class.reltoastrelid). • m: Value can be stored compressed inline. • x: Value can be stored compressed inline or stored in "secondary" storage. Note that m columns can also be moved out to secondary storage, but only as a last resort (e and x columns are moved first).
typnotnull	bool		typnotnull represents a not-null constraint on a type. Used for domains only
typbasetype	oid	pg_type.oid	If this is a domain (see typtype), then typbasetype identifies the type that this one is based on. Zero if this type is not a domain

Name	Type	References	Description
typtypmod	int4		Domains use typtypmod to record the typmod to be applied to their base type (-1 if base type does not use a typmod). -1 if this type is not a domain
typndims	int4		typndims is the number of array dimensions for a domain that is an array (that is, typbasetype is an array type; the domain's typelem will match the base type's typelem). Zero for types other than array domains
typdefaultbin	text		If typdefaultbin is not null, it is the nodeToString() representation of a default expression for the type. This is only used for domains
typdefault	text		typdefault is null if the type has no associated default value. If typdefaultbin is not null, typdefault must contain a human-readable version of the default expression represented by typdefaultbin. If typdefaultbin is null and typdefault is not, then typdefault is the external representation of the type's default value, which may be fed to the type's input converter to produce a constant

43.35. System Views

In addition to the system catalogs, PostgreSQL provides a number of built-in views. Some system views provide convenient access to some commonly used queries on the system catalogs. Other views provide access to internal server state.

The information schema (Chapter 32) provides an alternative set of views which overlap the functionality of the system views. Since the information schema is SQL-standard whereas the views described here are PostgreSQL-specific, it's usually better to use the information schema if it provides all the information you need.

Table 43-35 lists the system views described here. More detailed documentation of each view follows below. There are some additional views that provide access to the results of the statistics collector; they are described in Table 25-1.

Except where noted, all the views described here are read-only.

Table 43-35. System Views

View Name	Purpose
pg_cursors	open cursors
pg_group	groups of database users
pg_indexes	indexes
pg_locks	currently held locks
pg_prepared_statements	prepared statements
pg_prepared_xacts	prepared transactions
pg_roles	database roles
pg_rules	rules
pg_settings	parameter settings
pg_shadow	database users
pg_stats	planner statistics
pg_tables	tables
pg_timezone_abbrevs	time zone abbreviations
pg_timezone_names	time zone names
pg_user	database users
pg_views	views

43.36. pg_cursors

The pg_cursors view lists the cursors that are currently available. Cursors can be defined in several ways:

- via the *DECLARE* statement in SQL

- via the Bind message in the frontend/backend protocol, as described in Section 44.2.3
- via the Server Programming Interface (SPI), as described in Section 41.1

The pg_cursors view displays cursors created by any of these means. Cursors only exist for the duration of the transaction that defines them, unless they have been declared WITH HOLD. Therefore non-holdable cursors are only present in the view until the end of their creating transaction.

> **Note:** Cursors are used internally to implement some of the components of PostgreSQL, such as procedural languages. Therefore, the pg_cursors view may include cursors that have not been explicitly created by the user.

Table 43-36. pg_cursors Columns

Name	Type	Description	
name	text	The name of the cursor	
statement	text	The verbatim query string submitted to declare this cursor	
is_holdable	boolean	true if the cursor is holdable (that is, it can be accessed after the transaction that declared the cursor has committed); false otherwise	
is_binary	boolean	true if the cursor was declared BINARY; false otherwise	
is_scrollable	boolean	true if the cursor is scrollable (that is, it allows rows to be retrieved in a nonsequential manner); false otherwise	
creation_time	timestamptz	The time at which the cursor was declared	

The pg_cursors view is read only.

43.37. pg_group

The view pg_group exists for backwards compatibility: it emulates a catalog that existed in PostgreSQL before version 8.1. It shows the names and members of all roles that are marked as not rolcanlogin,

which is an approximation to the set of roles that are being used as groups.

Table 43-37. `pg_group` Columns

Name	Type	References	Description
groname	name	pg_authid.rolname	Name of the group
grosysid	oid	pg_authid.oid	ID of this group
grolist	oid[]	pg_authid.oid	An array containing the IDs of the roles in this group

43.38. pg_indexes

The view pg_indexes provides access to useful information about each index in the database.

Table 43-38. `pg_indexes` Columns

Name	Type	References	Description
schemaname	name	pg_namespace.nspname	Name of schema containing table and index
tablename	name	pg_class.relname	Name of table the index is for
indexname	name	pg_class.relname	Name of index
tablespace	name	pg_tablespace.spcname	Name of tablespace containing index (NULL if default for database)
indexdef	text		Index definition (a reconstructed CREATE INDEX command)

43.39. pg_locks

The view pg_locks provides access to information about the locks held by open transactions within the database server. See Chapter 12 for more discussion of locking.

pg_locks contains one row per active lockable object, requested lock mode, and relevant transaction. Thus, the same lockable object may appear many times, if multiple transactions are holding or waiting for locks on it. However, an object that currently has no locks on it will not appear at all.

There are several distinct types of lockable objects: whole relations (e.g., tables), individual pages of relations, individual tuples of relations, transaction IDs, and general database objects (identified by class OID and object OID, in the same way as in pg_description or pg_depend). Also, the right to extend

a relation is represented as a separate lockable object.

Table 43-39. `pg_locks` Columns

Name	Type	References	Description
`locktype`	`text`		type of the lockable object: `relation`, `extend`, `page`, `tuple`, `transactionid`, `object`, `userlock`, or `advisory`
`database`	`oid`	`pg_database.oid`	OID of the database in which the object exists, or zero if the object is a shared object, or NULL if the object is a transaction ID
`relation`	`oid`	`pg_class.oid`	OID of the relation, or NULL if the object is not a relation or part of a relation
`page`	`integer`		Page number within the relation, or NULL if the object is not a tuple or relation page
`tuple`	`smallint`		Tuple number within the page, or NULL if the object is not a tuple
`transactionid`	`xid`		ID of a transaction, or NULL if the object is not a transaction ID
`classid`	`oid`	`pg_class.oid`	OID of the system catalog containing the object, or NULL if the object is not a general database object
`objid`	`oid`	any OID column	OID of the object within its system catalog, or NULL if the object is not a general database object

Name	Type	References	Description
objsubid	smallint		For a table column, this is the column number (the classid and objid refer to the table itself). For all other object types, this column is zero. NULL if the object is not a general database object
transaction	xid		ID of the transaction that is holding or awaiting this lock
pid	integer		Process ID of the server process holding or awaiting this lock. NULL if the lock is held by a prepared transaction
mode	text		Name of the lock mode held or desired by this process (see Section 12.3.1)
granted	boolean		True if lock is held, false if lock is awaited

granted is true in a row representing a lock held by the indicated transaction. False indicates that this transaction is currently waiting to acquire this lock, which implies that some other transaction is holding a conflicting lock mode on the same lockable object. The waiting transaction will sleep until the other lock is released (or a deadlock situation is detected). A single transaction can be waiting to acquire at most one lock at a time.

Every transaction holds an exclusive lock on its transaction ID for its entire duration. If one transaction finds it necessary to wait specifically for another transaction, it does so by attempting to acquire share lock on the other transaction ID. That will succeed only when the other transaction terminates and releases its locks.

Although tuples are a lockable type of object, information about row-level locks is stored on disk, not in memory, and therefore row-level locks normally do not appear in this view. If a transaction is waiting for a row-level lock, it will usually appear in the view as waiting for the transaction ID of the current holder of that row lock.

Advisory locks can be acquired on keys consisting of either a single bigint value or two integer values. A bigint key is displayed with its high-order half in the classid column, its low-order half in the objid column, and objsubid equal to 1. Integer keys are displayed with the first key in the classid column, the second key in the objid column, and objsubid equal to 2. The actual meaning of the keys is up to the user. Advisory locks are local to each database, so the database column is meaningful for an advisory lock.

When the pg_locks view is accessed, the internal lock manager data structures are momentarily locked, and a copy is made for the view to display. This ensures that the view produces a consistent set of results, while not blocking normal lock manager operations longer than necessary. Nonetheless there could be some impact on database performance if this view is frequently accessed.

pg_locks provides a global view of all locks in the database cluster, not only those relevant to the current database. Although its relation column can be joined against pg_class.oid to identify locked relations, this will only work correctly for relations in the current database (those for which the database column is either the current database's OID or zero).

If you have enabled the statistics collector, the pid column can be joined to the procpid column of the pg_stat_activity view to get more information on the session holding or waiting to hold the lock. Also, if you are using prepared transactions, the transaction column can be joined to the transaction column of the pg_prepared_xacts view to get more information on prepared transactions that hold locks. (A prepared transaction can never be waiting for a lock, but it continues to hold the locks it acquired while running.)

43.40. pg_prepared_statements

The pg_prepared_statements view displays all the prepared statements that are available in the current session. See *PREPARE* for more information about prepared statements.

pg_prepared_statements contains one row for each prepared statement. Rows are added to the view when a new prepared statement is created and removed when a prepared statement is released (for example, via the *DEALLOCATE* command).

Table 43-40. pg_prepared_statements Columns

Name	Type	Description	
name	text	The identifier of the prepared statement	
statement	text	The query string submitted by the client to create this prepared statement. For prepared statements created via SQL, this is the PREPARE statement submitted by the client. For prepared statements created via the frontend/backend protocol, this is the text of the prepared statement itself	

Name	Type	Description	
prepare_time	timestamptz	The time at which the prepared statement was created	
parameter_types	regtype[]	The expected parameter types for the prepared statement in the form of an array of regtype. The OID corresponding to an element of this array can be obtained by casting the regtype value to oid	
from_sql	boolean	true if the prepared statement was created via the PREPARE SQL statement; false if the statement was prepared via the frontend/backend protocol	

The pg_prepared_statements view is read only.

43.41. pg_prepared_xacts

The view pg_prepared_xacts displays information about transactions that are currently prepared for two-phase commit (see *PREPARE TRANSACTION* for details).

pg_prepared_xacts contains one row per prepared transaction. An entry is removed when the transaction is committed or rolled back.

Table 43-41. pg_prepared_xacts Columns

Name	Type	References	Description
transaction	xid		Numeric transaction identifier of the prepared transaction
gid	text		Global transaction identifier that was assigned to the transaction
prepared	timestamp with time zone		Time at which the transaction was prepared for commit

Name	Type	References	Description
owner	name	pg_authid.rolname	Name of the user that executed the transaction
database	name	pg_database.datname	Name of the database in which the transaction was executed

When the pg_prepared_xacts view is accessed, the internal transaction manager data structures are momentarily locked, and a copy is made for the view to display. This ensures that the view produces a consistent set of results, while not blocking normal operations longer than necessary. Nonetheless there could be some impact on database performance if this view is frequently accessed.

43.42. pg_roles

The view pg_roles provides access to information about database roles. This is simply a publicly readable view of pg_authid that blanks out the password field.

This view explicitly exposes the OID column of the underlying table, since that is needed to do joins to other catalogs.

Table 43-42. pg_roles Columns

Name	Type	References	Description
rolname	name		Role name
rolsuper	bool		Role has superuser privileges
rolinherit	bool		Role automatically inherits privileges of roles it is a member of
rolcreaterole	bool		Role may create more roles
rolcreatedb	bool		Role may create databases
rolcatupdate	bool		Role may update system catalogs directly. (Even a superuser may not do this unless this column is true.)
rolcanlogin	bool		Role may log in. That is, this role can be given as the initial session authorization identifier

Name	Type	References	Description
rolconnlimit	int4		For roles that can log in, this sets maximum number of concurrent connections this role can make. -1 means no limit
rolpassword	text		Not the password (always reads as ********)
rolvaliduntil	timestamptz		Password expiry time (only used for password authentication); NULL if no expiration
rolconfig	text[]		Session defaults for run-time configuration variables
oid	oid	pg_authid.oid	ID of role

43.43. pg_rules

The view pg_rules provides access to useful information about query rewrite rules.

Table 43-43. pg_rules Columns

Name	Type	References	Description
schemaname	name	pg_namespace.nspname	Name of schema containing table
tablename	name	pg_class.relname	Name of table the rule is for
rulename	name	pg_rewrite.rulename	Name of rule
definition	text		Rule definition (a reconstructed creation command)

The pg_rules view excludes the ON SELECT rules of views; those can be seen in pg_views.

43.44. pg_settings

The view pg_settings provides access to run-time parameters of the server. It is essentially an alternative interface to the SHOW and SET commands. It also provides access to some facts about each parameter

that are not directly available from SHOW, such as minimum and maximum values.

Table 43-44. pg_settings Columns

Name	Type	Description	
name	text	Run-time configuration parameter name	
setting	text	Current value of the parameter	
unit	text	Implicit unit of the parameter	
category	text	Logical group of the parameter	
short_desc	text	A brief description of the parameter	
extra_desc	text	Additional, more detailed, information about the parameter	
context	text	Context required to set the parameter's value	
vartype	text	Parameter type (bool, integer, real, or string)	
source	text	Source of the current parameter value	
min_val	text	Minimum allowed value of the parameter (NULL for non-numeric values)	
max_val	text	Maximum allowed value of the parameter (NULL for non-numeric values)	

The pg_settings view cannot be inserted into or deleted from, but it can be updated. An UPDATE applied to a row of pg_settings is equivalent to executing the *SET* command on that named parameter. The change only affects the value used by the current session. If an UPDATE is issued within a transaction that is later aborted, the effects of the UPDATE command disappear when the transaction is rolled back. Once the surrounding transaction is committed, the effects will persist until the end of the session, unless overridden by another UPDATE or SET.

43.45. pg_shadow

The view pg_shadow exists for backwards compatibility: it emulates a catalog that existed in PostgreSQL before version 8.1. It shows properties of all roles that are marked as rolcanlogin.

The name stems from the fact that this table should not be readable by the public since it contains passwords. pg_user is a publicly readable view on pg_shadow that blanks out the password field.

Table 43-45. pg_shadow Columns

Name	Type	References	Description
usename	name	pg_authid.rolname	User name
usesysid	oid	pg_authid.oid	ID of this user
usecreatedb	bool		User may create databases
usesuper	bool		User is a superuser
usecatupd	bool		User may update system catalogs. (Even a superuser may not do this unless this column is true.)
passwd	text		Password (possibly encrypted)
valuntil	abstime		Password expiry time (only used for password authentication)
useconfig	text[]		Session defaults for run-time configuration variables

43.46. pg_stats

The view pg_stats provides access to the information stored in the pg_statistic catalog. This view allows access only to rows of pg_statistic that correspond to tables the user has permission to read, and therefore it is safe to allow public read access to this view.

pg_stats is also designed to present the information in a more readable format than the underlying catalog — at the cost that its schema must be extended whenever new slot types are defined for pg_statistic.

Table 43-46. pg_stats Columns

Name	Type	References	Description
schemaname	name	pg_namespace.nspname	Name of schema containing table
tablename	name	pg_class.relname	Name of table
attname	name	pg_attribute.attname	Name of the column described by this row

Name	Type	References	Description
null_frac	real		Fraction of column entries that are null
avg_width	integer		Average width in bytes of column's entries
n_distinct	real		If greater than zero, the estimated number of distinct values in the column. If less than zero, the negative of the number of distinct values divided by the number of rows. (The negated form is used when ANALYZE believes that the number of distinct values is likely to increase as the table grows; the positive form is used when the column seems to have a fixed number of possible values.) For example, -1 indicates a unique column in which the number of distinct values is the same as the number of rows
most_common_vals	anyarray		A list of the most common values in the column. (NULL if no values seem to be more common than any others.)
most_common_freqs	real[]		A list of the frequencies of the most common values, i.e., number of occurrences of each divided by total number of rows. (NULL when most_common_vals is.)

Name	Type	References	Description
histogram_bounds	anyarray		A list of values that divide the column's values into groups of approximately equal population. The values in most_common_vals, if present, are omitted from this histogram calculation. (This column is NULL if the column data type does not have a < operator or if the most_common_vals list accounts for the entire population.)
correlation	real		Statistical correlation between physical row ordering and logical ordering of the column values. This ranges from -1 to +1. When the value is near -1 or +1, an index scan on the column will be estimated to be cheaper than when it is near zero, due to reduction of random access to the disk. (This column is NULL if the column data type does not have a < operator.)

The maximum number of entries in the most_common_vals and histogram_bounds arrays can be set on a column-by-column basis using the ALTER TABLE SET STATISTICS command, or globally by setting the default_statistics_target run-time parameter.

43.47. pg_tables

The view pg_tables provides access to useful information about each table in the database.

Table 43-47. pg_tables Columns

Name	Type	References	Description

Name	Type	References	Description
schemaname	name	pg_namespace.nspname	Name of schema containing table
tablename	name	pg_class.relname	Name of table
tableowner	name	pg_authid.rolname	Name of table's owner
tablespace	name	pg_tablespace.spcname	Name of tablespace containing table (NULL if default for database)
hasindexes	boolean	pg_class.relhasindex	true if table has (or recently had) any indexes
hasrules	boolean	pg_class.relhasrules	true if table has rules
hastriggers	boolean	pg_class.reltriggers	true if table has triggers

43.48. pg_timezone_abbrevs

The view pg_timezone_abbrevs provides a list of time zone abbreviations that are currently recognized by the datetime input routines. The contents of this view change when the timezone_abbreviations run-time parameter is modified.

Table 43-48. pg_timezone_abbrevs Columns

Name	Type	Description
abbrev	text	Time zone abbreviation
utc_offset	interval	Offset from UTC (positive means east of Greenwich)
is_dst	boolean	True if this is a daylight-savings abbreviation

43.49. pg_timezone_names

The view pg_timezone_names provides a list of time zone names that are recognized by SET TIMEZONE, along with their associated abbreviations, UTC offsets, and daylight-savings status. Unlike the abbreviations shown in pg_timezone_abbrevs, many of these names imply a set of daylight-savings transition date rules. Therefore, the associated information changes across local DST boundaries. The displayed information is computed based on the current value of CURRENT_TIMESTAMP.

Table 43-49. pg_timezone_names Columns

Name	Type	Description
name	text	Time zone name
abbrev	text	Time zone abbreviation
utc_offset	interval	Offset from UTC (positive means east of Greenwich)
is_dst	boolean	True if currently observing daylight savings

43.50. pg_user

The view pg_user provides access to information about database users. This is simply a publicly readable view of pg_shadow that blanks out the password field.

Table 43-50. pg_user Columns

Name	Type	Description	
usename	name	User name	
usesysid	int4	User ID (arbitrary number used to reference this user)	
usecreatedb	bool	User may create databases	
usesuper	bool	User is a superuser	
usecatupd	bool	User may update system catalogs. (Even a superuser may not do this unless this column is true.)	
passwd	text	Not the password (always reads as ********)	
valuntil	abstime	Password expiry time (only used for password authentication)	
useconfig	text[]	Session defaults for run-time configuration variables	

43.51. pg_views

The view pg_views provides access to useful information about each view in the database.

Table 43-51. `pg_views` Columns

Name	Type	References	Description
`schemaname`	`name`	`pg_namespace.nspname`	Name of schema containing view
`viewname`	`name`	`pg_class.relname`	Name of view
`viewowner`	`name`	`pg_authid.rolname`	Name of view's owner
`definition`	`text`		View definition (a reconstructed `SELECT` query)

Chapter 44. Frontend/Backend Protocol

PostgreSQL uses a message-based protocol for communication between frontends and backends (clients and servers). The protocol is supported over TCP/IP and also over Unix-domain sockets. Port number 5432 has been registered with IANA as the customary TCP port number for servers supporting this protocol, but in practice any non-privileged port number may be used.

This document describes version 3.0 of the protocol, implemented in PostgreSQL 7.4 and later. For descriptions of the earlier protocol versions, see previous releases of the PostgreSQL documentation. A single server can support multiple protocol versions. The initial startup-request message tells the server which protocol version the client is attempting to use, and then the server follows that protocol if it is able.

Higher level features built on this protocol (for example, how libpq passes certain environment variables when the connection is established) are covered elsewhere.

In order to serve multiple clients efficiently, the server launches a new "backend" process for each client. In the current implementation, a new child process is created immediately after an incoming connection is detected. This is transparent to the protocol, however. For purposes of the protocol, the terms "backend" and "server" are interchangeable; likewise "frontend" and "client" are interchangeable.

44.1. Overview

The protocol has separate phases for startup and normal operation. In the startup phase, the frontend opens a connection to the server and authenticates itself to the satisfaction of the server. (This might involve a single message, or multiple messages depending on the authentication method being used.) If all goes well, the server then sends status information to the frontend, and finally enters normal operation. Except for the initial startup-request message, this part of the protocol is driven by the server.

During normal operation, the frontend sends queries and other commands to the backend, and the backend sends back query results and other responses. There are a few cases (such as `NOTIFY`) wherein the backend will send unsolicited messages, but for the most part this portion of a session is driven by frontend requests.

Termination of the session is normally by frontend choice, but can be forced by the backend in certain cases. In any case, when the backend closes the connection, it will roll back any open (incomplete) transaction before exiting.

Within normal operation, SQL commands can be executed through either of two sub-protocols. In the "simple query" protocol, the frontend just sends a textual query string, which is parsed and immediately executed by the backend. In the "extended query" protocol, processing of queries is separated into multiple steps: parsing, binding of parameter values, and execution. This offers flexibility and performance benefits, at the cost of extra complexity.

Normal operation has additional sub-protocols for special operations such as `COPY`.

44.1.1. Messaging Overview

All communication is through a stream of messages. The first byte of a message identifies the message type, and the next four bytes specify the length of the rest of the message (this length count includes itself, but not the message-type byte). The remaining contents of the message are determined by the message type. For historical reasons, the very first message sent by the client (the startup message) has no initial message-type byte.

To avoid losing synchronization with the message stream, both servers and clients typically read an entire message into a buffer (using the byte count) before attempting to process its contents. This allows easy recovery if an error is detected while processing the contents. In extreme situations (such as not having enough memory to buffer the message), the receiver may use the byte count to determine how much input to skip before it resumes reading messages.

Conversely, both servers and clients must take care never to send an incomplete message. This is commonly done by marshaling the entire message in a buffer before beginning to send it. If a communications failure occurs partway through sending or receiving a message, the only sensible response is to abandon the connection, since there is little hope of recovering message-boundary synchronization.

44.1.2. Extended Query Overview

In the extended-query protocol, execution of SQL commands is divided into multiple steps. The state retained between steps is represented by two types of objects: *prepared statements* and *portals*. A prepared statement represents the result of parsing, semantic analysis, and (optionally) planning of a textual query string. A prepared statement is not necessarily ready to execute, because it may lack specific values for *parameters*. A portal represents a ready-to-execute or already-partially-executed statement, with any missing parameter values filled in. (For `SELECT` statements, a portal is equivalent to an open cursor, but we choose to use a different term since cursors don't handle non-`SELECT` statements.)

The overall execution cycle consists of a *parse* step, which creates a prepared statement from a textual query string; a *bind* step, which creates a portal given a prepared statement and values for any needed parameters; and an *execute* step that runs a portal's query. In the case of a query that returns rows (`SELECT`, `SHOW`, etc), the execute step can be told to fetch only a limited number of rows, so that multiple execute steps may be needed to complete the operation.

The backend can keep track of multiple prepared statements and portals (but note that these exist only within a session, and are never shared across sessions). Existing prepared statements and portals are referenced by names assigned when they were created. In addition, an "unnamed" prepared statement and portal exist. Although these behave largely the same as named objects, operations on them are optimized for the case of executing a query only once and then discarding it, whereas operations on named objects are optimized on the expectation of multiple uses.

44.1.3. Formats and Format Codes

Data of a particular data type might be transmitted in any of several different *formats*. As of PostgreSQL 7.4 the only supported formats are "text" and "binary", but the protocol makes provision for future extensions. The desired format for any value is specified by a *format code*. Clients may specify a format code for each transmitted parameter value and for each column of a query result. Text has format code zero, binary has format code one, and all other format codes are reserved for future definition.

The text representation of values is whatever strings are produced and accepted by the input/output conversion functions for the particular data type. In the transmitted representation, there is no trailing null character; the frontend must add one to received values if it wants to process them as C strings. (The text format does not allow embedded nulls, by the way.)

Binary representations for integers use network byte order (most significant byte first). For other data types consult the documentation or source code to learn about the binary representation. Keep in mind that binary representations for complex data types may change across server versions; the text format is usually the more portable choice.

44.2. Message Flow

This section describes the message flow and the semantics of each message type. (Details of the exact representation of each message appear in Section 44.4.) There are several different sub-protocols depending on the state of the connection: start-up, query, function call, `COPY`, and termination. There are also special provisions for asynchronous operations (including notification responses and command cancellation), which can occur at any time after the start-up phase.

44.2.1. Start-Up

To begin a session, a frontend opens a connection to the server and sends a startup message. This message includes the names of the user and of the database the user wants to connect to; it also identifies the particular protocol version to be used. (Optionally, the startup message can include additional settings for run-time parameters.) The server then uses this information and the contents of its configuration files (such as `pg_hba.conf`) to determine whether the connection is provisionally acceptable, and what additional authentication is required (if any).

The server then sends an appropriate authentication request message, to which the frontend must reply with an appropriate authentication response message (such as a password). In principle the authentication request/response cycle could require multiple iterations, but none of the present authentication methods use more than one request and response. In some methods, no response at all is needed from the frontend, and so no authentication request occurs.

The authentication cycle ends with the server either rejecting the connection attempt (ErrorResponse), or sending AuthenticationOk.

The possible messages from the server in this phase are:

ErrorResponse

The connection attempt has been rejected. The server then immediately closes the connection.

AuthenticationOk

The authentication exchange is successfully completed.

AuthenticationKerberosV5

The frontend must now take part in a Kerberos V5 authentication dialog (not described here, part of the Kerberos specification) with the server. If this is successful, the server responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.

AuthenticationCleartextPassword

The frontend must now send a PasswordMessage containing the password in clear-text form. If this is the correct password, the server responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.

AuthenticationCryptPassword

The frontend must now send a PasswordMessage containing the password encrypted via crypt(3), using the 2-character salt specified in the AuthenticationCryptPassword message. If this is the correct password, the server responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.

AuthenticationMD5Password

The frontend must now send a PasswordMessage containing the password encrypted via MD5, using the 4-character salt specified in the AuthenticationMD5Password message. If this is the correct password, the server responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.

AuthenticationSCMCredential

This response is only possible for local Unix-domain connections on platforms that support SCM credential messages. The frontend must issue an SCM credential message and then send a single data byte. (The contents of the data byte are uninteresting; it's only used to ensure that the server waits long enough to receive the credential message.) If the credential is acceptable, the server responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.

If the frontend does not support the authentication method requested by the server, then it should immediately close the connection.

After having received AuthenticationOk, the frontend must wait for further messages from the server. In this phase a backend process is being started, and the frontend is just an interested bystander. It is still possible for the startup attempt to fail (ErrorResponse), but in the normal case the backend will send some ParameterStatus messages, BackendKeyData, and finally ReadyForQuery.

During this phase the backend will attempt to apply any additional run-time parameter settings that were given in the startup message. If successful, these values become session defaults. An error causes ErrorResponse and exit.

The possible messages from the backend in this phase are:

BackendKeyData

This message provides secret-key data that the frontend must save if it wants to be able to issue cancel requests later. The frontend should not respond to this message, but should continue listening for a ReadyForQuery message.

ParameterStatus

This message informs the frontend about the current (initial) setting of backend parameters, such as client_encoding or DateStyle. The frontend may ignore this message, or record the settings for its future use; see Section 44.2.6 for more details. The frontend should not respond to this message, but should continue listening for a ReadyForQuery message.

ReadyForQuery

Start-up is completed. The frontend may now issue commands.

ErrorResponse

Start-up failed. The connection is closed after sending this message.

NoticeResponse

A warning message has been issued. The frontend should display the message but continue listening for ReadyForQuery or ErrorResponse.

The ReadyForQuery message is the same one that the backend will issue after each command cycle. Depending on the coding needs of the frontend, it is reasonable to consider ReadyForQuery as starting a command cycle, or to consider ReadyForQuery as ending the start-up phase and each subsequent command cycle.

44.2.2. Simple Query

A simple query cycle is initiated by the frontend sending a Query message to the backend. The message includes an SQL command (or commands) expressed as a text string. The backend then sends one or more response messages depending on the contents of the query command string, and finally a ReadyForQuery response message. ReadyForQuery informs the frontend that it may safely send a new command. (It is not actually necessary for the frontend to wait for ReadyForQuery before issuing another command, but the frontend must then take responsibility for figuring out what happens if the earlier command fails and already-issued later commands succeed.)

The possible response messages from the backend are:

CommandComplete

An SQL command completed normally.

CopyInResponse

The backend is ready to copy data from the frontend to a table; see Section 44.2.5.

CopyOutResponse

The backend is ready to copy data from a table to the frontend; see Section 44.2.5.

RowDescription

Indicates that rows are about to be returned in response to a `SELECT`, `FETCH`, etc query. The contents of this message describe the column layout of the rows. This will be followed by a DataRow message for each row being returned to the frontend.

DataRow

One of the set of rows returned by a `SELECT`, `FETCH`, etc query.

EmptyQueryResponse

An empty query string was recognized.

ErrorResponse

An error has occurred.

ReadyForQuery

Processing of the query string is complete. A separate message is sent to indicate this because the query string may contain multiple SQL commands. (CommandComplete marks the end of processing one SQL command, not the whole string.) ReadyForQuery will always be sent, whether processing terminates successfully or with an error.

NoticeResponse

A warning message has been issued in relation to the query. Notices are in addition to other responses, i.e., the backend will continue processing the command.

The response to a `SELECT` query (or other queries that return row sets, such as `EXPLAIN` or `SHOW`) normally consists of RowDescription, zero or more DataRow messages, and then CommandComplete. `COPY` to or from the frontend invokes special protocol as described in Section 44.2.5. All other query types normally produce only a CommandComplete message.

Since a query string could contain several queries (separated by semicolons), there might be several such response sequences before the backend finishes processing the query string. ReadyForQuery is issued when the entire string has been processed and the backend is ready to accept a new query string.

If a completely empty (no contents other than whitespace) query string is received, the response is EmptyQueryResponse followed by ReadyForQuery.

In the event of an error, ErrorResponse is issued followed by ReadyForQuery. All further processing of the query string is aborted by ErrorResponse (even if more queries remained in it). Note that this may occur partway through the sequence of messages generated by an individual query.

In simple Query mode, the format of retrieved values is always text, except when the given command is a `FETCH` from a cursor declared with the `BINARY` option. In that case, the retrieved values are in binary format. The format codes given in the RowDescription message tell which format is being used.

A frontend must be prepared to accept ErrorResponse and NoticeResponse messages whenever it is expecting any other type of message. See also Section 44.2.6 concerning messages that the backend may generate due to outside events.

Recommended practice is to code frontends in a state-machine style that will accept any message type at any time that it could make sense, rather than wiring in assumptions about the exact sequence of messages.

44.2.3. Extended Query

The extended query protocol breaks down the above-described simple query protocol into multiple steps. The results of preparatory steps can be re-used multiple times for improved efficiency. Furthermore, additional features are available, such as the possibility of supplying data values as separate parameters instead of having to insert them directly into a query string.

In the extended protocol, the frontend first sends a Parse message, which contains a textual query string, optionally some information about data types of parameter placeholders, and the name of a destination prepared-statement object (an empty string selects the unnamed prepared statement). The response is

either ParseComplete or ErrorResponse. Parameter data types may be specified by OID; if not given, the parser attempts to infer the data types in the same way as it would do for untyped literal string constants.

> **Note:** A parameter data type can be left unspecified by setting it to zero, or by making the array of parameter type OIDs shorter than the number of parameter symbols (`$n`) used in the query string. Another special case is that a parameter's type can be specified as `void` (that is, the OID of the `void` pseudotype). This is meant to allow parameter symbols to be used for function parameters that are actually OUT parameters. Ordinarily there is no context in which a `void` parameter could be used, but if such a parameter symbol appears in a function's parameter list, it is effectively ignored. For example, a function call such as `foo($1,$2,$3,$4)` could match a function with two IN and two OUT arguments, if `$3` and `$4` are specified as having type `void`.

> **Note:** The query string contained in a Parse message cannot include more than one SQL statement; else a syntax error is reported. This restriction does not exist in the simple-query protocol, but it does exist in the extended protocol, because allowing prepared statements or portals to contain multiple commands would complicate the protocol unduly.

If successfully created, a named prepared-statement object lasts till the end of the current session, unless explicitly destroyed. An unnamed prepared statement lasts only until the next Parse statement specifying the unnamed statement as destination is issued. (Note that a simple Query message also destroys the unnamed statement.) Named prepared statements must be explicitly closed before they can be redefined by a Parse message, but this is not required for the unnamed statement. Named prepared statements can also be created and accessed at the SQL command level, using `PREPARE` and `EXECUTE`.

Once a prepared statement exists, it can be readied for execution using a Bind message. The Bind message gives the name of the source prepared statement (empty string denotes the unnamed prepared statement), the name of the destination portal (empty string denotes the unnamed portal), and the values to use for any parameter placeholders present in the prepared statement. The supplied parameter set must match those needed by the prepared statement. (If you declared any `void` parameters in the Parse message, pass NULL values for them in the Bind message.) Bind also specifies the format to use for any data returned by the query; the format can be specified overall, or per-column. The response is either BindComplete or ErrorResponse.

> **Note:** The choice between text and binary output is determined by the format codes given in Bind, regardless of the SQL command involved. The `BINARY` attribute in cursor declarations is irrelevant when using extended query protocol.

Query planning for named prepared-statement objects occurs when the Parse message is processed. If a query will be repeatedly executed with different parameters, it may be beneficial to send a single Parse message containing a parameterized query, followed by multiple Bind and Execute messages. This will avoid replanning the query on each execution.

The unnamed prepared statement is likewise planned during Parse processing if the Parse message defines no parameters. But if there are parameters, query planning occurs during Bind processing instead. This allows the planner to make use of the actual values of the parameters provided in the Bind message when planning the query.

> **Note:** Query plans generated from a parameterized query may be less efficient than query plans generated from an equivalent query with actual parameter values substituted. The query planner cannot make decisions based on actual parameter values (for example, index selectivity) when planning a parameterized query assigned to a named prepared-statement object. This possible penalty is avoided when using the unnamed statement, since it is not planned until actual parameter values are available. The cost is that planning must occur afresh for each Bind, even if the query stays the same.

If successfully created, a named portal object lasts till the end of the current transaction, unless explicitly destroyed. An unnamed portal is destroyed at the end of the transaction, or as soon as the next Bind statement specifying the unnamed portal as destination is issued. (Note that a simple Query message also destroys the unnamed portal.) Named portals must be explicitly closed before they can be redefined by a Bind message, but this is not required for the unnamed portal. Named portals can also be created and accessed at the SQL command level, using `DECLARE CURSOR` and `FETCH`.

Once a portal exists, it can be executed using an Execute message. The Execute message specifies the portal name (empty string denotes the unnamed portal) and a maximum result-row count (zero meaning "fetch all rows"). The result-row count is only meaningful for portals containing commands that return row sets; in other cases the command is always executed to completion, and the row count is ignored. The possible responses to Execute are the same as those described above for queries issued via simple query protocol, except that Execute doesn't cause ReadyForQuery or RowDescription to be issued.

If Execute terminates before completing the execution of a portal (due to reaching a nonzero result-row count), it will send a PortalSuspended message; the appearance of this message tells the frontend that another Execute should be issued against the same portal to complete the operation. The CommandComplete message indicating completion of the source SQL command is not sent until the portal's execution is completed. Therefore, an Execute phase is always terminated by the appearance of exactly one of these messages: CommandComplete, EmptyQueryResponse (if the portal was created from an empty query string), ErrorResponse, or PortalSuspended.

At completion of each series of extended-query messages, the frontend should issue a Sync message. This parameterless message causes the backend to close the current transaction if it's not inside a `BEGIN`/`COMMIT` transaction block ("close" meaning to commit if no error, or roll back if error). Then a ReadyForQuery response is issued. The purpose of Sync is to provide a resynchronization point for error recovery. When an error is detected while processing any extended-query message, the backend issues ErrorResponse, then reads and discards messages until a Sync is reached, then issues ReadyForQuery and returns to normal message processing. (But note that no skipping occurs if an error is detected *while* processing Sync — this ensures that there is one and only one ReadyForQuery sent for each Sync.)

> **Note:** Sync does not cause a transaction block opened with `BEGIN` to be closed. It is possible to detect this situation since the ReadyForQuery message includes transaction status information.

In addition to these fundamental, required operations, there are several optional operations that can be used with extended-query protocol.

The Describe message (portal variant) specifies the name of an existing portal (or an empty string for the unnamed portal). The response is a RowDescription message describing the rows that will be returned by executing the portal; or a NoData message if the portal does not contain a query that will return rows; or ErrorResponse if there is no such portal.

The Describe message (statement variant) specifies the name of an existing prepared statement (or an empty string for the unnamed prepared statement). The response is a ParameterDescription message describing the parameters needed by the statement, followed by a RowDescription message describing the rows that will be returned when the statement is eventually executed (or a NoData message if the statement will not return rows). ErrorResponse is issued if there is no such prepared statement. Note that since Bind has not yet been issued, the formats to be used for returned columns are not yet known to the backend; the format code fields in the RowDescription message will be zeroes in this case.

> **Tip:** In most scenarios the frontend should issue one or the other variant of Describe before issuing Execute, to ensure that it knows how to interpret the results it will get back.

The Close message closes an existing prepared statement or portal and releases resources. It is not an error to issue Close against a nonexistent statement or portal name. The response is normally CloseComplete, but could be ErrorResponse if some difficulty is encountered while releasing resources. Note that closing a prepared statement implicitly closes any open portals that were constructed from that statement.

The Flush message does not cause any specific output to be generated, but forces the backend to deliver any data pending in its output buffers. A Flush must be sent after any extended-query command except Sync, if the frontend wishes to examine the results of that command before issuing more commands. Without Flush, messages returned by the backend will be combined into the minimum possible number of packets to minimize network overhead.

> **Note:** The simple Query message is approximately equivalent to the series Parse, Bind, portal Describe, Execute, Close, Sync, using the unnamed prepared statement and portal objects and no parameters. One difference is that it will accept multiple SQL statements in the query string, automatically performing the bind/describe/execute sequence for each one in succession. Another difference is that it will not return ParseComplete, BindComplete, CloseComplete, or NoData messages.

44.2.4. Function Call

The Function Call sub-protocol allows the client to request a direct call of any function that exists in the database's `pg_proc` system catalog. The client must have execute permission for the function.

> **Note:** The Function Call sub-protocol is a legacy feature that is probably best avoided in new code. Similar results can be accomplished by setting up a prepared statement that does `SELECT function($1, ...)`. The Function Call cycle can then be replaced with Bind/Execute.

A Function Call cycle is initiated by the frontend sending a FunctionCall message to the backend. The backend then sends one or more response messages depending on the results of the function call, and finally a ReadyForQuery response message. ReadyForQuery informs the frontend that it may safely send a new query or function call.

The possible response messages from the backend are:

ErrorResponse

An error has occurred.

FunctionCallResponse

The function call was completed and returned the result given in the message. (Note that the Function Call protocol can only handle a single scalar result, not a row type or set of results.)

ReadyForQuery

Processing of the function call is complete. ReadyForQuery will always be sent, whether processing terminates successfully or with an error.

NoticeResponse

A warning message has been issued in relation to the function call. Notices are in addition to other responses, i.e., the backend will continue processing the command.

44.2.5. COPY Operations

The `COPY` command allows high-speed bulk data transfer to or from the server. Copy-in and copy-out operations each switch the connection into a distinct sub-protocol, which lasts until the operation is completed.

Copy-in mode (data transfer to the server) is initiated when the backend executes a `COPY FROM STDIN` SQL statement. The backend sends a CopyInResponse message to the frontend. The frontend should then send zero or more CopyData messages, forming a stream of input data. (The message boundaries are not required to have anything to do with row boundaries, although that is often a reasonable choice.) The frontend can terminate the copy-in mode by sending either a CopyDone message (allowing successful termination) or a CopyFail message (which will cause the `COPY` SQL statement to fail with an error). The backend then reverts to the command-processing mode it was in before the `COPY` started, which will be either simple or extended query protocol. It will next send either CommandComplete (if successful) or ErrorResponse (if not).

In the event of a backend-detected error during copy-in mode (including receipt of a CopyFail message), the backend will issue an ErrorResponse message. If the `COPY` command was issued via an extended-query message, the backend will now discard frontend messages until a Sync message is received, then it will issue ReadyForQuery and return to normal processing. If the `COPY` command was issued in a simple Query message, the rest of that message is discarded and ReadyForQuery is issued. In either case, any subsequent CopyData, CopyDone, or CopyFail messages issued by the frontend will simply be dropped.

The backend will ignore Flush and Sync messages received during copy-in mode. Receipt of any other non-copy message type constitutes an error that will abort the copy-in state as described above. (The exception for Flush and Sync is for the convenience of client libraries that always send Flush or Sync after an Execute message, without checking whether the command to be executed is a `COPY FROM STDIN`.)

Copy-out mode (data transfer from the server) is initiated when the backend executes a `COPY TO STDOUT` SQL statement. The backend sends a CopyOutResponse message to the frontend, followed by zero or more CopyData messages (always one per row), followed by CopyDone. The backend then reverts to the command-processing mode it was in before the `COPY` started, and sends CommandComplete. The frontend

cannot abort the transfer (except by closing the connection or issuing a Cancel request), but it can discard unwanted CopyData and CopyDone messages.

In the event of a backend-detected error during copy-out mode, the backend will issue an ErrorResponse message and revert to normal processing. The frontend should treat receipt of ErrorResponse (or indeed any message type other than CopyData or CopyDone) as terminating the copy-out mode.

The CopyInResponse and CopyOutResponse messages include fields that inform the frontend of the number of columns per row and the format codes being used for each column. (As of the present implementation, all columns in a given `COPY` operation will use the same format, but the message design does not assume this.)

44.2.6. Asynchronous Operations

There are several cases in which the backend will send messages that are not specifically prompted by the frontend's command stream. Frontends must be prepared to deal with these messages at any time, even when not engaged in a query. At minimum, one should check for these cases before beginning to read a query response.

It is possible for NoticeResponse messages to be generated due to outside activity; for example, if the database administrator commands a "fast" database shutdown, the backend will send a NoticeResponse indicating this fact before closing the connection. Accordingly, frontends should always be prepared to accept and display NoticeResponse messages, even when the connection is nominally idle.

ParameterStatus messages will be generated whenever the active value changes for any of the parameters the backend believes the frontend should know about. Most commonly this occurs in response to a `SET` SQL command executed by the frontend, and this case is effectively synchronous — but it is also possible for parameter status changes to occur because the administrator changed a configuration file and then sent the SIGHUP signal to the server. Also, if a `SET` command is rolled back, an appropriate ParameterStatus message will be generated to report the current effective value.

At present there is a hard-wired set of parameters for which ParameterStatus will be generated: they are `server_version`, `server_encoding`, `client_encoding`, `is_superuser`, `session_authorization`, `DateStyle`, `TimeZone`, `integer_datetimes`, and `standard_conforming_strings`. (`server_encoding`, `TimeZone`, and `integer_datetimes` were not reported by releases before 8.0; `standard_conforming_strings` was not reported by releases before 8.1.) Note that `server_version`, `server_encoding` and `integer_datetimes` are pseudo-parameters that cannot change after startup. This set might change in the future, or even become configurable. Accordingly, a frontend should simply ignore ParameterStatus for parameters that it does not understand or care about.

If a frontend issues a `LISTEN` command, then the backend will send a NotificationResponse message (not to be confused with NoticeResponse!) whenever a `NOTIFY` command is executed for the same notification name.

> **Note:** At present, NotificationResponse can only be sent outside a transaction, and thus it will not occur in the middle of a command-response series, though it may occur just before ReadyForQuery. It is unwise to design frontend logic that assumes that, however. Good practice is to be able to accept NotificationResponse at any point in the protocol.

44.2.7. Cancelling Requests in Progress

During the processing of a query, the frontend may request cancellation of the query. The cancel request is not sent directly on the open connection to the backend for reasons of implementation efficiency: we don't want to have the backend constantly checking for new input from the frontend during query processing. Cancel requests should be relatively infrequent, so we make them slightly cumbersome in order to avoid a penalty in the normal case.

To issue a cancel request, the frontend opens a new connection to the server and sends a CancelRequest message, rather than the StartupMessage message that would ordinarily be sent across a new connection. The server will process this request and then close the connection. For security reasons, no direct reply is made to the cancel request message.

A CancelRequest message will be ignored unless it contains the same key data (PID and secret key) passed to the frontend during connection start-up. If the request matches the PID and secret key for a currently executing backend, the processing of the current query is aborted. (In the existing implementation, this is done by sending a special signal to the backend process that is processing the query.)

The cancellation signal may or may not have any effect — for example, if it arrives after the backend has finished processing the query, then it will have no effect. If the cancellation is effective, it results in the current command being terminated early with an error message.

The upshot of all this is that for reasons of both security and efficiency, the frontend has no direct way to tell whether a cancel request has succeeded. It must continue to wait for the backend to respond to the query. Issuing a cancel simply improves the odds that the current query will finish soon, and improves the odds that it will fail with an error message instead of succeeding.

Since the cancel request is sent across a new connection to the server and not across the regular frontend/backend communication link, it is possible for the cancel request to be issued by any process, not just the frontend whose query is to be canceled. This may have some benefits of flexibility in building multiple-process applications. It also introduces a security risk, in that unauthorized persons might try to cancel queries. The security risk is addressed by requiring a dynamically generated secret key to be supplied in cancel requests.

44.2.8. Termination

The normal, graceful termination procedure is that the frontend sends a Terminate message and immediately closes the connection. On receipt of this message, the backend closes the connection and terminates.

In rare cases (such as an administrator-commanded database shutdown) the backend may disconnect without any frontend request to do so. In such cases the backend will attempt to send an error or notice message giving the reason for the disconnection before it closes the connection.

Other termination scenarios arise from various failure cases, such as core dump at one end or the other, loss of the communications link, loss of message-boundary synchronization, etc. If either frontend or backend sees an unexpected closure of the connection, it should clean up and terminate. The frontend has the option of launching a new backend by recontacting the server if it doesn't want to terminate itself. Closing the connection is also advisable if an unrecognizable message type is received, since this probably indicates loss of message-boundary sync.

For either normal or abnormal termination, any open transaction is rolled back, not committed. One should note however that if a frontend disconnects while a non-`SELECT` query is being processed, the backend will

probably finish the query before noticing the disconnection. If the query is outside any transaction block (BEGIN ... COMMIT sequence) then its results may be committed before the disconnection is recognized.

44.2.9. SSL Session Encryption

If PostgreSQL was built with SSL support, frontend/backend communications can be encrypted using SSL. This provides communication security in environments where attackers might be able to capture the session traffic. For more information on encrypting PostgreSQL sessions with SSL, see Section 16.7.

To initiate an SSL-encrypted connection, the frontend initially sends an SSLRequest message rather than a StartupMessage. The server then responds with a single byte containing S or N, indicating that it is willing or unwilling to perform SSL, respectively. The frontend may close the connection at this point if it is dissatisfied with the response. To continue after S, perform an SSL startup handshake (not described here, part of the SSL specification) with the server. If this is successful, continue with sending the usual StartupMessage. In this case the StartupMessage and all subsequent data will be SSL-encrypted. To continue after N, send the usual StartupMessage and proceed without encryption.

The frontend should also be prepared to handle an ErrorMessage response to SSLRequest from the server. This would only occur if the server predates the addition of SSL support to PostgreSQL. In this case the connection must be closed, but the frontend may choose to open a fresh connection and proceed without requesting SSL.

An initial SSLRequest may also be used in a connection that is being opened to send a CancelRequest message.

While the protocol itself does not provide a way for the server to force SSL encryption, the administrator may configure the server to reject unencrypted sessions as a byproduct of authentication checking.

44.3. Message Data Types

This section describes the base data types used in messages.

Int*n*(*i*)

An *n*-bit integer in network byte order (most significant byte first). If *i* is specified it is the exact value that will appear, otherwise the value is variable. Eg. Int16, Int32(42).

Int*n*[*k*]

An array of *k* *n*-bit integers, each in network byte order. The array length *k* is always determined by an earlier field in the message. Eg. Int16[M].

String(*s*)

A null-terminated string (C-style string). There is no specific length limitation on strings. If *s* is specified it is the exact value that will appear, otherwise the value is variable. Eg. String, String("user").

> **Note:** *There is no predefined limit* on the length of a string that can be returned by the backend. Good coding strategy for a frontend is to use an expandable buffer so that anything that fits in

memory can be accepted. If that's not feasible, read the full string and discard trailing characters that don't fit into your fixed-size buffer.

Byte*n*(*c*)

Exactly *n* bytes. If the field width *n* is not a constant, it is always determinable from an earlier field in the message. If *c* is specified it is the exact value. Eg. Byte2, Byte1('\n').

44.4. Message Formats

This section describes the detailed format of each message. Each is marked to indicate that it may be sent by a frontend (F), a backend (B), or both (F & B). Notice that although each message includes a byte count at the beginning, the message format is defined so that the message end can be found without reference to the byte count. This aids validity checking. (The CopyData message is an exception, because it forms part of a data stream; the contents of any individual CopyData message may not be interpretable on their own.)

AuthenticationOk (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(8)

Length of message contents in bytes, including self.

Int32(0)

Specifies that the authentication was successful.

AuthenticationKerberosV5 (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(8)

Length of message contents in bytes, including self.

Int32(2)

Specifies that Kerberos V5 authentication is required.

AuthenticationCleartextPassword (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(8)

Length of message contents in bytes, including self.

Int32(3)

Specifies that a clear-text password is required.

AuthenticationCryptPassword (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(10)

Length of message contents in bytes, including self.

Int32(4)

Specifies that a crypt()-encrypted password is required.

Byte2

The salt to use when encrypting the password.

AuthenticationMD5Password (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(12)

Length of message contents in bytes, including self.

Int32(5)

Specifies that an MD5-encrypted password is required.

Byte4

The salt to use when encrypting the password.

AuthenticationSCMCredential (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(8)

Length of message contents in bytes, including self.

Int32(6)

Specifies that an SCM credentials message is required.

BackendKeyData (B)

Byte1('K')

Identifies the message as cancellation key data. The frontend must save these values if it wishes to be able to issue CancelRequest messages later.

Int32(12)

Length of message contents in bytes, including self.

Int32

The process ID of this backend.

Int32

The secret key of this backend.

Bind (F)

Byte1('B')

Identifies the message as a Bind command.

Int32

Length of message contents in bytes, including self.

String

The name of the destination portal (an empty string selects the unnamed portal).

String

The name of the source prepared statement (an empty string selects the unnamed prepared statement).

Int16

The number of parameter format codes that follow (denoted *C* below). This can be zero to indicate that there are no parameters or that the parameters all use the default format (text); or one, in which case the specified format code is applied to all parameters; or it can equal the actual number of parameters.

Int16[*C*]

The parameter format codes. Each must presently be zero (text) or one (binary).

Int16

The number of parameter values that follow (possibly zero). This must match the number of parameters needed by the query.

Next, the following pair of fields appear for each parameter:

Int32

The length of the parameter value, in bytes (this count does not include itself). Can be zero. As a special case, -1 indicates a NULL parameter value. No value bytes follow in the NULL case.

Byte*n*

The value of the parameter, in the format indicated by the associated format code. *n* is the above length.

After the last parameter, the following fields appear:

Int16

The number of result-column format codes that follow (denoted *R* below). This can be zero to indicate that there are no result columns or that the result columns should all use the default format (text); or one, in which case the specified format code is applied to all result columns (if any); or it can equal the actual number of result columns of the query.

Int16[*R*]

The result-column format codes. Each must presently be zero (text) or one (binary).

BindComplete (B)

Byte1('2')

Identifies the message as a Bind-complete indicator.

Int32(4)

Length of message contents in bytes, including self.

CancelRequest (F)

Int32(16)

Length of message contents in bytes, including self.

Int32(80877102)

The cancel request code. The value is chosen to contain `1234` in the most significant 16 bits, and `5678` in the least 16 significant bits. (To avoid confusion, this code must not be the same as any protocol version number.)

Int32

The process ID of the target backend.

Int32

The secret key for the target backend.

Close (F)

Byte1('C')

Identifies the message as a Close command.

Int32

Length of message contents in bytes, including self.

Byte1

'S' to close a prepared statement; or 'P' to close a portal.

String

The name of the prepared statement or portal to close (an empty string selects the unnamed prepared statement or portal).

CloseComplete (B)

Byte1('3')

Identifies the message as a Close-complete indicator.

Int32(4)

Length of message contents in bytes, including self.

CommandComplete (B)

Byte1('C')

Identifies the message as a command-completed response.

Int32

Length of message contents in bytes, including self.

String

The command tag. This is usually a single word that identifies which SQL command was completed.

For an INSERT command, the tag is INSERT *oid rows*, where *rows* is the number of rows inserted. *oid* is the object ID of the inserted row if *rows* is 1 and the target table has OIDs; otherwise *oid* is 0.

For a DELETE command, the tag is DELETE *rows* where *rows* is the number of rows deleted.

For an UPDATE command, the tag is UPDATE *rows* where *rows* is the number of rows updated.

For a `MOVE` command, the tag is `MOVE` *`rows`* where *`rows`* is the number of rows the cursor's position has been changed by.

For a `FETCH` command, the tag is `FETCH` *`rows`* where *`rows`* is the number of rows that have been retrieved from the cursor.

For a `COPY` command, the tag is `COPY` *`rows`* where *`rows`* is the number of rows copied. (Note: the row count appears only in PostgreSQL 8.2 and later.)

CopyData (F & B)

Byte1('d')

Identifies the message as `COPY` data.

Int32

Length of message contents in bytes, including self.

Byte*n*

Data that forms part of a `COPY` data stream. Messages sent from the backend will always correspond to single data rows, but messages sent by frontends may divide the data stream arbitrarily.

CopyDone (F & B)

Byte1('c')

Identifies the message as a `COPY`-complete indicator.

Int32(4)

Length of message contents in bytes, including self.

CopyFail (F)

Byte1('f')

Identifies the message as a `COPY`-failure indicator.

Int32

Length of message contents in bytes, including self.

String

An error message to report as the cause of failure.

CopyInResponse (B)

Byte1('G')

Identifies the message as a Start Copy In response. The frontend must now send copy-in data (if not prepared to do so, send a CopyFail message).

Int32

Length of message contents in bytes, including self.

Int8

0 indicates the overall `COPY` format is textual (rows separated by newlines, columns separated by separator characters, etc). 1 indicates the overall copy format is binary (similar to DataRow format). See *COPY* for more information.

Int16

The number of columns in the data to be copied (denoted *N* below).

Int16[*N*]

The format codes to be used for each column. Each must presently be zero (text) or one (binary). All must be zero if the overall copy format is textual.

CopyOutResponse (B)

Byte1('H')

Identifies the message as a Start Copy Out response. This message will be followed by copy-out data.

Int32

Length of message contents in bytes, including self.

Int8

0 indicates the overall `COPY` format is textual (rows separated by newlines, columns separated by separator characters, etc). 1 indicates the overall copy format is binary (similar to DataRow format). See *COPY* for more information.

Int16

The number of columns in the data to be copied (denoted *N* below).

Int16[*N*]

The format codes to be used for each column. Each must presently be zero (text) or one (binary). All must be zero if the overall copy format is textual.

DataRow (B)

Byte1('D')

Identifies the message as a data row.

Int32

Length of message contents in bytes, including self.

Int16

The number of column values that follow (possibly zero).

Next, the following pair of fields appear for each column:

Int32

The length of the column value, in bytes (this count does not include itself). Can be zero. As a special case, -1 indicates a NULL column value. No value bytes follow in the NULL case.

Byte*n*

The value of the column, in the format indicated by the associated format code. *n* is the above length.

Describe (F)

Byte1('D')

Identifies the message as a Describe command.

Int32

Length of message contents in bytes, including self.

Byte1

'S' to describe a prepared statement; or 'P' to describe a portal.

String

The name of the prepared statement or portal to describe (an empty string selects the unnamed prepared statement or portal).

EmptyQueryResponse (B)

Byte1('I')

Identifies the message as a response to an empty query string. (This substitutes for CommandComplete.)

Int32(4)

Length of message contents in bytes, including self.

ErrorResponse (B)

Byte1('E')

Identifies the message as an error.

Int32

Length of message contents in bytes, including self.

The message body consists of one or more identified fields, followed by a zero byte as a terminator. Fields may appear in any order. For each field there is the following:

Byte1

A code identifying the field type; if zero, this is the message terminator and no string follows. The presently defined field types are listed in Section 44.5. Since more field types may be added in future, frontends should silently ignore fields of unrecognized type.

String

The field value.

Execute (F)

Byte1('E')

Identifies the message as an Execute command.

Int32

Length of message contents in bytes, including self.

String

The name of the portal to execute (an empty string selects the unnamed portal).

Int32

Maximum number of rows to return, if portal contains a query that returns rows (ignored otherwise). Zero denotes "no limit".

Flush (F)

Byte1('H')

Identifies the message as a Flush command.

Int32(4)

Length of message contents in bytes, including self.

FunctionCall (F)

Byte1('F')

Identifies the message as a function call.

Int32

Length of message contents in bytes, including self.

Int32

Specifies the object ID of the function to call.

Int16

The number of argument format codes that follow (denoted C below). This can be zero to indicate that there are no arguments or that the arguments all use the default format (text); or one,

in which case the specified format code is applied to all arguments; or it can equal the actual number of arguments.

Int16[*C*]

The argument format codes. Each must presently be zero (text) or one (binary).

Int16

Specifies the number of arguments being supplied to the function.

Next, the following pair of fields appear for each argument:

Int32

The length of the argument value, in bytes (this count does not include itself). Can be zero. As a special case, -1 indicates a NULL argument value. No value bytes follow in the NULL case.

Byte*n*

The value of the argument, in the format indicated by the associated format code. *n* is the above length.

After the last argument, the following field appears:

Int16

The format code for the function result. Must presently be zero (text) or one (binary).

FunctionCallResponse (B)

Byte1('V')

Identifies the message as a function call result.

Int32

Length of message contents in bytes, including self.

Int32

The length of the function result value, in bytes (this count does not include itself). Can be zero. As a special case, -1 indicates a NULL function result. No value bytes follow in the NULL case.

Byte*n*

The value of the function result, in the format indicated by the associated format code. *n* is the above length.

NoData (B)

Byte1('n')

Identifies the message as a no-data indicator.

Int32(4)

Length of message contents in bytes, including self.

NoticeResponse (B)

Byte1('N')

Identifies the message as a notice.

Int32

Length of message contents in bytes, including self.

The message body consists of one or more identified fields, followed by a zero byte as a terminator. Fields may appear in any order. For each field there is the following:

Byte1

A code identifying the field type; if zero, this is the message terminator and no string follows. The presently defined field types are listed in Section 44.5. Since more field types may be added in future, frontends should silently ignore fields of unrecognized type.

String

The field value.

NotificationResponse (B)

Byte1('A')

Identifies the message as a notification response.

Int32

Length of message contents in bytes, including self.

Int32

The process ID of the notifying backend process.

String

The name of the condition that the notify has been raised on.

String

Additional information passed from the notifying process. (Currently, this feature is unimplemented so the field is always an empty string.)

ParameterDescription (B)

Byte1('t')

Identifies the message as a parameter description.

Int32

Length of message contents in bytes, including self.

Int16

The number of parameters used by the statement (may be zero).

Then, for each parameter, there is the following:

Int32

Specifies the object ID of the parameter data type.

ParameterStatus (B)

Byte1('S')

Identifies the message as a run-time parameter status report.

Int32

Length of message contents in bytes, including self.

String

The name of the run-time parameter being reported.

String

The current value of the parameter.

Parse (F)

Byte1('P')

Identifies the message as a Parse command.

Int32

Length of message contents in bytes, including self.

String

The name of the destination prepared statement (an empty string selects the unnamed prepared statement).

String

The query string to be parsed.

Int16

The number of parameter data types specified (may be zero). Note that this is not an indication of the number of parameters that might appear in the query string, only the number that the frontend wants to prespecify types for.

Then, for each parameter, there is the following:

Int32

Specifies the object ID of the parameter data type. Placing a zero here is equivalent to leaving the type unspecified.

ParseComplete (B)

Byte1('1')

Identifies the message as a Parse-complete indicator.

Int32(4)

Length of message contents in bytes, including self.

PasswordMessage (F)

Byte1('p')

Identifies the message as a password response.

Int32

Length of message contents in bytes, including self.

String

The password (encrypted, if requested).

PortalSuspended (B)

Byte1('s')

Identifies the message as a portal-suspended indicator. Note this only appears if an Execute message's row-count limit was reached.

Int32(4)

Length of message contents in bytes, including self.

Query (F)

Byte1('Q')

Identifies the message as a simple query.

Int32

Length of message contents in bytes, including self.

String

The query string itself.

ReadyForQuery (B)

Byte1('Z')

Identifies the message type. ReadyForQuery is sent whenever the backend is ready for a new query cycle.

Int32(5)

Length of message contents in bytes, including self.

Byte1

Current backend transaction status indicator. Possible values are '`I`' if idle (not in a transaction block); '`T`' if in a transaction block; or '`E`' if in a failed transaction block (queries will be rejected until block is ended).

RowDescription (B)

Byte1('T')

Identifies the message as a row description.

Int32

Length of message contents in bytes, including self.

Int16

Specifies the number of fields in a row (may be zero).

Then, for each field, there is the following:

String

The field name.

Int32

If the field can be identified as a column of a specific table, the object ID of the table; otherwise zero.

Int16

If the field can be identified as a column of a specific table, the attribute number of the column; otherwise zero.

Int32

The object ID of the field's data type.

Int16

The data type size (see `pg_type.typlen`). Note that negative values denote variable-width types.

Int32

The type modifier (see `pg_attribute.atttypmod`). The meaning of the modifier is type-specific.

Int16

The format code being used for the field. Currently will be zero (text) or one (binary). In a RowDescription returned from the statement variant of Describe, the format code is not yet known and will always be zero.

SSLRequest (F)

Int32(8)

Length of message contents in bytes, including self.

Int32(80877103)

The SSL request code. The value is chosen to contain `1234` in the most significant 16 bits, and `5679` in the least 16 significant bits. (To avoid confusion, this code must not be the same as any protocol version number.)

StartupMessage (F)

Int32

Length of message contents in bytes, including self.

Int32(196608)

The protocol version number. The most significant 16 bits are the major version number (3 for the protocol described here). The least significant 16 bits are the minor version number (0 for the protocol described here).

The protocol version number is followed by one or more pairs of parameter name and value strings. A zero byte is required as a terminator after the last name/value pair. Parameters can appear in any order. `user` is required, others are optional. Each parameter is specified as:

String

The parameter name. Currently recognized names are:

`user`

The database user name to connect as. Required; there is no default.

`database`

The database to connect to. Defaults to the user name.

`options`

Command-line arguments for the backend. (This is deprecated in favor of setting individual run-time parameters.)

In addition to the above, any run-time parameter that can be set at backend start time may be listed. Such settings will be applied during backend start (after parsing the command-line options if any). The values will act as session defaults.

String

The parameter value.

Sync (F)

Byte1('S')

Identifies the message as a Sync command.

Int32(4)

Length of message contents in bytes, including self.

Terminate (F)

Byte1('X')

Identifies the message as a termination.

Int32(4)

Length of message contents in bytes, including self.

44.5. Error and Notice Message Fields

This section describes the fields that may appear in ErrorResponse and NoticeResponse messages. Each field type has a single-byte identification token. Note that any given field type should appear at most once per message.

`S`

Severity: the field contents are `ERROR`, `FATAL`, or `PANIC` (in an error message), or `WARNING`, `NOTICE`, `DEBUG`, `INFO`, or `LOG` (in a notice message), or a localized translation of one of these. Always present.

`C`

Code: the SQLSTATE code for the error (see Appendix A). Not localizable. Always present.

`M`

Message: the primary human-readable error message. This should be accurate but terse (typically one line). Always present.

`D`

Detail: an optional secondary error message carrying more detail about the problem. May run to multiple lines.

`H`

Hint: an optional suggestion what to do about the problem. This is intended to differ from Detail in that it offers advice (potentially inappropriate) rather than hard facts. May run to multiple lines.

`P`

Position: the field value is a decimal ASCII integer, indicating an error cursor position as an index into the original query string. The first character has index 1, and positions are measured in characters not bytes.

`p`

Internal position: this is defined the same as the `P` field, but it is used when the cursor position refers to an internally generated command rather than the one submitted by the client. The `q` field will always appear when this field appears.

`q`

Internal query: the text of a failed internally-generated command. This could be, for example, a SQL query issued by a PL/pgSQL function.

`W`

Where: an indication of the context in which the error occurred. Presently this includes a call stack traceback of active procedural language functions and internally-generated queries. The trace is one entry per line, most recent first.

`F`

File: the file name of the source-code location where the error was reported.

`L`

Line: the line number of the source-code location where the error was reported.

`R`

Routine: the name of the source-code routine reporting the error.

The client is responsible for formatting displayed information to meet its needs; in particular it should break long lines as needed. Newline characters appearing in the error message fields should be treated as paragraph breaks, not line breaks.

44.6. Summary of Changes since Protocol 2.0

This section provides a quick checklist of changes, for the benefit of developers trying to update existing client libraries to protocol 3.0.

The initial startup packet uses a flexible list-of-strings format instead of a fixed format. Notice that session default values for run-time parameters can now be specified directly in the startup packet. (Actually, you could do that before using the `options` field, but given the limited width of `options` and the lack of any way to quote whitespace in the values, it wasn't a very safe technique.)

All messages now have a length count immediately following the message type byte (except for startup packets, which have no type byte). Also note that PasswordMessage now has a type byte.

ErrorResponse and NoticeResponse ('`E`' and '`N`') messages now contain multiple fields, from which the client code may assemble an error message of the desired level of verbosity. Note that individual fields will typically not end with a newline, whereas the single string sent in the older protocol always did.

The ReadyForQuery ('`Z`') message includes a transaction status indicator.

The distinction between BinaryRow and DataRow message types is gone; the single DataRow message type serves for returning data in all formats. Note that the layout of DataRow has changed to make it easier to parse. Also, the representation of binary values has changed: it is no longer directly tied to the server's internal representation.

There is a new "extended query" sub-protocol, which adds the frontend message types Parse, Bind, Execute, Describe, Close, Flush, and Sync, and the backend message types ParseComplete, BindComplete, PortalSuspended, ParameterDescription, NoData, and CloseComplete. Existing clients do not have to concern themselves with this sub-protocol, but making use of it may allow improvements in performance or functionality.

`COPY` data is now encapsulated into CopyData and CopyDone messages. There is a well-defined way to recover from errors during `COPY`. The special "`\.`" last line is not needed anymore, and is not sent during `COPY OUT`. (It is still recognized as a terminator during `COPY IN`, but its use is deprecated and will eventually be removed.) Binary `COPY` is supported. The CopyInResponse and CopyOutResponse messages include fields indicating the number of columns and the format of each column.

The layout of FunctionCall and FunctionCallResponse messages has changed. FunctionCall can now support passing NULL arguments to functions. It also can handle passing parameters and retrieving results in either text or binary format. There is no longer any reason to consider FunctionCall a potential security hole, since it does not offer direct access to internal server data representations.

The backend sends ParameterStatus ('`S`') messages during connection startup for all parameters it considers interesting to the client library. Subsequently, a ParameterStatus message is sent whenever the active value changes for any of these parameters.

The RowDescription ('`T`') message carries new table OID and column number fields for each column of the described row. It also shows the format code for each column.

The CursorResponse ('`P`') message is no longer generated by the backend.

The NotificationResponse ('`A`') message has an additional string field, which is presently empty but may someday carry additional data passed from the `NOTIFY` event sender.

The EmptyQueryResponse ('`I`') message used to include an empty string parameter; this has been removed.

Chapter 45. PostgreSQL Coding Conventions

45.1. Formatting

Source code formatting uses 4 column tab spacing, with tabs preserved (i.e. tabs are not expanded to spaces). Each logical indentation level is one additional tab stop. Layout rules (brace positioning, etc) follow BSD conventions.

While submitted patches do not absolutely have to follow these formatting rules, it's a good idea to do so. Your code will get run through pgindent, so there's no point in making it look nice under some other set of formatting conventions.

The `src/tools` directory contains sample settings files that can be used with the emacs, xemacs or vim editors to help ensure that they format code according to these conventions.

The text browsing tools more and less can be invoked as

```
more -x4
less -x4
```

to make them show tabs appropriately.

45.2. Reporting Errors Within the Server

Error, warning, and log messages generated within the server code should be created using `ereport`, or its older cousin `elog`. The use of this function is complex enough to require some explanation.

There are two required elements for every message: a severity level (ranging from `DEBUG` to `PANIC`) and a primary message text. In addition there are optional elements, the most common of which is an error identifier code that follows the SQL spec's SQLSTATE conventions. `ereport` itself is just a shell function, that exists mainly for the syntactic convenience of making message generation look like a function call in the C source code. The only parameter accepted directly by `ereport` is the severity level. The primary message text and any optional message elements are generated by calling auxiliary functions, such as `errmsg`, within the `ereport` call.

A typical call to `ereport` might look like this:

```
ereport(ERROR,
        (errcode(ERRCODE_DIVISION_BY_ZERO),
         errmsg("division by zero")));
```

This specifies error severity level `ERROR` (a run-of-the-mill error). The `errcode` call specifies the SQLSTATE error code using a macro defined in `src/include/utils/errcodes.h`. The `errmsg` call provides the primary message text. Notice the extra set of parentheses surrounding the auxiliary function calls — these are annoying but syntactically necessary.

Here is a more complex example:

```
ereport(ERROR,
        (errcode(ERRCODE_AMBIGUOUS_FUNCTION),
         errmsg("function %s is not unique",
                func_signature_string(funcname, nargs,
                                      actual_arg_types)),
         errhint("Unable to choose a best candidate function. "
                 "You may need to add explicit typecasts.")));
```

This illustrates the use of format codes to embed run-time values into a message text. Also, an optional "hint" message is provided.

The available auxiliary routines for `ereport` are:

- `errcode(sqlerrcode)` specifies the SQLSTATE error identifier code for the condition. If this routine is not called, the error identifier defaults to `ERRCODE_INTERNAL_ERROR` when the error severity level is `ERROR` or higher, `ERRCODE_WARNING` when the error level is `WARNING`, otherwise (for `NOTICE` and below) `ERRCODE_SUCCESSFUL_COMPLETION`. While these defaults are often convenient, always think whether they are appropriate before omitting the `errcode()` call.
- `errmsg(const char *msg, ...)` specifies the primary error message text, and possibly run-time values to insert into it. Insertions are specified by `sprintf`-style format codes. In addition to the standard format codes accepted by `sprintf`, the format code `%m` can be used to insert the error message returned by `strerror` for the current value of `errno`.[1] `%m` does not require any corresponding entry in the parameter list for `errmsg`. Note that the message string will be run through `gettext` for possible localization before format codes are processed.
- `errmsg_internal(const char *msg, ...)` is the same as `errmsg`, except that the message string will not be included in the internationalization message dictionary. This should be used for "can't happen" cases that are probably not worth expending translation effort on.
- `errdetail(const char *msg, ...)` supplies an optional "detail" message; this is to be used when there is additional information that seems inappropriate to put in the primary message. The message string is processed in just the same way as for `errmsg`.
- `errhint(const char *msg, ...)` supplies an optional "hint" message; this is to be used when offering suggestions about how to fix the problem, as opposed to factual details about what went wrong. The message string is processed in just the same way as for `errmsg`.
- `errcontext(const char *msg, ...)` is not normally called directly from an `ereport` message site; rather it is used in `error_context_stack` callback functions to provide information about the context in which an error occurred, such as the current location in a PL function. The message string is processed in just the same way as for `errmsg`. Unlike the other auxiliary functions, this can be called more than once per `ereport` call; the successive strings thus supplied are concatenated with separating newlines.
- `errposition(int cursorpos)` specifies the textual location of an error within a query string. Currently it is only useful for errors detected in the lexical and syntactic analysis phases of query processing.

1. That is, the value that was current when the `ereport` call was reached; changes of `errno` within the auxiliary reporting routines will not affect it. That would not be true if you were to write `strerror(errno)` explicitly in `errmsg`'s parameter list; accordingly, do not do so.

- `errcode_for_file_access()` is a convenience function that selects an appropriate SQLSTATE error identifier for a failure in a file-access-related system call. It uses the saved `errno` to determine which error code to generate. Usually this should be used in combination with `%m` in the primary error message text.
- `errcode_for_socket_access()` is a convenience function that selects an appropriate SQLSTATE error identifier for a failure in a socket-related system call.

There is an older function `elog` that is still heavily used. An `elog` call

```
elog(level, "format string", ...);
```

is exactly equivalent to

```
ereport(level, (errmsg_internal("format string", ...)));
```

Notice that the SQLSTATE error code is always defaulted, and the message string is not included in the internationalization message dictionary. Therefore, `elog` should be used only for internal errors and low-level debug logging. Any message that is likely to be of interest to ordinary users should go through `ereport`. Nonetheless, there are enough internal "can't happen" error checks in the system that `elog` is still widely used; it is preferred for those messages for its notational simplicity.

Advice about writing good error messages can be found in Section 45.3.

45.3. Error Message Style Guide

This style guide is offered in the hope of maintaining a consistent, user-friendly style throughout all the messages generated by PostgreSQL.

45.3.1. What goes where

The primary message should be short, factual, and avoid reference to implementation details such as specific function names. "Short" means "should fit on one line under normal conditions". Use a detail message if needed to keep the primary message short, or if you feel a need to mention implementation details such as the particular system call that failed. Both primary and detail messages should be factual. Use a hint message for suggestions about what to do to fix the problem, especially if the suggestion might not always be applicable.

For example, instead of

```
IpcMemoryCreate: shmget(key=%d, size=%u, 0%o) failed: %m
(plus a long addendum that is basically a hint)
```

write

```
Primary:    could not create shared memory segment: %m
Detail:     Failed syscall was shmget(key=%d, size=%u, 0%o).
Hint:       the addendum
```

Rationale: keeping the primary message short helps keep it to the point, and lets clients lay out screen space on the assumption that one line is enough for error messages. Detail and hint messages may be relegated to a verbose mode, or perhaps a pop-up error-details window. Also, details and hints would normally be suppressed from the server log to save space. Reference to implementation details is best avoided since users don't know the details anyway.

45.3.2. Formatting

Don't put any specific assumptions about formatting into the message texts. Expect clients and the server log to wrap lines to fit their own needs. In long messages, newline characters (\n) may be used to indicate suggested paragraph breaks. Don't end a message with a newline. Don't use tabs or other formatting characters. (In error context displays, newlines are automatically added to separate levels of context such as function calls.)

Rationale: Messages are not necessarily displayed on terminal-type displays. In GUI displays or browsers these formatting instructions are at best ignored.

45.3.3. Quotation marks

English text should use double quotes when quoting is appropriate. Text in other languages should consistently use one kind of quotes that is consistent with publishing customs and computer output of other programs.

Rationale: The choice of double quotes over single quotes is somewhat arbitrary, but tends to be the preferred use. Some have suggested choosing the kind of quotes depending on the type of object according to SQL conventions (namely, strings single quoted, identifiers double quoted). But this is a language-internal technical issue that many users aren't even familiar with, it won't scale to other kinds of quoted terms, it doesn't translate to other languages, and it's pretty pointless, too.

45.3.4. Use of quotes

Use quotes always to delimit file names, user-supplied identifiers, and other variables that might contain words. Do not use them to mark up variables that will not contain words (for example, operator names).

There are functions in the backend that will double-quote their own output at need (for example, `format_type_be`()). Do not put additional quotes around the output of such functions.

Rationale: Objects can have names that create ambiguity when embedded in a message. Be consistent about denoting where a plugged-in name starts and ends. But don't clutter messages with unnecessary or duplicate quote marks.

45.3.5. Grammar and punctuation

The rules are different for primary error messages and for detail/hint messages:

Primary error messages: Do not capitalize the first letter. Do not end a message with a period. Do not even think about ending a message with an exclamation point.

Detail and hint messages: Use complete sentences, and end each with a period. Capitalize the first word of sentences.

Rationale: Avoiding punctuation makes it easier for client applications to embed the message into a variety of grammatical contexts. Often, primary messages are not grammatically complete sentences anyway. (And if they're long enough to be more than one sentence, they should be split into primary and detail parts.) However, detail and hint messages are longer and may need to include multiple sentences. For consistency, they should follow complete-sentence style even when there's only one sentence.

45.3.6. Upper case vs. lower case

Use lower case for message wording, including the first letter of a primary error message. Use upper case for SQL commands and key words if they appear in the message.

Rationale: It's easier to make everything look more consistent this way, since some messages are complete sentences and some not.

45.3.7. Avoid passive voice

Use the active voice. Use complete sentences when there is an acting subject ("A could not do B"). Use telegram style without subject if the subject would be the program itself; do not use "I" for the program.

Rationale: The program is not human. Don't pretend otherwise.

45.3.8. Present vs past tense

Use past tense if an attempt to do something failed, but could perhaps succeed next time (perhaps after fixing some problem). Use present tense if the failure is certainly permanent.

There is a nontrivial semantic difference between sentences of the form

```
could not open file "%s": %m
```

and

```
cannot open file "%s"
```

The first one means that the attempt to open the file failed. The message should give a reason, such as "disk full" or "file doesn't exist". The past tense is appropriate because next time the disk might not be full anymore or the file in question may exist.

The second form indicates the the functionality of opening the named file does not exist at all in the program, or that it's conceptually impossible. The present tense is appropriate because the condition will persist indefinitely.

Rationale: Granted, the average user will not be able to draw great conclusions merely from the tense of the message, but since the language provides us with a grammar we should use it correctly.

45.3.9. Type of the object

When citing the name of an object, state what kind of object it is.

Rationale: Otherwise no one will know what “foo.bar.baz” refers to.

45.3.10. Brackets

Square brackets are only to be used (1) in command synopses to denote optional arguments, or (2) to denote an array subscript.

Rationale: Anything else does not correspond to widely-known customary usage and will confuse people.

45.3.11. Assembling error messages

When a message includes text that is generated elsewhere, embed it in this style:

```
could not open file %s: %m
```

Rationale: It would be difficult to account for all possible error codes to paste this into a single smooth sentence, so some sort of punctuation is needed. Putting the embedded text in parentheses has also been suggested, but it’s unnatural if the embedded text is likely to be the most important part of the message, as is often the case.

45.3.12. Reasons for errors

Messages should always state the reason why an error occurred. For example:

```
BAD:    could not open file %s
BETTER: could not open file %s (I/O failure)
```

If no reason is known you better fix the code.

45.3.13. Function names

Don’t include the name of the reporting routine in the error text. We have other mechanisms for finding that out when needed, and for most users it’s not helpful information. If the error text doesn’t make as much sense without the function name, reword it.

```
BAD:    pg_atoi: error in "z": can't parse "z"
BETTER: invalid input syntax for integer: "z"
```

Avoid mentioning called function names, either; instead say what the code was trying to do:

```
BAD:    open() failed: %m
```

```
BETTER: could not open file %s: %m
```

If it really seems necessary, mention the system call in the detail message. (In some cases, providing the actual values passed to the system call might be appropriate information for the detail message.)

Rationale: Users don't know what all those functions do.

45.3.14. Tricky words to avoid

Unable. "Unable" is nearly the passive voice. Better use "cannot" or "could not", as appropriate.

Bad. Error messages like "bad result" are really hard to interpret intelligently. It's better to write why the result is "bad", e.g., "invalid format".

Illegal. "Illegal" stands for a violation of the law, the rest is "invalid". Better yet, say why it's invalid.

Unknown. Try to avoid "unknown". Consider "error: unknown response". If you don't know what the response is, how do you know it's erroneous? "Unrecognized" is often a better choice. Also, be sure to include the value being complained of.

```
BAD:    unknown node type
BETTER: unrecognized node type: 42
```

Find vs. Exists. If the program uses a nontrivial algorithm to locate a resource (e.g., a path search) and that algorithm fails, it is fair to say that the program couldn't "find" the resource. If, on the other hand, the expected location of the resource is known but the program cannot access it there then say that the resource doesn't "exist". Using "find" in this case sounds weak and confuses the issue.

45.3.15. Proper spelling

Spell out words in full. For instance, avoid:

- spec
- stats
- parens
- auth
- xact

Rationale: This will improve consistency.

45.3.16. Localization

Keep in mind that error message texts need to be translated into other languages. Follow the guidelines in Section 46.2.2 to avoid making life difficult for translators.

Chapter 46. Native Language Support

46.1. For the Translator

PostgreSQL programs (server and client) can issue their messages in your favorite language — if the messages have been translated. Creating and maintaining translated message sets needs the help of people who speak their own language well and want to contribute to the PostgreSQL effort. You do not have to be a programmer at all to do this. This section explains how to help.

46.1.1. Requirements

We won't judge your language skills — this section is about software tools. Theoretically, you only need a text editor. But this is only in the unlikely event that you do not want to try out your translated messages. When you configure your source tree, be sure to use the `--enable-nls` option. This will also check for the libintl library and the `msgfmt` program, which all end users will need anyway. To try out your work, follow the applicable portions of the installation instructions.

If you want to start a new translation effort or want to do a message catalog merge (described later), you will need the programs `xgettext` and `msgmerge`, respectively, in a GNU-compatible implementation. Later, we will try to arrange it so that if you use a packaged source distribution, you won't need `xgettext`. (From CVS, you will still need it.) GNU Gettext 0.10.36 or later is currently recommended.

Your local gettext implementation should come with its own documentation. Some of that is probably duplicated in what follows, but for additional details you should look there.

46.1.2. Concepts

The pairs of original (English) messages and their (possibly) translated equivalents are kept in *message catalogs*, one for each program (although related programs can share a message catalog) and for each target language. There are two file formats for message catalogs: The first is the "PO" file (for Portable Object), which is a plain text file with special syntax that translators edit. The second is the "MO" file (for Machine Object), which is a binary file generated from the respective PO file and is used while the internationalized program is run. Translators do not deal with MO files; in fact hardly anyone does.

The extension of the message catalog file is to no surprise either `.po` or `.mo`. The base name is either the name of the program it accompanies, or the language the file is for, depending on the situation. This is a bit confusing. Examples are `psql.po` (PO file for psql) or `fr.mo` (MO file in French).

The file format of the PO files is illustrated here:

```
# comment

msgid "original string"
msgstr "translated string"
```

```
msgid "more original"
msgstr "another translated"
"string can be broken up like this"

...
```

The msgid's are extracted from the program source. (They need not be, but this is the most common way.) The msgstr lines are initially empty and are filled in with useful strings by the translator. The strings can contain C-style escape characters and can be continued across lines as illustrated. (The next line must start at the beginning of the line.)

The # character introduces a comment. If whitespace immediately follows the # character, then this is a comment maintained by the translator. There may also be automatic comments, which have a non-whitespace character immediately following the #. These are maintained by the various tools that operate on the PO files and are intended to aid the translator.

```
#. automatic comment
#: filename.c:1023
#, flags, flags
```

The #. style comments are extracted from the source file where the message is used. Possibly the programmer has inserted information for the translator, such as about expected alignment. The #: comment indicates the exact location(s) where the message is used in the source. The translator need not look at the program source, but he can if there is doubt about the correct translation. The #, comments contain flags that describe the message in some way. There are currently two flags: `fuzzy` is set if the message has possibly been outdated because of changes in the program source. The translator can then verify this and possibly remove the fuzzy flag. Note that fuzzy messages are not made available to the end user. The other flag is `c-format`, which indicates that the message is a `printf`-style format template. This means that the translation should also be a format string with the same number and type of placeholders. There are tools that can verify this, which key off the c-format flag.

46.1.3. Creating and maintaining message catalogs

OK, so how does one create a "blank" message catalog? First, go into the directory that contains the program whose messages you want to translate. If there is a file `nls.mk`, then this program has been prepared for translation.

If there are already some `.po` files, then someone has already done some translation work. The files are named *`language`*`.po`, where *`language`* is the ISO 639-1 two-letter language code (in lower case)[1], e.g., `fr.po` for French. If there is really a need for more than one translation effort per language then the files may also be named *`language_region`*`.po` where *`region`* is the ISO 3166-1 two-letter country code (in upper case)[2], e.g., `pt_BR.po` for Portuguese in Brazil. If you find the language you wanted you can just start working on that file.

If you need to start a new translation effort, then first run the command

```
gmake init-po
```

1. http://lcweb.loc.gov/standards/iso639-2/englangn.html
2. http://www.din.de/gremien/nas/nabd/iso3166ma/codlstp1/en_listp1.html

This will create a file *progname*.pot. (.pot to distinguish it from PO files that are "in production". The T stands for "template".) Copy this file to *language*.po and edit it. To make it known that the new language is available, also edit the file nls.mk and add the language (or language and country) code to the line that looks like:

```
AVAIL_LANGUAGES := de fr
```

(Other languages may appear, of course.)

As the underlying program or library changes, messages may be changed or added by the programmers. In this case you do not need to start from scratch. Instead, run the command

```
gmake update-po
```

which will create a new blank message catalog file (the pot file you started with) and will merge it with the existing PO files. If the merge algorithm is not sure about a particular message it marks it "fuzzy" as explained above. For the case where something went really wrong, the old PO file is saved with a .po.old extension.

46.1.4. Editing the PO files

The PO files can be edited with a regular text editor. The translator should only change the area between the quotes after the msgstr directive, may add comments and alter the fuzzy flag. There is (unsurprisingly) a PO mode for Emacs, which I find quite useful.

The PO files need not be completely filled in. The software will automatically fall back to the original string if no translation (or an empty translation) is available. It is no problem to submit incomplete translations for inclusions in the source tree; that gives room for other people to pick up your work. However, you are encouraged to give priority to removing fuzzy entries after doing a merge. Remember that fuzzy entries will not be installed; they only serve as reference what might be the right translation.

Here are some things to keep in mind while editing the translations:

- Make sure that if the original ends with a newline, the translation does, too. Similarly for tabs, etc.
- If the original is a printf format string, the translation also needs to be. The translation also needs to have the same format specifiers in the same order. Sometimes the natural rules of the language make this impossible or at least awkward. In that case you can modify the format specifiers like this:

  ```
  msgstr "Die Datei %2$s hat %1$u Zeichen."
  ```

  Then the first placeholder will actually use the second argument from the list. The *digits*$ needs to follow the % immediately, before any other format manipulators. (This feature really exists in the printf family of functions. You may not have heard of it before because there is little use for it outside of message internationalization.)
- If the original string contains a linguistic mistake, report that (or fix it yourself in the program source) and translate normally. The corrected string can be merged in when the program sources have been updated. If the original string contains a factual mistake, report that (or fix it yourself) and do not translate it. Instead, you may mark the string with a comment in the PO file.

- Maintain the style and tone of the original string. Specifically, messages that are not sentences (`cannot open file %s`) should probably not start with a capital letter (if your language distinguishes letter case) or end with a period (if your language uses punctuation marks). It may help to read Section 45.3.
- If you don't know what a message means, or if it is ambiguous, ask on the developers' mailing list. Chances are that English speaking end users might also not understand it or find it ambiguous, so it's best to improve the message.

46.2. For the Programmer

46.2.1. Mechanics

This section describes how to implement native language support in a program or library that is part of the PostgreSQL distribution. Currently, it only applies to C programs.

Adding NLS support to a program

1. Insert this code into the start-up sequence of the program:

   ```
   #ifdef ENABLE_NLS
   #include <locale.h>
   #endif

   ...

   #ifdef ENABLE_NLS
   setlocale(LC_ALL, "");
   bindtextdomain("progname", LOCALEDIR);
   textdomain("progname");
   #endif
   ```

   (The *progname* can actually be chosen freely.)

2. Wherever a message that is a candidate for translation is found, a call to `gettext()` needs to be inserted. E.g.,

   ```
   fprintf(stderr, "panic level %d\n", lvl);
   ```

   would be changed to

   ```
   fprintf(stderr, gettext("panic level %d\n"), lvl);
   ```

   (`gettext` is defined as a no-op if no NLS is configured.)

   This may tend to add a lot of clutter. One common shortcut is to use

   ```
   #define _(x) gettext(x)
   ```

   Another solution is feasible if the program does much of its communication through one or a few functions, such as `ereport()` in the backend. Then you make this function call `gettext` internally on all input strings.

3. Add a file `nls.mk` in the directory with the program sources. This file will be read as a makefile. The following variable assignments need to be made here:

`CATALOG_NAME`

The program name, as provided in the `textdomain()` call.

`AVAIL_LANGUAGES`

List of provided translations — initially empty.

`GETTEXT_FILES`

List of files that contain translatable strings, i.e., those marked with `gettext` or an alternative solution. Eventually, this will include nearly all source files of the program. If this list gets too long you can make the first "file" be a `+` and the second word be a file that contains one file name per line.

`GETTEXT_TRIGGERS`

The tools that generate message catalogs for the translators to work on need to know what function calls contain translatable strings. By default, only `gettext()` calls are known. If you used _ or other identifiers you need to list them here. If the translatable string is not the first argument, the item needs to be of the form `func:2` (for the second argument).

The build system will automatically take care of building and installing the message catalogs.

46.2.2. Message-writing guidelines

Here are some guidelines for writing messages that are easily translatable.

- Do not construct sentences at run-time, like

  ```
  printf("Files were %s.\n", flag ? "copied" : "removed");
  ```

  The word order within the sentence may be different in other languages. Also, even if you remember to call gettext() on each fragment, the fragments may not translate well separately. It's better to duplicate a little code so that each message to be translated is a coherent whole. Only numbers, file names, and such-like run-time variables should be inserted at run time into a message text.

- For similar reasons, this won't work:

  ```
  printf("copied %d file%s", n, n!=1 ? "s" : "");
  ```

  because it assumes how the plural is formed. If you figured you could solve it like this

  ```
  if (n==1)
      printf("copied 1 file");
  else
      printf("copied %d files", n):
  ```

  then be disappointed. Some languages have more than two forms, with some peculiar rules. We may have a solution for this in the future, but for now the matter is best avoided altogether. You could write:

  ```
  printf("number of copied files: %d", n);
  ```

- If you want to communicate something to the translator, such as about how a message is intended to line up with other output, precede the occurrence of the string with a comment that starts with `translator`, e.g.,

```
/* translator: This message is not what it seems to be. */
```

These comments are copied to the message catalog files so that the translators can see them.

Chapter 47. Writing A Procedural Language Handler

All calls to functions that are written in a language other than the current "version 1" interface for compiled languages (this includes functions in user-defined procedural languages, functions written in SQL, and functions using the version 0 compiled language interface), go through a *call handler* function for the specific language. It is the responsibility of the call handler to execute the function in a meaningful way, such as by interpreting the supplied source text. This chapter outlines how a new procedural language's call handler can be written.

The call handler for a procedural language is a "normal" function that must be written in a compiled language such as C, using the version-1 interface, and registered with PostgreSQL as taking no arguments and returning the type `language_handler`. This special pseudotype identifies the function as a call handler and prevents it from being called directly in SQL commands.

The call handler is called in the same way as any other function: It receives a pointer to a `FunctionCallInfoData struct` containing argument values and information about the called function, and it is expected to return a `Datum` result (and possibly set the `isnull` field of the `FunctionCallInfoData` structure, if it wishes to return an SQL null result). The difference between a call handler and an ordinary callee function is that the `flinfo->fn_oid` field of the `FunctionCallInfoData` structure will contain the OID of the actual function to be called, not of the call handler itself. The call handler must use this field to determine which function to execute. Also, the passed argument list has been set up according to the declaration of the target function, not of the call handler.

It's up to the call handler to fetch the entry of the function from the system table `pg_proc` and to analyze the argument and return types of the called function. The `AS` clause from the `CREATE FUNCTION` command for the function will be found in the `prosrc` column of the `pg_proc` row. This is commonly source text in the procedural language, but in theory it could be something else, such as a path name to a file, or anything else that tells the call handler what to do in detail.

Often, the same function is called many times per SQL statement. A call handler can avoid repeated lookups of information about the called function by using the `flinfo->fn_extra` field. This will initially be `NULL`, but can be set by the call handler to point at information about the called function. On subsequent calls, if `flinfo->fn_extra` is already non-`NULL` then it can be used and the information lookup step skipped. The call handler must make sure that `flinfo->fn_extra` is made to point at memory that will live at least until the end of the current query, since an `FmgrInfo` data structure could be kept that long. One way to do this is to allocate the extra data in the memory context specified by `flinfo->fn_mcxt`; such data will normally have the same lifespan as the `FmgrInfo` itself. But the handler could also choose to use a longer-lived memory context so that it can cache function definition information across queries.

When a procedural-language function is invoked as a trigger, no arguments are passed in the usual way, but the `FunctionCallInfoData`'s `context` field points at a `TriggerData` structure, rather than being `NULL` as it is in a plain function call. A language handler should provide mechanisms for procedural-language functions to get at the trigger information.

This is a template for a procedural-language handler written in C:

```
#include "postgres.h"
#include "executor/spi.h"
#include "commands/trigger.h"
#include "fmgr.h"
#include "access/heapam.h"
#include "utils/syscache.h"
#include "catalog/pg_proc.h"
#include "catalog/pg_type.h"

PG_FUNCTION_INFO_V1(plsample_call_handler);

Datum
plsample_call_handler(PG_FUNCTION_ARGS)
{
    Datum          retval;

    if (CALLED_AS_TRIGGER(fcinfo))
    {
        /*
         * Called as a trigger procedure
         */
        TriggerData    *trigdata = (TriggerData *) fcinfo->context;

        retval = ...
    }
    else
    {
        /*
         * Called as a function
         */

        retval = ...
    }

    return retval;
}
```

Only a few thousand lines of code have to be added instead of the dots to complete the call handler.

After having compiled the handler function into a loadable module (see Section 33.9.6), the following commands then register the sample procedural language:

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS 'filename'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

The procedural languages included in the standard distribution are good references when trying to write your own call handler. Look into the `src/pl` subdirectory of the source tree.

Chapter 48. Genetic Query Optimizer

Author: Written by Martin Utesch (<utesch@aut.tu-freiberg.de>) for the Institute of Automatic Control at the University of Mining and Technology in Freiberg, Germany.

48.1. Query Handling as a Complex Optimization Problem

Among all relational operators the most difficult one to process and optimize is the *join*. The number of possible query plans grows exponentially with the number of joins in the query. Further optimization effort is caused by the support of a variety of *join methods* (e.g., nested loop, hash join, merge join in PostgreSQL) to process individual joins and a diversity of *indexes* (e.g., B-tree, hash, GiST and GIN in PostgreSQL) as access paths for relations.

The normal PostgreSQL query optimizer performs a *near-exhaustive search* over the space of alternative strategies. This algorithm, first introduced in IBM's System R database, produces a near-optimal join order, but can take an enormous amount of time and memory space when the number of joins in the query grows large. This makes the ordinary PostgreSQL query optimizer inappropriate for queries that join a large number of tables.

The Institute of Automatic Control at the University of Mining and Technology, in Freiberg, Germany, encountered some problems when it wanted to use PostgreSQL as the backend for a decision support knowledge based system for the maintenance of an electrical power grid. The DBMS needed to handle large join queries for the inference machine of the knowledge based system. The number of joins in these queries made using the normal query optimizer infeasible.

In the following we describe the implementation of a *genetic algorithm* to solve the join ordering problem in a manner that is efficient for queries involving large numbers of joins.

48.2. Genetic Algorithms

The genetic algorithm (GA) is a heuristic optimization method which operates through nondeterministic, randomized search. The set of possible solutions for the optimization problem is considered as a *population* of *individuals*. The degree of adaptation of an individual to its environment is specified by its *fitness*.

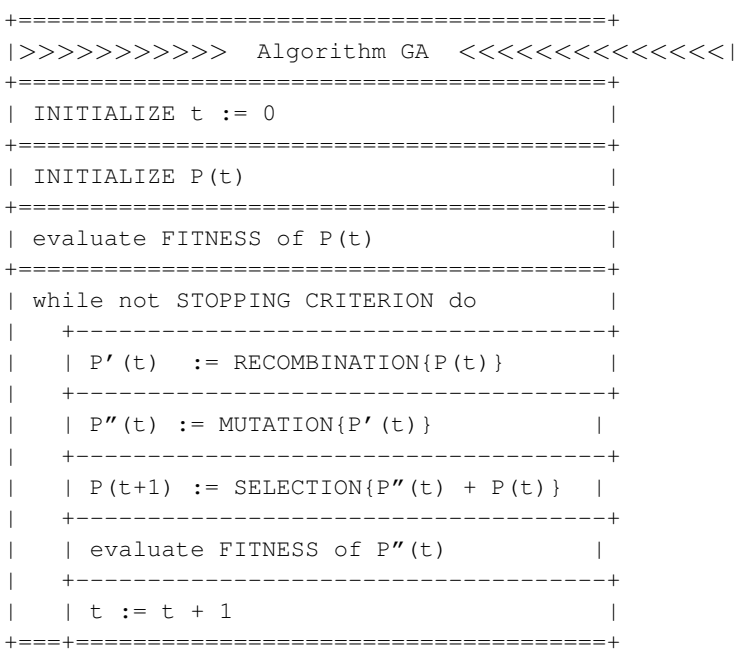
The coordinates of an individual in the search space are represented by *chromosomes*, in essence a set of character strings. A *gene* is a subsection of a chromosome which encodes the value of a single parameter being optimized. Typical encodings for a gene could be *binary* or *integer*.

Through simulation of the evolutionary operations *recombination*, *mutation*, and *selection* new generations of search points are found that show a higher average fitness than their ancestors.

According to the comp.ai.genetic FAQ it cannot be stressed too strongly that a GA is not a pure random search for a solution to a problem. A GA uses stochastic processes, but the result is distinctly non-random (better than random).

Figure 48-1. Structured Diagram of a Genetic Algorithm

P(t)	generation of ancestors at a time t
P”(t)	generation of descendants at a time t



48.3. Genetic Query Optimization (GEQO) in PostgreSQL

The GEQO module approaches the query optimization problem as though it were the well-known traveling salesman problem (TSP). Possible query plans are encoded as integer strings. Each string represents the join order from one relation of the query to the next. For example, the join tree

```
   /\
  /\ 2
 /\ 3
4  1
```

is encoded by the integer string ’4-1-3-2’, which means, first join relation ’4’ and ’1’, then ’3’, and then ’2’, where 1, 2, 3, 4 are relation IDs within the PostgreSQL optimizer.

Parts of the GEQO module are adapted from D. Whitley’s Genitor algorithm.

Specific characteristics of the GEQO implementation in PostgreSQL are:

- Usage of a *steady state* GA (replacement of the least fit individuals in a population, not whole-generational replacement) allows fast convergence towards improved query plans. This is essential for query handling with reasonable time;
- Usage of *edge recombination crossover* which is especially suited to keep edge losses low for the solution of the TSP by means of a GA;
- Mutation as genetic operator is deprecated so that no repair mechanisms are needed to generate legal TSP tours.

The GEQO module allows the PostgreSQL query optimizer to support large join queries effectively through non-exhaustive search.

48.3.1. Future Implementation Tasks for PostgreSQL GEQO

Work is still needed to improve the genetic algorithm parameter settings. In file `src/backend/optimizer/geqo/geqo_main.c`, routines `gimme_pool_size` and `gimme_number_generations`, we have to find a compromise for the parameter settings to satisfy two competing demands:

- Optimality of the query plan
- Computing time

At a more basic level, it is not clear that solving query optimization with a GA algorithm designed for TSP is appropriate. In the TSP case, the cost associated with any substring (partial tour) is independent of the rest of the tour, but this is certainly not true for query optimization. Thus it is questionable whether edge recombination crossover is the most effective mutation procedure.

48.4. Further Reading

The following resources contain additional information about genetic algorithms:

- The Hitch-Hiker's Guide to Evolutionary Computation[1], (FAQ for news://comp.ai.genetic)
- Evolutionary Computation and its application to art and design[2], by Craig Reynolds
- *Fundamentals of Database Systems*
- *The design and implementation of the POSTGRES query optimizer*

1. http://www.cs.bham.ac.uk/Mirrors/ftp.de.uu.net/EC/clife/www/location.htm
2. http://www.red3d.com/cwr/evolve.html

Chapter 49. Index Access Method Interface Definition

This chapter defines the interface between the core PostgreSQL system and *index access methods*, which manage individual index types. The core system knows nothing about indexes beyond what is specified here, so it is possible to develop entirely new index types by writing add-on code.

All indexes in PostgreSQL are what are known technically as *secondary indexes*; that is, the index is physically separate from the table file that it describes. Each index is stored as its own physical *relation* and so is described by an entry in the `pg_class` catalog. The contents of an index are entirely under the control of its index access method. In practice, all index access methods divide indexes into standard-size pages so that they can use the regular storage manager and buffer manager to access the index contents. (All the existing index access methods furthermore use the standard page layout described in Section 52.3, and they all use the same format for index tuple headers; but these decisions are not forced on an access method.)

An index is effectively a mapping from some data key values to *tuple identifiers*, or TIDs, of row versions (tuples) in the index's parent table. A TID consists of a block number and an item number within that block (see Section 52.3). This is sufficient information to fetch a particular row version from the table. Indexes are not directly aware that under MVCC, there may be multiple extant versions of the same logical row; to an index, each tuple is an independent object that needs its own index entry. Thus, an update of a row always creates all-new index entries for the row, even if the key values did not change. Index entries for dead tuples are reclaimed (by vacuuming) when the dead tuples themselves are reclaimed.

49.1. Catalog Entries for Indexes

Each index access method is described by a row in the `pg_am` system catalog (see Section 43.3). The principal contents of a `pg_am` row are references to `pg_proc` entries that identify the index access functions supplied by the access method. The APIs for these functions are defined later in this chapter. In addition, the `pg_am` row specifies a few fixed properties of the access method, such as whether it can support multicolumn indexes. There is not currently any special support for creating or deleting `pg_am` entries; anyone able to write a new access method is expected to be competent to insert an appropriate row for themselves.

To be useful, an index access method must also have one or more *operator classes* defined in `pg_opclass`, `pg_amop`, and `pg_amproc`. These entries allow the planner to determine what kinds of query qualifications can be used with indexes of this access method. Operator classes are described in Section 33.14, which is prerequisite material for reading this chapter.

An individual index is defined by a `pg_class` entry that describes it as a physical relation, plus a `pg_index` entry that shows the logical content of the index — that is, the set of index columns it has and the semantics of those columns, as captured by the associated operator classes. The index columns (key values) can be either simple columns of the underlying table or expressions over the table rows. The index access method normally has no interest in where the index key values come from (it is always handed precomputed key values) but it will be very interested in the operator class information in

pg_index. Both of these catalog entries can be accessed as part of the Relation data structure that is passed to all operations on the index.

Some of the flag columns of pg_am have nonobvious implications. The requirements of amcanunique are discussed in Section 49.5. The amcanmulticol flag asserts that the access method supports multicolumn indexes, while amoptionalkey asserts that it allows scans where no indexable restriction clause is given for the first index column. When amcanmulticol is false, amoptionalkey essentially says whether the access method allows full-index scans without any restriction clause. Access methods that support multiple index columns *must* support scans that omit restrictions on any or all of the columns after the first; however they are permitted to require some restriction to appear for the first index column, and this is signaled by setting amoptionalkey false. amindexnulls asserts that index entries are created for NULL key values. Since most indexable operators are strict and hence cannot return TRUE for NULL inputs, it is at first sight attractive to not store index entries for null values: they could never be returned by an index scan anyway. However, this argument fails when an index scan has no restriction clause for a given index column. In practice this means that indexes that have amoptionalkey true must index nulls, since the planner might decide to use such an index with no scan keys at all. A related restriction is that an index access method that supports multiple index columns *must* support indexing null values in columns after the first, because the planner will assume the index can be used for queries that do not restrict these columns. For example, consider an index on (a,b) and a query with WHERE a = 4. The system will assume the index can be used to scan for rows with a = 4, which is wrong if the index omits rows where b is null. It is, however, OK to omit rows where the first indexed column is null. Thus, amindexnulls should be set true only if the index access method indexes all rows, including arbitrary combinations of null values.

49.2. Index Access Method Functions

The index construction and maintenance functions that an index access method must provide are:

```
IndexBuildResult *
ambuild (Relation heapRelation,
         Relation indexRelation,
         IndexInfo *indexInfo);
```

Build a new index. The index relation has been physically created, but is empty. It must be filled in with whatever fixed data the access method requires, plus entries for all tuples already existing in the table. Ordinarily the ambuild function will call IndexBuildHeapScan() to scan the table for existing tuples and compute the keys that need to be inserted into the index. The function must return a palloc'd struct containing statistics about the new index.

```
bool
aminsert (Relation indexRelation,
          Datum *values,
          bool *isnull,
          ItemPointer heap_tid,
          Relation heapRelation,
          bool check_uniqueness);
```

Insert a new tuple into an existing index. The `values` and `isnull` arrays give the key values to be indexed, and `heap_tid` is the TID to be indexed. If the access method supports unique indexes (its `pg_am.amcanunique` flag is true) then `check_uniqueness` may be true, in which case the access method must verify that there is no conflicting row; this is the only situation in which the access method normally needs the `heapRelation` parameter. See Section 49.5 for details. The result is TRUE if an index entry was inserted, FALSE if not. (A FALSE result does not denote an error condition, but is used for cases such as an index AM refusing to index a NULL.)

```
IndexBulkDeleteResult *
ambulkdelete (IndexVacuumInfo *info,
              IndexBulkDeleteResult *stats,
              IndexBulkDeleteCallback callback,
              void *callback_state);
```

Delete tuple(s) from the index. This is a "bulk delete" operation that is intended to be implemented by scanning the whole index and checking each entry to see if it should be deleted. The passed-in `callback` function must be called, in the style `callback(`*`TID`*`, callback_state) returns bool`, to determine whether any particular index entry, as identified by its referenced TID, is to be deleted. Must return either NULL or a palloc'd struct containing statistics about the effects of the deletion operation. It is OK to return NULL if no information needs to be passed on to `amvacuumcleanup`.

Because of limited `maintenance_work_mem`, `ambulkdelete` may need to be called more than once when many tuples are to be deleted. The `stats` argument is the result of the previous call for this index (it is NULL for the first call within a `VACUUM` operation). This allows the AM to accumulate statistics across the whole operation. Typically, `ambulkdelete` will modify and return the same struct if the passed `stats` is not null.

```
IndexBulkDeleteResult *
amvacuumcleanup (IndexVacuumInfo *info,
                 IndexBulkDeleteResult *stats);
```

Clean up after a `VACUUM` operation (zero or more `ambulkdelete` calls). This does not have to do anything beyond returning index statistics, but it may perform bulk cleanup such as reclaiming empty index pages. `stats` is whatever the last `ambulkdelete` call returned, or NULL if `ambulkdelete` was not called because no tuples needed to be deleted. If the result is not NULL it must be a palloc'd struct. The statistics it contains will be used to update `pg_class`, and will be reported by `VACUUM` if `VERBOSE` is given. It is OK to return NULL if the index was not changed at all during the `VACUUM` operation, but otherwise correct stats should be returned.

```
void
amcostestimate (PlannerInfo *root,
                IndexOptInfo *index,
                List *indexQuals,
                RelOptInfo *outer_rel,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation);
```

Estimate the costs of an index scan. This function is described fully in Section 49.6, below.

```
bytea *
amoptions (ArrayType *reloptions,
           bool validate);
```

Parse and validate the reloptions array for an index. This is called only when a non-null reloptions array exists for the index. `reloptions` is a `text` array containing entries of the form *`name=value`*. The function should construct a `bytea` value, which will be copied into the `rd_options` field of the index's relcache entry. The data contents of the `bytea` value are open for the access method to define, but the standard access methods currently all use struct `StdRdOptions`. When `validate` is true, the function should report a suitable error message if any of the options are unrecognized or have invalid values; when `validate` is false, invalid entries should be silently ignored. (`validate` is false when loading options already stored in `pg_catalog`; an invalid entry could only be found if the access method has changed its rules for options, and in that case ignoring obsolete entries is appropriate.) It is OK to return NULL if default behavior is wanted.

The purpose of an index, of course, is to support scans for tuples matching an indexable `WHERE` condition, often called a *qualifier* or *scan key*. The semantics of index scanning are described more fully in Section 49.3, below. The scan-related functions that an index access method must provide are:

```
IndexScanDesc
ambeginscan (Relation indexRelation,
             int nkeys,
             ScanKey key);
```

Begin a new scan. The `key` array (of length `nkeys`) describes the scan key(s) for the index scan. The result must be a palloc'd struct. For implementation reasons the index access method *must* create this struct by calling `RelationGetIndexScan()`. In most cases `ambeginscan` itself does little beyond making that call; the interesting parts of index-scan startup are in `amrescan`.

```
boolean
amgettuple (IndexScanDesc scan,
            ScanDirection direction);
```

Fetch the next tuple in the given scan, moving in the given direction (forward or backward in the index). Returns TRUE if a tuple was obtained, FALSE if no matching tuples remain. In the TRUE case the tuple TID is stored into the `scan` structure. Note that "success" means only that the index contains an entry that matches the scan keys, not that the tuple necessarily still exists in the heap or will pass the caller's snapshot test.

```
boolean
amgetmulti (IndexScanDesc scan,
            ItemPointer tids,
            int32 max_tids,
            int32 *returned_tids);
```

Fetch multiple tuples in the given scan. Returns TRUE if the scan should continue, FALSE if no matching tuples remain. `tids` points to a caller-supplied array of `max_tids` `ItemPointerData` records, which the call fills with TIDs of matching tuples. `*returned_tids` is set to the number of TIDs actually returned. This can be less than `max_tids`, or even zero, even when the return value is TRUE. (This provision allows the access method to choose the most efficient stopping points in its scan, for example index

page boundaries.) amgetmulti and amgettuple cannot be used in the same index scan; there are other restrictions too when using amgetmulti, as explained in Section 49.3.

```
void
amrescan (IndexScanDesc scan,
          ScanKey key);
```

Restart the given scan, possibly with new scan keys (to continue using the old keys, NULL is passed for key). Note that it is not possible for the number of keys to be changed. In practice the restart feature is used when a new outer tuple is selected by a nested-loop join and so a new key comparison value is needed, but the scan key structure remains the same. This function is also called by RelationGetIndexScan(), so it is used for initial setup of an index scan as well as rescanning.

```
void
amendscan (IndexScanDesc scan);
```

End a scan and release resources. The scan struct itself should not be freed, but any locks or pins taken internally by the access method must be released.

```
void
ammarkpos (IndexScanDesc scan);
```

Mark current scan position. The access method need only support one remembered scan position per scan.

```
void
amrestrpos (IndexScanDesc scan);
```

Restore the scan to the most recently marked position.

By convention, the pg_proc entry for an index access method function should show the correct number of arguments, but declare them all as type internal (since most of the arguments have types that are not known to SQL, and we don't want users calling the functions directly anyway). The return type is declared as void, internal, or boolean as appropriate. The only exception is amoptions, which should be correctly declared as taking text[] and bool and returning bytea. This provision allows client code to execute amoptions to test validity of options settings.

49.3. Index Scanning

In an index scan, the index access method is responsible for regurgitating the TIDs of all the tuples it has been told about that match the *scan keys*. The access method is *not* involved in actually fetching those tuples from the index's parent table, nor in determining whether they pass the scan's time qualification test or other conditions.

A scan key is the internal representation of a WHERE clause of the form *index_key operator constant*, where the index key is one of the columns of the index and the operator is one of the members of the operator class associated with that index column. An index scan has zero or more scan keys, which are implicitly ANDed — the returned tuples are expected to satisfy all the indicated conditions.

The operator class may indicate that the index is *lossy* for a particular operator; this implies that the index scan will return all the entries that pass the scan key, plus possibly additional entries that do not. The core system's index-scan machinery will then apply that operator again to the heap tuple to verify whether or not it really should be selected. For non-lossy operators, the index scan must return exactly the set of matching entries, as there is no recheck.

Note that it is entirely up to the access method to ensure that it correctly finds all and only the entries passing all the given scan keys. Also, the core system will simply hand off all the `WHERE` clauses that match the index keys and operator classes, without any semantic analysis to determine whether they are redundant or contradictory. As an example, given `WHERE x > 4 AND x > 14` where `x` is a b-tree indexed column, it is left to the b-tree `amrescan` function to realize that the first scan key is redundant and can be discarded. The extent of preprocessing needed during `amrescan` will depend on the extent to which the index access method needs to reduce the scan keys to a "normalized" form.

The `amgettuple` function has a `direction` argument, which can be either `ForwardScanDirection` (the normal case) or `BackwardScanDirection`. If the first call after `amrescan` specifies `BackwardScanDirection`, then the set of matching index entries is to be scanned back-to-front rather than in the normal front-to-back direction, so `amgettuple` must return the last matching tuple in the index, rather than the first one as it normally would. (This will only occur for access methods that advertise they support ordered scans by setting `pg_am.amorderstrategy` nonzero.) After the first call, `amgettuple` must be prepared to advance the scan in either direction from the most recently returned entry.

The access method must support "marking" a position in a scan and later returning to the marked position. The same position may be restored multiple times. However, only one position need be remembered per scan; a new `ammarkpos` call overrides the previously marked position.

Both the scan position and the mark position (if any) must be maintained consistently in the face of concurrent insertions or deletions in the index. It is OK if a freshly-inserted entry is not returned by a scan that would have found the entry if it had existed when the scan started, or for the scan to return such an entry upon rescanning or backing up even though it had not been returned the first time through. Similarly, a concurrent delete may or may not be reflected in the results of a scan. What is important is that insertions or deletions not cause the scan to miss or multiply return entries that were not themselves being inserted or deleted.

Instead of using `amgettuple`, an index scan can be done with `amgetmulti` to fetch multiple tuples per call. This can be noticeably more efficient than `amgettuple` because it allows avoiding lock/unlock cycles within the access method. In principle `amgetmulti` should have the same effects as repeated `amgettuple` calls, but we impose several restrictions to simplify matters. In the first place, `amgetmulti` does not take a `direction` argument, and therefore it does not support backwards scan nor intrascan reversal of direction. The access method need not support marking or restoring scan positions during an `amgetmulti` scan, either. (These restrictions cost little since it would be difficult to use these features in an `amgetmulti` scan anyway: adjusting the caller's buffered list of TIDs would be complex.) Finally, `amgetmulti` does not guarantee any locking of the returned tuples, with implications spelled out in Section 49.4.

49.4. Index Locking Considerations

Index access methods must handle concurrent updates of the index by multiple processes. The core Post-

greSQL system obtains `AccessShareLock` on the index during an index scan, and `RowExclusiveLock` when updating the index (including plain `VACUUM`). Since these lock types do not conflict, the access method is responsible for handling any fine-grained locking it may need. An exclusive lock on the index as a whole will be taken only during index creation, destruction, `REINDEX`, or `VACUUM FULL`.

Building an index type that supports concurrent updates usually requires extensive and subtle analysis of the required behavior. For the b-tree and hash index types, you can read about the design decisions involved in `src/backend/access/nbtree/README` and `src/backend/access/hash/README`.

Aside from the index's own internal consistency requirements, concurrent updates create issues about consistency between the parent table (the *heap*) and the index. Because PostgreSQL separates accesses and updates of the heap from those of the index, there are windows in which the index may be inconsistent with the heap. We handle this problem with the following rules:

- A new heap entry is made before making its index entries. (Therefore a concurrent index scan is likely to fail to see the heap entry. This is okay because the index reader would be uninterested in an uncommitted row anyway. But see Section 49.5.)
- When a heap entry is to be deleted (by `VACUUM`), all its index entries must be removed first.
- An index scan must maintain a pin on the index page holding the item last returned by `amgettuple`, and `ambulkdelete` cannot delete entries from pages that are pinned by other backends. The need for this rule is explained below.

Without the third rule, it is possible for an index reader to see an index entry just before it is removed by `VACUUM`, and then to arrive at the corresponding heap entry after that was removed by `VACUUM`. This creates no serious problems if that item number is still unused when the reader reaches it, since an empty item slot will be ignored by `heap_fetch()`. But what if a third backend has already re-used the item slot for something else? When using an MVCC-compliant snapshot, there is no problem because the new occupant of the slot is certain to be too new to pass the snapshot test. However, with a non-MVCC-compliant snapshot (such as `SnapshotNow`), it would be possible to accept and return a row that does not in fact match the scan keys. We could defend against this scenario by requiring the scan keys to be rechecked against the heap row in all cases, but that is too expensive. Instead, we use a pin on an index page as a proxy to indicate that the reader may still be "in flight" from the index entry to the matching heap entry. Making `ambulkdelete` block on such a pin ensures that `VACUUM` cannot delete the heap entry before the reader is done with it. This solution costs little in run time, and adds blocking overhead only in the rare cases where there actually is a conflict.

This solution requires that index scans be "synchronous": we have to fetch each heap tuple immediately after scanning the corresponding index entry. This is expensive for a number of reasons. An "asynchronous" scan in which we collect many TIDs from the index, and only visit the heap tuples sometime later, requires much less index locking overhead and may allow a more efficient heap access pattern. Per the above analysis, we must use the synchronous approach for non-MVCC-compliant snapshots, but an asynchronous scan is workable for a query using an MVCC snapshot.

In an `amgetmulti` index scan, the access method need not guarantee to keep an index pin on any of the returned tuples. (It would be impractical to pin more than the last one anyway.) Therefore it is only safe to use such scans with MVCC-compliant snapshots.

49.5. Index Uniqueness Checks

PostgreSQL enforces SQL uniqueness constraints using *unique indexes*, which are indexes that disallow multiple entries with identical keys. An access method that supports this feature sets `pg_am.amcanunique` true. (At present, only b-tree supports it.)

Because of MVCC, it is always necessary to allow duplicate entries to exist physically in an index: the entries might refer to successive versions of a single logical row. The behavior we actually want to enforce is that no MVCC snapshot could include two rows with equal index keys. This breaks down into the following cases that must be checked when inserting a new row into a unique index:

- If a conflicting valid row has been deleted by the current transaction, it's okay. (In particular, since an UPDATE always deletes the old row version before inserting the new version, this will allow an UPDATE on a row without changing the key.)
- If a conflicting row has been inserted by an as-yet-uncommitted transaction, the would-be inserter must wait to see if that transaction commits. If it rolls back then there is no conflict. If it commits without deleting the conflicting row again, there is a uniqueness violation. (In practice we just wait for the other transaction to end and then redo the visibility check in toto.)
- Similarly, if a conflicting valid row has been deleted by an as-yet-uncommitted transaction, the would-be inserter must wait for that transaction to commit or abort, and then repeat the test.

Furthermore, immediately before raising a uniqueness violation according to the above rules, the access method must recheck the liveness of the row being inserted. If it is committed dead then no error should be raised. (This case cannot occur during the ordinary scenario of inserting a row that's just been created by the current transaction. It can happen during `CREATE UNIQUE INDEX CONCURRENTLY`, however.)

We require the index access method to apply these tests itself, which means that it must reach into the heap to check the commit status of any row that is shown to have a duplicate key according to the index contents. This is without a doubt ugly and non-modular, but it saves redundant work: if we did a separate probe then the index lookup for a conflicting row would be essentially repeated while finding the place to insert the new row's index entry. What's more, there is no obvious way to avoid race conditions unless the conflict check is an integral part of insertion of the new index entry.

The main limitation of this scheme is that it has no convenient way to support deferred uniqueness checks.

49.6. Index Cost Estimation Functions

The amcostestimate function is given a list of WHERE clauses that have been determined to be usable with the index. It must return estimates of the cost of accessing the index and the selectivity of the WHERE clauses (that is, the fraction of parent-table rows that will be retrieved during the index scan). For simple cases, nearly all the work of the cost estimator can be done by calling standard routines in the optimizer; the point of having an amcostestimate function is to allow index access methods to provide index-type-specific knowledge, in case it is possible to improve on the standard estimates.

Each amcostestimate function must have the signature:

```
void
amcostestimate (PlannerInfo *root,
```

```
IndexOptInfo *index,
List *indexQuals,
RelOptInfo *outer_rel,
Cost *indexStartupCost,
Cost *indexTotalCost,
Selectivity *indexSelectivity,
double *indexCorrelation);
```

The first four parameters are inputs:

root

The planner's information about the query being processed.

index

The index being considered.

indexQuals

List of index qual clauses (implicitly ANDed); a NIL list indicates no qualifiers are available. Note that the list contains expression trees, not ScanKeys.

outer_rel

If the index is being considered for use in a join inner indexscan, the planner's information about the outer side of the join. Otherwise NULL. When non-NULL, some of the qual clauses will be join clauses with this rel rather than being simple restriction clauses. Also, the cost estimator should expect that the index scan will be repeated for each row of the outer rel.

The last four parameters are pass-by-reference outputs:

*indexStartupCost

Set to cost of index start-up processing

*indexTotalCost

Set to total cost of index processing

*indexSelectivity

Set to index selectivity

*indexCorrelation

Set to correlation coefficient between index scan order and underlying table's order

Note that cost estimate functions must be written in C, not in SQL or any available procedural language, because they must access internal data structures of the planner/optimizer.

The index access costs should be computed using the parameters used by `src/backend/optimizer/path/costsize.c`: a sequential disk block fetch has cost `seq_page_cost`, a nonsequential fetch has cost `random_page_cost`, and the cost of processing one index row should usually be taken as `cpu_index_tuple_cost`. In addition, an appropriate multiple of

`cpu_operator_cost` should be charged for any comparison operators invoked during index processing (especially evaluation of the indexQuals themselves).

The access costs should include all disk and CPU costs associated with scanning the index itself, but *not* the costs of retrieving or processing the parent-table rows that are identified by the index.

The "start-up cost" is the part of the total scan cost that must be expended before we can begin to fetch the first row. For most indexes this can be taken as zero, but an index type with a high start-up cost might want to set it nonzero.

The indexSelectivity should be set to the estimated fraction of the parent table rows that will be retrieved during the index scan. In the case of a lossy index, this will typically be higher than the fraction of rows that actually pass the given qual conditions.

The indexCorrelation should be set to the correlation (ranging between -1.0 and 1.0) between the index order and the table order. This is used to adjust the estimate for the cost of fetching rows from the parent table.

In the join case, the returned numbers should be averages expected for any one scan of the index.

Cost Estimation

A typical cost estimator will proceed as follows:

1. Estimate and return the fraction of parent-table rows that will be visited based on the given qual conditions. In the absence of any index-type-specific knowledge, use the standard optimizer function `clauselist_selectivity()`:

   ```
   *indexSelectivity = clauselist_selectivity(root, indexQuals,
                                              index->rel->relid, JOIN_INNER);
   ```

2. Estimate the number of index rows that will be visited during the scan. For many index types this is the same as indexSelectivity times the number of rows in the index, but it might be more. (Note that the index's size in pages and rows is available from the IndexOptInfo struct.)

3. Estimate the number of index pages that will be retrieved during the scan. This might be just indexSelectivity times the index's size in pages.

4. Compute the index access cost. A generic estimator might do this:

   ```
   /*
    * Our generic assumption is that the index pages will be read
    * sequentially, so they cost seq_page_cost each, not random_page_cost.
    * Also, we charge for evaluation of the indexquals at each index row.
    * All the costs are assumed to be paid incrementally during the scan.
    */
   cost_qual_eval(&index_qual_cost, indexQuals);
   *indexStartupCost = index_qual_cost.startup;
   *indexTotalCost = seq_page_cost * numIndexPages +
       (cpu_index_tuple_cost + index_qual_cost.per_tuple) * numIndexTuples;
   ```

   However, the above does not account for amortization of index reads across repeated index scans in the join case.

5. Estimate the index correlation. For a simple ordered index on a single field, this can be retrieved from pg_statistic. If the correlation is not known, the conservative estimate is zero (no correlation).

Examples of cost estimator functions can be found in `src/backend/utils/adt/selfuncs.c`.

Chapter 50. GiST Indexes

50.1. Introduction

GiST stands for Generalized Search Tree. It is a balanced, tree-structured access method, that acts as a base template in which to implement arbitrary indexing schemes. B-trees, R-trees and many other indexing schemes can be implemented in GiST.

One advantage of GiST is that it allows the development of custom data types with the appropriate access methods, by an expert in the domain of the data type, rather than a database expert.

Some of the information here is derived from the University of California at Berkeley's GiST Indexing Project web site[1] and Marcel Kornacker's thesis, Access Methods for Next-Generation Database Systems[2]. The GiST implementation in PostgreSQL is primarily maintained by Teodor Sigaev and Oleg Bartunov, and there is more information on their website[3].

50.2. Extensibility

Traditionally, implementing a new index access method meant a lot of difficult work. It was necessary to understand the inner workings of the database, such as the lock manager and Write-Ahead Log. The GiST interface has a high level of abstraction, requiring the access method implementer to only implement the semantics of the data type being accessed. The GiST layer itself takes care of concurrency, logging and searching the tree structure.

This extensibility should not be confused with the extensibility of the other standard search trees in terms of the data they can handle. For example, PostgreSQL supports extensible B-trees and hash indexes. That means that you can use PostgreSQL to build a B-tree or hash over any data type you want. But B-trees only support range predicates (<, =, >), and hash indexes only support equality queries.

So if you index, say, an image collection with a PostgreSQL B-tree, you can only issue queries such as "is imagex equal to imagey", "is imagex less than imagey" and "is imagex greater than imagey"? Depending on how you define "equals", "less than" and "greater than" in this context, this could be useful. However, by using a GiST based index, you could create ways to ask domain-specific questions, perhaps "find all images of horses" or "find all over-exposed images".

All it takes to get a GiST access method up and running is to implement seven user-defined methods, which define the behavior of keys in the tree. Of course these methods have to be pretty fancy to support fancy queries, but for all the standard queries (B-trees, R-trees, etc.) they're relatively straightforward. In short, GiST combines extensibility along with generality, code reuse, and a clean interface.

1. http://gist.cs.berkeley.edu/
2. http://www.sai.msu.su/~megera/postgres/gist/papers/concurrency/access-methods-for-next-generation.pdf.gz
3. http://www.sai.msu.su/~megera/postgres/gist/

50.3. Implementation

There are seven methods that an index operator class for GiST must provide:

consistent

Given a predicate `p` on a tree page, and a user query, `q`, this method will return false if it is certain that both `p` and `q` cannot be true for a given data item.

union

This method consolidates information in the tree. Given a set of entries, this function generates a new predicate that is true for all the entries.

compress

Converts the data item into a format suitable for physical storage in an index page.

decompress

The reverse of the `compress` method. Converts the index representation of the data item into a format that can be manipulated by the database.

penalty

Returns a value indicating the "cost" of inserting the new entry into a particular branch of the tree. items will be inserted down the path of least `penalty` in the tree.

picksplit

When a page split is necessary, this function decides which entries on the page are to stay on the old page, and which are to move to the new page.

same

Returns true if two entries are identical, false otherwise.

50.4. Examples

The PostgreSQL source distribution includes several examples of index methods implemented using GiST. The core system currently provides R-Tree equivalent functionality for some of the built-in geometric data types (see `src/backend/access/gist/gistproc.c`). The following `contrib` modules also contain GiST operator classes:

btree_gist

B-Tree equivalent functionality for several data types

cube

Indexing for multidimensional cubes

intarray

RD-Tree for one-dimensional array of int4 values

ltree

Indexing for tree-like structures

pg_trgm

Text similarity using trigram matching

seg

Indexing for “float ranges”

tsearch2

Full text indexing

50.5. Crash Recovery

Usually, replay of the WAL log is sufficient to restore the integrity of a GiST index following a database crash. However, there are some corner cases in which the index state is not fully rebuilt. The index will still be functionally correct, but there may be some performance degradation. When this occurs, the index can be repaired by `VACUUM`ing its table, or by rebuilding the index using `REINDEX`. In some cases a plain `VACUUM` is not sufficient, and either `VACUUM FULL` or `REINDEX` is needed. The need for one of these procedures is indicated by occurrence of this log message during crash recovery:

```
LOG:  index NNN/NNN/NNN needs VACUUM or REINDEX to finish crash recovery
```

or this log message during routine index insertions:

```
LOG:  index "FOO" needs VACUUM or REINDEX to finish crash recovery
```

If a plain `VACUUM` finds itself unable to complete recovery fully, it will return a notice:

```
NOTICE:  index "FOO" needs VACUUM FULL or REINDEX to finish crash recovery
```

Chapter 51. GIN Indexes

51.1. Introduction

GIN stands for Generalized Inverted Index. It is an index structure storing a set of (key, posting list) pairs, where a "posting list" is a set of rows in which the key occurs. Each indexed value may contain many keys, so the same row ID may appear in multiple posting lists.

It is generalized in the sense that a GIN index does not need to be aware of the operation that it accelerates. Instead, it uses custom strategies defined for particular data types.

One advantage of GIN is that it allows the development of custom data types with the appropriate access methods, by an expert in the domain of the data type, rather than a database expert. This is much the same advantage as using GiST.

The GIN implementation in PostgreSQL is primarily maintained by Teodor Sigaev and Oleg Bartunov. There is more information about GIN on their website[1].

51.2. Extensibility

The GIN interface has a high level of abstraction, requiring the access method implementer only to implement the semantics of the data type being accessed. The GIN layer itself takes care of concurrency, logging and searching the tree structure.

All it takes to get a GIN access method working is to implement four user-defined methods, which define the behavior of keys in the tree and the relationships between keys, indexed values, and indexable queries. In short, GIN combines extensibility with generality, code reuse, and a clean interface.

The four methods that an index operator class for GIN must provide are:

int compare(Datum a, Datum b)

Compares keys (not indexed values!) and returns an integer less than zero, zero, or greater than zero, indicating whether the first key is less than, equal to, or greater than the second.

Datum* extractValue(Datum inputValue, uint32 *nkeys)

Returns an array of keys given a value to be indexed. The number of returned keys must be stored into `*nkeys`.

Datum* extractQuery(Datum query, uint32 *nkeys, StrategyNumber n)

Returns an array of keys given a value to be queried; that is, `query` is the value on the right-hand side of an indexable operator whose left-hand side is the indexed column. `n` is the strategy number of the operator within the operator class (see Section 33.14.2). Often, `extractQuery` will need to consult

1. http://www.sai.msu.su/~megera/oddmuse/index.cgi/Gin

`n` to determine the data type of `query` and the key values that need to be extracted. The number of returned keys must be stored into `*nkeys`.

bool consistent(bool check[], StrategyNumber n, Datum query)

Returns TRUE if the indexed value satisfies the query operator with strategy number `n` (or may satisfy, if the operator is marked RECHECK in the operator class). The `check` array has the same length as the number of keys previously returned by `extractQuery` for this query. Each element of the `check` array is TRUE if the indexed value contains the corresponding query key, ie, if (check[i] == TRUE) the i-th key of the `extractQuery` result array is present in the indexed value. The original `query` datum (not the extracted key array!) is passed in case the `consistent` method needs to consult it.

51.3. Implementation

Internally, a GIN index contains a B-tree index constructed over keys, where each key is an element of the indexed value (a member of an array, for example) and where each tuple in a leaf page is either a pointer to a B-tree over heap pointers (PT, posting tree), or a list of heap pointers (PL, posting list) if the list is small enough.

51.4. GIN tips and tricks

Create vs insert

In most cases, insertion into a GIN index is slow due to the likelihood of many keys being inserted for each value. So, for bulk insertions into a table it is advisable to drop the GIN index and recreate it after finishing bulk insertion.

gin_fuzzy_search_limit

The primary goal of developing GIN indexes was to create support for highly scalable, full-text search in PostgreSQL, and there are often situations when a full-text search returns a very large set of results. Moreover, this often happens when the query contains very frequent words, so that the large result set is not even useful. Since reading many tuples from the disk and sorting them could take a lot of time, this is unacceptable for production. (Note that the index search itself is very fast.)

To facilitate controlled execution of such queries GIN has a configurable soft upper limit on the size of the returned set, the `gin_fuzzy_search_limit` configuration parameter. It is set to 0 (meaning no limit) by default. If a non-zero limit is set, then the returned set is a subset of the whole result set, chosen at random.

“Soft” means that the actual number of returned results could differ slightly from the specified limit, depending on the query and the quality of the system’s random number generator.

51.5. Limitations

GIN doesn't support full index scans: because there are often many keys per value, each heap pointer would be returned many times, and there is no easy way to prevent this.

When `extractQuery` returns zero keys, GIN will emit an error. Depending on the operator, a void query might match all, some, or none of the indexed values (for example, every array contains the empty array, but does not overlap the empty array), and GIN can't determine the correct answer, nor produce a full-index-scan result if it could determine that that was correct.

It is not an error for `extractValue` to return zero keys, but in this case the indexed value will be unrepresented in the index. This is another reason why full index scan is not useful — it would miss such rows.

GIN searches keys only by equality matching. This may be improved in future.

51.6. Examples

The PostgreSQL source distribution includes GIN classes for one-dimensional arrays of all internal types. The following `contrib` modules also contain GIN operator classes:

intarray

Enhanced support for int4[]

tsearch2

Support for inverted text indexing. This is much faster for very large, mostly-static sets of documents.

Chapter 52. Database Physical Storage

This chapter provides an overview of the physical storage format used by PostgreSQL databases.

52.1. Database File Layout

This section describes the storage format at the level of files and directories.

All the data needed for a database cluster is stored within the cluster's data directory, commonly referred to as `PGDATA` (after the name of the environment variable that can be used to define it). A common location for `PGDATA` is `/var/lib/pgsql/data`. Multiple clusters, managed by different server instances, can exist on the same machine.

The `PGDATA` directory contains several subdirectories and control files, as shown in Table 52-1. In addition to these required items, the cluster configuration files `postgresql.conf`, `pg_hba.conf`, and `pg_ident.conf` are traditionally stored in `PGDATA` (although in PostgreSQL 8.0 and later, it is possible to keep them elsewhere).

Table 52-1. Contents of `PGDATA`

Item	Description
`PG_VERSION`	A file containing the major version number of PostgreSQL
`base`	Subdirectory containing per-database subdirectories
`global`	Subdirectory containing cluster-wide tables, such as `pg_database`
`pg_clog`	Subdirectory containing transaction commit status data
`pg_multixact`	Subdirectory containing multitransaction status data (used for shared row locks)
`pg_subtrans`	Subdirectory containing subtransaction status data
`pg_tblspc`	Subdirectory containing symbolic links to tablespaces
`pg_twophase`	Subdirectory containing state files for prepared transactions
`pg_xlog`	Subdirectory containing WAL (Write Ahead Log) files
`postmaster.opts`	A file recording the command-line options the server was last started with

Item	Description
postmaster.pid	A lock file recording the current server PID and shared memory segment ID (not present after server shutdown)

For each database in the cluster there is a subdirectory within PGDATA/base, named after the database's OID in pg_database. This subdirectory is the default location for the database's files; in particular, its system catalogs are stored there.

Each table and index is stored in a separate file, named after the table or index's *filenode* number, which can be found in pg_class.relfilenode.

Caution

Note that while a table's filenode often matches its OID, this is *not* necessarily the case; some operations, like TRUNCATE, REINDEX, CLUSTER and some forms of ALTER TABLE, can change the filenode while preserving the OID. Avoid assuming that filenode and table OID are the same.

When a table or index exceeds 1 GB, it is divided into gigabyte-sized *segments*. The first segment's file name is the same as the filenode; subsequent segments are named filenode.1, filenode.2, etc. This arrangement avoids problems on platforms that have file size limitations. The contents of tables and indexes are discussed further in Section 52.3.

A table that has columns with potentially large entries will have an associated *TOAST* table, which is used for out-of-line storage of field values that are too large to keep in the table rows proper. pg_class.reltoastrelid links from a table to its TOAST table, if any. See Section 52.2 for more information.

Tablespaces make the scenario more complicated. Each user-defined tablespace has a symbolic link inside the PGDATA/pg_tblspc directory, which points to the physical tablespace directory (as specified in its CREATE TABLESPACE command). The symbolic link is named after the tablespace's OID. Inside the physical tablespace directory there is a subdirectory for each database that has elements in the tablespace, named after the database's OID. Tables within that directory follow the filenode naming scheme. The pg_default tablespace is not accessed through pg_tblspc, but corresponds to PGDATA/base. Similarly, the pg_global tablespace is not accessed through pg_tblspc, but corresponds to PGDATA/global.

52.2. TOAST

This section provides an overview of TOAST (The Oversized-Attribute Storage Technique).

PostgreSQL uses a fixed page size (commonly 8 kB), and does not allow tuples to span multiple pages. Therefore, it is not possible to store very large field values directly. To overcome this limitation, large field values are compressed and/or broken up into multiple physical rows. This happens transparently to the user, with only small impact on most of the backend code. The technique is affectionately known as TOAST (or "the best thing since sliced bread").

Only certain data types support TOAST — there is no need to impose the overhead on data types that cannot produce large field values. To support TOAST, a data type must have a variable-length (*varlena*)

representation, in which the first 32-bit word of any stored value contains the total length of the value in bytes (including itself). TOAST does not constrain the rest of the representation. All the C-level functions supporting a TOAST-able data type must be careful to handle TOASTed input values. (This is normally done by invoking PG_DETOAST_DATUM before doing anything with an input value, but in some cases more efficient approaches are possible.)

TOAST usurps the high-order two bits of the varlena length word, thereby limiting the logical size of any value of a TOAST-able data type to 1 GB (2^{30} - 1 bytes). When both bits are zero, the value is an ordinary un-TOASTed value of the data type. One of these bits, if set, indicates that the value has been compressed and must be decompressed before use. The other bit, if set, indicates that the value has been stored out-of-line. In this case the remainder of the value is actually just a pointer, and the correct data has to be found elsewhere. When both bits are set, the out-of-line data has been compressed too. In each case the length in the low-order bits of the varlena word indicates the actual size of the datum, not the size of the logical value that would be extracted by decompression or fetching of the out-of-line data.

If any of the columns of a table are TOAST-able, the table will have an associated TOAST table, whose OID is stored in the table's pg_class.reltoastrelid entry. Out-of-line TOASTed values are kept in the TOAST table, as described in more detail below.

The compression technique used is a fairly simple and very fast member of the LZ family of compression techniques. See src/backend/utils/adt/pg_lzcompress.c for the details.

Out-of-line values are divided (after compression if used) into chunks of at most TOAST_MAX_CHUNK_SIZE bytes (this value is a little less than BLCKSZ/4, or about 2000 bytes by default). Each chunk is stored as a separate row in the TOAST table for the owning table. Every TOAST table has the columns chunk_id (an OID identifying the particular TOASTed value), chunk_seq (a sequence number for the chunk within its value), and chunk_data (the actual data of the chunk). A unique index on chunk_id and chunk_seq provides fast retrieval of the values. A pointer datum representing an out-of-line TOASTed value therefore needs to store the OID of the TOAST table in which to look and the OID of the specific value (its chunk_id). For convenience, pointer datums also store the logical datum size (original uncompressed data length) and actual stored size (different if compression was applied). Allowing for the varlena header word, the total size of a TOAST pointer datum is therefore 20 bytes regardless of the actual size of the represented value.

The TOAST code is triggered only when a row value to be stored in a table is wider than BLCKSZ/4 bytes (normally 2 kB). The TOAST code will compress and/or move field values out-of-line until the row value is shorter than BLCKSZ/4 bytes or no more gains can be had. During an UPDATE operation, values of unchanged fields are normally preserved as-is; so an UPDATE of a row with out-of-line values incurs no TOAST costs if none of the out-of-line values change.

The TOAST code recognizes four different strategies for storing TOAST-able columns:

- PLAIN prevents either compression or out-of-line storage. This is the only possible strategy for columns of non-TOAST-able data types.
- EXTENDED allows both compression and out-of-line storage. This is the default for most TOAST-able data types. Compression will be attempted first, then out-of-line storage if the row is still too big.
- EXTERNAL allows out-of-line storage but not compression. Use of EXTERNAL will make substring operations on wide text and bytea columns faster (at the penalty of increased storage space) because these operations are optimized to fetch only the required parts of the out-of-line value when it is not compressed.

- `MAIN` allows compression but not out-of-line storage. (Actually, out-of-line storage will still be performed for such columns, but only as a last resort when there is no other way to make the row small enough.)

Each TOAST-able data type specifies a default strategy for columns of that data type, but the strategy for a given table column can be altered with `ALTER TABLE SET STORAGE`.

This scheme has a number of advantages compared to a more straightforward approach such as allowing row values to span pages. Assuming that queries are usually qualified by comparisons against relatively small key values, most of the work of the executor will be done using the main row entry. The big values of TOASTed attributes will only be pulled out (if selected at all) at the time the result set is sent to the client. Thus, the main table is much smaller and more of its rows fit in the shared buffer cache than would be the case without any out-of-line storage. Sort sets shrink also, and sorts will more often be done entirely in memory. A little test showed that a table containing typical HTML pages and their URLs was stored in about half of the raw data size including the TOAST table, and that the main table contained only about 10% of the entire data (the URLs and some small HTML pages). There was no run time difference compared to an un-TOASTed comparison table, in which all the HTML pages were cut down to 7 kB to fit.

52.3. Database Page Layout

This section provides an overview of the page format used within PostgreSQL tables and indexes.[1] Sequences and TOAST tables are formatted just like a regular table.

In the following explanation, a *byte* is assumed to contain 8 bits. In addition, the term *item* refers to an individual data value that is stored on a page. In a table, an item is a row; in an index, an item is an index entry.

Every table and index is stored as an array of *pages* of a fixed size (usually 8 kB, although a different page size can be selected when compiling the server). In a table, all the pages are logically equivalent, so a particular item (row) can be stored in any page. In indexes, the first page is generally reserved as a *metapage* holding control information, and there may be different types of pages within the index, depending on the index access method.

Table 52-2 shows the overall layout of a page. There are five parts to each page.

Table 52-2. Overall Page Layout

Item	Description
PageHeaderData	20 bytes long. Contains general information about the page, including free space pointers.
ItemIdData	Array of (offset,length) pairs pointing to the actual items. 4 bytes per item.
Free space	The unallocated space. New item pointers are allocated from the start of this area, new items from the end.

1. Actually, index access methods need not use this page format. All the existing index methods do use this basic format, but the data kept on index metapages usually doesn't follow the item layout rules.

Item	Description
Items	The actual items themselves.
Special space	Index access method specific data. Different methods store different data. Empty in ordinary tables.

The first 20 bytes of each page consists of a page header (PageHeaderData). Its format is detailed in Table 52-3. The first two fields track the most recent WAL entry related to this page. They are followed by three 2-byte integer fields (`pd_lower`, `pd_upper`, and `pd_special`). These contain byte offsets from the page start to the start of unallocated space, to the end of unallocated space, and to the start of the special space. The last 2 bytes of the page header, `pd_pagesize_version`, store both the page size and a version indicator. Beginning with PostgreSQL 8.1 the version number is 3; PostgreSQL 8.0 used version number 2; PostgreSQL 7.3 and 7.4 used version number 1; prior releases used version number 0. (The basic page layout and header format has not changed in these versions, but the layout of heap row headers has.) The page size is basically only present as a cross-check; there is no support for having more than one page size in an installation.

Table 52-3. PageHeaderData Layout

Field	Type	Length	Description
pd_lsn	XLogRecPtr	8 bytes	LSN: next byte after last byte of xlog record for last change to this page
pd_tli	TimeLineID	4 bytes	TLI of last change
pd_lower	LocationIndex	2 bytes	Offset to start of free space
pd_upper	LocationIndex	2 bytes	Offset to end of free space
pd_special	LocationIndex	2 bytes	Offset to start of special space
pd_pagesize_version	uint16	2 bytes	Page size and layout version number information

All the details may be found in `src/include/storage/bufpage.h`.

Following the page header are item identifiers (`ItemIdData`), each requiring four bytes. An item identifier contains a byte-offset to the start of an item, its length in bytes, and a few attribute bits which affect its interpretation. New item identifiers are allocated as needed from the beginning of the unallocated space. The number of item identifiers present can be determined by looking at `pd_lower`, which is increased to allocate a new identifier. Because an item identifier is never moved until it is freed, its index may be used on a long-term basis to reference an item, even when the item itself is moved around on the page to compact free space. In fact, every pointer to an item (`ItemPointer`, also known as `CTID`) created by PostgreSQL consists of a page number and the index of an item identifier.

The items themselves are stored in space allocated backwards from the end of unallocated space. The exact structure varies depending on what the table is to contain. Tables and sequences both use a structure named `HeapTupleHeaderData`, described below.

The final section is the "special section" which may contain anything the access method wishes to store. For example, b-tree indexes store links to the page's left and right siblings, as well as some other data relevant to the index structure. Ordinary tables do not use a special section at all (indicated by setting `pd_special` to equal the page size).

All table rows are structured in the same way. There is a fixed-size header (occupying 27 bytes on most machines), followed by an optional null bitmap, an optional object ID field, and the user data. The header is detailed in Table 52-4. The actual user data (columns of the row) begins at the offset indicated by `t_hoff`, which must always be a multiple of the MAXALIGN distance for the platform. The null bitmap is only present if the *HEAP_HASNULL* bit is set in `t_infomask`. If it is present it begins just after the fixed header and occupies enough bytes to have one bit per data column (that is, `t_natts` bits altogether). In this list of bits, a 1 bit indicates not-null, a 0 bit is a null. When the bitmap is not present, all columns are assumed not-null. The object ID is only present if the *HEAP_HASOID* bit is set in `t_infomask`. If present, it appears just before the `t_hoff` boundary. Any padding needed to make `t_hoff` a MAXALIGN multiple will appear between the null bitmap and the object ID. (This in turn ensures that the object ID is suitably aligned.)

Table 52-4. HeapTupleHeaderData Layout

Field	Type	Length	Description
t_xmin	TransactionId	4 bytes	insert XID stamp
t_cmin	CommandId	4 bytes	insert CID stamp
t_xmax	TransactionId	4 bytes	delete XID stamp
t_cmax	CommandId	4 bytes	delete CID stamp (overlays with t_xvac)
t_xvac	TransactionId	4 bytes	XID for VACUUM operation moving a row version
t_ctid	ItemPointerData	6 bytes	current TID of this or newer row version
t_natts	int16	2 bytes	number of attributes
t_infomask	uint16	2 bytes	various flag bits
t_hoff	uint8	1 byte	offset to user data

All the details may be found in `src/include/access/htup.h`.

Interpreting the actual data can only be done with information obtained from other tables, mostly `pg_attribute`. The key values needed to identify field locations are `attlen` and `attalign`. There is no way to directly get a particular attribute, except when there are only fixed width fields and no null values. All this trickery is wrapped up in the functions *heap_getattr*, *fastgetattr* and *heap_getsysattr*.

To read the data you need to examine each attribute in turn. First check whether the field is NULL according to the null bitmap. If it is, go to the next. Then make sure you have the right alignment. If the field is a fixed width field, then all the bytes are simply placed. If it's a variable length field (attlen = -1) then it's a bit more complicated. All variable-length datatypes share the common header structure `varattrib`, which includes the total length of the stored value and some flag bits. Depending on the flags, the data may be either inline or in a TOAST table; it might be compressed, too (see Section 52.2).

Chapter 53. BKI Backend Interface

Backend Interface (BKI) files are scripts in a special language that is understood by the PostgreSQL backend when running in the "bootstrap" mode. The bootstrap mode allows system catalogs to be created and filled from scratch, whereas ordinary SQL commands require the catalogs to exist already. BKI files can therefore be used to create the database system in the first place. (And they are probably not useful for anything else.)

initdb uses a BKI file to do part of its job when creating a new database cluster. The input file used by initdb is created as part of building and installing PostgreSQL by a program named `genbki.sh`, which reads some specially formatted C header files in the `src/include/catalog/` directory of the source tree. The created BKI file is called `postgres.bki` and is normally installed in the `share` subdirectory of the installation tree.

Related information may be found in the documentation for initdb.

53.1. BKI File Format

This section describes how the PostgreSQL backend interprets BKI files. This description will be easier to understand if the `postgres.bki` file is at hand as an example.

BKI input consists of a sequence of commands. Commands are made up of a number of tokens, depending on the syntax of the command. Tokens are usually separated by whitespace, but need not be if there is no ambiguity. There is no special command separator; the next token that syntactically cannot belong to the preceding command starts a new one. (Usually you would put a new command on a new line, for clarity.) Tokens can be certain key words, special characters (parentheses, commas, etc.), numbers, or double-quoted strings. Everything is case sensitive.

Lines starting with `#` are ignored.

53.2. BKI Commands

`create` [`bootstrap`] [`shared_relation`] [`without_oids`] *`tablename`* *`tableoid`* (*`name1`* = *`type1`* [, *`name2`* = *`type2`*, ...])

Create a table named *`tablename`*, and having the OID *`tableoid`*, with the columns given in parentheses.

The following column types are supported directly by `bootstrap.c`: `bool`, `bytea`, `char` (1 byte), `name`, `int2`, `int4`, `regproc`, `regclass`, `regtype`, `text`, `oid`, `tid`, `xid`, `cid`, `int2vector`, `oidvector`, `_int4` (array), `_text` (array), `_oid` (array), `_char` (array), `_aclitem` (array). Although it is possible to create tables containing columns of other types, this cannot be done until after `pg_type` has been created and filled with appropriate entries. (That effectively means that only

these column types can be used in bootstrapped tables, but non-bootstrap catalogs can contain any built-in type.)

When `bootstrap` is specified, the table will only be created on disk; nothing is entered into `pg_class`, `pg_attribute`, etc, for it. Thus the table will not be accessible by ordinary SQL operations until such entries are made the hard way (with `insert` commands). This option is used for creating `pg_class` etc themselves.

The table is created as shared if `shared_relation` is specified. It will have OIDs unless `without_oids` is specified.

`open` *`tablename`*

Open the table named *`tablename`* for insertion of data. Any currently open table is closed.

`close` [*`tablename`*]

Close the open table. The name of the table can be given as a cross-check, but this is not required.

`insert` [`OID =` *`oid_value`*] `(` *`value1 value2`* ... `)`

Insert a new row into the open table using *`value1`*, *`value2`*, etc., for its column values and *`oid_value`* for its OID. If *`oid_value`* is zero (0) or the clause is omitted, and the table has OIDs, then the next available OID is assigned.

NULL values can be specified using the special key word `_null_`. Values containing spaces must be double quoted.

`declare` [`unique`] `index` *`indexname indexoid`* `on` *`tablename`* `using` *`amname`* `(` *`opclass1 name1`* [, ...] `)`

Create an index named *`indexname`*, having OID *`indexoid`*, on the table named *`tablename`*, using the *`amname`* access method. The fields to index are called *`name1`*, *`name2`* etc., and the operator classes to use are *`opclass1`*, *`opclass2`* etc., respectively. The index file is created and appropriate catalog entries are made for it, but the index contents are not initialized by this command.

`declare toast` *`toasttableoid toastindexoid`* `on` *`tablename`*

Create a TOAST table for the table named *`tablename`*. The TOAST table is assigned OID *`toasttableoid`* and its index is assigned OID *`toastindexoid`*. As with `declare index`, filling of the index is postponed.

`build indices`

Fill in the indices that have previously been declared.

53.3. Structure of the Bootstrap BKI File

The `open` command cannot be used until the tables it uses exist and have entries for the table that is to be opened. (These minimum tables are `pg_class`, `pg_attribute`, `pg_proc`, and `pg_type`.) To allow those tables themselves to be filled, `create` with the `bootstrap` option implicitly opens the created table for data insertion.

Also, the `declare index` and `declare toast` commands cannot be used until the system catalogs they need have been created and filled in.

Thus, the structure of the `postgres.bki` file has to be:

1. `create bootstrap` one of the critical tables
2. `insert` data describing at least the critical tables
3. `close`
4. Repeat for the other critical tables.
5. `create` (without `bootstrap`) a noncritical table
6. `open`
7. `insert` desired data
8. `close`
9. Repeat for the other noncritical tables.
10. Define indexes and toast tables.
11. `build indices`

There are doubtless other, undocumented ordering dependencies.

53.4. Example

The following sequence of commands will create the table `test_table` with OID 420, having two columns `cola` and `colb` of type `int4` and `text`, respectively, and insert two rows into the table.

```
create test_table 420 (cola = int4, colb = text)
open test_table
insert OID=421 ( 1 "value1" )
insert OID=422 ( 2 _null_ )
close test_table
```

Chapter 54. How the Planner Uses Statistics

This chapter builds on the material covered in Section 13.1 and Section 13.2, and shows how the planner uses the system statistics to estimate the number of rows each stage in a query might return. This is a significant part of the planning / optimizing process, providing much of the raw material for cost calculation.

The intent of this chapter is not to document the code — better done in the code itself, but to present an overview of how it works. This will perhaps ease the learning curve for someone who subsequently wishes to read the code. As a consequence, the approach chosen is to analyze a series of incrementally more complex examples.

The outputs and algorithms shown below are taken from version 8.0. The behavior of earlier (or later) versions may vary.

54.1. Row Estimation Examples

Using examples drawn from the regression test database, let's start with a very simple query:

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..445.00 rows=10000 width=244)
```

How the planner determines the cardinality of `tenk1` is covered in Section 13.1, but is repeated here for completeness. The number of rows is looked up from `pg_class`:

```
SELECT reltuples, relpages FROM pg_class WHERE relname = 'tenk1';

 relpages | reltuples
----------+-----------
      345 |     10000
```

The planner will check the `relpages` estimate (this is a cheap operation) and if incorrect may scale `reltuples` to obtain a row estimate. In this case it does not, thus:

```
rows = 10000
```

let's move on to an example with a range condition in its `WHERE` clause:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..470.00 rows=1031 width=244)
```

```
   Filter: (unique1 < 1000)
```

The planner examines the WHERE clause condition:

```
unique1 < 1000
```

and looks up the restriction function for the operator < in pg_operator. This is held in the column oprrest, and the result in this case is scalarltsel. The scalarltsel function retrieves the histogram for unique1 from pg_statistics - we can follow this by using the simpler pg_stats view:

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='unique1';

                    histogram_bounds
------------------------------------------------------
 {1,970,1943,2958,3971,5069,6028,7007,7919,8982,9995}
```

Next the fraction of the histogram occupied by "< 1000" is worked out. This is the selectivity. The histogram divides the range into equal frequency buckets, so all we have to do is locate the bucket that our value is in and count *part* of it and *all* of the ones before. The value 1000 is clearly in the second (970 - 1943) bucket, so by assuming a linear distribution of values inside each bucket we can calculate the selectivity as:

```
selectivity = (1 + (1000 - bckt[2].min)/(bckt[2].max - bckt[2].min))/num_bckts
            = (1 + (1000 - 970)/(1943 - 970))/10
            = 0.1031
```

that is, one whole bucket plus a linear fraction of the second, divided by the number of buckets. The estimated number of rows can now be calculated as the product of the selectivity and the cardinality of tenk1:

```
rows = rel_cardinality * selectivity
     = 10000 * 0.1031
     = 1031
```

Next let's consider an example with equality condition in its WHERE clause:

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'ATAAAA';

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..470.00 rows=31 width=244)
   Filter: (stringu1 = 'ATAAAA'::name)
```

Again the planner examines the WHERE clause condition:

```
stringu1 = 'ATAAAA'
```

and looks up the restriction function for =, which is eqsel. This case is a bit different, as the most common values — MCVs, are used to determine the selectivity. Let's have a look at these, with some extra columns that will be useful later:

```
SELECT null_frac, n_distinct, most_common_vals, most_common_freqs FROM pg_stats
WHERE tablename='tenk1' AND attname='stringu1';

null_frac         | 0
n_distinct        | 672
most_common_vals  | {FDAAAA,NHAAAA,ATAAAA,BGAAAA,EBAAAA,MOAAAA,NDAAAA,OWAAAA,BHAAAA,BJAAAA}
most_common_freqs | {0.00333333,0.00333333,0.003,0.003,0.003,0.003,0.003,0.003,0.00266667,0
```

The selectivity is merely the most common frequency (MCF) corresponding to the third MCV — 'ATAAAA':

```
selectivity = mcf[3]
            = 0.003
```

The estimated number of rows is just the product of this with the cardinality of `tenk1` as before:

```
rows = 10000 * 0.003
     = 30
```

The number displayed by `EXPLAIN` is one more than this, due to some post estimation checks.

Now consider the same query, but with a constant that is not in the MCV list:

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'xxx';

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..470.00 rows=15 width=244)
   Filter: (stringu1 = 'xxx'::name)
```

This is quite a different problem, how to estimate the selectivity when the value is *not* in the MCV list. The approach is to use the fact that the value is not in the list, combined with the knowledge of the frequencies for all of the MCVs:

```
selectivity = (1 - sum(mvf))/(num_distinct - num_mcv)
            = (1 - (0.00333333 + 0.00333333 + 0.003 + 0.003 + 0.003
            + 0.003 + 0.003 + 0.003 + 0.00266667 + 0.00266667))/(672 - 10)
            = 0.001465
```

That is, add up all the frequencies for the MCVs and subtract them from one — because it is *not* one of these, and divide by the *remaining* distinct values. Notice that there are no null values so we don't have to worry about those. The estimated number of rows is calculated as usual:

```
rows = 10000 * 0.001465
     = 15
```

Let's increase the complexity to consider a case with more than one condition in the `WHERE` clause:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000 AND stringu1 = 'xxx';

                        QUERY PLAN
-----------------------------------------------------------
```

```
 Seq Scan on tenk1  (cost=0.00..495.00 rows=2 width=244)
   Filter: ((unique1 < 1000) AND (stringu1 = 'xxx'::name))
```

An assumption of independence is made and the selectivities of the individual restrictions are multiplied together:

```
selectivity = selectivity(unique1 < 1000) * selectivity(stringu1 = 'xxx')
            = 0.1031 * 0.001465
            = 0.00015104
```

The row estimates are calculated as before:

```
rows = 10000 * 0.00015104
     = 2
```

Finally we will examine a query that includes a `JOIN` together with a `WHERE` clause:

```
EXPLAIN SELECT *  FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;

                                        QUERY PLAN
------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..346.90 rows=51 width=488)
   ->  Index Scan using tenk1_unique1 on tenk1 t1  (cost=0.00..192.57 rows=51 width=244)
         Index Cond: (unique1 < 50)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.00..3.01 rows=1 width=244)
         Index Cond: ("outer".unique2 = t2.unique2)
```

The restriction on `tenk1` "unique1 < 50" is evaluated before the nested-loop join. This is handled analogously to the previous range example. The restriction operator for < is `scalarlteqsel` as before, but this time the value 50 is in the first bucket of the `unique1` histogram:

```
selectivity = (0 + (50 - bckt[1].min)/(bckt[1].max - bckt[1].min))/num_bckts
            = (0 + (50 - 1)/(970 - 1))/10
            = 0.005057

rows        = 10000 * 0.005057
            = 51
```

The restriction for the join is:

```
t2.unique2 = t1.unique2
```

This is due to the join method being nested-loop, with `tenk1` being in the outer loop. The operator is just our familiar =, however the restriction function is obtained from the `oprjoin` column of `pg_operator` - and is `eqjoinsel`. Additionally we use the statistical information for both `tenk2` and `tenk1`:

```
SELECT tablename, null_frac,n_distinct, most_common_vals FROM pg_stats
WHERE tablename IN ('tenk1', 'tenk2') AND attname='unique2';

tablename  | null_frac | n_distinct | most_common_vals
-----------+-----------+------------+------------------
```

```
 tenk1     |          0 |         -1 |
 tenk2     |          0 |         -1 |
```

In this case there is no MCV information for `unique2` because all the values appear to be unique, so we can use an algorithm that relies only on the number of distinct values for both relations together with their null fractions:

```
selectivity = (1 - null_frac1) * (1 - null_frac2) * min(1/num_distinct1, 1/num_distinct2)
            = (1 - 0) * (1 - 0) * min(1/10000, 1/1000)
            = 0.0001
```

This is, subtract the null fraction from one for each of the relations, and divide by the maximum of the two distinct values. The number of rows that the join is likely to emit is calculated as the cardinality of Cartesian product of the two nodes in the nested-loop, multiplied by the selectivity:

```
rows = (outer_cardinality * inner_cardinality) * selectivity
     = (51 * 10000) * 0.0001
     = 51
```

For those interested in further details, estimation of the number of rows in a relation is covered in `src/backend/optimizer/util/plancat.c`. The calculation logic for clause selectivities is in `src/backend/optimizer/path/clausesel.c`. The actual implementations of the operator and join restriction functions can be found in `src/backend/utils/adt/selfuncs.c`.

VIII. Appendixes

Appendix A. PostgreSQL Error Codes

All messages emitted by the PostgreSQL server are assigned five-character error codes that follow the SQL standard's conventions for "SQLSTATE" codes. Applications that need to know which error condition has occurred should usually test the error code, rather than looking at the textual error message. The error codes are less likely to change across PostgreSQL releases, and also are not subject to change due to localization of error messages. Note that some, but not all, of the error codes produced by PostgreSQL are defined by the SQL standard; some additional error codes for conditions not defined by the standard have been invented or borrowed from other databases.

According to the standard, the first two characters of an error code denote a class of errors, while the last three characters indicate a specific condition within that class. Thus, an application that does not recognize the specific error code may still be able to infer what to do from the error class.

Table A-1 lists all the error codes defined in PostgreSQL 8.2.6. (Some are not actually used at present, but are defined by the SQL standard.) The error classes are also shown. For each error class there is a "standard" error code having the last three characters `000`. This code is used only for error conditions that fall within the class but do not have any more-specific code assigned.

The PL/pgSQL condition name for each error code is the same as the phrase shown in the table, with underscores substituted for spaces. For example, code `22012`, DIVISION BY ZERO, has condition name `DIVISION_BY_ZERO`. Condition names can be written in either upper or lower case. (Note that PL/pgSQL does not recognize warning, as opposed to error, condition names; those are classes 00, 01, and 02.)

Table A-1. PostgreSQL Error Codes

Error Code	Meaning	Constant
Class 00 — Successful Completion		
`00000`	SUCCESSFUL COMPLETION	successful_completion
Class 01 — Warning		
`01000`	WARNING	warning
`0100C`	DYNAMIC RESULT SETS RETURNED	dynamic_result_sets_returned
`01008`	IMPLICIT ZERO BIT PADDING	implicit_zero_bit_padding
`01003`	NULL VALUE ELIMINATED IN SET FUNCTION	null_value_eliminated_in_set_function
`01007`	PRIVILEGE NOT GRANTED	privilege_not_granted
`01006`	PRIVILEGE NOT REVOKED	privilege_not_revoked
`01004`	STRING DATA RIGHT TRUNCATION	string_data_right_truncation
`01P01`	DEPRECATED FEATURE	deprecated_feature
Class 02 — No Data (this is also a warning class per the SQL standard)		
`02000`	NO DATA	no_data
`02001`	NO ADDITIONAL DYNAMIC RESULT SETS RETURNED	no_additional_dynamic_result_sets_returned

Error Code	Meaning	Constant
Class 03 — SQL Statement Not Yet Complete		
03000	SQL STATEMENT NOT YET COMPLETE	sql_statement_not_yet_complete
Class 08 — Connection Exception		
08000	CONNECTION EXCEPTION	connection_exception
08003	CONNECTION DOES NOT EXIST	connection_does_not_exist
08006	CONNECTION FAILURE	connection_failure
08001	SQLCLIENT UNABLE TO ESTABLISH SQLCONNECTION	sqlclient_unable_to_establish_sqlconnection
08004	SQLSERVER REJECTED ESTABLISHMENT OF SQLCONNECTION	sqlserver_rejected_establishment_of_sqlconnection
08007	TRANSACTION RESOLUTION UNKNOWN	transaction_resolution_unknown
08P01	PROTOCOL VIOLATION	protocol_violation
Class 09 — Triggered Action Exception		
09000	TRIGGERED ACTION EXCEPTION	triggered_action_exception
Class 0A — Feature Not Supported		
0A000	FEATURE NOT SUPPORTED	feature_not_supported
Class 0B — Invalid Transaction Initiation		
0B000	INVALID TRANSACTION INITIATION	invalid_transaction_initiation
Class 0F — Locator Exception		
0F000	LOCATOR EXCEPTION	locator_exception
0F001	INVALID LOCATOR SPECIFICATION	invalid_locator_specification
Class 0L — Invalid Grantor		
0L000	INVALID GRANTOR	invalid_grantor
0LP01	INVALID GRANT OPERATION	invalid_grant_operation
Class 0P — Invalid Role Specification		
0P000	INVALID ROLE SPECIFICATION	invalid_role_specification
Class 21 — Cardinality Violation		
21000	CARDINALITY VIOLATION	cardinality_violation
Class 22 — Data Exception		
22000	DATA EXCEPTION	data_exception

Error Code	Meaning	Constant
2202E	ARRAY SUBSCRIPT ERROR	array_subscript_error
22021	CHARACTER NOT IN REPERTOIRE	character_not_in_repertoire
22008	DATETIME FIELD OVERFLOW	datetime_field_overflow
22012	DIVISION BY ZERO	division_by_zero
22005	ERROR IN ASSIGNMENT	error_in_assignment
2200B	ESCAPE CHARACTER CONFLICT	escape_character_conflict
22022	INDICATOR OVERFLOW	indicator_overflow
22015	INTERVAL FIELD OVERFLOW	interval_field_overflow
2201E	INVALID ARGUMENT FOR LOGARITHM	invalid_argument_for_logarithm
2201F	INVALID ARGUMENT FOR POWER FUNCTION	invalid_argument_for_power_function
2201G	INVALID ARGUMENT FOR WIDTH BUCKET FUNCTION	invalid_argument_for_width_bucket_function
22018	INVALID CHARACTER VALUE FOR CAST	invalid_character_value_for_cast
22007	INVALID DATETIME FORMAT	invalid_datetime_format
22019	INVALID ESCAPE CHARACTER	invalid_escape_character
2200D	INVALID ESCAPE OCTET	invalid_escape_octet
22025	INVALID ESCAPE SEQUENCE	invalid_escape_sequence
22P06	NONSTANDARD USE OF ESCAPE CHARACTER	nonstandard_use_of_escape_character
22010	INVALID INDICATOR PARAMETER VALUE	invalid_indicator_parameter_value
22020	INVALID LIMIT VALUE	invalid_limit_value
22023	INVALID PARAMETER VALUE	invalid_parameter_value
2201B	INVALID REGULAR EXPRESSION	invalid_regular_expression
22009	INVALID TIME ZONE DISPLACEMENT VALUE	invalid_time_zone_displacement_value
2200C	INVALID USE OF ESCAPE CHARACTER	invalid_use_of_escape_character

Error Code	Meaning	Constant
2200G	MOST SPECIFIC TYPE MISMATCH	most_specific_type_mismatch
22004	NULL VALUE NOT ALLOWED	null_value_not_allowed
22002	NULL VALUE NO INDICATOR PARAMETER	null_value_no_indicator_parameter
22003	NUMERIC VALUE OUT OF RANGE	numeric_value_out_of_range
22026	STRING DATA LENGTH MISMATCH	string_data_length_mismatch
22001	STRING DATA RIGHT TRUNCATION	string_data_right_truncation
22011	SUBSTRING ERROR	substring_error
22027	TRIM ERROR	trim_error
22024	UNTERMINATED C STRING	unterminated_c_string
2200F	ZERO LENGTH CHARACTER STRING	zero_length_character_string
22P01	FLOATING POINT EXCEPTION	floating_point_exception
22P02	INVALID TEXT REPRESENTATION	invalid_text_representation
22P03	INVALID BINARY REPRESENTATION	invalid_binary_representation
22P04	BAD COPY FILE FORMAT	bad_copy_file_format
22P05	UNTRANSLATABLE CHARACTER	untranslatable_character
Class 23 — Integrity Constraint Violation		
23000	INTEGRITY CONSTRAINT VIOLATION	integrity_constraint_violation
23001	RESTRICT VIOLATION	restrict_violation
23502	NOT NULL VIOLATION	not_null_violation
23503	FOREIGN KEY VIOLATION	foreign_key_violation
23505	UNIQUE VIOLATION	unique_violation
23514	CHECK VIOLATION	check_violation
Class 24 — Invalid Cursor State		
24000	INVALID CURSOR STATE	invalid_cursor_state
Class 25 — Invalid Transaction State		
25000	INVALID TRANSACTION STATE	invalid_transaction_state
25001	ACTIVE SQL TRANSACTION	active_sql_transaction

Error Code	Meaning	Constant
25002	BRANCH TRANSACTION ALREADY ACTIVE	branch_transaction_already_active
25008	HELD CURSOR REQUIRES SAME ISOLATION LEVEL	held_cursor_requires_same_isolation_level
25003	INAPPROPRIATE ACCESS MODE FOR BRANCH TRANSACTION	inappropriate_access_mode_for_branch_transaction
25004	INAPPROPRIATE ISOLATION LEVEL FOR BRANCH TRANSACTION	inappropriate_isolation_level_for_branch_transactio
25005	NO ACTIVE SQL TRANSACTION FOR BRANCH TRANSACTION	no_active_sql_transaction_for_branch_transaction
25006	READ ONLY SQL TRANSACTION	read_only_sql_transaction
25007	SCHEMA AND DATA STATEMENT MIXING NOT SUPPORTED	schema_and_data_statement_mixing_not_supported
25P01	NO ACTIVE SQL TRANSACTION	no_active_sql_transaction
25P02	IN FAILED SQL TRANSACTION	in_failed_sql_transaction
Class 26 — Invalid SQL Statement Name		
26000	INVALID SQL STATEMENT NAME	invalid_sql_statement_name
Class 27 — Triggered Data Change Violation		
27000	TRIGGERED DATA CHANGE VIOLATION	triggered_data_change_violation
Class 28 — Invalid Authorization Specification		
28000	INVALID AUTHORIZATION SPECIFICATION	invalid_authorization_specification
Class 2B — Dependent Privilege Descriptors Still Exist		
2B000	DEPENDENT PRIVILEGE DESCRIPTORS STILL EXIST	dependent_privilege_descriptors_still_exist
2BP01	DEPENDENT OBJECTS STILL EXIST	dependent_objects_still_exist
Class 2D — Invalid Transaction Termination		
2D000	INVALID TRANSACTION TERMINATION	invalid_transaction_termination
Class 2F — SQL Routine Exception		
2F000	SQL ROUTINE EXCEPTION	sql_routine_exception

Error Code	Meaning	Constant
2F005	FUNCTION EXECUTED NO RETURN STATEMENT	function_executed_no_return_statement
2F002	MODIFYING SQL DATA NOT PERMITTED	modifying_sql_data_not_permitted
2F003	PROHIBITED SQL STATEMENT ATTEMPTED	prohibited_sql_statement_attempted
2F004	READING SQL DATA NOT PERMITTED	reading_sql_data_not_permitted
Class 34 — Invalid Cursor Name		
34000	INVALID CURSOR NAME	invalid_cursor_name
Class 38 — External Routine Exception		
38000	EXTERNAL ROUTINE EXCEPTION	external_routine_exception
38001	CONTAINING SQL NOT PERMITTED	containing_sql_not_permitted
38002	MODIFYING SQL DATA NOT PERMITTED	modifying_sql_data_not_permitted
38003	PROHIBITED SQL STATEMENT ATTEMPTED	prohibited_sql_statement_attempted
38004	READING SQL DATA NOT PERMITTED	reading_sql_data_not_permitted
Class 39 — External Routine Invocation Exception		
39000	EXTERNAL ROUTINE INVOCATION EXCEPTION	external_routine_invocation_exception
39001	INVALID SQLSTATE RETURNED	invalid_sqlstate_returned
39004	NULL VALUE NOT ALLOWED	null_value_not_allowed
39P01	TRIGGER PROTOCOL VIOLATED	trigger_protocol_violated
39P02	SRF PROTOCOL VIOLATED	srf_protocol_violated
Class 3B — Savepoint Exception		
3B000	SAVEPOINT EXCEPTION	savepoint_exception
3B001	INVALID SAVEPOINT SPECIFICATION	invalid_savepoint_specification
Class 3D — Invalid Catalog Name		
3D000	INVALID CATALOG NAME	invalid_catalog_name
Class 3F — Invalid Schema Name		
3F000	INVALID SCHEMA NAME	invalid_schema_name
Class 40 — Transaction Rollback		
40000	TRANSACTION ROLLBACK	transaction_rollback

Error Code	Meaning	Constant
40002	TRANSACTION INTEGRITY CONSTRAINT VIOLATION	transaction_integrity_constraint_violation
40001	SERIALIZATION FAILURE	serialization_failure
40003	STATEMENT COMPLETION UNKNOWN	statement_completion_unknown
40P01	DEADLOCK DETECTED	deadlock_detected
Class 42 — Syntax Error or Access Rule Violation		
42000	SYNTAX ERROR OR ACCESS RULE VIOLATION	syntax_error_or_access_rule_violation
42601	SYNTAX ERROR	syntax_error
42501	INSUFFICIENT PRIVILEGE	insufficient_privilege
42846	CANNOT COERCE	cannot_coerce
42803	GROUPING ERROR	grouping_error
42830	INVALID FOREIGN KEY	invalid_foreign_key
42602	INVALID NAME	invalid_name
42622	NAME TOO LONG	name_too_long
42939	RESERVED NAME	reserved_name
42804	DATATYPE MISMATCH	datatype_mismatch
42P18	INDETERMINATE DATATYPE	indeterminate_datatype
42809	WRONG OBJECT TYPE	wrong_object_type
42703	UNDEFINED COLUMN	undefined_column
42883	UNDEFINED FUNCTION	undefined_function
42P01	UNDEFINED TABLE	undefined_table
42P02	UNDEFINED PARAMETER	undefined_parameter
42704	UNDEFINED OBJECT	undefined_object
42701	DUPLICATE COLUMN	duplicate_column
42P03	DUPLICATE CURSOR	duplicate_cursor
42P04	DUPLICATE DATABASE	duplicate_database
42723	DUPLICATE FUNCTION	duplicate_function
42P05	DUPLICATE PREPARED STATEMENT	duplicate_prepared_statement
42P06	DUPLICATE SCHEMA	duplicate_schema
42P07	DUPLICATE TABLE	duplicate_table
42712	DUPLICATE ALIAS	duplicate_alias
42710	DUPLICATE OBJECT	duplicate_object
42702	AMBIGUOUS COLUMN	ambiguous_column
42725	AMBIGUOUS FUNCTION	ambiguous_function
42P08	AMBIGUOUS PARAMETER	ambiguous_parameter
42P09	AMBIGUOUS ALIAS	ambiguous_alias

Error Code	Meaning	Constant
42P10	INVALID COLUMN REFERENCE	invalid_column_reference
42611	INVALID COLUMN DEFINITION	invalid_column_definition
42P11	INVALID CURSOR DEFINITION	invalid_cursor_definition
42P12	INVALID DATABASE DEFINITION	invalid_database_definition
42P13	INVALID FUNCTION DEFINITION	invalid_function_definition
42P14	INVALID PREPARED STATEMENT DEFINITION	invalid_prepared_statement_definition
42P15	INVALID SCHEMA DEFINITION	invalid_schema_definition
42P16	INVALID TABLE DEFINITION	invalid_table_definition
42P17	INVALID OBJECT DEFINITION	invalid_object_definition
Class 44 — WITH CHECK OPTION Violation		
44000	WITH CHECK OPTION VIOLATION	with_check_option_violation
Class 53 — Insufficient Resources		
53000	INSUFFICIENT RESOURCES	insufficient_resources
53100	DISK FULL	disk_full
53200	OUT OF MEMORY	out_of_memory
53300	TOO MANY CONNECTIONS	too_many_connections
Class 54 — Program Limit Exceeded		
54000	PROGRAM LIMIT EXCEEDED	program_limit_exceeded
54001	STATEMENT TOO COMPLEX	statement_too_complex
54011	TOO MANY COLUMNS	too_many_columns
54023	TOO MANY ARGUMENTS	too_many_arguments
Class 55 — Object Not In Prerequisite State		
55000	OBJECT NOT IN PREREQUISITE STATE	object_not_in_prerequisite_state
55006	OBJECT IN USE	object_in_use
55P02	CANT CHANGE RUNTIME PARAM	cant_change_runtime_param
55P03	LOCK NOT AVAILABLE	lock_not_available
Class 57 — Operator Intervention		
57000	OPERATOR INTERVENTION	operator_intervention

Error Code	Meaning	Constant
57014	QUERY CANCELED	query_canceled
57P01	ADMIN SHUTDOWN	admin_shutdown
57P02	CRASH SHUTDOWN	crash_shutdown
57P03	CANNOT CONNECT NOW	cannot_connect_now
Class 58 — System Error (errors external to PostgreSQL itself)		
58030	IO ERROR	io_error
58P01	UNDEFINED FILE	undefined_file
58P02	DUPLICATE FILE	duplicate_file
Class F0 — Configuration File Error		
F0000	CONFIG FILE ERROR	config_file_error
F0001	LOCK FILE EXISTS	lock_file_exists
Class P0 — PL/pgSQL Error		
P0000	PLPGSQL ERROR	plpgsql_error
P0001	RAISE EXCEPTION	raise_exception
P0002	NO DATA FOUND	no_data_found
P0003	TOO MANY ROWS	too_many_rows
Class XX — Internal Error		
XX000	INTERNAL ERROR	internal_error
XX001	DATA CORRUPTED	data_corrupted
XX002	INDEX CORRUPTED	index_corrupted

Appendix B. Date/Time Support

PostgreSQL uses an internal heuristic parser for all date/time input support. Dates and times are input as strings, and are broken up into distinct fields with a preliminary determination of what kind of information may be in the field. Each field is interpreted and either assigned a numeric value, ignored, or rejected. The parser contains internal lookup tables for all textual fields, including months, days of the week, and time zones.

This appendix includes information on the content of these lookup tables and describes the steps used by the parser to decode dates and times.

B.1. Date/Time Input Interpretation

The date/time type inputs are all decoded using the following procedure.

1. Break the input string into tokens and categorize each token as a string, time, time zone, or number.
    a. If the numeric token contains a colon (`:`), this is a time string. Include all subsequent digits and colons.
    b. If the numeric token contains a dash (`-`), slash (`/`), or two or more dots (`.`), this is a date string which may have a text month. If a date token has already been seen, it is instead interpreted as a time zone name (e.g., `America/New_York`).
    c. If the token is numeric only, then it is either a single field or an ISO 8601 concatenated date (e.g., `19990113` for January 13, 1999) or time (e.g., `141516` for 14:15:16).
    d. If the token starts with a plus (`+`) or minus (`-`), then it is either a numeric time zone or a special field.
2. If the token is a text string, match up with possible strings:
    a. Do a binary-search table lookup for the token as a time zone abbreviation.
    b. If not found, do a similar binary-search table lookup to match the token as either a special string (e.g., `today`), day (e.g., `Thursday`), month (e.g., `January`), or noise word (e.g., `at`, `on`).
    c. If still not found, throw an error.
3. When the token is a number or number field:
    a. If there are eight or six digits, and if no other date fields have been previously read, then interpret as a "concatenated date" (e.g., `19990118` or `990118`). The interpretation is `YYYYMMDD` or `YYMMDD`.
    b. If the token is three digits and a year has already been read, then interpret as day of year.
    c. If four or six digits and a year has already been read, then interpret as a time (`HHMM` or `HHMMSS`).

d. If three or more digits and no date fields have yet been found, interpret as a year (this forces yy-mm-dd ordering of the remaining date fields).

e. Otherwise the date field ordering is assumed to follow the `DateStyle` setting: mm-dd-yy, dd-mm-yy, or yy-mm-dd. Throw an error if a month or day field is found to be out of range.

4. If BC has been specified, negate the year and add one for internal storage. (There is no year zero in the Gregorian calendar, so numerically 1 BC becomes year zero.)

5. If BC was not specified, and if the year field was two digits in length, then adjust the year to four digits. If the field is less than 70, then add 2000, otherwise add 1900.

Tip: Gregorian years AD 1-99 may be entered by using 4 digits with leading zeros (e.g., `0099` is AD 99).

B.2. Date/Time Key Words

Table B-1 shows the tokens that are recognized as names of months.

Table B-1. Month Names

Month	Abbreviations
January	Jan
February	Feb
March	Mar
April	Apr
May	
June	Jun
July	Jul
August	Aug
September	Sep, Sept
October	Oct
November	Nov
December	Dec

Table B-2 shows the tokens that are recognized as names of days of the week.

Table B-2. Day of the Week Names

Day	Abbreviations
Sunday	Sun
Monday	Mon

Day	Abbreviations
Tuesday	Tue, Tues
Wednesday	Wed, Weds
Thursday	Thu, Thur, Thurs
Friday	Fri
Saturday	Sat

Table B-3 shows the tokens that serve various modifier purposes.

Table B-3. Date/Time Field Modifiers

Identifier	Description
ABSTIME	Ignored
AM	Time is before 12:00
AT	Ignored
JULIAN, JD, J	Next field is Julian Day
ON	Ignored
PM	Time is on or after 12:00
T	Next field is time

The key word ABSTIME is ignored for historical reasons: In very old releases of PostgreSQL, invalid values of type abstime were emitted as Invalid Abstime. This is no longer the case however and this key word will likely be dropped in a future release.

B.3. Date/Time Configuration Files

Since timezone abbreviations are not well standardized, PostgreSQL provides a means to customize the set of abbreviations accepted by the server. The timezone_abbreviations run-time parameter determines the active set of abbreviations. While this parameter can be altered by any database user, the possible values for it are under the control of the database administrator — they are in fact names of configuration files stored in .../share/timezonesets/ of the installation directory. By adding or altering files in that directory, the administrator can set local policy for timezone abbreviations.

timezone_abbreviations can be set to any file name found in .../share/timezonesets/, if the file's name is entirely alphabetic. (The prohibition against non-alphabetic characters in timezone_abbreviations prevents reading files outside the intended directory, as well as reading editor backup files and other extraneous files.)

A timezone abbreviation file may contain blank lines and comments beginning with #. Non-comment lines must have one of these formats:

```
time_zone_name offset
time_zone_name offset D
@INCLUDE file_name
@OVERRIDE
```

A *time_zone_name* is just the abbreviation being defined. The *offset* is the zone's offset in seconds from UTC, positive being east from Greenwich and negative being west. For example, -18000 would be five hours west of Greenwich, or North American east coast standard time. `D` indicates that the zone name represents local daylight-savings time rather than standard time. Since all known time zone offsets are on 15 minute boundaries, the number of seconds has to be a multiple of 900.

The `@INCLUDE` syntax allows inclusion of another file in the `.../share/timezonesets/` directory. Inclusion can be nested, to a limited depth.

The `@OVERRIDE` syntax indicates that subsequent entries in the file may override previous entries (i.e., entries obtained from included files). Without this, conflicting definitions of the same timezone abbreviation are considered an error.

In an unmodified installation, the file `Default` contains all the non-conflicting time zone abbreviations for most of the world. Additional files `Australia` and `India` are provided for those regions: these files first include the `Default` file and then add or modify timezones as needed.

For reference purposes, a standard installation also contains files `Africa.txt`, `America.txt`, etc, containing information about every time zone abbreviation known to be in use according to the `zic` timezone database. The zone name definitions found in these files can be copied and pasted into a custom configuration file as needed. Note that these files cannot be directly referenced as `timezone_abbreviations` settings, because of the dot embedded in their names.

Note: If an error occurs while reading the time zone data sets, no new value is applied but the old set is kept. If the error occurs while starting the database, startup fails.

Caution

Time zone abbreviations defined in the configuration file override non-timezone meanings built into PostgreSQL. For example, the `Australia` configuration file defines `SAT` (for South Australian Standard Time). When this file is active, `SAT` will not be recognized as an abbreviation for Saturday.

Caution

If you modify files in `.../share/timezonesets/`, it is up to you to make backups — a normal database dump will not include this directory.

B.4. History of Units

The Julian Date was invented by the French scholar Joseph Justus Scaliger (1540-1609) and probably takes its name from Scaliger's father, the Italian scholar Julius Caesar Scaliger (1484-1558). Astronomers have used the Julian period to assign a unique number to every day since 1 January 4713 BC. This is the so-called Julian Date (JD). JD 0 designates the 24 hours from noon UTC on 1 January 4713 BC to noon UTC on 2 January 4713 BC.

The "Julian Date" is different from the "Julian Calendar". The Julian calendar was introduced by Julius Caesar in 45 BC. It was in common use until the year 1582, when countries started changing to the Gregorian calendar. In the Julian calendar, the tropical year is approximated as 365 1/4 days = 365.25 days. This gives an error of about 1 day in 128 years.

The accumulating calendar error prompted Pope Gregory XIII to reform the calendar in accordance with instructions from the Council of Trent. In the Gregorian calendar, the tropical year is approximated as 365 + 97 / 400 days = 365.2425 days. Thus it takes approximately 3300 years for the tropical year to shift one day with respect to the Gregorian calendar.

The approximation 365+97/400 is achieved by having 97 leap years every 400 years, using the following rules:

Every year divisible by 4 is a leap year.
However, every year divisible by 100 is not a leap year.
However, every year divisible by 400 is a leap year after all.

So, 1700, 1800, 1900, 2100, and 2200 are not leap years. But 1600, 2000, and 2400 are leap years. By contrast, in the older Julian calendar all years divisible by 4 are leap years.

The papal bull of February 1582 decreed that 10 days should be dropped from October 1582 so that 15 October should follow immediately after 4 October. This was observed in Italy, Poland, Portugal, and Spain. Other Catholic countries followed shortly after, but Protestant countries were reluctant to change, and the Greek orthodox countries didn't change until the start of the 20th century. The reform was observed by Great Britain and Dominions (including what is now the USA) in 1752. Thus 2 September 1752 was followed by 14 September 1752. This is why Unix systems have the `cal` program produce the following:

```
$ cal 9 1752
   September 1752
 S  M Tu  W Th  F  S
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

> **Note:** The SQL standard states that "Within the definition of a 'datetime literal', the 'datetime value's are constrained by the natural rules for dates and times according to the Gregorian calendar". Dates between 1752-09-03 and 1752-09-13, although eliminated in some countries by Papal fiat, conform to "natural rules" and are hence valid dates.

Different calendars have been developed in various parts of the world, many predating the Gregorian system. For example, the beginnings of the Chinese calendar can be traced back to the 14th century BC. Legend has it that the Emperor Huangdi invented the calendar in 2637 BC. The People's Republic of China uses the Gregorian calendar for civil purposes. The Chinese calendar is used for determining festivals.

Appendix C. SQL Key Words

Table C-1 lists all tokens that are key words in the SQL standard and in PostgreSQL 8.2.6. Background information can be found in Section 4.1.1.

SQL distinguishes between *reserved* and *non-reserved* key words. According to the standard, reserved key words are the only real key words; they are never allowed as identifiers. Non-reserved key words only have a special meaning in particular contexts and can be used as identifiers in other contexts. Most non-reserved key words are actually the names of built-in tables and functions specified by SQL. The concept of non-reserved key words essentially only exists to declare that some predefined meaning is attached to a word in some contexts.

In the PostgreSQL parser life is a bit more complicated. There are several different classes of tokens ranging from those that can never be used as an identifier to those that have absolutely no special status in the parser as compared to an ordinary identifier. (The latter is usually the case for functions specified by SQL.) Even reserved key words are not completely reserved in PostgreSQL, but can be used as column labels (for example, `SELECT 55 AS CHECK`, even though `CHECK` is a reserved key word).

In Table C-1 in the column for PostgreSQL we classify as "non-reserved" those key words that are explicitly known to the parser but are allowed in most or all contexts where an identifier is expected. Some key words that are otherwise non-reserved cannot be used as function or data type names and are marked accordingly. (Most of these words represent built-in functions or data types with special syntax. The function or type is still available but it cannot be redefined by the user.) Labeled "reserved" are those tokens that are only allowed as "AS" column label names (and perhaps in very few other contexts). Some reserved key words are allowable as names for functions; this is also shown in the table.

As a general rule, if you get spurious parser errors for commands that contain any of the listed key words as an identifier you should try to quote the identifier to see if the problem goes away.

It is important to understand before studying Table C-1 that the fact that a key word is not reserved in PostgreSQL does not mean that the feature related to the word is not implemented. Conversely, the presence of a key word does not indicate the existence of a feature.

Table C-1. SQL Key Words

Key Word	PostgreSQL	SQL:2003	SQL:1999	SQL-92
A		non-reserved		
ABORT	non-reserved			
ABS		reserved	non-reserved	
ABSOLUTE	non-reserved	non-reserved	reserved	reserved
ACCESS	non-reserved			
ACTION	non-reserved	non-reserved	reserved	reserved
ADA		non-reserved	non-reserved	non-reserved
ADD	non-reserved	non-reserved	reserved	reserved
ADMIN	non-reserved	non-reserved	reserved	

Key Word	PostgreSQL	SQL:2003	SQL:1999	SQL-92
AFTER	non-reserved	non-reserved	reserved	
AGGREGATE	non-reserved		reserved	
ALIAS			reserved	
ALL	reserved	reserved	reserved	reserved
ALLOCATE		reserved	reserved	reserved
ALSO	non-reserved			
ALTER	non-reserved	reserved	reserved	reserved
ALWAYS		non-reserved		
ANALYSE	reserved			
ANALYZE	reserved			
AND	reserved	reserved	reserved	reserved
ANY	reserved	reserved	reserved	reserved
ARE		reserved	reserved	reserved
ARRAY	reserved	reserved	reserved	
AS	reserved	reserved	reserved	reserved
ASC	reserved	non-reserved	reserved	reserved
ASENSITIVE		reserved	non-reserved	
ASSERTION	non-reserved	non-reserved	reserved	reserved
ASSIGNMENT	non-reserved	non-reserved	non-reserved	
ASYMMETRIC	reserved	reserved	non-reserved	
AT	non-reserved	reserved	reserved	reserved
ATOMIC		reserved	non-reserved	
ATTRIBUTE		non-reserved		
ATTRIBUTES		non-reserved		
AUTHORIZATION	reserved (can be function)	reserved	reserved	reserved
AVG		reserved	non-reserved	reserved
BACKWARD	non-reserved			
BEFORE	non-reserved	non-reserved	reserved	
BEGIN	non-reserved	reserved	reserved	reserved
BERNOULLI		non-reserved		
BETWEEN	reserved (can be function)	reserved	non-reserved	reserved
BIGINT	non-reserved (cannot be function or type)	reserved		
BINARY	reserved (can be function)	reserved	reserved	

Key Word	PostgreSQL	SQL:2003	SQL:1999	SQL-92
BIT	non-reserved (cannot be function or type)		reserved	reserved
BITVAR			non-reserved	
BIT_LENGTH			non-reserved	reserved
BLOB		reserved	reserved	
BOOLEAN	non-reserved (cannot be function or type)	reserved	reserved	
BOTH	reserved	reserved	reserved	reserved
BREADTH		non-reserved	reserved	
BY	non-reserved	reserved	reserved	reserved
C		non-reserved	non-reserved	non-reserved
CACHE	non-reserved			
CALL		reserved	reserved	
CALLED	non-reserved	reserved	non-reserved	
CARDINALITY		reserved	non-reserved	
CASCADE	non-reserved	non-reserved	reserved	reserved
CASCADED	non-reserved	reserved	reserved	reserved
CASE	reserved	reserved	reserved	reserved
CAST	reserved	reserved	reserved	reserved
CATALOG		non-reserved	reserved	reserved
CATALOG_NAME		non-reserved	non-reserved	non-reserved
CEIL		reserved		
CEILING		reserved		
CHAIN	non-reserved	non-reserved	non-reserved	
CHAR	non-reserved (cannot be function or type)	reserved	reserved	reserved
CHARACTER	non-reserved (cannot be function or type)	reserved	reserved	reserved
CHARACTERISTICS	non-reserved	non-reserved		
CHARACTERS		non-reserved		
CHARACTER_LENGTH		reserved	non-reserved	reserved
CHARACTER_SET_CATALOG		non-reserved	non-reserved	non-reserved

Key Word	PostgreSQL	SQL:2003	SQL:1999	SQL-92
CHARACTER_SET_NAME		non-reserved	non-reserved	non-reserved
CHARACTER_SET_SCHEMA		non-reserved	non-reserved	non-reserved
CHAR_LENGTH		reserved	non-reserved	reserved
CHECK	reserved	reserved	reserved	reserved
CHECKED			non-reserved	
CHECKPOINT	non-reserved			
CLASS	non-reserved		reserved	
CLASS_ORIGIN		non-reserved	non-reserved	non-reserved
CLOB		reserved	reserved	
CLOSE	non-reserved	reserved	reserved	reserved
CLUSTER	non-reserved			
COALESCE	non-reserved (cannot be function or type)	reserved	non-reserved	reserved
COBOL		non-reserved	non-reserved	non-reserved
COLLATE	reserved	reserved	reserved	reserved
COLLATION		non-reserved	reserved	reserved
COLLATION_CATALOG		non-reserved	non-reserved	non-reserved
COLLATION_NAME		non-reserved	non-reserved	non-reserved
COLLATION_SCHEMA		non-reserved	non-reserved	non-reserved
COLLECT		reserved		
COLUMN	reserved	reserved	reserved	reserved
COLUMN_NAME		non-reserved	non-reserved	non-reserved
COMMAND_FUNCTION		non-reserved	non-reserved	non-reserved
COMMAND_FUNCTION_CODE		non-reserved	non-reserved	
COMMENT	non-reserved			
COMMIT	non-reserved	reserved	reserved	reserved
COMMITTED	non-reserved	non-reserved	non-reserved	non-reserved
COMPLETION			reserved	
CONCURRENTLY	non-reserved			
CONDITION		reserved		
CONDITION_NUMBER		non-reserved	non-reserved	non-reserved
CONNECT		reserved	reserved	reserved

Key Word	PostgreSQL	SQL:2003	SQL:1999	SQL-92
CONNECTION	non-reserved	non-reserved	reserved	reserved
CONNECTION_NAME		non-reserved	non-reserved	non-reserved
CONSTRAINT	reserved	reserved	reserved	reserved
CONSTRAINTS	non-reserved	non-reserved	reserved	reserved
CONSTRAINT_CATALOG		non-reserved	non-reserved	non-reserved
CONSTRAINT_NAME		non-reserved	non-reserved	non-reserved
CONSTRAINT_SCHEMA		non-reserved	non-reserved	non-reserved
CONSTRUCTOR		non-reserved	reserved	
CONTAINS		non-reserved	non-reserved	
CONTINUE		non-reserved	reserved	reserved
CONVERSION	non-reserved			
CONVERT	non-reserved (cannot be function or type)	reserved	non-reserved	reserved
COPY	non-reserved			
CORR		reserved		
CORRESPONDING		reserved	reserved	reserved
COUNT		reserved	non-reserved	reserved
COVAR_POP		reserved		
COVAR_SAMP		reserved		
CREATE	reserved	reserved	reserved	reserved
CREATEDB	non-reserved			
CREATEROLE	non-reserved			
CREATEUSER	non-reserved			
CROSS	reserved (can be function)	reserved	reserved	reserved
CSV	non-reserved			
CUBE		reserved	reserved	
CUME_DIST		reserved		
CURRENT		reserved	reserved	reserved
CURRENT_DATE	reserved	reserved	reserved	reserved
CURRENT_DEFAULT_TRANSFORM_GROUP		reserved		
CURRENT_PATH		reserved	reserved	
CURRENT_ROLE	reserved	reserved	reserved	
CURRENT_TIME	reserved	reserved	reserved	reserved

Key Word	PostgreSQL	SQL:2003	SQL:1999	SQL-92
CURRENT_TIMESTAMP	reserved	reserved	reserved	reserved
CURRENT_TRANSFORM_GROUP_FOR_TYPE	reserved			
CURRENT_USER	reserved	reserved	reserved	reserved
CURSOR	non-reserved	reserved	reserved	reserved
CURSOR_NAME		non-reserved	non-reserved	non-reserved
CYCLE	non-reserved	reserved	reserved	
DATA		non-reserved	reserved	non-reserved
DATABASE	non-reserved			
DATE		reserved	reserved	reserved
DATETIME_INTERVAL_CODE		non-reserved	non-reserved	non-reserved
DATETIME_INTERVAL_PRECISION		non-reserved	non-reserved	non-reserved
DAY	non-reserved	reserved	reserved	reserved
DEALLOCATE	non-reserved	reserved	reserved	reserved
DEC	non-reserved (cannot be function or type)	reserved	reserved	reserved
DECIMAL	non-reserved (cannot be function or type)	reserved	reserved	reserved
DECLARE	non-reserved	reserved	reserved	reserved
DEFAULT	reserved	reserved	reserved	reserved
DEFAULTS	non-reserved	non-reserved		
DEFERRABLE	reserved	non-reserved	reserved	reserved
DEFERRED	non-reserved	non-reserved	reserved	reserved
DEFINED		non-reserved	non-reserved	
DEFINER	non-reserved	non-reserved	non-reserved	
DEGREE		non-reserved		
DELETE	non-reserved	reserved	reserved	reserved
DELIMITER	non-reserved			
DELIMITERS	non-reserved			
DENSE_RANK		reserved		
DEPTH		non-reserved	reserved	
DEREF		reserved	reserved	
DERIVED		non-reserved		
DESC	reserved	non-reserved	reserved	reserved
DESCRIBE		reserved	reserved	reserved

Key Word	PostgreSQL	SQL:2003	SQL:1999	SQL-92
DESCRIPTOR		non-reserved	reserved	reserved
DESTROY			reserved	
DESTRUCTOR			reserved	
DETERMINISTIC		reserved	reserved	
DIAGNOSTICS		non-reserved	reserved	reserved
DICTIONARY			reserved	
DISABLE	non-reserved			
DISCONNECT		reserved	reserved	reserved
DISPATCH		non-reserved	non-reserved	
DISTINCT	reserved	reserved	reserved	reserved
DO	reserved			
DOMAIN	non-reserved	non-reserved	reserved	reserved
DOUBLE	non-reserved	reserved	reserved	reserved
DROP	non-reserved	reserved	reserved	reserved
DYNAMIC		reserved	reserved	
DYNAMIC_FUNCTION		non-reserved	non-reserved	non-reserved
DYNAMIC_FUNCTION_CODE		non-reserved	non-reserved	
EACH	non-reserved	reserved	reserved	
ELEMENT		reserved		
ELSE	reserved	reserved	reserved	reserved
ENABLE	non-reserved			
ENCODING	non-reserved			
ENCRYPTED	non-reserved			
END	reserved	reserved	reserved	reserved
END-EXEC		reserved	reserved	reserved
EQUALS		non-reserved	reserved	
ESCAPE	non-reserved	reserved	reserved	reserved
EVERY		reserved	reserved	
EXCEPT	reserved	reserved	reserved	reserved
EXCEPTION		non-reserved	reserved	reserved
EXCLUDE		non-reserved		
EXCLUDING	non-reserved	non-reserved		
EXCLUSIVE	non-reserved			
EXEC		reserved	reserved	reserved
EXECUTE	non-reserved	reserved	reserved	reserved
EXISTING			non-reserved	

Key Word	PostgreSQL	SQL:2003	SQL:1999	SQL-92
EXISTS	non-reserved (cannot be function or type)	reserved	non-reserved	reserved
EXP		reserved		
EXPLAIN	non-reserved			
EXTERNAL	non-reserved	reserved	reserved	reserved
EXTRACT	non-reserved (cannot be function or type)	reserved	non-reserved	reserved
FALSE	reserved	reserved	reserved	reserved
FETCH	non-reserved	reserved	reserved	reserved
FILTER		reserved		
FINAL		non-reserved	non-reserved	
FIRST	non-reserved	non-reserved	reserved	reserved
FLOAT	non-reserved (cannot be function or type)	reserved	reserved	reserved
FLOOR		reserved		
FOLLOWING		non-reserved		
FOR	reserved	reserved	reserved	reserved
FORCE	non-reserved			
FOREIGN	reserved	reserved	reserved	reserved
FORTRAN		non-reserved	non-reserved	non-reserved
FORWARD	non-reserved			
FOUND		non-reserved	reserved	reserved
FREE		reserved	reserved	
FREEZE	reserved (can be function)			
FROM	reserved	reserved	reserved	reserved
FULL	reserved (can be function)	reserved	reserved	reserved
FUNCTION	non-reserved	reserved	reserved	
FUSION		reserved		
G		non-reserved	non-reserved	
GENERAL		non-reserved	reserved	
GENERATED		non-reserved	non-reserved	
GET		reserved	reserved	reserved
GLOBAL	non-reserved	reserved	reserved	reserved
GO		non-reserved	reserved	reserved
GOTO		non-reserved	reserved	reserved

Key Word	PostgreSQL	SQL:2003	SQL:1999	SQL-92
GRANT	reserved	reserved	reserved	reserved
GRANTED	non-reserved	non-reserved	non-reserved	
GREATEST	non-reserved (cannot be function or type)			
GROUP	reserved	reserved	reserved	reserved
GROUPING		reserved	reserved	
HANDLER	non-reserved			
HAVING	reserved	reserved	reserved	reserved
HEADER	non-reserved			
HIERARCHY		non-reserved	non-reserved	
HOLD	non-reserved	reserved	non-reserved	
HOST			reserved	
HOUR	non-reserved	reserved	reserved	reserved
IDENTITY		reserved	reserved	reserved
IF	non-reserved			
IGNORE			reserved	
ILIKE	reserved (can be function)			
IMMEDIATE	non-reserved	non-reserved	reserved	reserved
IMMUTABLE	non-reserved			
IMPLEMENTATION		non-reserved	non-reserved	
IMPLICIT	non-reserved			
IN	reserved	reserved	reserved	reserved
INCLUDING	non-reserved	non-reserved		
INCREMENT	non-reserved	non-reserved		
INDEX	non-reserved			
INDEXES	non-reserved			
INDICATOR		reserved	reserved	reserved
INFIX			non-reserved	
INHERIT	non-reserved			
INHERITS	non-reserved			
INITIALIZE			reserved	
INITIALLY	reserved	non-reserved	reserved	reserved
INNER	reserved (can be function)	reserved	reserved	reserved
INOUT	non-reserved (cannot be function or type)	reserved	reserved	

Key Word	PostgreSQL	SQL:2003	SQL:1999	SQL-92
INPUT	non-reserved	non-reserved	reserved	reserved
INSENSITIVE	non-reserved	reserved	non-reserved	reserved
INSERT	non-reserved	reserved	reserved	reserved
INSTANCE		non-reserved	non-reserved	
INSTANTIABLE		non-reserved	non-reserved	
INSTEAD	non-reserved			
INT	non-reserved (cannot be function or type)	reserved	reserved	reserved
INTEGER	non-reserved (cannot be function or type)	reserved	reserved	reserved
INTERSECT	reserved	reserved	reserved	reserved
INTERSECTION		reserved		
INTERVAL	non-reserved (cannot be function or type)	reserved	reserved	reserved
INTO	reserved	reserved	reserved	reserved
INVOKER	non-reserved	non-reserved	non-reserved	
IS	reserved (can be function)	reserved	reserved	reserved
ISNULL	reserved (can be function)			
ISOLATION	non-reserved	non-reserved	reserved	reserved
ITERATE			reserved	
JOIN	reserved (can be function)	reserved	reserved	reserved
K		non-reserved	non-reserved	
KEY	non-reserved	non-reserved	reserved	reserved
KEY_MEMBER		non-reserved	non-reserved	
KEY_TYPE		non-reserved	non-reserved	
LANCOMPILER	non-reserved			
LANGUAGE	non-reserved	reserved	reserved	reserved
LARGE	non-reserved	reserved	reserved	
LAST	non-reserved	non-reserved	reserved	reserved
LATERAL		reserved	reserved	
LEADING	reserved	reserved	reserved	reserved
LEAST	non-reserved (cannot be function or type)			

Key Word	PostgreSQL	SQL:2003	SQL:1999	SQL-92
LEFT	reserved (can be function)	reserved	reserved	reserved
LENGTH		non-reserved	non-reserved	non-reserved
LESS			reserved	
LEVEL	non-reserved	non-reserved	reserved	reserved
LIKE	reserved (can be function)	reserved	reserved	reserved
LIMIT	reserved		reserved	
LISTEN	non-reserved			
LN		reserved		
LOAD	non-reserved			
LOCAL	non-reserved	reserved	reserved	reserved
LOCALTIME	reserved	reserved	reserved	
LOCALTIMESTAMP	reserved	reserved	reserved	
LOCATION	non-reserved			
LOCATOR		non-reserved	reserved	
LOCK	non-reserved			
LOGIN	non-reserved			
LOWER		reserved	non-reserved	reserved
M		non-reserved	non-reserved	
MAP		non-reserved	reserved	
MATCH	non-reserved	reserved	reserved	reserved
MATCHED		non-reserved		
MAX		reserved	non-reserved	reserved
MAXVALUE	non-reserved	non-reserved		
MEMBER		reserved		
MERGE		reserved		
MESSAGE_LENGTH		non-reserved	non-reserved	non-reserved
MESSAGE_OCTET_LENGTH		non-reserved	non-reserved	non-reserved
MESSAGE_TEXT		non-reserved	non-reserved	non-reserved
METHOD		reserved	non-reserved	
MIN		reserved	non-reserved	reserved
MINUTE	non-reserved	reserved	reserved	reserved
MINVALUE	non-reserved	non-reserved		
MOD		reserved	non-reserved	
MODE	non-reserved			
MODIFIES		reserved	reserved	
MODIFY			reserved	

Key Word	PostgreSQL	SQL:2003	SQL:1999	SQL-92
MODULE		reserved	reserved	reserved
MONTH	non-reserved	reserved	reserved	reserved
MORE		non-reserved	non-reserved	non-reserved
MOVE	non-reserved			
MULTISET		reserved		
MUMPS		non-reserved	non-reserved	non-reserved
NAME		non-reserved	non-reserved	non-reserved
NAMES	non-reserved	non-reserved	reserved	reserved
NATIONAL	non-reserved (cannot be function or type)	reserved	reserved	reserved
NATURAL	reserved (can be function)	reserved	reserved	reserved
NCHAR	non-reserved (cannot be function or type)	reserved	reserved	reserved
NCLOB		reserved	reserved	
NESTING		non-reserved		
NEW	reserved	reserved	reserved	
NEXT	non-reserved	non-reserved	reserved	reserved
NO	non-reserved	reserved	reserved	reserved
NOCREATEDB	non-reserved			
NOCREATEROLE	non-reserved			
NOCREATEUSER	non-reserved			
NOINHERIT	non-reserved			
NOLOGIN	non-reserved			
NONE	non-reserved (cannot be function or type)	reserved	reserved	
NORMALIZE		reserved		
NORMALIZED		non-reserved		
NOSUPERUSER	non-reserved			
NOT	reserved	reserved	reserved	reserved
NOTHING	non-reserved			
NOTIFY	non-reserved			
NOTNULL	reserved (can be function)			
NOWAIT	non-reserved			
NULL	reserved	reserved	reserved	reserved
NULLABLE		non-reserved	non-reserved	non-reserved

Key Word	PostgreSQL	SQL:2003	SQL:1999	SQL-92
NULLIF	non-reserved (cannot be function or type)	reserved	non-reserved	reserved
NULLS		non-reserved		
NUMBER		non-reserved	non-reserved	non-reserved
NUMERIC	non-reserved (cannot be function or type)	reserved	reserved	reserved
OBJECT	non-reserved	non-reserved	reserved	
OCTETS		non-reserved		
OCTET_LENGTH		reserved	non-reserved	reserved
OF	non-reserved	reserved	reserved	reserved
OFF	reserved		reserved	
OFFSET	reserved			
OIDS	non-reserved			
OLD	reserved	reserved	reserved	
ON	reserved	reserved	reserved	reserved
ONLY	reserved	reserved	reserved	reserved
OPEN		reserved	reserved	reserved
OPERATION			reserved	
OPERATOR	non-reserved			
OPTION	non-reserved	non-reserved	reserved	reserved
OPTIONS		non-reserved	non-reserved	
OR	reserved	reserved	reserved	reserved
ORDER	reserved	reserved	reserved	reserved
ORDERING		non-reserved		
ORDINALITY		non-reserved	reserved	
OTHERS		non-reserved		
OUT	non-reserved (cannot be function or type)	reserved	reserved	
OUTER	reserved (can be function)	reserved	reserved	reserved
OUTPUT		non-reserved	reserved	reserved
OVER		reserved		
OVERLAPS	reserved (can be function)	reserved	non-reserved	reserved
OVERLAY	non-reserved (cannot be function or type)	reserved	non-reserved	

Key Word	PostgreSQL	SQL:2003	SQL:1999	SQL-92
OVERRIDING		non-reserved	non-reserved	
OWNED	non-reserved			
OWNER	non-reserved			
PAD		non-reserved	reserved	reserved
PARAMETER		reserved	reserved	
PARAMETERS			reserved	
PARAMETER_MODE		non-reserved	non-reserved	
PARAMETER_NAME		non-reserved	non-reserved	
PARAMETER_ORDINAL_POSITION		non-reserved	non-reserved	
PARAMETER_SPECIFIC_CATALOG		non-reserved	non-reserved	
PARAMETER_SPECIFIC_NAME		non-reserved	non-reserved	
PARAMETER_SPECIFIC_SCHEMA		non-reserved	non-reserved	
PARTIAL	non-reserved	non-reserved	reserved	reserved
PARTITION		reserved		
PASCAL		non-reserved	non-reserved	non-reserved
PASSWORD	non-reserved			
PATH		non-reserved	reserved	
PERCENTILE_CONT		reserved		
PERCENTILE_DISC		reserved		
PERCENT_RANK		reserved		
PLACING	reserved	non-reserved		
PLI		non-reserved	non-reserved	non-reserved
POSITION	non-reserved (cannot be function or type)	reserved	non-reserved	reserved
POSTFIX			reserved	
POWER		reserved		
PRECEDING		non-reserved		
PRECISION	non-reserved (cannot be function or type)	reserved	reserved	reserved
PREFIX			reserved	
PREORDER			reserved	
PREPARE	non-reserved	reserved	reserved	reserved

Key Word	PostgreSQL	SQL:2003	SQL:1999	SQL-92
PREPARED	non-reserved			
PRESERVE	non-reserved	non-reserved	reserved	reserved
PRIMARY	reserved	reserved	reserved	reserved
PRIOR	non-reserved	non-reserved	reserved	reserved
PRIVILEGES	non-reserved	non-reserved	reserved	reserved
PROCEDURAL	non-reserved			
PROCEDURE	non-reserved	reserved	reserved	reserved
PUBLIC		non-reserved	reserved	reserved
QUOTE	non-reserved			
RANGE		reserved		
RANK		reserved		
READ	non-reserved	non-reserved	reserved	reserved
READS		reserved	reserved	
REAL	non-reserved (cannot be function or type)	reserved	reserved	reserved
REASSIGN	non-reserved			
RECHECK	non-reserved			
RECURSIVE		reserved	reserved	
REF		reserved	reserved	
REFERENCES	reserved	reserved	reserved	reserved
REFERENCING		reserved	reserved	
REGR_AVGX		reserved		
REGR_AVGY		reserved		
REGR_COUNT		reserved		
REGR_INTERCEPT		reserved		
REGR_R2		reserved		
REGR_SLOPE		reserved		
REGR_SXX		reserved		
REGR_SXY		reserved		
REGR_SYY		reserved		
REINDEX	non-reserved			
RELATIVE	non-reserved	non-reserved	reserved	reserved
RELEASE	non-reserved	reserved		
RENAME	non-reserved			
REPEATABLE	non-reserved	non-reserved	non-reserved	non-reserved
REPLACE	non-reserved			
RESET	non-reserved			
RESTART	non-reserved	non-reserved		

Key Word	PostgreSQL	SQL:2003	SQL:1999	SQL-92
RESTRICT	non-reserved	non-reserved	reserved	reserved
RESULT		reserved	reserved	
RETURN		reserved	reserved	
RETURNED_CARDINALITY		non-reserved		
RETURNED_LENGTH		non-reserved	non-reserved	non-reserved
RETURNED_OCTET_LENGTH		non-reserved	non-reserved	non-reserved
RETURNED_SQLSTATE		non-reserved	non-reserved	non-reserved
RETURNING	reserved			
RETURNS	non-reserved	reserved	reserved	
REVOKE	non-reserved	reserved	reserved	reserved
RIGHT	reserved (can be function)	reserved	reserved	reserved
ROLE	non-reserved	non-reserved	reserved	
ROLLBACK	non-reserved	reserved	reserved	reserved
ROLLUP		reserved	reserved	
ROUTINE		non-reserved	reserved	
ROUTINE_CATALOG		non-reserved	non-reserved	
ROUTINE_NAME		non-reserved	non-reserved	
ROUTINE_SCHEMA		non-reserved	non-reserved	
ROW	non-reserved (cannot be function or type)	reserved	reserved	
ROWS	non-reserved	reserved	reserved	reserved
ROW_COUNT		non-reserved	non-reserved	non-reserved
ROW_NUMBER		reserved		
RULE	non-reserved			
SAVEPOINT	non-reserved	reserved	reserved	
SCALE		non-reserved	non-reserved	non-reserved
SCHEMA	non-reserved	non-reserved	reserved	reserved
SCHEMA_NAME		non-reserved	non-reserved	non-reserved
SCOPE		reserved	reserved	
SCOPE_CATALOG		non-reserved		
SCOPE_NAME		non-reserved		
SCOPE_SCHEMA		non-reserved		
SCROLL	non-reserved	reserved	reserved	reserved

Key Word	PostgreSQL	SQL:2003	SQL:1999	SQL-92
SEARCH		reserved	reserved	
SECOND	non-reserved	reserved	reserved	reserved
SECTION		non-reserved	reserved	reserved
SECURITY	non-reserved	non-reserved	non-reserved	
SELECT	reserved	reserved	reserved	reserved
SELF		non-reserved	non-reserved	
SENSITIVE		reserved	non-reserved	
SEQUENCE	non-reserved	non-reserved	reserved	
SERIALIZABLE	non-reserved	non-reserved	non-reserved	non-reserved
SERVER_NAME		non-reserved	non-reserved	non-reserved
SESSION	non-reserved	non-reserved	reserved	reserved
SESSION_USER	reserved	reserved	reserved	reserved
SET	non-reserved	reserved	reserved	reserved
SETOF	non-reserved (cannot be function or type)			
SETS		non-reserved	reserved	
SHARE	non-reserved			
SHOW	non-reserved			
SIMILAR	reserved (can be function)	reserved	non-reserved	
SIMPLE	non-reserved	non-reserved	non-reserved	
SIZE		non-reserved	reserved	reserved
SMALLINT	non-reserved (cannot be function or type)	reserved	reserved	reserved
SOME	reserved	reserved	reserved	reserved
SOURCE		non-reserved	non-reserved	
SPACE		non-reserved	reserved	reserved
SPECIFIC		reserved	reserved	
SPECIFICTYPE		reserved	reserved	
SPECIFIC_NAME		non-reserved	non-reserved	
SQL		reserved	reserved	reserved
SQLCODE				reserved
SQLERROR				reserved
SQLEXCEPTION		reserved	reserved	
SQLSTATE		reserved	reserved	reserved
SQLWARNING		reserved	reserved	
SQRT		reserved		

Key Word	PostgreSQL	SQL:2003	SQL:1999	SQL-92
STABLE	non-reserved			
START	non-reserved	reserved	reserved	
STATE		non-reserved	reserved	
STATEMENT	non-reserved	non-reserved	reserved	
STATIC		reserved	reserved	
STATISTICS	non-reserved			
STDDEV_POP		reserved		
STDDEV_SAMP		reserved		
STDIN	non-reserved			
STDOUT	non-reserved			
STORAGE	non-reserved			
STRICT	non-reserved			
STRUCTURE		non-reserved	reserved	
STYLE		non-reserved	non-reserved	
SUBCLASS_ORIGIN		non-reserved	non-reserved	non-reserved
SUBLIST			non-reserved	
SUBMULTISET		reserved		
SUBSTRING	non-reserved (cannot be function or type)	reserved	non-reserved	reserved
SUM		reserved	non-reserved	reserved
SUPERUSER	non-reserved			
SYMMETRIC	reserved	reserved	non-reserved	
SYSID	non-reserved			
SYSTEM	non-reserved	reserved	non-reserved	
SYSTEM_USER		reserved	reserved	reserved
TABLE	reserved	reserved	reserved	reserved
TABLESAMPLE		reserved		
TABLESPACE	non-reserved			
TABLE_NAME		non-reserved	non-reserved	non-reserved
TEMP	non-reserved			
TEMPLATE	non-reserved			
TEMPORARY	non-reserved	non-reserved	reserved	reserved
TERMINATE			reserved	
THAN			reserved	
THEN	reserved	reserved	reserved	reserved
TIES		non-reserved		

Key Word	PostgreSQL	SQL:2003	SQL:1999	SQL-92
TIME	non-reserved (cannot be function or type)	reserved	reserved	reserved
TIMESTAMP	non-reserved (cannot be function or type)	reserved	reserved	reserved
TIMEZONE_HOUR		reserved	reserved	reserved
TIMEZONE_MINUTE		reserved	reserved	reserved
TO	reserved	reserved	reserved	reserved
TOP_LEVEL_COUNT		non-reserved		
TRAILING	reserved	reserved	reserved	reserved
TRANSACTION	non-reserved	non-reserved	reserved	reserved
TRANSACTIONS_COMMITTED		non-reserved	non-reserved	
TRANSACTIONS_ROLLED_BACK		non-reserved	non-reserved	
TRANSACTION_ACTIVE		non-reserved	non-reserved	
TRANSFORM		non-reserved	non-reserved	
TRANSFORMS		non-reserved	non-reserved	
TRANSLATE		reserved	non-reserved	reserved
TRANSLATION		reserved	reserved	reserved
TREAT	non-reserved (cannot be function or type)	reserved	reserved	
TRIGGER	non-reserved	reserved	reserved	
TRIGGER_CATALOG		non-reserved	non-reserved	
TRIGGER_NAME		non-reserved	non-reserved	
TRIGGER_SCHEMA		non-reserved	non-reserved	
TRIM	non-reserved (cannot be function or type)	reserved	non-reserved	reserved
TRUE	reserved	reserved	reserved	reserved
TRUNCATE	non-reserved			
TRUSTED	non-reserved			
TYPE	non-reserved	non-reserved	non-reserved	non-reserved
UESCAPE		reserved		
UNBOUNDED		non-reserved		

Key Word	PostgreSQL	SQL:2003	SQL:1999	SQL-92
UNCOMMITTED	non-reserved	non-reserved	non-reserved	non-reserved
UNDER		non-reserved	reserved	
UNENCRYPTED	non-reserved			
UNION	reserved	reserved	reserved	reserved
UNIQUE	reserved	reserved	reserved	reserved
UNKNOWN	non-reserved	reserved	reserved	reserved
UNLISTEN	non-reserved			
UNNAMED		non-reserved	non-reserved	non-reserved
UNNEST		reserved	reserved	
UNTIL	non-reserved			
UPDATE	non-reserved	reserved	reserved	reserved
UPPER		reserved	non-reserved	reserved
USAGE		non-reserved	reserved	reserved
USER	reserved	reserved	reserved	reserved
USER_DEFINED_TYPE_CATALOG		non-reserved	non-reserved	
USER_DEFINED_TYPE_CODE		non-reserved		
USER_DEFINED_TYPE_NAME		non-reserved	non-reserved	
USER_DEFINED_TYPE_SCHEMA		non-reserved	non-reserved	
USING	reserved	reserved	reserved	reserved
VACUUM	non-reserved			
VALID	non-reserved			
VALIDATOR	non-reserved			
VALUE		reserved	reserved	reserved
VALUES	non-reserved (cannot be function or type)	reserved	reserved	reserved
VARCHAR	non-reserved (cannot be function or type)	reserved	reserved	reserved
VARIABLE			reserved	
VARYING	non-reserved	reserved	reserved	reserved
VAR_POP		reserved		
VAR_SAMP		reserved		
VERBOSE	reserved (can be function)			
VIEW	non-reserved	non-reserved	reserved	reserved

Key Word	PostgreSQL	SQL:2003	SQL:1999	SQL-92
VOLATILE	non-reserved			
WHEN	reserved	reserved	reserved	reserved
WHENEVER		reserved	reserved	reserved
WHERE	reserved	reserved	reserved	reserved
WIDTH_BUCKET		reserved		
WINDOW		reserved		
WITH	non-reserved	reserved	reserved	reserved
WITHIN		reserved		
WITHOUT	non-reserved	reserved	reserved	
WORK	non-reserved	non-reserved	reserved	reserved
WRITE	non-reserved	non-reserved	reserved	reserved
YEAR	non-reserved		reserved	reserved
ZONE	non-reserved	non-reserved	reserved	reserved

Appendix D. SQL Conformance

This section attempts to outline to what extent PostgreSQL conforms to the current SQL standard. The following information is not a full statement of conformance, but it presents the main topics in as much detail as is both reasonable and useful for users.

The formal name of the SQL standard is ISO/IEC 9075 “Database Language SQL”. A revised version of the standard is released from time to time; the most recent one appearing in late 2003. That version is referred to as ISO/IEC 9075:2003, or simply as SQL:2003. The versions prior to that were SQL:1999 and SQL-92. Each version replaces the previous one, so claims of conformance to earlier versions have no official merit. PostgreSQL development aims for conformance with the latest official version of the standard where such conformance does not contradict traditional features or common sense. The PostgreSQL project was not represented in the ISO/IEC 9075 Working Group during the preparation of SQL:2003. Even so, many of the features required by SQL:2003 are already supported, though sometimes with slightly differing syntax or function. Further moves towards conformance may be expected in later releases.

SQL-92 defined three feature sets for conformance: Entry, Intermediate, and Full. Most database management systems claiming SQL standard conformance were conforming at only the Entry level, since the entire set of features in the Intermediate and Full levels was either too voluminous or in conflict with legacy behaviors.

Starting with SQL:1999, the SQL standard defines a large set of individual features rather than the ineffectively broad three levels found in SQL-92. A large subset of these features represents the “Core” features, which every conforming SQL implementation must supply. The rest of the features are purely optional. Some optional features are grouped together to form “packages”, which SQL implementations can claim conformance to, thus claiming conformance to particular groups of features.

The SQL:2003 standard is also split into a number of parts. Each is known by a shorthand name. Note that these parts are not consecutively numbered.

- ISO/IEC 9075-1 Framework (SQL/Framework)
- ISO/IEC 9075-2 Foundation (SQL/Foundation)
- ISO/IEC 9075-3 Call Level Interface (SQL/CLI)
- ISO/IEC 9075-4 Persistent Stored Modules (SQL/PSM)
- ISO/IEC 9075-9 Management of External Data (SQL/MED)
- ISO/IEC 9075-10 Object Language Bindings (SQL/OLB)
- ISO/IEC 9075-11 Information and Definition Schemas (SQL/Schemata)
- ISO/IEC 9075-13 Routines and Types using the Java Language (SQL/JRT)
- ISO/IEC 9075-14 XML-related specifications (SQL/XML)

PostgreSQL covers parts 1, 2, and 11. Part 3 is similar to the ODBC interface, and part 4 is similar to the PL/pgSQL programming language, but exact conformance is not specifically intended or verified in either case.

PostgreSQL supports most of the major features of SQL:2003. Out of 164 mandatory features required for full Core conformance, PostgreSQL conforms to at least 150. In addition, there is a long list of supported optional features. It may be worth noting that at the time of writing, no current version of any database management system claims full conformance to Core SQL:2003.

In the following two sections, we provide a list of those features that PostgreSQL supports, followed by a list of the features defined in SQL:2003 which are not yet supported in PostgreSQL. Both of these lists are approximate: There may be minor details that are nonconforming for a feature that is listed as supported, and large parts of an unsupported feature may in fact be implemented. The main body of the documentation always contains the most accurate information about what does and does not work.

> **Note:** Feature codes containing a hyphen are subfeatures. Therefore, if a particular subfeature is not supported, the main feature is listed as unsupported even if some other subfeatures are supported.

D.1. Supported Features

Identifier	Package	Description	Comment
B012		Embedded C	
B021		Direct SQL	
E011	Core	Numeric data types	
E011-01	Core	INTEGER and SMALLINT data types	
E011-02	Core	REAL, DOUBLE PRECISION, and FLOAT data types	
E011-03	Core	DECIMAL and NUMERIC data types	
E011-04	Core	Arithmetic operators	
E011-05	Core	Numeric comparison	
E011-06	Core	Implicit casting among the numeric data types	
E021	Core	Character data types	
E021-01	Core	CHARACTER data type	
E021-02	Core	CHARACTER VARYING data type	
E021-03	Core	Character literals	

Identifier	Package	Description	Comment
E021-04	Core	CHARACTER_LENGTH function	trims trailing spaces from CHARACTER values before counting
E021-05	Core	OCTET_LENGTH function	
E021-06	Core	SUBSTRING function	
E021-07	Core	Character concatenation	
E021-08	Core	UPPER and LOWER functions	
E021-09	Core	TRIM function	
E021-10	Core	Implicit casting among the character string types	
E021-11	Core	POSITION function	
E021-12	Core	Character comparison	
E031	Core	Identifiers	
E031-01	Core	Delimited identifiers	
E031-02	Core	Lower case identifiers	
E031-03	Core	Trailing underscore	
E051	Core	Basic query specification	
E051-01	Core	SELECT DISTINCT	
E051-02	Core	GROUP BY clause	
E051-04	Core	GROUP BY can contain columns not in <select list>	
E051-05	Core	Select list items can be renamed	AS is required
E051-06	Core	HAVING clause	
E051-07	Core	Qualified * in select list	
E051-08	Core	Correlation names in the FROM clause	
E051-09	Core	Rename columns in the FROM clause	
E061	Core	Basic predicates and search conditions	
E061-01	Core	Comparison predicate	
E061-02	Core	BETWEEN predicate	
E061-03	Core	IN predicate with list of values	
E061-04	Core	LIKE predicate	

Identifier	Package	Description	Comment
E061-05	Core	LIKE predicate ESCAPE clause	
E061-06	Core	NULL predicate	
E061-07	Core	Quantified comparison predicate	
E061-08	Core	EXISTS predicate	
E061-09	Core	Subqueries in comparison predicate	
E061-11	Core	Subqueries in IN predicate	
E061-12	Core	Subqueries in quantified comparison predicate	
E061-13	Core	Correlated subqueries	
E061-14	Core	Search condition	
E071	Core	Basic query expressions	
E071-01	Core	UNION DISTINCT table operator	
E071-02	Core	UNION ALL table operator	
E071-03	Core	EXCEPT DISTINCT table operator	
E071-05	Core	Columns combined via table operators need not have exactly the same data type	
E071-06	Core	Table operators in subqueries	
E081-01	Core	SELECT privilege	
E081-02	Core	DELETE privilege	
E081-03	Core	INSERT privilege at the table level	
E081-04	Core	UPDATE privilege at the table level	
E081-06	Core	REFERENCES privilege at the table level	
E081-08	Core	WITH GRANT OPTION	
E081-10	Core	EXECUTE privilege	
E091	Core	Set functions	
E091-01	Core	AVG	
E091-02	Core	COUNT	

Identifier	Package	Description	Comment
E091-03	Core	MAX	
E091-04	Core	MIN	
E091-05	Core	SUM	
E091-06	Core	ALL quantifier	
E091-07	Core	DISTINCT quantifier	
E101	Core	Basic data manipulation	
E101-01	Core	INSERT statement	
E101-03	Core	Searched UPDATE statement	
E101-04	Core	Searched DELETE statement	
E111	Core	Single row SELECT statement	
E121-01	Core	DECLARE CURSOR	
E121-02	Core	ORDER BY columns need not be in select list	
E121-03	Core	Value expressions in ORDER BY clause	
E121-04	Core	OPEN statement	
E121-08	Core	CLOSE statement	
E121-10	Core	FETCH statement implicit NEXT	
E121-17	Core	WITH HOLD cursors	
E131	Core	Null value support (nulls in lieu of values)	
E141	Core	Basic integrity constraints	
E141-01	Core	NOT NULL constraints	
E141-02	Core	UNIQUE constraints of NOT NULL columns	
E141-03	Core	PRIMARY KEY constraints	
E141-04	Core	Basic FOREIGN KEY constraint with the NO ACTION default for both referential delete action and referential update action	
E141-06	Core	CHECK constraints	
E141-07	Core	Column defaults	
E141-08	Core	NOT NULL inferred on PRIMARY KEY	

Identifier	Package	Description	Comment
E141-10	Core	Names in a foreign key can be specified in any order	
E151	Core	Transaction support	
E151-01	Core	COMMIT statement	
E151-02	Core	ROLLBACK statement	
E152	Core	Basic SET TRANSACTION statement	
E152-01	Core	SET TRANSACTION statement: ISOLATION LEVEL SERIALIZABLE clause	
E152-02	Core	SET TRANSACTION statement: READ ONLY and READ WRITE clauses	
E161	Core	SQL comments using leading double minus	
E171	Core	SQLSTATE support	
F021	Core	Basic information schema	
F021-01	Core	COLUMNS view	
F021-02	Core	TABLES view	
F021-03	Core	VIEWS view	
F021-04	Core	TABLE_CONSTRAINTS view	
F021-05	Core	REFERENTIAL_CONSTRAINTS view	
F021-06	Core	CHECK_CONSTRAINTS view	
F031	Core	Basic schema manipulation	
F031-01	Core	CREATE TABLE statement to create persistent base tables	
F031-02	Core	CREATE VIEW statement	
F031-03	Core	GRANT statement	

Identifier	Package	Description	Comment
F031-04	Core	ALTER TABLE statement: ADD COLUMN clause	
F031-13	Core	DROP TABLE statement: RESTRICT clause	
F031-16	Core	DROP VIEW statement: RESTRICT clause	
F031-19	Core	REVOKE statement: RESTRICT clause	
F032		CASCADE drop behavior	
F033		ALTER TABLE statement: DROP COLUMN clause	
F034		Extended REVOKE statement	
F034-01		REVOKE statement performed by other than the owner of a schema object	
F034-02		REVOKE statement: GRANT OPTION FOR clause	
F034-03		REVOKE statement to revoke a privilege that the grantee has WITH GRANT OPTION	
F041	Core	Basic joined table	
F041-01	Core	Inner join (but not necessarily the INNER keyword)	
F041-02	Core	INNER keyword	
F041-03	Core	LEFT OUTER JOIN	
F041-04	Core	RIGHT OUTER JOIN	
F041-05	Core	Outer joins can be nested	
F041-07	Core	The inner table in a left or right outer join can also be used in an inner join	

Identifier	Package	Description	Comment
F041-08	Core	All comparison operators are supported (rather than just =)	
F051	Core	Basic date and time	
F051-01	Core	DATE data type (including support of DATE literal)	
F051-02	Core	TIME data type (including support of TIME literal) with fractional seconds precision of at least 0	
F051-03	Core	TIMESTAMP data type (including support of TIMESTAMP literal) with fractional seconds precision of at least 0 and 6	
F051-04	Core	Comparison predicate on DATE, TIME, and TIMESTAMP data types	
F051-05	Core	Explicit CAST between datetime types and character string types	
F051-06	Core	CURRENT_DATE	
F051-07	Core	LOCALTIME	
F051-08	Core	LOCALTIMESTAMP	
F052	Enhanced datetime facilities	Intervals and datetime arithmetic	
F053		OVERLAPS predicate	
F081	Core	UNION and EXCEPT in views	
F111		Isolation levels other than SERIALIZABLE	
F111-01		READ UNCOMMITTED isolation level	
F111-02		READ COMMITTED isolation level	
F111-03		REPEATABLE READ isolation level	
F131	Core	Grouped operations	

Identifier	Package	Description	Comment
F131-01	Core	WHERE, GROUP BY, and HAVING clauses supported in queries with grouped views	
F131-02	Core	Multiple tables supported in queries with grouped views	
F131-03	Core	Set functions supported in queries with grouped views	
F131-04	Core	Subqueries with GROUP BY and HAVING clauses and grouped views	
F131-05	Core	Single row SELECT with GROUP BY and HAVING clauses and grouped views	
F171		Multiple schemas per user	
F191	Enhanced integrity management	Referential delete actions	
F201	Core	CAST function	
F221	Core	Explicit defaults	
F222		INSERT statement: DEFAULT VALUES clause	
F231		Privilege tables	
F231-01		TABLE_PRIVILEGES view	
F231-02		COLUMN_PRIVILEGES view	
F231-03		USAGE_PRIVILEGES view	
F251		Domain support	
F261	Core	CASE expression	
F261-01	Core	Simple CASE	
F261-02	Core	Searched CASE	
F261-03	Core	NULLIF	
F261-04	Core	COALESCE	
F271		Compound character literals	

Identifier	Package	Description	Comment
F281		LIKE enhancements	
F302		INTERSECT table operator	
F302-01		INTERSECT DISTINCT table operator	
F302-02		INTERSECT ALL table operator	
F304		EXCEPT ALL table operator	
F311-01	Core	CREATE SCHEMA	
F311-02	Core	CREATE TABLE for persistent base tables	
F311-03	Core	CREATE VIEW	
F311-05	Core	GRANT statement	
F321		User authorization	
F361		Subprogram support	
F381		Extended schema manipulation	
F381-01		ALTER TABLE statement: ALTER COLUMN clause	
F381-02		ALTER TABLE statement: ADD CONSTRAINT clause	
F381-03		ALTER TABLE statement: DROP CONSTRAINT clause	
F391		Long identifiers	
F401		Extended joined table	
F401-01		NATURAL JOIN	
F401-02		FULL OUTER JOIN	
F401-04		CROSS JOIN	
F411	Enhanced datetime facilities	Time zone specification	differences regarding literal interpretation
F421		National character	
F431		Read-only scrollable cursors	
F431-01		FETCH with explicit NEXT	
F431-02		FETCH FIRST	
F431-03		FETCH LAST	

Identifier	Package	Description	Comment
F431-04		FETCH PRIOR	
F431-05		FETCH ABSOLUTE	
F431-06		FETCH RELATIVE	
F441		Extended set function support	
F471	Core	Scalar subquery values	
F481	Core	Expanded NULL predicate	
F491	Enhanced integrity management	Constraint management	
F501	Core	Features and conformance views	
F501-01	Core	SQL_FEATURES view	
F501-02	Core	SQL_SIZING view	
F501-03	Core	SQL_LANGUAGES view	
F502		Enhanced documentation tables	
F502-01		SQL_SIZING_PROFILES view	
F502-02		SQL_IMPLEMENTATION_INFO view	
F502-03		SQL_PACKAGES view	
F531		Temporary tables	
F555	Enhanced datetime facilities	Enhanced seconds precision	
F561		Full value expressions	
F571		Truth value tests	
F591		Derived tables	
F611		Indicator data types	
F651		Catalog name qualifiers	
F672		Retrospective check constraints	
F701	Enhanced integrity management	Referential update actions	
F711		ALTER domain	
F761		Session management	
F771		Connection management	
F781		Self-referencing operations	

Identifier	Package	Description	Comment
F791		Insensitive cursors	
F801		Full set function	
S071	Enhanced object support	SQL paths in function and type name resolution	
S111	Enhanced object support	ONLY in query expressions	
S211	Enhanced object support	User-defined cast functions	
T031		BOOLEAN data type	
T071		BIGINT data type	
T141		SIMILAR predicate	
T151		DISTINCT predicate	
T171		LIKE clause in table definition	
T191	Enhanced integrity management	Referential action RESTRICT	
T201	Enhanced integrity management	Comparable data types for referential constraints	
T211-01	Active database, Enhanced integrity management	Triggers activated on UPDATE, INSERT, or DELETE of one base table	
T211-02	Active database, Enhanced integrity management	BEFORE triggers	
T211-03	Active database, Enhanced integrity management	AFTER triggers	
T211-04	Active database, Enhanced integrity management	FOR EACH ROW triggers	
T211-07	Active database, Enhanced integrity management	TRIGGER privilege	
T212	Enhanced integrity management	Enhanced trigger capability	
T231		Sensitive cursors	
T241		START TRANSACTION statement	

Identifier	Package	Description	Comment
T271		Savepoints	
T312		OVERLAY function	
T321-01	Core	User-defined functions with no overloading	
T321-03	Core	Function invocation	
T321-06	Core	ROUTINES view	
T321-07	Core	PARAMETERS view	
T322	PSM	Overloading of SQL-invoked functions and procedures	
T323		Explicit security for external routines	
T351		Bracketed SQL comments (/*...*/ comments)	
T441		ABS and MOD functions	
T461		Symmetric BETWEEN predicate	
T501		Enhanced EXISTS predicate	
T551		Optional key words for default syntax	
T581		Regular expression substring function	
T591		UNIQUE constraints of possibly null columns	

D.2. Unsupported Features

The following features defined in SQL:2003 are not implemented in this release of PostgreSQL. In a few cases, equivalent functionality is available.

Identifier	Package	Description	Comment
B011		Embedded Ada	
B013		Embedded COBOL	
B014		Embedded Fortran	
B015		Embedded MUMPS	
B016		Embedded Pascal	

Identifier	Package	Description	Comment
B017		Embedded PL/I	
B031		Basic dynamic SQL	
B032		Extended dynamic SQL	
B032-01		<describe input statement>	
B033		Untyped SQL-invoked function arguments	
B034		Dynamic specification of cursor attributes	
B041		Extensions to embedded SQL exception declarations	
B051		Enhanced execution rights	
B111		Module language Ada	
B112		Module language C	
B113		Module language COBOL	
B114		Module language Fortran	
B115		Module language MUMPS	
B116		Module language Pascal	
B117		Module language PL/I	
B121		Routine language Ada	
B122		Routine language C	
B123		Routine language COBOL	
B124		Routine language Fortran	
B125		Routine language MUMPS	
B126		Routine language Pascal	
B127		Routine language PL/I	
B128		Routine language SQL	
C011	Core	Call-Level Interface	
E081	Core	Basic Privileges	
E081-05	Core	UPDATE privilege at the column level	

Identifier	Package	Description	Comment
E081-07	Core	REFERENCES privilege at the column level	
E081-09	Core	USAGE privilege	
E121	Core	Basic cursor support	
E121-06	Core	Positioned UPDATE statement	
E121-07	Core	Positioned DELETE statement	
E153	Core	Updatable queries with subqueries	
E182	Core	Module language	
F121		Basic diagnostics management	
F121-01		GET DIAGNOSTICS statement	
F121-02		SET TRANSACTION statement: DIAGNOSTICS SIZE clause	
F181	Core	Multiple module support	
F262		Extended CASE expression	
F263		Comma-separated predicates in simple CASE expression	
F291		UNIQUE predicate	
F301		CORRESPONDING in query expressions	
F311	Core	Schema definition statement	
F311-04	Core	CREATE VIEW: WITH CHECK OPTION	
F312		MERGE statement	
F341		Usage tables	
F392		Unicode escapes in identifiers	
F393		Unicode escapes in literals	

Identifier	Package	Description	Comment
F402		Named column joins for LOBs, arrays, and multisets	
F442		Mixed column references in set functions	
F451		Character set definition	
F461		Named character sets	
F521	Enhanced integrity management	Assertions	
F641		Row and table constructors	
F661		Simple tables	
F671	Enhanced integrity management	Subqueries in CHECK	intentionally omitted
F691		Collation and translation	
F692		Enhanced collation support	
F693		SQL-session and client module collations	
F695		Translation support	
F696		Additional translation documentation	
F721		Deferrable constraints	foreign keys only
F731		INSERT column privileges	
F741		Referential MATCH types	no partial match yet
F751		View CHECK enhancements	
F811		Extended flagging	
F812	Core	Basic flagging	
F813		Extended flagging	
F821		Local table references	
F831		Full cursor update	
F831-01		Updatable scrollable cursors	
F831-02		Updatable ordered cursors	
S011	Core	Distinct data types	

Identifier	Package	Description	Comment
S011-01	Core	USER_DEFINED_TYPES view	
S023	Basic object support	Basic structured types	
S024	Enhanced object support	Enhanced structured types	
S025		Final structured types	
S026		Self-referencing structured types	
S027		Create method by specific method name	
S028		Permutable UDT options list	
S041	Basic object support	Basic reference types	
S043	Enhanced object support	Enhanced reference types	
S051	Basic object support	Create table of type	
S081	Enhanced object support	Subtables	
S091		Basic array support	
S091-01		Arrays of built-in data types	
S091-02		Arrays of distinct types	
S091-03		Array expressions	
S092		Arrays of user-defined types	
S094		Arrays of reference types	
S095		Array constructors by query	
S096		Optional array bounds	
S097		Array element assignment	
S151	Basic object support	Type predicate	
S161	Enhanced object support	Subtype treatment	
S162		Subtype treatment for references	
S201		SQL-invoked routines on arrays	
S201-01		Array parameters	
S201-02		Array as result type of functions	

Identifier	Package	Description	Comment
S202		SQL-invoked routines on multisets	
S231	Enhanced object support	Structured type locators	
S232		Array locators	
S233		Multiset locators	
S241		Transform functions	
S242		Alter transform statement	
S251		User-defined orderings	
S261		Specific type method	
S271		Basic multiset support	
S272		Multisets of user-defined types	
S274		Multisets of reference types	
S275		Advanced multiset support	
S281		Nested collection types	
S291		Unique constraint on entire row	
T011		Timestamp in Information Schema	
T041	Basic object support	Basic LOB data type support	
T041-01	Basic object support	BLOB data type	
T041-02	Basic object support	CLOB data type	
T041-03	Basic object support	POSITION, LENGTH, LOWER, TRIM, UPPER, and SUBSTRING functions for LOB data types	
T041-04	Basic object support	Concatenation of LOB data types	
T041-05	Basic object support	LOB locator: non-holdable	
T042		Extended LOB data type support	
T051		Row types	
T052		MAX and MIN for row types	

Identifier	Package	Description	Comment
T053		Explicit aliases for all-fields reference	
T061		UCS support	
T111		Updatable joins, unions, and columns	
T121		WITH (excluding RECURSIVE) in query expression	
T122		WITH (excluding RECURSIVE) in subquery	
T131		Recursive query	
T132		Recursive query in subquery	
T152		DISTINCT predicate with negation	
T172		AS subquery clause in table definition	
T173		Extended LIKE clause in table definition	
T174		Identity columns	
T175		Generated columns	
T176		Sequence generator support	
T211	Active database, Enhanced integrity management	Basic trigger capability	
T211-05	Active database, Enhanced integrity management	Ability to specify a search condition that must be true before the trigger is invoked	
T211-06	Active database, Enhanced integrity management	Support for run-time rules for the interaction of triggers and constraints	
T211-08	Active database, Enhanced integrity management	Multiple triggers for the same event are executed in the order in which they were created in the catalog	intentionally omitted
T251		SET TRANSACTION statement: LOCAL option	

Identifier	Package	Description	Comment
T261		Chained transactions	
T272		Enhanced savepoint management	
T281		SELECT privilege with column granularity	
T301		Functional dependencies	
T321	Core	Basic SQL-invoked routines	
T321-02	Core	User-defined stored procedures with no overloading	
T321-04	Core	CALL statement	
T321-05	Core	RETURN statement	
T324		Explicit security for SQL routines	
T325		Qualified SQL parameter references	
T326		Table functions	
T331		Basic roles	
T332		Extended roles	
T401		INSERT into a cursor	
T411		UPDATE statement: SET ROW option	
T431	OLAP	Extended grouping capabilities	
T432		Nested and concatenated GROUPING SETS	
T433		Multiargument GROUPING function	
T434		GROUP BY DISTINCT	
T471		Result sets return value	
T491		LATERAL derived table	
T511		Transaction counts	
T541		Updatable table references	
T561		Holdable locators	
T571		Array-returning external SQL-invoked functions	

Identifier	Package	Description	Comment
T572		Multiset-returning external SQL-invoked functions	
T601		Local cursor references	
T611	OLAP	Elementary OLAP operations	
T612		Advanced OLAP operations	
T613		Sampling	
T621		Enhanced numeric functions	
T631	Core	IN predicate with one list element	
T641		Multiple column assignment	
T651		SQL-schema statements in SQL routines	
T652		SQL-dynamic statements in SQL routines	
T653		SQL-schema statements in external routines	
T654		SQL-dynamic statements in external routines	
T655		Cyclically dependent routines	

Appendix E. Release Notes

E.1. Release 8.2.6

Release date: 2008-01-07

This release contains a variety of fixes from 8.2.5, including fixes for significant security issues.

E.1.1. Migration to Version 8.2.6

A dump/restore is not required for those running 8.2.X.

E.1.2. Changes

- Prevent functions in indexes from executing with the privileges of the user running `VACUUM`, `ANALYZE`, etc (Tom)

  Functions used in index expressions and partial-index predicates are evaluated whenever a new table entry is made. It has long been understood that this poses a risk of trojan-horse code execution if one modifies a table owned by an untrustworthy user. (Note that triggers, defaults, check constraints, etc. pose the same type of risk.) But functions in indexes pose extra danger because they will be executed by routine maintenance operations such as `VACUUM FULL`, which are commonly performed automatically under a superuser account. For example, a nefarious user can execute code with superuser privileges by setting up a trojan-horse index definition and waiting for the next routine vacuum. The fix arranges for standard maintenance operations (including `VACUUM`, `ANALYZE`, `REINDEX`, and `CLUSTER`) to execute as the table owner rather than the calling user, using the same privilege-switching mechanism already used for `SECURITY DEFINER` functions. To prevent bypassing this security measure, execution of `SET SESSION AUTHORIZATION` and `SET ROLE` is now forbidden within a `SECURITY DEFINER` context. (CVE-2007-6600)

- Repair assorted bugs in the regular-expression package (Tom, Will Drewry)

  Suitably crafted regular-expression patterns could cause crashes, infinite or near-infinite looping, and/or massive memory consumption, all of which pose denial-of-service hazards for applications that accept regex search patterns from untrustworthy sources. (CVE-2007-4769, CVE-2007-4772, CVE-2007-6067)

- Require non-superusers who use `/contrib/dblink` to use only password authentication, as a security measure (Joe)

The fix that appeared for this in 8.2.5 was incomplete, as it plugged the hole for only some `dblink` functions. (CVE-2007-6601, CVE-2007-3278)

- Fix bugs in WAL replay for GIN indexes (Teodor)
- Fix GIN index build to work properly when `maintenance_work_mem` is 4GB or more (Tom)
- Update time zone data files to tzdata release 2007k (in particular, recent Argentina changes) (Tom)
- Improve planner's handling of LIKE/regex estimation in non-C locales (Tom)
- Fix planning-speed problem for deep outer-join nests, as well as possible poor choice of join order (Tom)
- Fix planner failure in some cases of `WHERE false AND var IN (SELECT ...)` (Tom)
- Make `CREATE TABLE ... SERIAL` and `ALTER SEQUENCE ... OWNED BY` not change the `currval()` state of the sequence (Tom)
- Preserve the tablespace and storage parameters of indexes that are rebuilt by `ALTER TABLE ... ALTER COLUMN TYPE` (Tom)
- Make archive recovery always start a new WAL timeline, rather than only when a recovery stop time was used (Simon)

  This avoids a corner-case risk of trying to overwrite an existing archived copy of the last WAL segment, and seems simpler and cleaner than the original definition.
- Make `VACUUM` not use all of `maintenance_work_mem` when the table is too small for it to be useful (Alvaro)
- Fix potential crash in `translate()` when using a multibyte database encoding (Tom)
- Make `corr()` return the correct result for negative correlation values (Neil)
- Fix overflow in `extract(epoch from interval)` for intervals exceeding 68 years (Tom)
- Fix PL/Perl to not fail when a UTF-8 regular expression is used in a trusted function (Andrew)
- Fix PL/Perl to cope when platform's Perl defines type `bool` as `int` rather than `char` (Tom)

  While this could theoretically happen anywhere, no standard build of Perl did things this way ... until Mac OS X 10.5.
- Fix PL/Python to work correctly with Python 2.5 on 64-bit machines (Marko Kreen)
- Fix PL/Python to not crash on long exception messages (Alvaro)
- Fix pg_dump to correctly handle inheritance child tables that have default expressions different from their parent's (Tom)
- Fix libpq crash when `PGPASSFILE` refers to a file that is not a plain file (Martin Pitt)
- ecpg parser fixes (Michael)
- Make `contrib/pgcrypto` defend against OpenSSL libraries that fail on keys longer than 128 bits; which is the case at least on some Solaris versions (Marko Kreen)
- Make `contrib/tablefunc`'s `crosstab()` handle NULL rowid as a category in its own right, rather than crashing (Joe)
- Fix `tsvector` and `tsquery` output routines to escape backslashes correctly (Teodor, Bruce)
- Fix crash of `to_tsvector()` on huge input strings (Teodor)

- Require a specific version of Autoconf to be used when re-generating the `configure` script (Peter)

  This affects developers and packagers only. The change was made to prevent accidental use of untested combinations of Autoconf and PostgreSQL versions. You can remove the version check if you really want to use a different Autoconf version, but it's your responsibility whether the result works or not.

- Update `gettimeofday` configuration check so that PostgreSQL can be built on newer versions of MinGW (Magnus)

E.2. Release 8.2.5

Release date: 2007-09-17

This release contains a variety of fixes from 8.2.4.

E.2.1. Migration to Version 8.2.5

A dump/restore is not required for those running 8.2.X.

E.2.2. Changes

- Prevent index corruption when a transaction inserts rows and then aborts close to the end of a concurrent `VACUUM` on the same table (Tom)
- Fix `ALTER DOMAIN ADD CONSTRAINT` for cases involving domains over domains (Tom)
- Make `CREATE DOMAIN ... DEFAULT NULL` work properly (Tom)
- Fix some planner problems with outer joins, notably poor size estimation for `t1 LEFT JOIN t2 WHERE t2.col IS NULL` (Tom)
- Allow the `interval` data type to accept input consisting only of milliseconds or microseconds (Neil)
- Allow timezone name to appear before the year in `timestamp` input (Tom)
- Fixes for GIN indexes used by `/contrib/tsearch2` (Teodor)
- Speed up rtree index insertion (Teodor)
- Fix excessive logging of SSL error messages (Tom)
- Fix logging so that log messages are never interleaved when using the syslogger process (Andrew)
- Fix crash when `log_min_error_statement` logging runs out of memory (Tom)
- Fix incorrect handling of some foreign-key corner cases (Tom)
- Fix `stddev_pop(numeric)` and `var_pop(numeric)` (Tom)

- Prevent REINDEX and CLUSTER from failing due to attempting to process temporary tables of other sessions (Alvaro)
- Update the time zone database rules, particularly New Zealand's upcoming changes (Tom)
- Windows socket and semaphore improvements (Magnus)
- Make pg_ctl -w work properly in Windows service mode (Dave Page)
- Fix memory allocation bug when using MIT Kerberos on Windows (Magnus)
- Suppress timezone name (%Z) in log timestamps on Windows because of possible encoding mismatches (Tom)
- Require non-superusers who use /contrib/dblink to use only password authentication, as a security measure (Joe)
- Restrict /contrib/pgstattuple functions to superusers, for security reasons (Tom)
- Do not let /contrib/intarray try to make its GIN opclass the default (this caused problems at dump/restore) (Tom)

E.3. Release 8.2.4

Release date: 2007-04-23

This release contains a variety of fixes from 8.2.3, including a security fix.

E.3.1. Migration to Version 8.2.4

A dump/restore is not required for those running 8.2.X.

E.3.2. Changes

- Support explicit placement of the temporary-table schema within search_path, and disable searching it for functions and operators (Tom)

  This is needed to allow a security-definer function to set a truly secure value of search_path. Without it, an unprivileged SQL user can use temporary objects to execute code with the privileges of the security-definer function (CVE-2007-2138). See CREATE FUNCTION for more information.
- Fix shared_preload_libraries for Windows by forcing reload in each backend (Korry Douglas)
- Fix to_char() so it properly upper/lower cases localized day or month names (Pavel Stehule)
- /contrib/tsearch2 crash fixes (Teodor)
- Require COMMIT PREPARED to be executed in the same database as the transaction was prepared in (Heikki)

- Allow `pg_dump` to do binary backups larger than two gigabytes on Windows (Magnus)
- New traditional (Taiwan) Chinese FAQ (Zhou Daojing)
- Prevent the statistics collector from writing to disk too frequently (Tom)
- Fix potential-data-corruption bug in how `VACUUM FULL` handles `UPDATE` chains (Tom, Pavan Deolasee)
- Fix bug in domains that use array types (Tom)
- Fix `pg_dump` so it can dump a serial column's sequence using `-t` when not also dumping the owning table (Tom)
- Planner fixes, including improving outer join and bitmap scan selection logic (Tom)
- Fix possible wrong answers or crash when a PL/pgSQL function tries to `RETURN` from within an `EXCEPTION` block (Tom)
- Fix PANIC during enlargement of a hash index (Tom)
- Fix POSIX-style timezone specs to follow new USA DST rules (Tom)

E.4. Release 8.2.3

Release date: 2007-02-07

This release contains two fixes from 8.2.2.

E.4.1. Migration to Version 8.2.3

A dump/restore is not required for those running 8.2.X.

E.4.2. Changes

- Remove overly-restrictive check for type length in constraints and functional indexes(Tom)
- Fix optimization so MIN/MAX in subqueries can again use indexes (Tom)

E.5. Release 8.2.2

Release date: 2007-02-05

This release contains a variety of fixes from 8.2.1, including a security fix.

E.5.1. Migration to Version 8.2.2

A dump/restore is not required for those running 8.2.X.

E.5.2. Changes

- Remove security vulnerabilities that allowed connected users to read backend memory (Tom)

  The vulnerabilities involve suppressing the normal check that a SQL function returns the data type it's declared to, and changing the data type of a table column (CVE-2007-0555, CVE-2007-0556). These errors can easily be exploited to cause a backend crash, and in principle might be used to read database content that the user should not be able to access.
- Fix not-so-rare-anymore bug wherein btree index page splits could fail due to choosing an infeasible split point (Heikki Linnakangas)
- Fix Borland C compile scripts (L Bayuk)
- Properly handle `to_char('CC')` for years ending in `00` (Tom)

  Year 2000 is in the twentieth century, not the twenty-first.
- `/contrib/tsearch2` localization improvements (Tatsuo, Teodor)
- Fix incorrect permission check in `information_schema.key_column_usage` view (Tom)

  The symptom is "relation with OID nnnnn does not exist" errors. To get this fix without using `initdb`, use `CREATE OR REPLACE VIEW` to install the corrected definition found in `share/information_schema.sql`. Note you will need to do this in each database.
- Improve `VACUUM` performance for databases with many tables (Tom)
- Fix for rare Assert() crash triggered by `UNION` (Tom)
- Fix potentially incorrect results from index searches using `ROW` inequality conditions (Tom)
- Tighten security of multi-byte character processing for UTF8 sequences over three bytes long (Tom)
- Fix bogus "permission denied" failures occurring on Windows due to attempts to fsync already-deleted files (Magnus, Tom)
- Fix bug that could cause the statistics collector to hang on Windows (Magnus)

  This would in turn lead to autovacuum not working.
- Fix possible crashes when an already-in-use PL/pgSQL function is updated (Tom)
- Improve PL/pgSQL handling of domain types (Sergiy Vyshnevetskiy, Tom)
- Fix possible errors in processing PL/pgSQL exception blocks (Tom)

E.6. Release 8.2.1

Release date: 2007-01-08

This release contains a variety of fixes from 8.2.

E.6.1. Migration to Version 8.2.1

A dump/restore is not required for those running 8.2.

E.6.2. Changes

- Fix crash with `SELECT ... LIMIT ALL` (also `LIMIT NULL`) (Tom)
- `Several /contrib/tsearch2` fixes (Teodor)
- On Windows, make log messages coming from the operating system use ASCII encoding (Hiroshi Saito)

  This fixes a conversion problem when there is a mismatch between the encoding of the operating system and database server.
- Fix Windows linking of pg_dump using `win32.mak` (Hiroshi Saito)
- Fix planner mistakes for outer join queries (Tom)
- Fix several problems in queries involving sub-SELECTs (Tom)
- Fix potential crash in SPI during subtransaction abort (Tom)

  This affects all PL functions since they all use SPI.
- Improve build speed of PDF documentation (Peter)
- Re-add JST (Japan) timezone abbreviation (Tom)
- Improve optimization decisions related to index scans (Tom)
- Have psql print multi-byte combining characters as before, rather than output as `\u` (Tom)
- Improve index usage of regular expressions that use parentheses (Tom)

  This improves psql `\d` performance also.
- Make pg_dumpall assume that databases have public `CONNECT` privilege, when dumping from a pre-8.2 server (Tom)

  This preserves the previous behavior that anyone can connect to a database if allowed by `pg_hba.conf`.

E.7. Release 8.2

Release date: 2006-12-05

E.7.1. Overview

This release adds many functionality and performance improvements that were requested by users, including:

- Query language enhancements including `INSERT/UPDATE/DELETE RETURNING`, multirow `VALUES` lists, and optional target-table alias in `UPDATE`/`DELETE`
- Index creation without blocking concurrent `INSERT`/`UPDATE`/`DELETE` operations
- Many query optimization improvements, including support for reordering outer joins
- Improved sorting performance with lower memory usage
- More efficient locking with better concurrency
- More efficient vacuuming
- Easier administration of warm standby servers
- New `FILLFACTOR` support for tables and indexes
- Monitoring, logging, and performance tuning additions
- More control over creating and dropping objects
- Table inheritance relationships can be defined for and removed from pre-existing tables
- `COPY TO` can copy the output of an arbitrary `SELECT` statement
- Array improvements, including nulls in arrays
- Aggregate-function improvements, including multiple-input aggregates and SQL:2003 statistical functions
- Many `contrib/` improvements

E.7.2. Migration to Version 8.2

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release.

Observe the following incompatibilities:

- Set `escape_string_warning` to `on` by default (Bruce)

  This issues a warning if backslash escapes are used in non-escape (non-`E''`) strings.

- Change the row constructor syntax (`ROW(...)`) so that list elements `foo.*` will be expanded to a list of their member fields, rather than creating a nested row type field as formerly (Tom)

  The new behavior is substantially more useful since it allows, for example, triggers to check for data changes with `IF row(new.*) IS DISTINCT FROM row(old.*)`. The old behavior is still available by omitting `.*`.

- Make row comparisons follow SQL standard semantics and allow them to be used in index scans (Tom)

Previously, row = and <> comparisons followed the standard but < <= > >= did not. A row comparison can now be used as an index constraint for a multicolumn index matching the row value.

- Make row `IS [NOT] NULL` tests follow SQL standard semantics (Tom)

  The former behavior conformed to the standard for simple cases with `IS NULL`, but `IS NOT NULL` would return true if any row field was non-null, whereas the standard says it should return true only when all fields are non-null.

- Make `SET CONSTRAINT` affect only one constraint (Kris Jurka)

  In previous releases, `SET CONSTRAINT` modified all constraints with a matching name. In this release, the schema search path is used to modify only the first matching constraint. A schema specification is also supported. This more nearly conforms to the SQL standard.

- Remove `RULE` permission for tables, for security reasons (Tom)

  As of this release, only a table's owner can create or modify rules for the table. For backwards compatibility, `GRANT`/`REVOKE RULE` is still accepted, but it does nothing.

- Array comparison improvements (Tom)

  Now array dimensions are also compared.

- Change array concatenation to match documented behavior (Tom)

  This changes the previous behavior where concatenation would modify the array lower bound.

- Make command-line options of postmaster and postgres identical (Peter)

  This allows the postmaster to pass arguments to each backend without using `-o`. Note that some options are now only available as long-form options, because there were conflicting single-letter options.

- Deprecate use of postmaster symbolic link (Peter)

  postmaster and postgres commands now act identically, with the behavior determined by command-line options. The postmaster symbolic link is kept for compatibility, but is not really needed.

- Change `log_duration` to output even if the query is not output (Tom)

  In prior releases, `log_duration` only printed if the query appeared earlier in the log.

- Make `to_char(time)` and `to_char(interval)` treat `HH` and `HH12` as 12-hour intervals

  Most applications should use `HH24` unless they want a 12-hour display.

- Zero unmasked bits in conversion from `INET` to `CIDR` (Tom)

  This ensures that the converted value is actually valid for `CIDR`.

- Remove `australian_timezones` configuration variable (Joachim Wieland)

  This variable has been superseded by a more general facility for configuring timezone abbreviations.

- Improve cost estimation for nested-loop index scans (Tom)

  This might eliminate the need to set unrealistically small values of `random_page_cost`. If you have been using a very small `random_page_cost`, please recheck your test cases.

- Change behavior of `pg_dump -n` and `-t` options. (Greg Sabino Mullane)

  See the `pg_dump` manual page for details.

- Change libpq `PQdsplen()` to return a useful value (Martijn van Oosterhout)

- Declare libpq `PQgetssl()` as returning `void *`, rather than `SSL *` (Martijn van Oosterhout)

  This allows applications to use the function without including the OpenSSL headers.

- C-language loadable modules must now include a `PG_MODULE_MAGIC` macro call for version compatibility checking (Martijn van Oosterhout)

- For security's sake, modules used by a PL/PerlU function are no longer available to PL/Perl functions (Andrew)

  > **Note:** This also implies that data can no longer be shared between a PL/Perl function and a PL/PerlU function. Some Perl installations have not been compiled with the correct flags to allow multiple interpreters to exist within a single process. In this situation PL/Perl and PL/PerlU cannot both be used in a single backend. The solution is to get a Perl installation which supports multiple interpreters.

- In `contrib/xml2/`, rename `xml_valid()` to `xml_is_well_formed()` (Tom)

  `xml_valid()` will remain for backward compatibility, but its behavior will change to do schema checking in a future release.

- Remove `contrib/ora2pg/`, now at http://www.samse.fr/GPL/ora2pg

- Remove contrib modules that have been migrated to PgFoundry: `adddepend`, `dbase`, `dbmirror`, `fulltextindex`, `mac`, `userlock`

- Remove abandoned contrib modules: `mSQL-interface`, `tips`

- Remove QNX and BEOS ports (Bruce)

  These ports no longer had active maintainers.

E.7.3. Changes

Below you will find a detailed account of the changes between PostgreSQL 8.2 and the previous major release.

E.7.3.1. Performance Improvements

- Allow the planner to reorder outer joins in some circumstances (Tom)

  In previous releases, outer joins would always be evaluated in the order written in the query. This change allows the query optimizer to consider reordering outer joins, in cases where it can determine that the join order can be changed without altering the meaning of the query. This can make a considerable performance difference for queries involving multiple outer joins or mixed inner and outer joins.

- Improve efficiency of `IN` (list-of-expressions) clauses (Tom)

- Improve sorting speed and reduce memory usage (Simon, Tom)

- Improve subtransaction performance (Alvaro, Itagaki Takahiro, Tom)

- Add `FILLFACTOR` to table and index creation (ITAGAKI Takahiro)

  This leaves extra free space in each table or index page, allowing improved performance as the database grows. This is particularly valuable to maintain clustering.

- Increase default values for `shared_buffers` and `max_fsm_pages` (Andrew)
- Improve locking performance by breaking the lock manager tables into sections (Tom)

  This allows locking to be more fine-grained, reducing contention.

- Reduce locking requirements of sequential scans (Qingqing Zhou)
- Reduce locking required for database creation and destruction (Tom)
- Improve the optimizer's selectivity estimates for `LIKE`, `ILIKE`, and regular expression operations (Tom)
- Improve planning of joins to inherited tables and `UNION ALL` views (Tom)
- Allow constraint exclusion to be applied to inherited `UPDATE` and `DELETE` queries (Tom)

  `SELECT` already honored constraint exclusion.

- Improve planning of constant `WHERE` clauses, such as a condition that depends only on variables inherited from an outer query level (Tom)
- Protocol-level unnamed prepared statements are re-planned for each set of `BIND` values (Tom)

  This improves performance because the exact parameter values can be used in the plan.

- Speed up vacuuming of B-Tree indexes (Heikki Linnakangas, Tom)
- Avoid extra scan of tables without indexes during `VACUUM` (Greg Stark)
- Improve multicolumn GiST indexing (Oleg, Teodor)
- Remove dead index entries before B-Tree page split (Junji Teramoto)

E.7.3.2. Server Changes

- Allow a forced switch to a new transaction log file (Simon, Tom)

  This is valuable for keeping warm standby slave servers in sync with the master. Transaction log file switching now also happens automatically during `pg_stop_backup()`. This ensures that all transaction log files needed for recovery can be archived immediately.

- Add WAL informational functions (Simon)

  Add functions for interrogating the current transaction log insertion point and determining WAL filenames from the hex WAL locations displayed by `pg_stop_backup()` and related functions.

- Improve recovery from a crash during WAL replay (Simon)

  The server now does periodic checkpoints during WAL recovery, so if there is a crash, future WAL recovery is shortened. This also eliminates the need for warm standby servers to replay the entire log since the base backup if they crash.

- Improve reliability of long-term WAL replay (Heikki, Simon, Tom)

  Formerly, trying to roll forward through more than 2 billion transactions would not work due to XID wraparound. This meant warm standby servers had to be reloaded from fresh base backups periodically.

- Add `archive_timeout` to force transaction log file switches at a given interval (Simon)

  This enforces a maximum replication delay for warm standby servers.

- Add native LDAP authentication (Magnus Hagander)

  This is particularly useful for platforms that do not support PAM, such as Windows.

- Add `GRANT CONNECT ON DATABASE` (Gevik Babakhani)

  This gives SQL-level control over database access. It works as an additional filter on top of the existing `pg_hba.conf` controls.

- Add support for SSL Certificate Revocation List (CRL) files (Libor Hohoš)

  The server and libpq both recognize CRL files now.

- GiST indexes are now clusterable (Teodor)

- Remove routine autovacuum server log entries (Bruce)

  `pg_stat_activity` now shows autovacuum activity.

- Track maximum XID age within individual tables, instead of whole databases (Alvaro)

  This reduces the overhead involved in preventing transaction ID wraparound, by avoiding unnecessary VACUUMs.

- Add last vacuum and analyze timestamp columns to the stats collector (Larry Rosenman)

  These values now appear in the `pg_stat_*_tables` system views.

- Improve performance of statistics monitoring, especially `stats_command_string` (Tom, Bruce)

  This release enables `stats_command_string` by default, now that its overhead is minimal. This means `pg_stat_activity` will now show all active queries by default.

- Add a `waiting` column to `pg_stat_activity` (Tom)

  This allows `pg_stat_activity` to show all the information included in the ps display.

- Add configuration parameter `update_process_title` to control whether the ps display is updated for every command (Bruce)

  On platforms where it is expensive to update the ps display, it might be worthwhile to turn this off and rely solely on `pg_stat_activity` for status information.

- Allow units to be specified in configuration settings (Peter)

  For example, you can now set `shared_buffers` to `32MB` rather than mentally converting sizes.

- Add support for include directives in `postgresql.conf` (Joachim Wieland)

- Improve logging of protocol-level prepare/bind/execute messages (Bruce, Tom)

  Such logging now shows statement names, bind parameter values, and the text of the query being executed. Also, the query text is properly included in logged error messages when enabled by `log_min_error_statement`.

- Prevent `max_stack_depth` from being set to unsafe values

  On platforms where we can determine the actual kernel stack depth limit (which is most), make sure that the initial default value of `max_stack_depth` is safe, and reject attempts to set it to unsafely large values.

- Enable highlighting of error location in query in more cases (Tom)

  The server is now able to report a specific error location for some semantic errors (such as unrecognized column name), rather than just for basic syntax errors as before.

- Fix "failed to re-find parent key" errors in `VACUUM` (Tom)
- Clean out `pg_internal.init` cache files during server restart (Simon)

  This avoids a hazard that the cache files might contain stale data after PITR recovery.

- Fix race condition for truncation of a large relation across a gigabyte boundary by `VACUUM` (Tom)
- Fix bug causing needless deadlock errors on row-level locks (Tom)
- Fix bugs affecting multi-gigabyte hash indexes (Tom)
- Each backend process is now its own process group leader (Tom)

  This allows query cancel to abort subprocesses invoked from a backend or archive/recovery process.

E.7.3.3. Query Changes

- Add `INSERT`/`UPDATE`/`DELETE` `RETURNING` (Jonah Harris, Tom)

  This allows these commands to return values, such as the computed serial key for a new row. In the `UPDATE` case, values from the updated version of the row are returned.

- Add support for multiple-row `VALUES` clauses, per SQL standard (Joe, Tom)

  This allows `INSERT` to insert multiple rows of constants, or queries to generate result sets using constants. For example, `INSERT ... VALUES (...), (...), ...`, and `SELECT * FROM (VALUES (...), (...), ....) AS alias(f1, ...)`.

- Allow `UPDATE` and `DELETE` to use an alias for the target table (Atsushi Ogawa)

  The SQL standard does not permit an alias in these commands, but many database systems allow one anyway for notational convenience.

- Allow `UPDATE` to set multiple columns with a list of values (Susanne Ebrecht)

  This is basically a short-hand for assigning the columns and values in pairs. The syntax is `UPDATE tab SET (`*`column`*`, ...) = (`*`val`*`, ...)`.

- Make row comparisons work per standard (Tom)

  The forms <, <=, >, >= now compare rows lexicographically, that is, compare the first elements, if equal compare the second elements, and so on. Formerly they expanded to an AND condition across all the elements, which was neither standard nor very useful.

- Add `CASCADE` option to `TRUNCATE` (Joachim Wieland)

  This causes `TRUNCATE` to automatically include all tables that reference the specified table(s) via foreign keys. While convenient, this is a dangerous tool — use with caution!

- Support `FOR UPDATE` and `FOR SHARE` in the same `SELECT` command (Tom)
- Add `IS NOT DISTINCT FROM` (Pavel Stehule)

This operator is similar to equality (=), but evaluates to true when both left and right operands are NULL, and to false when just one is, rather than yielding NULL in these cases.

- Improve the length output used by UNION/INTERSECT/EXCEPT (Tom)

  When all corresponding columns are of the same defined length, that length is used for the result, rather than a generic length.

- Allow ILIKE to work for multi-byte encodings (Tom)

  Internally, ILIKE now calls lower() and then uses LIKE. Locale-specific regular expression patterns still do not work in these encodings.

- Enable standard_conforming_strings to be turned on (Kevin Grittner)

  This allows backslash escaping in strings to be disabled, making PostgreSQL more standards-compliant. The default is off for backwards compatibility, but future releases will default this to on.

- Do not flatten subqueries that contain volatile functions in their target lists (Jaime Casanova)

  This prevents surprising behavior due to multiple evaluation of a volatile function (such as random() or nextval()). It might cause performance degradation in the presence of functions that are unnecessarily marked as volatile.

- Add system views pg_prepared_statements and pg_cursors to show prepared statements and open cursors (Joachim Wieland, Neil)

  These are very useful in pooled connection setups.

- Support portal parameters in EXPLAIN and EXECUTE (Tom)

  This allows, for example, JDBC ? parameters to work in these commands.

- If SQL-level PREPARE parameters are unspecified, infer their types from the content of the query (Neil)

  Protocol-level PREPARE already did this.

- Allow LIMIT and OFFSET to exceed two billion (Dhanaraj M)

E.7.3.4. Object Manipulation Changes

- Add TABLESPACE clause to CREATE TABLE AS (Neil)

  This allows a tablespace to be specified for the new table.

- Add ON COMMIT clause to CREATE TABLE AS (Neil)

  This allows temporary tables to be truncated or dropped on transaction commit. The default behavior is for the table to remain until the session ends.

- Add INCLUDING CONSTRAINTS to CREATE TABLE LIKE (Greg Stark)

  This allows easy copying of CHECK constraints to a new table.

- Allow the creation of placeholder (shell) types (Martijn van Oosterhout)

  A shell type declaration creates a type name, without specifying any of the details of the type. Making a shell type is useful because it allows cleaner declaration of the type's input/output functions, which must exist before the type can be defined "for real". The syntax is CREATE TYPE *typename*.

- Aggregate functions now support multiple input parameters (Sergey Koposov, Tom)
- Add new aggregate creation syntax (Tom)

  The new syntax is `CREATE AGGREGATE` *`aggname`* `(`*`input_type`*`) (`*`parameter_list`*`)`. This more naturally supports the new multi-parameter aggregate functionality. The previous syntax is still supported.
- Add `ALTER ROLE PASSWORD NULL` to remove a previously set role password (Peter)
- Add `DROP` object `IF EXISTS` for many object types (Andrew)

  This allows `DROP` operations on non-existent objects without generating an error.
- Add `DROP OWNED` to drop all objects owned by a role (Alvaro)
- Add `REASSIGN OWNED` to reassign ownership of all objects owned by a role (Alvaro)

  This, and `DROP OWNED` above, facilitate dropping roles.
- Add `GRANT ON SEQUENCE` syntax (Bruce)

  This was added for setting sequence-specific permissions. `GRANT ON TABLE` for sequences is still supported for backward compatibility.
- Add `USAGE` permission for sequences that allows only `currval()` and `nextval()`, not `setval()` (Bruce)

  `USAGE` permission allows more fine-grained control over sequence access. Granting `USAGE` allows users to increment a sequence, but prevents them from setting the sequence to an arbitrary value using `setval()`.
- Add `ALTER TABLE [ NO ] INHERIT` (Greg Stark)

  This allows inheritance to be adjusted dynamically, rather than just at table creation and destruction. This is very valuable when using inheritance to implement table partitioning.
- Allow comments on global objects to be stored globally (Kris Jurka)

  Previously, comments attached to databases were stored in individual databases, making them ineffective, and there was no provision at all for comments on roles or tablespaces. This change adds a new shared catalog `pg_shdescription` and stores comments on databases, roles, and tablespaces therein.

E.7.3.5. Utility Command Changes

- Add option to allow indexes to be created without blocking concurrent writes to the table (Greg Stark, Tom)

  The new syntax is `CREATE INDEX CONCURRENTLY`. The default behavior is still to block table modification while a index is being created.
- Provide advisory locking functionality (Abhijit Menon-Sen, Tom)

  This is a new locking API designed to replace what used to be in /contrib/userlock. The userlock code is now on pgfoundry.
- Allow `COPY` to dump a `SELECT` query (Zoltan Boszormenyi, Karel Zak)

  This allows `COPY` to dump arbitrary SQL queries. The syntax is `COPY (SELECT ...) TO`.

- Make the `COPY` command return a command tag that includes the number of rows copied (Volkan YAZICI)
- Allow `VACUUM` to expire rows without being affected by other concurrent `VACUUM` operations (Hannu Krossing, Alvaro, Tom)
- Make initdb detect the operating system locale and set the default `DateStyle` accordingly (Peter)

  This makes it more likely that the installed `postgresql.conf` `DateStyle` value will be as desired.
- Reduce number of progress messages displayed by initdb (Tom)

E.7.3.6. Date/Time Changes

- Allow full timezone names in `timestamp` input values (Joachim Wieland)

  For example, `'2006-05-24 21:11 America/New_York'::timestamptz`.
- Support configurable timezone abbreviations (Joachim Wieland)

  A desired set of timezone abbreviations can be chosen via the configuration parameter `timezone_abbreviations`.
- Add `pg_timezone_abbrevs` and `pg_timezone_names` views to show supported timezones (Magnus Hagander)
- Add `clock_timestamp()`, `statement_timestamp()`, and `transaction_timestamp()` (Bruce)

  `clock_timestamp()` is the current wall-clock time, `statement_timestamp()` is the time the current statement arrived at the server, and `transaction_timestamp()` is an alias for `now()`.
- Allow `to_char()` to print localized month and day names (Euler Taveira de Oliveira)
- Allow `to_char(time)` and `to_char(interval)` to output AM/PM specifications (Bruce)

  Intervals and times are treated as 24-hour periods, e.g. `25 hours` is considered AM.
- Add new function `justify_interval()` to adjust interval units (Mark Dilger)
- Allow timezone offsets up to 14:59 away from GMT

  Kiribati uses GMT+14, so we'd better accept that.
- Interval computation improvements (Michael Glaesemann, Bruce)

E.7.3.7. Other Data Type and Function Changes

- Allow arrays to contain `NULL` elements (Tom)
- Allow assignment to array elements not contiguous with the existing entries (Tom)

  The intervening array positions will be filled with nulls. This is per SQL standard.
- New built-in operators for array-subset comparisons (`@>`, `<@`, `&&`) (Teodor, Tom)

  These operators can be indexed for many data types using GiST or GIN indexes.
- Add convenient arithmetic operations on `INET`/`CIDR` values (Stephen R. van den Berg)

The new operators are `&` (and), `|` (or), `~` (not), `inet + int8`, `inet - int8`, and `inet - inet`.

- Add new aggregate functions from SQL:2003 (Neil)

  The new functions are `var_pop()`, `var_samp()`, `stddev_pop()`, and `stddev_samp()`. `var_samp()` and `stddev_samp()` are merely renamings of the existing aggregates `variance()` and `stddev()`. The latter names remain available for backward compatibility.

- Add SQL:2003 statistical aggregates (Sergey Koposov)

  New functions: `regr_intercept()`, `regr_slope()`, `regr_r2()`, `corr()`, `covar_samp()`, `covar_pop()`, `regr_avgx()`, `regr_avgy()`, `regr_sxy()`, `regr_sxx()`, `regr_syy()`, `regr_count()`.

- Allow domains to be based on other domains (Tom)

- Properly enforce domain `CHECK` constraints everywhere (Neil, Tom)

  For example, the result of a user-defined function that is declared to return a domain type is now checked against the domain's constraints. This closes a significant hole in the domain implementation.

- Fix problems with dumping renamed `SERIAL` columns (Tom)

  The fix is to dump a `SERIAL` column by explicitly specifying its `DEFAULT` and sequence elements, and reconstructing the `SERIAL` column on reload using a new `ALTER SEQUENCE OWNED BY` command. This also allows dropping a `SERIAL` column specification.

- Add a server-side sleep function `pg_sleep()` (Joachim Wieland)

- Add all comparison operators for the `tid` (tuple id) data type (Mark Kirkwood, Greg Stark, Tom)

E.7.3.8. PL/PgSQL Server-Side Language Changes

- Add `TG_table_name` and `TG_table_schema` to trigger parameters (Andrew)

  `TG_relname` is now deprecated. Comparable changes have been made in the trigger parameters for the other PLs as well.

- Allow `FOR` statements to return values to scalars as well as records and row types (Pavel Stehule)

- Add a `BY` clause to the `FOR` loop, to control the iteration increment (Jaime Casanova)

- Add `STRICT` to `SELECT INTO` (Matt Miller)

  `STRICT` mode throws an exception if more or less than one row is returned by the `SELECT`, for Oracle PL/SQL compatibility.

E.7.3.9. PL/Perl Server-Side Language Changes

- Add `table_name` and `table_schema` to trigger parameters (Adam Sjøgren)

- Add prepared queries (Dmitry Karasik)

- Make `$_TD` trigger data a global variable (Andrew)

  Previously, it was lexical, which caused unexpected sharing violations.

- Run PL/Perl and PL/PerlU in separate interpreters, for security reasons (Andrew)

  In consequence, they can no longer share data nor loaded modules. Also, if Perl has not been compiled with the requisite flags to allow multiple interpreters, only one of these languages can be used in any given backend process.

E.7.3.10. PL/Python Server-Side Language Changes

- Named parameters are passed as ordinary variables, as well as in the `args[]` array (Sven Suursoho)
- Add `table_name` and `table_schema` to trigger parameters (Andrew)
- Allow returning of composite types and result sets (Sven Suursoho)
- Return result-set as `list`, `iterator`, or `generator` (Sven Suursoho)
- Allow functions to return `void` (Neil)
- Python 2.5 is now supported (Tom)

E.7.3.11. psql Changes

- Add new command `\password` for changing role password with client-side password encryption (Peter)
- Allow `\c` to connect to a new host and port number (David, Volkan YAZICI)
- Add tablespace display to `\l+` (Philip Yarra)
- Improve `\df` slash command to include the argument names and modes (`OUT` or `INOUT`) of the function (David Fetter)
- Support binary `COPY` (Andreas Pflug)
- Add option to run the entire session in a single transaction (Simon)

  Use option `-1` or `--single-transaction`.
- Support for automatically retrieving `SELECT` results in batches using a cursor (Chris Mair)

  This is enabled using `\set FETCH_COUNT` *`n`*. This feature allows large result sets to be retrieved in psql without attempting to buffer the entire result set in memory.
- Make multi-line values align in the proper column (Martijn van Oosterhout)

  Field values containing newlines are now displayed in a more readable fashion.
- Save multi-line statements as a single entry, rather than one line at a time (Sergey E. Koposov)

  This makes up-arrow recall of queries easier. (This is not available on Windows, because that platform uses the native command-line editing present in the operating system.)
- Make the line counter 64-bit so it can handle files with more than two billion lines (David Fetter)
- Report both the returned data and the command status tag for `INSERT`/`UPDATE`/`DELETE RETURNING` (Tom)

E.7.3.12. pg_dump Changes

- Allow complex selection of objects to be included or excluded by pg_dump (Greg Sabino Mullane)

  pg_dump now supports multiple `-n` (schema) and `-t` (table) options, and adds `-N` and `-T` options to exclude objects. Also, the arguments of these switches can now be wild-card expressions rather than single object names, for example `-t 'foo*'`, and a schema can be part of a `-t` or `-T` switch, for example `-t schema1.table1`.

- Add pg_restore `--no-data-for-failed-tables` option to suppress loading data if table creation failed (i.e., the table already exists) (Martin Pitt)

- Add pg_restore option to run the entire session in a single transaction (Simon)

  Use option `-1` or `--single-transaction`.

E.7.3.13. libpq Changes

- Add `PQencryptPassword()` to encrypt passwords (Tom)

  This allows passwords to be sent pre-encrypted for commands like `ALTER ROLE ... PASSWORD`.

- Add function `PQisthreadsafe()` (Bruce)

  This allows applications to query the thread-safety status of the library.

- Add `PQdescribePrepared()`, `PQdescribePortal()`, and related functions to return information about previously prepared statements and open cursors (Volkan YAZICI)

- Allow LDAP lookups from `pg_service.conf` (Laurenz Albe)

- Allow a hostname in `~/.pgpass` to match the default socket directory (Bruce)

  A blank hostname continues to match any Unix-socket connection, but this addition allows entries that are specific to one of several postmasters on the machine.

E.7.3.14. ecpg Changes

- Allow `SHOW` to put its result into a variable (Joachim Wieland)
- Add `COPY TO STDOUT` (Joachim Wieland)
- Add regression tests (Joachim Wieland, Michael)
- Major source code cleanups (Joachim Wieland, Michael)

E.7.3.15. Windows Port

- Allow MSVC to compile the PostgreSQL server (Magnus, Hiroshi Saito)
- Add MSVC support for utility commands and pg_dump (Hiroshi Saito)
- Add support for Windows code pages `1253`, `1254`, `1255`, and `1257` (Kris Jurka)

- Drop privileges on startup, so that the server can be started from an administrative account (Magnus)
- Stability fixes (Qingqing Zhou, Magnus)
- Add native semaphore implementation (Qingqing Zhou)

  The previous code mimicked SysV semaphores.

E.7.3.16. Source Code Changes

- Add GIN (Generalized Inverted iNdex) index access method (Teodor, Oleg)
- Remove R-tree indexing (Tom)

  Rtree has been re-implemented using GiST. Among other differences, this means that rtree indexes now have support for crash recovery via write-ahead logging (WAL).
- Reduce libraries needlessly linked into the backend (Martijn van Oosterhout, Tom)
- Add a configure flag to allow libedit to be preferred over GNU readline (Bruce)

  Use configure `--with-libedit-preferred`.
- Allow installation into directories containing spaces (Peter)
- Improve ability to relocate installation directories (Tom)
- Add support for Solaris x86_64 using the Solaris compiler (Pierre Girard, Theo Schlossnagle, Bruce)
- Add DTrace support (Robert Lor)
- Add `PG_VERSION_NUM` for use by third-party applications wanting to test the backend version in C using > and < comparisons (Bruce)
- Add `XLOG_BLCKSZ` as independent from `BLCKSZ` (Mark Wong)
- Add `LWLOCK_STATS` define to report locking activity (Tom)
- Emit warnings for unknown configure options (Martijn van Oosterhout)
- Add server support for "plugin" libraries that can be used for add-on tasks such as debugging and performance measurement (Korry Douglas)

  This consists of two features: a table of "rendezvous variables" that allows separately-loaded shared libraries to communicate, and a new configuration parameter `local_preload_libraries` that allows libraries to be loaded into specific sessions without explicit cooperation from the client application. This allows external add-ons to implement features such as a PL/PgSQL debugger.
- Rename existing configuration parameter `preload_libraries` to `shared_preload_libraries` (Tom)

  This was done for clarity in comparison to `local_preload_libraries`.
- Add new configuration parameter `server_version_num` (Greg Sabino Mullane)

  This is like `server_version`, but is an integer, e.g. `80200`. This allows applications to make version checks more easily.
- Add a configuration parameter `seq_page_cost` (Tom)
- Re-implement the regression test script as a C program (Magnus, Tom)

- Allow loadable modules to allocate shared memory and lightweight locks (Marc Munro)
- Add automatic initialization and finalization of dynamically loaded libraries (Ralf Engelschall, Tom)

  New functions `_PG_init()` and `_PG_fini()` are called if the library defines such symbols. Hence we no longer need to specify an initialization function in `shared_preload_libraries`; we can assume that the library used the `_PG_init()` convention instead.
- Add `PG_MODULE_MAGIC` header block to all shared object files (Martijn van Oosterhout)

  The magic block prevents version mismatches between loadable object files and servers.
- Add shared library support for AIX (Laurenz Albe)
- New XML documentation section (Bruce)

E.7.3.17. Contrib Changes

- Major tsearch2 improvements (Oleg, Teodor)
  - multibyte encoding support, including UTF8
  - query rewriting support
  - improved ranking functions
  - thesaurus dictionary support
  - Ispell dictionaries now recognize MySpell format, used by OpenOffice
  - GIN support

- Add adminpack module containing Pgadmin administration functions (Dave)

  These functions provide additional file system access routines not present in the default PostgreSQL server.
- Add sslinfo module (Victor Wagner)

  Reports information about the current connection's SSL certificate.
- Add pgrowlocks module (Tatsuo)

  This shows row locking information for a specified table.
- Add hstore module (Oleg, Teodor)
- Add isn module, replacing isbn_issn (Jeremy Kronuz)

  This new implementation supports EAN13, UPC, ISBN (books), ISMN (music), and ISSN (serials).
- Add index information functions to pgstattuple (ITAGAKI Takahiro, Satoshi Nagayasu)
- Add pg_freespacemap module to display free space map information (Mark Kirkwood)
- pgcrypto now has all planned functionality (Marko Kreen)
  - Include iMath library in pgcrypto to have the public-key encryption functions always available.
  - Add SHA224 algorithm that was missing in OpenBSD code.

- Activate builtin code for SHA224/256/384/512 hashes on older OpenSSL to have those algorithms always available.
- New function gen_random_bytes() that returns cryptographically strong randomness. Useful for generating encryption keys.
- Remove digest_exists(), hmac_exists() and cipher_exists() functions.

- Improvements to cube module (Joshua Reich)

  New functions are `cube(float[])`, `cube(float[], float[])`, and `cube_subset(cube, int4[])`.
- Add async query capability to dblink (Kai Londenberg, Joe Conway)
- New operators for array-subset comparisons (`@>`, `<@`, `&&`) (Tom)

  Various contrib packages already had these operators for their datatypes, but the naming wasn't consistent. We have now added consistently named array-subset comparison operators to the core code and all the contrib packages that have such functionality. (The old names remain available, but are deprecated.)
- Add uninstall scripts for all contrib packages that have install scripts (David, Josh Drake)

E.8. Release 8.1.11

Release date: 2008-01-07

This release contains a variety of fixes from 8.1.10, including fixes for significant security issues.

This is the last 8.1.X release for which the PostgreSQL community will produce binary packages for Windows. Windows users are encouraged to move to 8.2.X or later, since there are Windows-specific fixes in 8.2.X that are impractical to back-port. 8.1.X will continue to be supported on other platforms.

E.8.1. Migration to Version 8.1.11

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see the release notes for 8.1.2.

E.8.2. Changes

- Prevent functions in indexes from executing with the privileges of the user running `VACUUM`, `ANALYZE`, etc (Tom)

  Functions used in index expressions and partial-index predicates are evaluated whenever a new table entry is made. It has long been understood that this poses a risk of trojan-horse code execution if one

modifies a table owned by an untrustworthy user. (Note that triggers, defaults, check constraints, etc. pose the same type of risk.) But functions in indexes pose extra danger because they will be executed by routine maintenance operations such as `VACUUM FULL`, which are commonly performed automatically under a superuser account. For example, a nefarious user can execute code with superuser privileges by setting up a trojan-horse index definition and waiting for the next routine vacuum. The fix arranges for standard maintenance operations (including `VACUUM`, `ANALYZE`, `REINDEX`, and `CLUSTER`) to execute as the table owner rather than the calling user, using the same privilege-switching mechanism already used for `SECURITY DEFINER` functions. To prevent bypassing this security measure, execution of `SET SESSION AUTHORIZATION` and `SET ROLE` is now forbidden within a `SECURITY DEFINER` context. (CVE-2007-6600)

- Repair assorted bugs in the regular-expression package (Tom, Will Drewry)

  Suitably crafted regular-expression patterns could cause crashes, infinite or near-infinite looping, and/or massive memory consumption, all of which pose denial-of-service hazards for applications that accept regex search patterns from untrustworthy sources. (CVE-2007-4769, CVE-2007-4772, CVE-2007-6067)

- Require non-superusers who use `/contrib/dblink` to use only password authentication, as a security measure (Joe)

  The fix that appeared for this in 8.1.10 was incomplete, as it plugged the hole for only some `dblink` functions. (CVE-2007-6601, CVE-2007-3278)

- Update time zone data files to tzdata release 2007k (in particular, recent Argentina changes) (Tom)
- Improve planner's handling of LIKE/regex estimation in non-C locales (Tom)
- Fix planner failure in some cases of `WHERE false AND var IN (SELECT ...)` (Tom)
- Preserve the tablespace of indexes that are rebuilt by `ALTER TABLE ... ALTER COLUMN TYPE` (Tom)
- Make archive recovery always start a new WAL timeline, rather than only when a recovery stop time was used (Simon)

  This avoids a corner-case risk of trying to overwrite an existing archived copy of the last WAL segment, and seems simpler and cleaner than the original definition.

- Make `VACUUM` not use all of `maintenance_work_mem` when the table is too small for it to be useful (Alvaro)
- Fix potential crash in `translate()` when using a multibyte database encoding (Tom)
- Fix overflow in `extract(epoch from interval)` for intervals exceeding 68 years (Tom)
- Fix PL/Perl to not fail when a UTF-8 regular expression is used in a trusted function (Andrew)
- Fix PL/Perl to cope when platform's Perl defines type `bool` as `int` rather than `char` (Tom)

  While this could theoretically happen anywhere, no standard build of Perl did things this way ... until Mac OS X 10.5.

- Fix PL/Python to not crash on long exception messages (Alvaro)
- Fix pg_dump to correctly handle inheritance child tables that have default expressions different from their parent's (Tom)
- Fix libpq crash when `PGPASSFILE` refers to a file that is not a plain file (Martin Pitt)

- ecpg parser fixes (Michael)
- Make `contrib/pgcrypto` defend against OpenSSL libraries that fail on keys longer than 128 bits; which is the case at least on some Solaris versions (Marko Kreen)
- Make `contrib/tablefunc`'s `crosstab()` handle NULL rowid as a category in its own right, rather than crashing (Joe)
- Fix `tsvector` and `tsquery` output routines to escape backslashes correctly (Teodor, Bruce)
- Fix crash of `to_tsvector()` on huge input strings (Teodor)
- Require a specific version of Autoconf to be used when re-generating the `configure` script (Peter)

  This affects developers and packagers only. The change was made to prevent accidental use of untested combinations of Autoconf and PostgreSQL versions. You can remove the version check if you really want to use a different Autoconf version, but it's your responsibility whether the result works or not.

E.9. Release 8.1.10

Release date: 2007-09-17

This release contains a variety of fixes from 8.1.9.

E.9.1. Migration to Version 8.1.10

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see the release notes for 8.1.2.

E.9.2. Changes

- Prevent index corruption when a transaction inserts rows and then aborts close to the end of a concurrent `VACUUM` on the same table (Tom)
- Make `CREATE DOMAIN ... DEFAULT NULL` work properly (Tom)
- Allow the `interval` data type to accept input consisting only of milliseconds or microseconds (Neil)
- Speed up rtree index insertion (Teodor)
- Fix excessive logging of SSL error messages (Tom)
- Fix logging so that log messages are never interleaved when using the syslogger process (Andrew)
- Fix crash when `log_min_error_statement` logging runs out of memory (Tom)
- Fix incorrect handling of some foreign-key corner cases (Tom)
- Prevent `REINDEX` and `CLUSTER` from failing due to attempting to process temporary tables of other sessions (Alvaro)

- Update the time zone database rules, particularly New Zealand's upcoming changes (Tom)
- Windows socket improvements (Magnus)
- Suppress timezone name (`%Z`) in log timestamps on Windows because of possible encoding mismatches (Tom)
- Require non-superusers who use `/contrib/dblink` to use only password authentication, as a security measure (Joe)

E.10. Release 8.1.9

Release date: 2007-04-23

This release contains a variety of fixes from 8.1.8, including a security fix.

E.10.1. Migration to Version 8.1.9

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see the release notes for 8.1.2.

E.10.2. Changes

- Support explicit placement of the temporary-table schema within `search_path`, and disable searching it for functions and operators (Tom)

  This is needed to allow a security-definer function to set a truly secure value of `search_path`. Without it, an unprivileged SQL user can use temporary objects to execute code with the privileges of the security-definer function (CVE-2007-2138). See `CREATE FUNCTION` for more information.
- `/contrib/tsearch2` crash fixes (Teodor)
- Require `COMMIT PREPARED` to be executed in the same database as the transaction was prepared in (Heikki)
- Fix potential-data-corruption bug in how `VACUUM FULL` handles `UPDATE` chains (Tom, Pavan Deolasee)
- Planner fixes, including improving outer join and bitmap scan selection logic (Tom)
- Fix PANIC during enlargement of a hash index (bug introduced in 8.1.6) (Tom)
- Fix POSIX-style timezone specs to follow new USA DST rules (Tom)

E.11. Release 8.1.8

Release date: 2007-02-07

This release contains one fix from 8.1.7.

E.11.1. Migration to Version 8.1.8

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see the release notes for 8.1.2.

E.11.2. Changes

- Remove overly-restrictive check for type length in constraints and functional indexes(Tom)

E.12. Release 8.1.7

Release date: 2007-02-05

This release contains a variety of fixes from 8.1.6, including a security fix.

E.12.1. Migration to Version 8.1.7

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see the release notes for 8.1.2.

E.12.2. Changes

- Remove security vulnerabilities that allowed connected users to read backend memory (Tom)

  The vulnerabilities involve suppressing the normal check that a SQL function returns the data type it's declared to, and changing the data type of a table column (CVE-2007-0555, CVE-2007-0556). These errors can easily be exploited to cause a backend crash, and in principle might be used to read database content that the user should not be able to access.
- Fix rare bug wherein btree index page splits could fail due to choosing an infeasible split point (Heikki Linnakangas)

- Improve VACUUM performance for databases with many tables (Tom)
- Fix autovacuum to avoid leaving non-permanent transaction IDs in non-connectable databases (Alvaro)

  This bug affects the 8.1 branch only.
- Fix for rare Assert() crash triggered by UNION (Tom)
- Tighten security of multi-byte character processing for UTF8 sequences over three bytes long (Tom)
- Fix bogus "permission denied" failures occurring on Windows due to attempts to fsync already-deleted files (Magnus, Tom)
- Fix possible crashes when an already-in-use PL/pgSQL function is updated (Tom)

E.13. Release 8.1.6

Release date: 2007-01-08

This release contains a variety of fixes from 8.1.5.

E.13.1. Migration to Version 8.1.6

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see the release notes for 8.1.2.

E.13.2. Changes

- Improve handling of `getaddrinfo()` on AIX (Tom)

  This fixes a problem with starting the statistics collector, among other things.
- Fix pg_restore to handle a tar-format backup that contains large objects (blobs) with comments (Tom)
- Fix "failed to re-find parent key" errors in VACUUM (Tom)
- Clean out `pg_internal.init` cache files during server restart (Simon)

  This avoids a hazard that the cache files might contain stale data after PITR recovery.
- Fix race condition for truncation of a large relation across a gigabyte boundary by VACUUM (Tom)
- Fix bug causing needless deadlock errors on row-level locks (Tom)
- Fix bugs affecting multi-gigabyte hash indexes (Tom)
- Fix possible deadlock in Windows signal handling (Teodor)
- Fix error when constructing an `ARRAY[]` made up of multiple empty elements (Tom)
- Fix ecpg memory leak during connection (Michael)

- Fix for Darwin (OS X) compilation (Tom)
- `to_number()` and `to_char(numeric)` are now `STABLE`, not `IMMUTABLE`, for new initdb installs (Tom)

  This is because `lc_numeric` can potentially change the output of these functions.
- Improve index usage of regular expressions that use parentheses (Tom)

  This improves psql `\d` performance also.
- Update timezone database

  This affects Australian and Canadian daylight-savings rules in particular.

E.14. Release 8.1.5

Release date: 2006-10-16

This release contains a variety of fixes from 8.1.4.

E.14.1. Migration to Version 8.1.5

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see the release notes for 8.1.2.

E.14.2. Changes

- Disallow aggregate functions in `UPDATE` commands, except within sub-SELECTs (Tom)

  The behavior of such an aggregate was unpredictable, and in 8.1.X could cause a crash, so it has been disabled. The SQL standard does not allow this either.
- Fix core dump when an untyped literal is taken as ANYARRAY
- Fix core dump in duration logging for extended query protocol when a `COMMIT` or `ROLLBACK` is executed
- Fix mishandling of AFTER triggers when query contains a SQL function returning multiple rows (Tom)
- Fix `ALTER TABLE ... TYPE` to recheck `NOT NULL` for `USING` clause (Tom)
- Fix `string_to_array()` to handle overlapping matches for the separator string

  For example, `string_to_array('123xx456xxx789', 'xx')`.
- Fix `to_timestamp()` for `AM`/`PM` formats (Bruce)
- Fix autovacuum's calculation that decides whether `ANALYZE` is needed (Alvaro)
- Fix corner cases in pattern matching for psql's `\d` commands

- Fix index-corrupting bugs in /contrib/ltree (Teodor)
- Numerous robustness fixes in ecpg (Joachim Wieland)
- Fix backslash escaping in /contrib/dbmirror
- Minor fixes in /contrib/dblink and /contrib/tsearch2
- Efficiency improvements in hash tables and bitmap index scans (Tom)
- Fix instability of statistics collection on Windows (Tom, Andrew)
- Fix `statement_timeout` to use the proper units on Win32 (Bruce)

  In previous Win32 8.1.X versions, the delay was off by a factor of 100.
- Fixes for MSVC and Borland C++ compilers (Hiroshi Saito)
- Fixes for AIX and Intel compilers (Tom)
- Fix rare bug in continuous archiving (Tom)

E.15. Release 8.1.4

Release date: 2006-05-23

This release contains a variety of fixes from 8.1.3, including patches for extremely serious security issues.

E.15.1. Migration to Version 8.1.4

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see the release notes for 8.1.2.

Full security against the SQL-injection attacks described in CVE-2006-2313 and CVE-2006-2314 might require changes in application code. If you have applications that embed untrustworthy strings into SQL commands, you should examine them as soon as possible to ensure that they are using recommended escaping techniques. In most cases, applications should be using subroutines provided by libraries or drivers (such as libpq's `PQescapeStringConn()`) to perform string escaping, rather than relying on *ad hoc* code to do it.

E.15.2. Changes

- Change the server to reject invalidly-encoded multibyte characters in all cases (Tatsuo, Tom)

  While PostgreSQL has been moving in this direction for some time, the checks are now applied uniformly to all encodings and all textual input, and are now always errors not merely warnings. This change defends against SQL-injection attacks of the type described in CVE-2006-2313.
- Reject unsafe uses of `\'` in string literals

As a server-side defense against SQL-injection attacks of the type described in CVE-2006-2314, the server now only accepts '' and not \' as a representation of ASCII single quote in SQL string literals. By default, \' is rejected only when client_encoding is set to a client-only encoding (SJIS, BIG5, GBK, GB18030, or UHC), which is the scenario in which SQL injection is possible. A new configuration parameter backslash_quote is available to adjust this behavior when needed. Note that full security against CVE-2006-2314 might require client-side changes; the purpose of backslash_quote is in part to make it obvious that insecure clients are insecure.

- Modify libpq's string-escaping routines to be aware of encoding considerations and standard_conforming_strings

  This fixes libpq-using applications for the security issues described in CVE-2006-2313 and CVE-2006-2314, and also future-proofs them against the planned changeover to SQL-standard string literal syntax. Applications that use multiple PostgreSQL connections concurrently should migrate to PQescapeStringConn() and PQescapeByteaConn() to ensure that escaping is done correctly for the settings in use in each database connection. Applications that do string escaping "by hand" should be modified to rely on library routines instead.

- Fix weak key selection in pgcrypto (Marko Kreen)

  Errors in fortuna PRNG reseeding logic could cause a predictable session key to be selected by pgp_sym_encrypt() in some cases. This only affects non-OpenSSL-using builds.

- Fix some incorrect encoding conversion functions

  win1251_to_iso, win866_to_iso, euc_tw_to_big5, euc_tw_to_mic, mic_to_euc_tw were all broken to varying extents.

- Clean up stray remaining uses of \' in strings (Bruce, Jan)

- Make autovacuum visible in pg_stat_activity (Alvaro)

- Disable full_page_writes (Tom)

  In certain cases, having full_page_writes off would cause crash recovery to fail. A proper fix will appear in 8.2; for now it's just disabled.

- Various planner fixes, particularly for bitmap index scans and MIN/MAX optimization (Tom)

- Fix incorrect optimization in merge join (Tom)

  Outer joins could sometimes emit multiple copies of unmatched rows.

- Fix crash from using and modifying a plpgsql function in the same transaction

- Fix WAL replay for case where a B-Tree index has been truncated

- Fix SIMILAR TO for patterns involving | (Tom)

- Fix SELECT INTO and CREATE TABLE AS to create tables in the default tablespace, not the base directory (Kris Jurka)

- Fix server to use custom DH SSL parameters correctly (Michael Fuhr)

- Improve qsort performance (Dann Corbit)

  Currently this code is only used on Solaris.

- Fix for OS/X Bonjour on x86 systems (Ashley Clark)

- Fix various minor memory leaks

- Fix problem with password prompting on some Win32 systems (Robert Kinberg)
- Improve pg_dump's handling of default values for domains
- Fix pg_dumpall to handle identically-named users and groups reasonably (only possible when dumping from a pre-8.1 server) (Tom)

  The user and group will be merged into a single role with `LOGIN` permission. Formerly the merged role wouldn't have `LOGIN` permission, making it unusable as a user.
- Fix pg_restore `-n` to work as documented (Tom)

E.16. Release 8.1.3

Release date: 2006-02-14

This release contains a variety of fixes from 8.1.2, including one very serious security issue.

E.16.1. Migration to Version 8.1.3

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see the release notes for 8.1.2.

E.16.2. Changes

- Fix bug that allowed any logged-in user to `SET ROLE` to any other database user id (CVE-2006-0553)

  Due to inadequate validity checking, a user could exploit the special case that `SET ROLE` normally uses to restore the previous role setting after an error. This allowed ordinary users to acquire superuser status, for example. The escalation-of-privilege risk exists only in 8.1.0-8.1.2. However, in all releases back to 7.3 there is a related bug in `SET SESSION AUTHORIZATION` that allows unprivileged users to crash the server, if it has been compiled with Asserts enabled (which is not the default). Thanks to Akio Ishida for reporting this problem.
- Fix bug with row visibility logic in self-inserted rows (Tom)

  Under rare circumstances a row inserted by the current command could be seen as already valid, when it should not be. Repairs bug created in 8.0.4, 7.4.9, and 7.3.11 releases.
- Fix race condition that could lead to "file already exists" errors during pg_clog and pg_subtrans file creation (Tom)
- Fix cases that could lead to crashes if a cache-invalidation message arrives at just the wrong time (Tom)
- Properly check `DOMAIN` constraints for `UNKNOWN` parameters in prepared statements (Neil)
- Ensure `ALTER COLUMN TYPE` will process `FOREIGN KEY`, `UNIQUE`, and `PRIMARY KEY` constraints in the proper order (Nakano Yoshihisa)

- Fixes to allow restoring dumps that have cross-schema references to custom operators or operator classes (Tom)
- Allow pg_restore to continue properly after a `COPY` failure; formerly it tried to treat the remaining `COPY` data as SQL commands (Stephen Frost)
- Fix pg_ctl `unregister` crash when the data directory is not specified (Magnus)
- Fix libpq `PQprint` HTML tags (Christoph Zwerschke)
- Fix ecpg crash on AMD64 and PPC (Neil)
- Allow `SETOF` and `%TYPE` to be used together in function result type declarations
- Recover properly if error occurs during argument passing in PL/python (Neil)
- Fix memory leak in `plperl_return_next` (Neil)
- Fix PL/perl's handling of locales on Win32 to match the backend (Andrew)
- Various optimizer fixes (Tom)
- Fix crash when `log_min_messages` is set to `DEBUG3` or above in `postgresql.conf` on Win32 (Bruce)
- Fix pgxs `-L` library path specification for Win32, Cygwin, OS X, AIX (Bruce)
- Check that SID is enabled while checking for Win32 admin privileges (Magnus)
- Properly reject out-of-range date inputs (Kris Jurka)
- Portability fix for testing presence of `finite` and `isinf` during configure (Tom)
- Improve speed of `COPY IN` via libpq, by avoiding a kernel call per data line (Alon Goldshuv)
- Improve speed of `/contrib/tsearch2` index creation (Tom)

E.17. Release 8.1.2

Release date: 2006-01-09

This release contains a variety of fixes from 8.1.1.

E.17.1. Migration to Version 8.1.2

A dump/restore is not required for those running 8.1.X. However, you might need to `REINDEX` indexes on textual columns after updating, if you are affected by the locale or plperl issues described below.

E.17.2. Changes

- Fix Windows code so that postmaster will continue rather than exit if there is no more room in ShmemBackendArray (Magnus)

  The previous behavior could lead to a denial-of-service situation if too many connection requests arrive close together. This applies *only* to the Windows port.

- Fix bug introduced in 8.0 that could allow ReadBuffer to return an already-used page as new, potentially causing loss of recently-committed data (Tom)

- Fix for protocol-level Describe messages issued outside a transaction or in a failed transaction (Tom)

- Fix character string comparison for locales that consider different character combinations as equal, such as Hungarian (Tom)

  This might require `REINDEX` to fix existing indexes on textual columns.

- Set locale environment variables during postmaster startup to ensure that plperl won't change the locale later

  This fixes a problem that occurred if the postmaster was started with environment variables specifying a different locale than what initdb had been told. Under these conditions, any use of plperl was likely to lead to corrupt indexes. You might need `REINDEX` to fix existing indexes on textual columns if this has happened to you.

- Allow more flexible relocation of installation directories (Tom)

  Previous releases supported relocation only if all installation directory paths were the same except for the last component.

- Prevent crashes caused by the use of `ISO-8859-5` and `ISO-8859-9` encodings (Tatsuo)

- Fix longstanding bug in strpos() and regular expression handling in certain rarely used Asian multi-byte character sets (Tatsuo)

- Fix bug where COPY CSV mode considered any `\.` to terminate the copy data

  The new code requires `\.` to appear alone on a line, as per documentation.

- Make COPY CSV mode quote a literal data value of `\.` to ensure it cannot be interpreted as the end-of-data marker (Bruce)

- Various fixes for functions returning `RECORD`s (Tom)

- Fix processing of `postgresql.conf` so a final line with no newline is processed properly (Tom)

- Fix bug in `/contrib/pgcrypto` gen_salt, which caused it not to use all available salt space for MD5 and XDES algorithms (Marko Kreen, Solar Designer)

  Salts for Blowfish and standard DES are unaffected.

- Fix autovacuum crash when processing expression indexes

- Fix `/contrib/dblink` to throw an error, rather than crashing, when the number of columns specified is different from what's actually returned by the query (Joe)

E.18. Release 8.1.1

Release date: 2005-12-12

This release contains a variety of fixes from 8.1.0.

E.18.1. Migration to Version 8.1.1

A dump/restore is not required for those running 8.1.X.

E.18.2. Changes

- Fix incorrect optimizations of outer-join conditions (Tom)
- Fix problems with wrong reported column names in cases involving sub-selects flattened by the optimizer (Tom)
- Fix update failures in scenarios involving CHECK constraints, toasted columns, *and* indexes (Tom)
- Fix bgwriter problems after recovering from errors (Tom)

  The background writer was found to leak buffer pins after write errors. While not fatal in itself, this might lead to mysterious blockages of later VACUUM commands.
- Prevent failure if client sends Bind protocol message when current transaction is already aborted
- `/contrib/tsearch2` and `/contrib/ltree` fixes (Teodor)
- Fix problems with translated error messages in languages that require word reordering, such as Turkish; also problems with unexpected truncation of output strings and wrong display of the smallest possible bigint value (Andrew, Tom)

  These problems only appeared on platforms that were using our `port/snprintf.c` code, which includes BSD variants if `--enable-nls` was given, and perhaps others. In addition, a different form of the translated-error-message problem could appear on Windows depending on which version of `libintl` was used.
- Re-allow `AM`/`PM`, `HH`, `HH12`, and `D` format specifiers for `to_char(time)` and `to_char(interval)`. (`to_char(interval)` should probably use `HH24`.) (Bruce)
- AIX, HPUX, and MSVC compile fixes (Tom, Hiroshi Saito)
- Optimizer improvements (Tom)
- Retry file reads and writes after Windows NO_SYSTEM_RESOURCES error (Qingqing Zhou)
- Prevent autovacuum from crashing during ANALYZE of expression index (Alvaro)
- Fix problems with ON COMMIT DELETE ROWS temp tables
- Fix problems when a trigger alters the output of a SELECT DISTINCT query
- Add 8.1.0 release note item on how to migrate invalid `UTF-8` byte sequences (Paul Lindner)

E.19. Release 8.1

Release date: 2005-11-08

E.19.1. Overview

Major changes in this release:

Improve concurrent access to the shared buffer cache (Tom)

Access to the shared buffer cache was identified as a significant scalability problem, particularly on multi-CPU systems. In this release, the way that locking is done in the buffer manager has been overhauled to reduce lock contention and improve scalability. The buffer manager has also been changed to use a "clock sweep" replacement policy.

Allow index scans to use an intermediate in-memory bitmap (Tom)

In previous releases, only a single index could be used to do lookups on a table. With this feature, if a query has `WHERE tab.col1 = 4 and tab.col2 = 9`, and there is no multicolumn index on `col1` and `col2`, but there is an index on `col1` and another on `col2`, it is possible to search both indexes and combine the results in memory, then do heap fetches for only the rows matching both the `col1` and `col2` restrictions. This is very useful in environments that have a lot of unstructured queries where it is impossible to create indexes that match all possible access conditions. Bitmap scans are useful even with a single index, as they reduce the amount of random access needed; a bitmap index scan is efficient for retrieving fairly large fractions of the complete table, whereas plain index scans are not.

Add two-phase commit (Heikki Linnakangas, Alvaro, Tom)

Two-phase commit allows transactions to be "prepared" on several computers, and once all computers have successfully prepared their transactions (none failed), all transactions can be committed. Even if a machine crashes after a prepare, the prepared transaction can be committed after the machine is restarted. New syntax includes `PREPARE TRANSACTION` and `COMMIT/ROLLBACK PREPARED`. A new system view `pg_prepared_xacts` has also been added.

Create a new role system that replaces users and groups (Stephen Frost)

Roles are a combination of users and groups. Like users, they can have login capability, and like groups, a role can have other roles as members. Roles basically remove the distinction between users and groups. For example, a role can:

- Have login capability (optionally)
- Own objects
- Hold access permissions for database objects
- Inherit permissions from other roles it is a member of

Once a user logs into a role, she obtains capabilities of the login role plus any inherited roles, and can use `SET ROLE` to switch to other roles she is a member of. This feature is a generalization of the SQL standard's concept of roles. This change also replaces `pg_shadow` and `pg_group` by new

role-capable catalogs pg_authid and pg_auth_members. The old tables are redefined as read-only views on the new role tables.

Automatically use indexes for MIN() and MAX() (Tom)

In previous releases, the only way to use an index for MIN() or MAX() was to rewrite the query as SELECT col FROM tab ORDER BY col LIMIT 1. Index usage now happens automatically.

Move /contrib/pg_autovacuum into the main server (Alvaro)

Integrating autovacuum into the server allows it to be automatically started and stopped in sync with the database server, and allows autovacuum to be configured from postgresql.conf.

Add shared row level locks using SELECT ... FOR SHARE (Alvaro)

While PostgreSQL's MVCC locking allows SELECT to never be blocked by writers and therefore does not need shared row locks for typical operations, shared locks are useful for applications that require shared row locking. In particular this reduces the locking requirements imposed by referential integrity checks.

Add dependencies on shared objects, specifically roles (Alvaro)

This extension of the dependency mechanism prevents roles from being dropped while there are still database objects they own. Formerly it was possible to accidentally "orphan" objects by deleting their owner. While this could be recovered from, it was messy and unpleasant.

Improve performance for partitioned tables (Simon)

The new constraint_exclusion configuration parameter avoids lookups on child tables where constraints indicate that no matching rows exist in the child table.

This allows for a basic type of table partitioning. If child tables store separate key ranges and this is enforced using appropriate CHECK constraints, the optimizer will skip child table accesses when the constraint guarantees no matching rows exist in the child table.

E.19.2. Migration to Version 8.1

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release.

The 8.0 release announced that the to_char() function for intervals would be removed in 8.1. However, since no better API has been suggested, to_char(interval) has been enhanced in 8.1 and will remain in the server.

Observe the following incompatibilities:

- add_missing_from is now false by default (Neil)

  By default, we now generate an error if a table is used in a query without a FROM reference. The old behavior is still available, but the parameter must be set to 'true' to obtain it.

  It might be necessary to set add_missing_from to true in order to load an existing dump file, if the dump contains any views or rules created using the implicit-FROM syntax. This should be a one-time annoyance, because PostgreSQL 8.1 will convert such views and rules to standard explicit-FROM syntax. Subsequent dumps will therefore not have the problem.

- Cause input of a zero-length string (″) for `float4`/`float8`/`oid` to throw an error, rather than treating it as a zero (Neil)

  This change is consistent with the current handling of zero-length strings for integers. The schedule for this change was announced in 8.0.

- `default_with_oids` is now false by default (Neil)

  With this option set to false, user-created tables no longer have an OID column unless `WITH OIDS` is specified in `CREATE TABLE`. Though OIDs have existed in all releases of PostgreSQL, their use is limited because they are only four bytes long and the counter is shared across all installed databases. The preferred way of uniquely identifying rows is via sequences and the `SERIAL` type, which have been supported since PostgreSQL 6.4.

- Add `E″` syntax so eventually ordinary strings can treat backslashes literally (Bruce)

  Currently PostgreSQL processes a backslash in a string literal as introducing a special escape sequence, e.g. `\n` or `\010`. While this allows easy entry of special values, it is nonstandard and makes porting of applications from other databases more difficult. For this reason, the PostgreSQL project is planning to remove the special meaning of backslashes in strings. For backward compatibility and for users who want special backslash processing, a new string syntax has been created. This new string syntax is formed by writing an `E` immediately preceding the single quote that starts the string, e.g. `E′hi\n′`. While this release does not change the handling of backslashes in strings, it does add new configuration parameters to help users migrate applications for future releases:

  - `standard_conforming_strings` — does this release treat backslashes literally in ordinary strings?
  - `escape_string_warning` — warn about backslashes in ordinary (non-E) strings

  The `standard_conforming_strings` value is read-only. Applications can retrieve the value to know how backslashes are processed. (Presence of the parameter can also be taken as an indication that `E″` string syntax is supported.) In a future release, `standard_conforming_strings` will be true, meaning backslashes will be treated literally in non-E strings. To prepare for this change, use `E″` strings in places that need special backslash processing, and turn on `escape_string_warning` to find additional strings that need to be converted to use `E″`. Also, use two single-quotes (″) to embed a literal single-quote in a string, rather than the PostgreSQL-supported syntax of backslash single-quote (`\′`). The former is standards-conforming and does not require the use of the `E″` string syntax. You can also use the `$$` string syntax, which does not treat backslashes specially.

- Make `REINDEX DATABASE` reindex all indexes in the database (Tom)

  Formerly, `REINDEX DATABASE` reindexed only system tables. This new behavior seems more intuitive. A new command `REINDEX SYSTEM` provides the old functionality of reindexing just the system tables.

- Read-only large object descriptors now obey MVCC snapshot semantics

  When a large object is opened with `INV_READ` (and not `INV_WRITE`), the data read from the descriptor will now reflect a "snapshot" of the large object's state at the time of the transaction snapshot in use by the query that called `lo_open()`. To obtain the old behavior of always returning the latest committed data, include `INV_WRITE` in the mode flags for `lo_open()`.

- Add proper dependencies for arguments of sequence functions (Tom)

  In previous releases, sequence names passed to `nextval()`, `currval()`, and `setval()` were stored as simple text strings, meaning that renaming or dropping a sequence used in a `DEFAULT` clause made

the clause invalid. This release stores all newly-created sequence function arguments as internal OIDs, allowing them to track sequence renaming, and adding dependency information that prevents improper sequence removal. It also makes such `DEFAULT` clauses immune to schema renaming and search path changes.

Some applications might rely on the old behavior of run-time lookup for sequence names. This can still be done by explicitly casting the argument to `text`, for example `nextval('myseq'::text)`.

Pre-8.1 database dumps loaded into 8.1 will use the old text-based representation and therefore will not have the features of OID-stored arguments. However, it is possible to update a database containing text-based `DEFAULT` clauses. First, save this query into a file, such as `fixseq.sql`:

```
SELECT  'ALTER TABLE ' ||
   pg_catalog.quote_ident(n.nspname) || '.' ||
   pg_catalog.quote_ident(c.relname) ||
   ' ALTER COLUMN ' || pg_catalog.quote_ident(a.attname) ||
   ' SET DEFAULT ' ||
   regexp_replace(d.adsrc,
                  $$val\(\(('[^']*')::text\)::regclass$$,
                  $$val(\1$$,
                  'g') ||
   ';'
FROM    pg_namespace n, pg_class c, pg_attribute a, pg_attrdef d
WHERE   n.oid = c.relnamespace AND
   c.oid = a.attrelid AND
   a.attrelid = d.adrelid AND
   a.attnum = d.adnum AND
   d.adsrc ~ $$val\(\('[^']*'::text\)::regclass$$;
```

Next, run the query against a database to find what adjustments are required, like this for database `db1`:

```
psql -t -f fixseq.sql db1
```

This will show the `ALTER TABLE` commands needed to convert the database to the newer OID-based representation. If the commands look reasonable, run this to update the database:

```
psql -t -f fixseq.sql db1 | psql -e db1
```

This process must be repeated in each database to be updated.

- In psql, treat unquoted `\{digit}+` sequences as octal (Bruce)

  In previous releases, `\{digit}+` sequences were treated as decimal, and only `\0{digit}+` were treated as octal. This change was made for consistency.

- Remove grammar productions for prefix and postfix `%` and `^` operators (Tom)

  These have never been documented and complicated the use of the modulus operator (`%`) with negative numbers.

- Make `&<` and `&>` for polygons consistent with the box "over" operators (Tom)

- `CREATE LANGUAGE` can ignore the provided arguments in favor of information from `pg_pltemplate` (Tom)

  A new system catalog `pg_pltemplate` has been defined to carry information about the preferred definitions of procedural languages (such as whether they have validator functions). When an entry exists in this catalog for the language being created, `CREATE LANGUAGE` will ignore all its parameters except the language name and instead use the catalog information. This measure was taken because of increasing problems with obsolete language definitions being loaded by old dump files. As of 8.1,

pg_dump will dump procedural language definitions as just `CREATE LANGUAGE name`, relying on a template entry to exist at load time. We expect this will be a more future-proof representation.

- Make `pg_cancel_backend(int)` return a `boolean` rather than an `integer` (Neil)
- Some users are having problems loading UTF-8 data into 8.1.X. This is because previous versions allowed invalid UTF-8 byte sequences to be entered into the database, and this release properly accepts only valid UTF-8 sequences. One way to correct a dumpfile is to run the command `iconv -c -f UTF-8 -t UTF-8 -o cleanfile.sql dumpfile.sql`. The `-c` option removes invalid character sequences. A diff of the two files will show the sequences that are invalid. `iconv` reads the entire input file into memory so it might be necessary to use split to break up the dump into multiple smaller files for processing.

E.19.3. Additional Changes

Below you will find a detailed account of the additional changes between PostgreSQL 8.1 and the previous major release.

E.19.3.1. Performance Improvements

- Improve GiST and R-tree index performance (Neil)
- Improve the optimizer, including auto-resizing of hash joins (Tom)
- Overhaul internal API in several areas
- Change WAL record CRCs from 64-bit to 32-bit (Tom)

  We determined that the extra cost of computing 64-bit CRCs was significant, and the gain in reliability too marginal to justify it.
- Prevent writing large empty gaps in WAL pages (Tom)
- Improve spinlock behavior on SMP machines, particularly Opterons (Tom)
- Allow nonconsecutive index columns to be used in a multicolumn index (Tom)

  For example, this allows an index on columns a,b,c to be used in a query with `WHERE a = 4 and c = 10`.
- Skip WAL logging for `CREATE TABLE AS` / `SELECT INTO` (Simon)

  Since a crash during `CREATE TABLE AS` would cause the table to be dropped during recovery, there is no reason to WAL log as the table is loaded. (Logging still happens if WAL archiving is enabled, however.)
- Allow concurrent GiST index access (Teodor, Oleg)
- Add configuration parameter `full_page_writes` to control writing full pages to WAL (Bruce)

  To prevent partial disk writes from corrupting the database, PostgreSQL writes a complete copy of each database disk page to WAL the first time it is modified after a checkpoint. This option turns off that functionality for more speed. This is safe to use with battery-backed disk caches where partial page writes cannot happen.
- Use `O_DIRECT` if available when using `O_SYNC` for `wal_sync_method` (Itagaki Takahiro)

O_DIRECT causes disk writes to bypass the kernel cache, and for WAL writes, this improves performance.

- Improve COPY FROM performance (Alon Goldshuv)

  This was accomplished by reading COPY input in larger chunks, rather than character by character.

- Improve the performance of COUNT(), SUM, AVG(), STDDEV(), and VARIANCE() (Neil, Tom)

E.19.3.2. Server Changes

- Prevent problems due to transaction ID (XID) wraparound (Tom)

  The server will now warn when the transaction counter approaches the wraparound point. If the counter becomes too close to wraparound, the server will stop accepting queries. This ensures that data is not lost before needed vacuuming is performed.

- Fix problems with object IDs (OIDs) conflicting with existing system objects after the OID counter has wrapped around (Tom)

- Add warning about the need to increase max_fsm_relations and max_fsm_pages during VACUUM (Ron Mayer)

- Add temp_buffers configuration parameter to allow users to determine the size of the local buffer area for temporary table access (Tom)

- Add session start time and client IP address to pg_stat_activity (Magnus)

- Adjust pg_stat views for bitmap scans (Tom)

  The meanings of some of the fields have changed slightly.

- Enhance pg_locks view (Tom)

- Log queries for client-side PREPARE and EXECUTE (Simon)

- Allow Kerberos name and user name case sensitivity to be specified in postgresql.conf (Magnus)

- Add configuration parameter krb_server_hostname so that the server host name can be specified as part of service principal (Todd Kover)

  If not set, any service principal matching an entry in the keytab can be used. This is new Kerberos matching behavior in this release.

- Add log_line_prefix options for millisecond timestamps (%m) and remote host (%h) (Ed L.)

- Add WAL logging for GiST indexes (Teodor, Oleg)

  GiST indexes are now safe for crash and point-in-time recovery.

- Remove old *.backup files when we do pg_stop_backup() (Bruce)

  This prevents a large number of *.backup files from existing in pg_xlog/.

- Add configuration parameters to control TCP/IP keep-alive times for idle, interval, and count (Oliver Jowett)

  These values can be changed to allow more rapid detection of lost client connections.

- Add per-user and per-database connection limits (Petr Jelinek)

Using ALTER USER and ALTER DATABASE, limits can now be enforced on the maximum number of sessions that can concurrently connect as a specific user or to a specific database. Setting the limit to zero disables user or database connections.

- Allow more than two gigabytes of shared memory and per-backend work memory on 64-bit machines (Koichi Suzuki)
- New system catalog pg_pltemplate allows overriding obsolete procedural-language definitions in dump files (Tom)

E.19.3.3. Query Changes

- Add temporary views (Koju Iijima, Neil)
- Fix HAVING without any aggregate functions or GROUP BY so that the query returns a single group (Tom)

  Previously, such a case would treat the HAVING clause the same as a WHERE clause. This was not per spec.
- Add USING clause to allow additional tables to be specified to DELETE (Euler Taveira de Oliveira, Neil)

  In prior releases, there was no clear method for specifying additional tables to be used for joins in a DELETE statement. UPDATE already has a FROM clause for this purpose.
- Add support for \x hex escapes in backend and ecpg strings (Bruce)

  This is just like the standard C \x escape syntax. Octal escapes were already supported.
- Add BETWEEN SYMMETRIC query syntax (Pavel Stehule)

  This feature allows BETWEEN comparisons without requiring the first value to be less than the second. For example, 2 BETWEEN [ASYMMETRIC] 3 AND 1 returns false, while 2 BETWEEN SYMMETRIC 3 AND 1 returns true. BETWEEN ASYMMETRIC was already supported.
- Add NOWAIT option to SELECT ... FOR UPDATE/SHARE (Hans-Juergen Schoenig)

  While the statement_timeout configuration parameter allows a query taking more than a certain amount of time to be cancelled, the NOWAIT option allows a query to be canceled as soon as a SELECT ... FOR UPDATE/SHARE command cannot immediately acquire a row lock.

E.19.3.4. Object Manipulation Changes

- Track dependencies of shared objects (Alvaro)

  PostgreSQL allows global tables (users, databases, tablespaces) to reference information in multiple databases. This addition adds dependency information for global tables, so, for example, user ownership can be tracked across databases, so a user who owns something in any database can no longer be removed. Dependency tracking already existed for database-local objects.
- Allow limited ALTER OWNER commands to be performed by the object owner (Stephen Frost)

  Prior releases allowed only superusers to change object owners. Now, ownership can be transferred if the user executing the command owns the object and would be able to create it as the new owner (that

is, the user is a member of the new owning role and that role has the CREATE permission that would be needed to create the object afresh).

- Add `ALTER` object `SET SCHEMA` capability for some object types (tables, functions, types) (Bernd Helmle)

  This allows objects to be moved to different schemas.

- Add `ALTER TABLE ENABLE/DISABLE TRIGGER` to disable triggers (Satoshi Nagayasu)

E.19.3.5. Utility Command Changes

- Allow `TRUNCATE` to truncate multiple tables in a single command (Alvaro)

  Because of referential integrity checks, it is not allowed to truncate a table that is part of a referential integrity constraint. Using this new functionality, `TRUNCATE` can be used to truncate such tables, if both tables involved in a referential integrity constraint are truncated in a single `TRUNCATE` command.

- Properly process carriage returns and line feeds in `COPY CSV` mode (Andrew)

  In release 8.0, carriage returns and line feeds in `CSV COPY TO` were processed in an inconsistent manner. (This was documented on the TODO list.)

- Add `COPY WITH CSV HEADER` to allow a header line as the first line in `COPY` (Andrew)

  This allows handling of the common `CSV` usage of placing the column names on the first line of the data file. For `COPY TO`, the first line contains the column names, and for `COPY FROM`, the first line is ignored.

- On Windows, display better sub-second precision in `EXPLAIN ANALYZE` (Magnus)

- Add trigger duration display to `EXPLAIN ANALYZE` (Tom)

  Prior releases included trigger execution time as part of the total execution time, but did not show it separately. It is now possible to see how much time is spent in each trigger.

- Add support for `\x` hex escapes in `COPY` (Sergey Ten)

  Previous releases only supported octal escapes.

- Make `SHOW ALL` include variable descriptions (Matthias Schmidt)

  `SHOW` varname still only displays the variable's value and does not include the description.

- Make initdb create a new standard database called `postgres`, and convert utilities to use `postgres` rather than `template1` for standard lookups (Dave)

  In prior releases, `template1` was used both as a default connection for utilities like createuser, and as a template for new databases. This caused `CREATE DATABASE` to sometimes fail, because a new database cannot be created if anyone else is in the template database. With this change, the default connection database is now `postgres`, meaning it is much less likely someone will be using `template1` during `CREATE DATABASE`.

- Create new reindexdb command-line utility by moving `/contrib/reindexdb` into the server (Euler Taveira de Oliveira)

E.19.3.6. Data Type and Function Changes

- Add `MAX()` and `MIN()` aggregates for array types (Koju Iijima)
- Fix `to_date()` and `to_timestamp()` to behave reasonably when `CC` and `YY` fields are both used (Karel Zak)

  If the format specification contains `CC` and a year specification is `YYY` or longer, ignore the `CC`. If the year specification is `YY` or shorter, interpret `CC` as the previous century.
- Add `md5(bytea)` (Abhijit Menon-Sen)

  `md5(text)` already existed.
- Add support for `numeric ^ numeric` based on `power(numeric, numeric)`

  The function already existed, but there was no operator assigned to it.
- Fix `NUMERIC` modulus by properly truncating the quotient during computation (Bruce)

  In previous releases, modulus for large values sometimes returned negative results due to rounding of the quotient.
- Add a function `lastval()` (Dennis Björklund)

  `lastval()` is a simplified version of `currval()`. It automatically determines the proper sequence name based on the most recent `nextval()` or `setval()` call performed by the current session.
- Add `to_timestamp(DOUBLE PRECISION) (Michael Glaesemann)`

  Converts Unix seconds since 1970 to a `TIMESTAMP WITH TIMEZONE`.
- Add `pg_postmaster_start_time()` function (Euler Taveira de Oliveira, Matthias Schmidt)
- Allow the full use of time zone names in `AT TIME ZONE`, not just the short list previously available (Magnus)

  Previously, only a predefined list of time zone names were supported by `AT TIME ZONE`. Now any supported time zone name can be used, e.g.:

  ```
  SELECT CURRENT_TIMESTAMP AT TIME ZONE 'Europe/London';
  ```

  In the above query, the time zone used is adjusted based on the daylight saving time rules that were in effect on the supplied date.
- Add `GREATEST()` and `LEAST()` variadic functions (Pavel Stehule)

  These functions take a variable number of arguments and return the greatest or least value among the arguments.
- Add `pg_column_size()` (Mark Kirkwood)

  This returns storage size of a column, which might be compressed.
- Add `regexp_replace()` (Atsushi Ogawa)

  This allows regular expression replacement, like sed. An optional flag argument allows selection of global (replace all) and case-insensitive modes.
- Fix interval division and multiplication (Bruce)

Previous versions sometimes returned unjustified results, like `'4 months'::interval / 5` returning `'1 mon -6 days'`.

- Fix roundoff behavior in timestamp, time, and interval output (Tom)

  This fixes some cases in which the seconds field would be shown as `60` instead of incrementing the higher-order fields.

- Add a separate day field to type `interval` so a one day interval can be distinguished from a 24 hour interval (Michael Glaesemann)

  Days that contain a daylight saving time adjustment are not 24 hours long, but typically 23 or 25 hours. This change creates a conceptual distinction between intervals of "so many days" and intervals of "so many hours". Adding `1 day` to a timestamp now gives the same local time on the next day even if a daylight saving time adjustment occurs between, whereas adding `24 hours` will give a different local time when this happens. For example, under US DST rules:

  ```
  '2005-04-03 00:00:00-05' + '1 day' = '2005-04-04 00:00:00-04'
  '2005-04-03 00:00:00-05' + '24 hours' = '2005-04-04 01:00:00-04'
  ```

- Add `justify_days()` and `justify_hours()` (Michael Glaesemann)

  These functions, respectively, adjust days to an appropriate number of full months and days, and adjust hours to an appropriate number of full days and hours.

- Move `/contrib/dbsize` into the backend, and rename some of the functions (Dave Page, Andreas Pflug)

  - `pg_tablespace_size()`
  - `pg_database_size()`
  - `pg_relation_size()`
  - `pg_total_relation_size()`
  - `pg_size_pretty()`

  `pg_total_relation_size()` includes indexes and TOAST tables.

- Add functions for read-only file access to the cluster directory (Dave Page, Andreas Pflug)

  - `pg_stat_file()`
  - `pg_read_file()`
  - `pg_ls_dir()`

- Add `pg_reload_conf()` to force reloading of the configuration files (Dave Page, Andreas Pflug)
- Add `pg_rotate_logfile()` to force rotation of the server log file (Dave Page, Andreas Pflug)
- Change `pg_stat_*` views to include TOAST tables (Tom)

E.19.3.7. Encoding and Locale Changes

- Rename some encodings to be more consistent and to follow international standards (Bruce)
  - `UNICODE` is now `UTF8`
  - `ALT` is now `WIN866`
  - `WIN` is now `WIN1251`
  - `TCVN` is now `WIN1258`

  The original names still work.
- Add support for `WIN1252` encoding (Roland Volkmann)
- Add support for four-byte `UTF8` characters (John Hansen)

  Previously only one, two, and three-byte `UTF8` characters were supported. This is particularly important for support for some Chinese character sets.
- Allow direct conversion between `EUC_JP` and `SJIS` to improve performance (Atsushi Ogawa)
- Allow the UTF8 encoding to work on Windows (Magnus)

  This is done by mapping UTF8 to the Windows-native UTF16 implementation.

E.19.3.8. General Server-Side Language Changes

- Fix `ALTER LANGUAGE RENAME` (Sergey Yatskevich)
- Allow function characteristics, like strictness and volatility, to be modified via `ALTER FUNCTION` (Neil)
- Increase the maximum number of function arguments to 100 (Tom)
- Allow SQL and PL/PgSQL functions to use `OUT` and `INOUT` parameters (Tom)

  `OUT` is an alternate way for a function to return values. Instead of using `RETURN`, values can be returned by assigning to parameters declared as `OUT` or `INOUT`. This is notationally simpler in some cases, particularly so when multiple values need to be returned. While returning multiple values from a function was possible in previous releases, this greatly simplifies the process. (The feature will be extended to other server-side languages in future releases.)
- Move language handler functions into the `pg_catalog` schema

  This makes it easier to drop the public schema if desired.
- Add `SPI_getnspname()` to SPI (Neil)

E.19.3.9. PL/PgSQL Server-Side Language Changes

- Overhaul the memory management of PL/PgSQL functions (Neil)

The parsetree of each function is now stored in a separate memory context. This allows this memory to be easily reclaimed when it is no longer needed.

- Check function syntax at `CREATE FUNCTION` time, rather than at runtime (Neil)

  Previously, most syntax errors were reported only when the function was executed.

- Allow `OPEN` to open non-`SELECT` queries like `EXPLAIN` and `SHOW` (Tom)
- No longer require functions to issue a `RETURN` statement (Tom)

  This is a byproduct of the newly added `OUT` and `INOUT` functionality. `RETURN` can be omitted when it is not needed to provide the function's return value.

- Add support for an optional `INTO` clause to PL/PgSQL's `EXECUTE` statement (Pavel Stehule, Neil)
- Make `CREATE TABLE AS` set `ROW_COUNT` (Tom)
- Define `SQLSTATE` and `SQLERRM` to return the `SQLSTATE` and error message of the current exception (Pavel Stehule, Neil)

  These variables are only defined inside exception blocks.

- Allow the parameters to the `RAISE` statement to be expressions (Pavel Stehule, Neil)
- Add a loop `CONTINUE` statement (Pavel Stehule, Neil)
- Allow block and loop labels (Pavel Stehule)

E.19.3.10. PL/Perl Server-Side Language Changes

- Allow large result sets to be returned efficiently (Abhijit Menon-Sen)

  This allows functions to use `return_next()` to avoid building the entire result set in memory.

- Allow one-row-at-a-time retrieval of query results (Abhijit Menon-Sen)

  This allows functions to use `spi_query()` and `spi_fetchrow()` to avoid accumulating the entire result set in memory.

- Force PL/Perl to handle strings as `UTF8` if the server encoding is `UTF8` (David Kamholz)
- Add a validator function for PL/Perl (Andrew)

  This allows syntax errors to be reported at definition time, rather than execution time.

- Allow PL/Perl to return a Perl array when the function returns an array type (Andrew)

  This basically maps PostgreSQL arrays to Perl arrays.

- Allow Perl nonfatal warnings to generate `NOTICE` messages (Andrew)
- Allow Perl's `strict` mode to be enabled (Andrew)

E.19.3.11. psql Changes

- Add `\set ON_ERROR_ROLLBACK` to allow statements in a transaction to error without affecting the rest of the transaction (Greg Sabino Mullane)

This is basically implemented by wrapping every statement in a sub-transaction.

- Add support for `\x` hex strings in psql variables (Bruce)

  Octal escapes were already supported.

- Add support for `troff -ms` output format (Roger Leigh)

- Allow the history file location to be controlled by `HISTFILE` (Andreas Seltenreich)

  This allows configuration of per-database history storage.

- Prevent `\x` (expanded mode) from affecting the output of `\d tablename` (Neil)

- Add `-L` option to psql to log sessions (Lorne Sunley)

  This option was added because some operating systems do not have simple command-line activity logging functionality.

- Make `\d` show the tablespaces of indexes (Qingqing Zhou)

- Allow psql help (`\h`) to make a best guess on the proper help information (Greg Sabino Mullane)

  This allows the user to just add `\h` to the front of the syntax error query and get help on the supported syntax. Previously any additional query text beyond the command name had to be removed to use `\h`.

- Add `\pset numericlocale` to allow numbers to be output in a locale-aware format (Eugen Nedelcu)

  For example, using `C` locale `100000` would be output as `100,000.0` while a European locale might output this value as `100.000,0`.

- Make startup banner show both server version number and psql's version number, when they are different (Bruce)

  Also, a warning will be shown if the server and psql are from different major releases.

E.19.3.12. pg_dump Changes

- Add `-n` / `--schema` switch to pg_restore (Richard van den Berg)

  This allows just the objects in a specified schema to be restored.

- Allow pg_dump to dump large objects even in text mode (Tom)

  With this change, large objects are now always dumped; the former `-b` switch is a no-op.

- Allow pg_dump to dump a consistent snapshot of large objects (Tom)

- Dump comments for large objects (Tom)

- Add `--encoding` to pg_dump (Magnus Hagander)

  This allows a database to be dumped in an encoding that is different from the server's encoding. This is valuable when transferring the dump to a machine with a different encoding.

- Rely on `pg_pltemplate` for procedural languages (Tom)

  If the call handler for a procedural language is in the `pg_catalog` schema, pg_dump does not dump the handler. Instead, it dumps the language using just `CREATE LANGUAGE` *`name`*, relying on the `pg_pltemplate` catalog to provide the language's creation parameters at load time.

E.19.3.13. libpq Changes

- Add a `PGPASSFILE` environment variable to specify the password file's filename (Andrew)
- Add `lo_create()`, that is similar to `lo_creat()` but allows the OID of the large object to be specified (Tom)
- Make libpq consistently return an error to the client application on `malloc()` failure (Neil)

E.19.3.14. Source Code Changes

- Fix pgxs to support building against a relocated installation
- Add spinlock support for the Itanium processor using Intel compiler (Vikram Kalsi)
- Add Kerberos 5 support for Windows (Magnus)
- Add Chinese FAQ (laser@pgsqldb.com)
- Rename Rendezvous to Bonjour to match OS/X feature renaming (Bruce)
- Add support for `fsync_writethrough` on Darwin (Chris Campbell)
- Streamline the passing of information within the server, the optimizer, and the lock system (Tom)
- Allow pg_config to be compiled using MSVC (Andrew)

  This is required to build DBD::Pg using MSVC.
- Remove support for Kerberos V4 (Magnus)

  Kerberos 4 had security vulnerabilities and is no longer maintained.
- Code cleanups (Coverity static analysis performed by EnterpriseDB)
- Modify `postgresql.conf` to use documentation defaults `on`/`off` rather than `true`/`false` (Bruce)
- Enhance pg_config to be able to report more build-time values (Tom)
- Allow libpq to be built thread-safe on Windows (Dave Page)
- Allow IPv6 connections to be used on Windows (Andrew)
- Add Server Administration documentation about I/O subsystem reliability (Bruce)
- Move private declarations from `gist.h` to `gist_private.h` (Neil)

  In previous releases, `gist.h` contained both the public GiST API (intended for use by authors of GiST index implementations) as well as some private declarations used by the implementation of GiST itself. The latter have been moved to a separate file, `gist_private.h`. Most GiST index implementations should be unaffected.
- Overhaul GiST memory management (Neil)

  GiST methods are now always invoked in a short-lived memory context. Therefore, memory allocated via `palloc()` will be reclaimed automatically, so GiST index implementations do not need to manually release allocated memory via `pfree()`.

E.19.3.15. Contrib Changes

- Add `/contrib/pg_buffercache` contrib module (Mark Kirkwood)

  This displays the contents of the buffer cache, for debugging and performance tuning purposes.
- Remove `/contrib/array` because it is obsolete (Tom)
- Clean up the `/contrib/lo` module (Tom)
- Move `/contrib/findoidjoins` to `/src/tools` (Tom)
- Remove the <<, >>, &<, and &> operators from `/contrib/cube`

  These operators were not useful.
- Improve `/contrib/btree_gist` (Janko Richter)
- Improve `/contrib/pgbench` (Tomoaki Sato, Tatsuo)

  There is now a facility for testing with SQL command scripts given by the user, instead of only a hard-wired command sequence.
- Improve `/contrib/pgcrypto` (Marko Kreen)
  - Implementation of OpenPGP symmetric-key and public-key encryption

    Both RSA and Elgamal public-key algorithms are supported.
  - Stand alone build: include SHA256/384/512 hashes, Fortuna PRNG
  - OpenSSL build: support 3DES, use internal AES with OpenSSL < 0.9.7
  - Take build parameters (OpenSSL, zlib) from `configure` result

    There is no need to edit the `Makefile` anymore.
  - Remove support for `libmhash` and `libmcrypt`

E.20. Release 8.0.15

Release date: 2008-01-07

This release contains a variety of fixes from 8.0.14, including fixes for significant security issues.

This is the last 8.0.X release for which the PostgreSQL community will produce binary packages for Windows. Windows users are encouraged to move to 8.2.X or later, since there are Windows-specific fixes in 8.2.X that are impractical to back-port. 8.0.X will continue to be supported on other platforms.

E.20.1. Migration to Version 8.0.15

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see the release notes for 8.0.6.

E.20.2. Changes

- Prevent functions in indexes from executing with the privileges of the user running `VACUUM`, `ANALYZE`, etc (Tom)

  Functions used in index expressions and partial-index predicates are evaluated whenever a new table entry is made. It has long been understood that this poses a risk of trojan-horse code execution if one modifies a table owned by an untrustworthy user. (Note that triggers, defaults, check constraints, etc. pose the same type of risk.) But functions in indexes pose extra danger because they will be executed by routine maintenance operations such as `VACUUM FULL`, which are commonly performed automatically under a superuser account. For example, a nefarious user can execute code with superuser privileges by setting up a trojan-horse index definition and waiting for the next routine vacuum. The fix arranges for standard maintenance operations (including `VACUUM`, `ANALYZE`, `REINDEX`, and `CLUSTER`) to execute as the table owner rather than the calling user, using the same privilege-switching mechanism already used for `SECURITY DEFINER` functions. To prevent bypassing this security measure, execution of `SET SESSION AUTHORIZATION` and `SET ROLE` is now forbidden within a `SECURITY DEFINER` context. (CVE-2007-6600)

- Repair assorted bugs in the regular-expression package (Tom, Will Drewry)

  Suitably crafted regular-expression patterns could cause crashes, infinite or near-infinite looping, and/or massive memory consumption, all of which pose denial-of-service hazards for applications that accept regex search patterns from untrustworthy sources. (CVE-2007-4769, CVE-2007-4772, CVE-2007-6067)

- Require non-superusers who use `/contrib/dblink` to use only password authentication, as a security measure (Joe)

  The fix that appeared for this in 8.0.14 was incomplete, as it plugged the hole for only some `dblink` functions. (CVE-2007-6601, CVE-2007-3278)

- Update time zone data files to tzdata release 2007k (in particular, recent Argentina changes) (Tom)

- Fix planner failure in some cases of `WHERE false AND var IN (SELECT ...)` (Tom)

- Preserve the tablespace of indexes that are rebuilt by `ALTER TABLE ... ALTER COLUMN TYPE` (Tom)

- Make archive recovery always start a new WAL timeline, rather than only when a recovery stop time was used (Simon)

  This avoids a corner-case risk of trying to overwrite an existing archived copy of the last WAL segment, and seems simpler and cleaner than the original definition.

- Make `VACUUM` not use all of `maintenance_work_mem` when the table is too small for it to be useful (Alvaro)

- Fix potential crash in `translate()` when using a multibyte database encoding (Tom)

- Fix PL/Perl to cope when platform's Perl defines type `bool` as `int` rather than `char` (Tom)

  While this could theoretically happen anywhere, no standard build of Perl did things this way ... until Mac OS X 10.5.

- Fix PL/Python to not crash on long exception messages (Alvaro)
- Fix pg_dump to correctly handle inheritance child tables that have default expressions different from their parent's (Tom)
- ecpg parser fixes (Michael)
- Make `contrib/tablefunc`'s `crosstab()` handle NULL rowid as a category in its own right, rather than crashing (Joe)
- Fix `tsvector` and `tsquery` output routines to escape backslashes correctly (Teodor, Bruce)
- Fix crash of `to_tsvector()` on huge input strings (Teodor)
- Require a specific version of Autoconf to be used when re-generating the `configure` script (Peter)

  This affects developers and packagers only. The change was made to prevent accidental use of untested combinations of Autoconf and PostgreSQL versions. You can remove the version check if you really want to use a different Autoconf version, but it's your responsibility whether the result works or not.

E.21. Release 8.0.14

Release date: 2007-09-17

This release contains a variety of fixes from 8.0.13.

E.21.1. Migration to Version 8.0.14

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see the release notes for 8.0.6.

E.21.2. Changes

- Prevent index corruption when a transaction inserts rows and then aborts close to the end of a concurrent `VACUUM` on the same table (Tom)
- Make `CREATE DOMAIN ... DEFAULT NULL` work properly (Tom)
- Fix excessive logging of SSL error messages (Tom)
- Fix logging so that log messages are never interleaved when using the syslogger process (Andrew)
- Fix crash when `log_min_error_statement` logging runs out of memory (Tom)
- Fix incorrect handling of some foreign-key corner cases (Tom)

- Prevent `CLUSTER` from failing due to attempting to process temporary tables of other sessions (Alvaro)
- Update the time zone database rules, particularly New Zealand's upcoming changes (Tom)
- Windows socket improvements (Magnus)
- Suppress timezone name (`%Z`) in log timestamps on Windows because of possible encoding mismatches (Tom)
- Require non-superusers who use `/contrib/dblink` to use only password authentication, as a security measure (Joe)

E.22. Release 8.0.13

Release date: 2007-04-23

This release contains a variety of fixes from 8.0.12, including a security fix.

E.22.1. Migration to Version 8.0.13

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see the release notes for 8.0.6.

E.22.2. Changes

- Support explicit placement of the temporary-table schema within `search_path`, and disable searching it for functions and operators (Tom)

  This is needed to allow a security-definer function to set a truly secure value of `search_path`. Without it, an unprivileged SQL user can use temporary objects to execute code with the privileges of the security-definer function (CVE-2007-2138). See `CREATE FUNCTION` for more information.
- `/contrib/tsearch2` crash fixes (Teodor)
- Fix potential-data-corruption bug in how `VACUUM FULL` handles `UPDATE` chains (Tom, Pavan Deolasee)
- Fix PANIC during enlargement of a hash index (bug introduced in 8.0.10) (Tom)
- Fix POSIX-style timezone specs to follow new USA DST rules (Tom)

E.23. Release 8.0.12

Release date: 2007-02-07

This release contains one fix from 8.0.11.

E.23.1. Migration to Version 8.0.12

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see the release notes for 8.0.6.

E.23.2. Changes

- Remove overly-restrictive check for type length in constraints and functional indexes(Tom)

E.24. Release 8.0.11

Release date: 2007-02-05

This release contains a variety of fixes from 8.0.10, including a security fix.

E.24.1. Migration to Version 8.0.11

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see the release notes for 8.0.6.

E.24.2. Changes

- Remove security vulnerabilities that allowed connected users to read backend memory (Tom)

  The vulnerabilities involve suppressing the normal check that a SQL function returns the data type it's declared to, and changing the data type of a table column (CVE-2007-0555, CVE-2007-0556). These errors can easily be exploited to cause a backend crash, and in principle might be used to read database content that the user should not be able to access.
- Fix rare bug wherein btree index page splits could fail due to choosing an infeasible split point (Heikki Linnakangas)
- Fix for rare Assert() crash triggered by `UNION` (Tom)
- Tighten security of multi-byte character processing for UTF8 sequences over three bytes long (Tom)

E.25. Release 8.0.10

Release date: 2007-01-08

This release contains a variety of fixes from 8.0.9.

E.25.1. Migration to Version 8.0.10

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see the release notes for 8.0.6.

E.25.2. Changes

- Improve handling of `getaddrinfo()` on AIX (Tom)

  This fixes a problem with starting the statistics collector, among other things.
- Fix "failed to re-find parent key" errors in `VACUUM` (Tom)
- Fix race condition for truncation of a large relation across a gigabyte boundary by `VACUUM` (Tom)
- Fix bugs affecting multi-gigabyte hash indexes (Tom)
- Fix possible deadlock in Windows signal handling (Teodor)
- Fix error when constructing an `ARRAY[]` made up of multiple empty elements (Tom)
- Fix ecpg memory leak during connection (Michael)
- `to_number()` and `to_char(numeric)` are now `STABLE`, not `IMMUTABLE`, for new initdb installs (Tom)

  This is because `lc_numeric` can potentially change the output of these functions.
- Improve index usage of regular expressions that use parentheses (Tom)

  This improves psql `\d` performance also.
- Update timezone database

  This affects Australian and Canadian daylight-savings rules in particular.

E.26. Release 8.0.9

Release date: 2006-10-16

This release contains a variety of fixes from 8.0.8.

E.26.1. Migration to Version 8.0.9

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see the release notes for 8.0.6.

E.26.2. Changes

- Fix crash when referencing `NEW` row values in rule WHERE expressions (Tom)
- Fix core dump when an untyped literal is taken as ANYARRAY
- Fix mishandling of AFTER triggers when query contains a SQL function returning multiple rows (Tom)
- Fix `ALTER TABLE ... TYPE` to recheck `NOT NULL` for `USING` clause (Tom)
- Fix `string_to_array()` to handle overlapping matches for the separator string

  For example, `string_to_array('123xx456xxx789', 'xx')`.
- Fix corner cases in pattern matching for psql's `\d` commands
- Fix index-corrupting bugs in /contrib/ltree (Teodor)
- Numerous robustness fixes in ecpg (Joachim Wieland)
- Fix backslash escaping in /contrib/dbmirror
- Fix instability of statistics collection on Win32 (Tom, Andrew)
- Fixes for AIX and Intel compilers (Tom)

E.27. Release 8.0.8

Release date: 2006-05-23

This release contains a variety of fixes from 8.0.7, including patches for extremely serious security issues.

E.27.1. Migration to Version 8.0.8

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see the release notes for 8.0.6.

Full security against the SQL-injection attacks described in CVE-2006-2313 and CVE-2006-2314 might require changes in application code. If you have applications that embed untrustworthy strings into SQL commands, you should examine them as soon as possible to ensure that they are using recommended escaping techniques. In most cases, applications should be using subroutines provided by libraries or drivers (such as libpq's `PQescapeStringConn()`) to perform string escaping, rather than relying on *ad hoc* code to do it.

E.27.2. Changes

- Change the server to reject invalidly-encoded multibyte characters in all cases (Tatsuo, Tom)

  While PostgreSQL has been moving in this direction for some time, the checks are now applied uniformly to all encodings and all textual input, and are now always errors not merely warnings. This change defends against SQL-injection attacks of the type described in CVE-2006-2313.

- Reject unsafe uses of `\'` in string literals

  As a server-side defense against SQL-injection attacks of the type described in CVE-2006-2314, the server now only accepts `''` and not `\'` as a representation of ASCII single quote in SQL string literals. By default, `\'` is rejected only when `client_encoding` is set to a client-only encoding (SJIS, BIG5, GBK, GB18030, or UHC), which is the scenario in which SQL injection is possible. A new configuration parameter `backslash_quote` is available to adjust this behavior when needed. Note that full security against CVE-2006-2314 might require client-side changes; the purpose of `backslash_quote` is in part to make it obvious that insecure clients are insecure.

- Modify libpq's string-escaping routines to be aware of encoding considerations and `standard_conforming_strings`

  This fixes libpq-using applications for the security issues described in CVE-2006-2313 and CVE-2006-2314, and also future-proofs them against the planned changeover to SQL-standard string literal syntax. Applications that use multiple PostgreSQL connections concurrently should migrate to `PQescapeStringConn()` and `PQescapeByteaConn()` to ensure that escaping is done correctly for the settings in use in each database connection. Applications that do string escaping "by hand" should be modified to rely on library routines instead.

- Fix some incorrect encoding conversion functions

  `win1251_to_iso`, `alt_to_iso`, `euc_tw_to_big5`, `euc_tw_to_mic`, `mic_to_euc_tw` were all broken to varying extents.

- Clean up stray remaining uses of `\'` in strings (Bruce, Jan)

- Fix bug that sometimes caused OR'd index scans to miss rows they should have returned

- Fix WAL replay for case where a btree index has been truncated

- Fix `SIMILAR TO` for patterns involving `|` (Tom)

- Fix `SELECT INTO` and `CREATE TABLE AS` to create tables in the default tablespace, not the base directory (Kris Jurka)

- Fix server to use custom DH SSL parameters correctly (Michael Fuhr)

- Fix for Bonjour on Intel Macs (Ashley Clark)

- Fix various minor memory leaks

- Fix problem with password prompting on some Win32 systems (Robert Kinberg)

E.28. Release 8.0.7

Release date: 2006-02-14

This release contains a variety of fixes from 8.0.6.

E.28.1. Migration to Version 8.0.7

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see the release notes for 8.0.6.

E.28.2. Changes

- Fix potential crash in `SET SESSION AUTHORIZATION` (CVE-2006-0553)

  An unprivileged user could crash the server process, resulting in momentary denial of service to other users, if the server has been compiled with Asserts enabled (which is not the default). Thanks to Akio Ishida for reporting this problem.

- Fix bug with row visibility logic in self-inserted rows (Tom)

  Under rare circumstances a row inserted by the current command could be seen as already valid, when it should not be. Repairs bug created in 8.0.4, 7.4.9, and 7.3.11 releases.

- Fix race condition that could lead to "file already exists" errors during pg_clog and pg_subtrans file creation (Tom)

- Fix cases that could lead to crashes if a cache-invalidation message arrives at just the wrong time (Tom)

- Properly check `DOMAIN` constraints for `UNKNOWN` parameters in prepared statements (Neil)

- Ensure `ALTER COLUMN TYPE` will process `FOREIGN KEY`, `UNIQUE`, and `PRIMARY KEY` constraints in the proper order (Nakano Yoshihisa)

- Fixes to allow restoring dumps that have cross-schema references to custom operators or operator classes (Tom)

- Allow pg_restore to continue properly after a `COPY` failure; formerly it tried to treat the remaining `COPY` data as SQL commands (Stephen Frost)

- Fix pg_ctl `unregister` crash when the data directory is not specified (Magnus)

- Fix ecpg crash on AMD64 and PPC (Neil)

- Recover properly if error occurs during argument passing in PL/python (Neil)

- Fix PL/perl's handling of locales on Win32 to match the backend (Andrew)

- Fix crash when `log_min_messages` is set to `DEBUG3` or above in `postgresql.conf` on Win32 (Bruce)

- Fix pgxs `-L` library path specification for Win32, Cygwin, OS X, AIX (Bruce)

- Check that SID is enabled while checking for Win32 admin privileges (Magnus)
- Properly reject out-of-range date inputs (Kris Jurka)
- Portability fix for testing presence of `finite` and `isinf` during configure (Tom)

E.29. Release 8.0.6

Release date: 2006-01-09

This release contains a variety of fixes from 8.0.5.

E.29.1. Migration to Version 8.0.6

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.3, see the release notes for 8.0.3. Also, you might need to `REINDEX` indexes on textual columns after updating, if you are affected by the locale or plperl issues described below.

E.29.2. Changes

- Fix Windows code so that postmaster will continue rather than exit if there is no more room in ShmemBackendArray (Magnus)

  The previous behavior could lead to a denial-of-service situation if too many connection requests arrive close together. This applies *only* to the Windows port.
- Fix bug introduced in 8.0 that could allow ReadBuffer to return an already-used page as new, potentially causing loss of recently-committed data (Tom)
- Fix for protocol-level Describe messages issued outside a transaction or in a failed transaction (Tom)
- Fix character string comparison for locales that consider different character combinations as equal, such as Hungarian (Tom)

  This might require `REINDEX` to fix existing indexes on textual columns.
- Set locale environment variables during postmaster startup to ensure that plperl won't change the locale later

  This fixes a problem that occurred if the postmaster was started with environment variables specifying a different locale than what initdb had been told. Under these conditions, any use of plperl was likely to lead to corrupt indexes. You might need `REINDEX` to fix existing indexes on textual columns if this has happened to you.
- Allow more flexible relocation of installation directories (Tom)

  Previous releases supported relocation only if all installation directory paths were the same except for the last component.

- Fix longstanding bug in strpos() and regular expression handling in certain rarely used Asian multi-byte character sets (Tatsuo)
- Various fixes for functions returning `RECORD`s (Tom)
- Fix bug in `/contrib/pgcrypto` gen_salt, which caused it not to use all available salt space for MD5 and XDES algorithms (Marko Kreen, Solar Designer)

  Salts for Blowfish and standard DES are unaffected.
- Fix `/contrib/dblink` to throw an error, rather than crashing, when the number of columns specified is different from what's actually returned by the query (Joe)

E.30. Release 8.0.5

Release date: 2005-12-12

This release contains a variety of fixes from 8.0.4.

E.30.1. Migration to Version 8.0.5

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.3, see the release notes for 8.0.3.

E.30.2. Changes

- Fix race condition in transaction log management

  There was a narrow window in which an I/O operation could be initiated for the wrong page, leading to an Assert failure or data corruption.
- Fix bgwriter problems after recovering from errors (Tom)

  The background writer was found to leak buffer pins after write errors. While not fatal in itself, this might lead to mysterious blockages of later VACUUM commands.
- Prevent failure if client sends Bind protocol message when current transaction is already aborted
- `/contrib/ltree` fixes (Teodor)
- AIX and HPUX compile fixes (Tom)
- Retry file reads and writes after Windows NO_SYSTEM_RESOURCES error (Qingqing Zhou)
- Fix intermittent failure when `log_line_prefix` includes `%i`
- Fix psql performance issue with long scripts on Windows (Merlin Moncure)
- Fix missing updates of `pg_group` flat file

- Fix longstanding planning error for outer joins

  This bug sometimes caused a bogus error "RIGHT JOIN is only supported with merge-joinable join conditions".

- Postpone timezone initialization until after `postmaster.pid` is created

  This avoids confusing startup scripts that expect the pid file to appear quickly.

- Prevent core dump in pg_autovacuum when a table has been dropped

- Fix problems with whole-row references (`foo.*`) to subquery results

E.31. Release 8.0.4

Release date: 2005-10-04

This release contains a variety of fixes from 8.0.3.

E.31.1. Migration to Version 8.0.4

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.3, see the release notes for 8.0.3.

E.31.2. Changes

- Fix error that allowed `VACUUM` to remove `ctid` chains too soon, and add more checking in code that follows `ctid` links

  This fixes a long-standing problem that could cause crashes in very rare circumstances.

- Fix `CHAR()` to properly pad spaces to the specified length when using a multiple-byte character set (Yoshiyuki Asaba)

  In prior releases, the padding of `CHAR()` was incorrect because it only padded to the specified number of bytes without considering how many characters were stored.

- Force a checkpoint before committing `CREATE DATABASE`

  This should fix recent reports of "index is not a btree" failures when a crash occurs shortly after `CREATE DATABASE`.

- Fix the sense of the test for read-only transaction in `COPY`

  The code formerly prohibited `COPY TO`, where it should prohibit `COPY FROM`.

- Handle consecutive embedded newlines in `COPY` CSV-mode input

- Fix `date_trunc(week)` for dates near year end

- Fix planning problem with outer-join ON clauses that reference only the inner-side relation
- Further fixes for `x FULL JOIN y ON true` corner cases
- Fix overenthusiastic optimization of `x IN (SELECT DISTINCT ...)` and related cases
- Fix mis-planning of queries with small `LIMIT` values due to poorly thought out "fuzzy" cost comparison
- Make `array_in` and `array_recv` more paranoid about validating their OID parameter
- Fix missing rows in queries like `UPDATE a=... WHERE a...` with GiST index on column `a`
- Improve robustness of datetime parsing
- Improve checking for partially-written WAL pages
- Improve robustness of signal handling when SSL is enabled
- Improve MIPS and M68K spinlock code
- Don't try to open more than `max_files_per_process` files during postmaster startup
- Various memory leakage fixes
- Various portability improvements
- Update timezone data files
- Improve handling of DLL load failures on Windows
- Improve random-number generation on Windows
- Make `psql -f filename` return a nonzero exit code when opening the file fails
- Change pg_dump to handle inherited check constraints more reliably
- Fix password prompting in pg_restore on Windows
- Fix PL/PgSQL to handle `var := var` correctly when the variable is of pass-by-reference type
- Fix PL/Perl `%_SHARED` so it's actually shared
- Fix `contrib/pg_autovacuum` to allow sleep intervals over 2000 sec
- Update `contrib/tsearch2` to use current Snowball code

E.32. Release 8.0.3

Release date: 2005-05-09

This release contains a variety of fixes from 8.0.2, including several security-related issues.

E.32.1. Migration to Version 8.0.3

A dump/restore is not required for those running 8.0.X. However, it is one possible way of handling two significant security problems that have been found in the initial contents of 8.0.X system catalogs. A dump/initdb/reload sequence using 8.0.3's initdb will automatically correct these problems.

The larger security problem is that the built-in character set encoding conversion functions can be invoked from SQL commands by unprivileged users, but the functions were not designed for such use and are not secure against malicious choices of arguments. The fix involves changing the declared parameter list of these functions so that they can no longer be invoked from SQL commands. (This does not affect their normal use by the encoding conversion machinery.)

The lesser problem is that the `contrib/tsearch2` module creates several functions that are improperly declared to return `internal` when they do not accept `internal` arguments. This breaks type safety for all functions using `internal` arguments.

It is strongly recommended that all installations repair these errors, either by initdb or by following the manual repair procedure given below. The errors at least allow unprivileged database users to crash their server process, and might allow unprivileged users to gain the privileges of a database superuser.

If you wish not to do an initdb, perform the same manual repair procedures shown in the 7.4.8 release notes.

E.32.2. Changes

- Change encoding function signature to prevent misuse
- Change `contrib/tsearch2` to avoid unsafe use of `INTERNAL` function results
- Guard against incorrect second parameter to `record_out`
- Repair ancient race condition that allowed a transaction to be seen as committed for some purposes (eg SELECT FOR UPDATE) slightly sooner than for other purposes

  This is an extremely serious bug since it could lead to apparent data inconsistencies being briefly visible to applications.
- Repair race condition between relation extension and VACUUM

  This could theoretically have caused loss of a page's worth of freshly-inserted data, although the scenario seems of very low probability. There are no known cases of it having caused more than an Assert failure.
- Fix comparisons of `TIME WITH TIME ZONE` values

  The comparison code was wrong in the case where the `--enable-integer-datetimes` configuration switch had been used. NOTE: if you have an index on a `TIME WITH TIME ZONE` column, it will need to be `REINDEX`ed after installing this update, because the fix corrects the sort order of column values.
- Fix `EXTRACT(EPOCH)` for `TIME WITH TIME ZONE` values
- Fix mis-display of negative fractional seconds in `INTERVAL` values

  This error only occurred when the `--enable-integer-datetimes` configuration switch had been used.
- Fix pg_dump to dump trigger names containing `%` correctly (Neil)
- Still more 64-bit fixes for `contrib/intagg`
- Prevent incorrect optimization of functions returning `RECORD`

- Prevent crash on `COALESCE(NULL,NULL)`
- Fix Borland makefile for libpq
- Fix `contrib/btree_gist` for `timetz` type (Teodor)
- Make `pg_ctl` check the PID found in `postmaster.pid` to see if it is still a live process
- Fix `pg_dump`/`pg_restore` problems caused by addition of dump timestamps
- Fix interaction between materializing holdable cursors and firing deferred triggers during transaction commit
- Fix memory leak in SQL functions returning pass-by-reference data types

E.33. Release 8.0.2

Release date: 2005-04-07

This release contains a variety of fixes from 8.0.1.

E.33.1. Migration to Version 8.0.2

A dump/restore is not required for those running 8.0.*. This release updates the major version number of the PostgreSQL libraries, so it might be necessary to re-link some user applications if they cannot find the properly-numbered shared library.

E.33.2. Changes

- Increment the major version number of all interface libraries (Bruce)

  This should have been done in 8.0.0. It is required so 7.4.X versions of PostgreSQL client applications, like psql, can be used on the same machine as 8.0.X applications. This might require re-linking user applications that use these libraries.

- Add Windows-only `wal_sync_method` setting of `fsync_writethrough` (Magnus, Bruce)

  This setting causes PostgreSQL to write through any disk-drive write cache when writing to WAL. This behavior was formerly called `fsync`, but was renamed because it acts quite differently from `fsync` on other platforms.

- Enable the `wal_sync_method` setting of `open_datasync` on Windows, and make it the default for that platform (Magnus, Bruce)

  Because the default is no longer `fsync_writethrough`, data loss is possible during a power failure if the disk drive has write caching enabled. To turn off the write cache on Windows, from the Device Manager, choose the drive properties, then `Policies`.

- New cache management algorithm 2Q replaces ARC (Tom)

  This was done to avoid a pending US patent on ARC. The 2Q code might be a few percentage points slower than ARC for some work loads. A better cache management algorithm will appear in 8.1.

- Planner adjustments to improve behavior on freshly-created tables (Tom)
- Allow plpgsql to assign to an element of an array that is initially `NULL` (Tom)

  Formerly the array would remain `NULL`, but now it becomes a single-element array. The main SQL engine was changed to handle `UPDATE` of a null array value this way in 8.0, but the similar case in plpgsql was overlooked.

- Convert `\r\n` and `\r` to `\n` in plpython function bodies (Michael Fuhr)

  This prevents syntax errors when plpython code is written on a Windows or Mac client.

- Allow SPI cursors to handle utility commands that return rows, such as `EXPLAIN` (Tom)
- Fix `CLUSTER` failure after `ALTER TABLE SET WITHOUT OIDS` (Tom)
- Reduce memory usage of `ALTER TABLE ADD COLUMN` (Neil)
- Fix `ALTER LANGUAGE RENAME` (Tom)
- Document the Windows-only `register` and `unregister` options of pg_ctl (Magnus)
- Ensure operations done during backend shutdown are counted by statistics collector

  This is expected to resolve reports of pg_autovacuum not vacuuming the system catalogs often enough — it was not being told about catalog deletions caused by temporary table removal during backend exit.

- Change the Windows default for configuration parameter `log_destination` to `eventlog` (Magnus)

  By default, a server running on Windows will now send log output to the Windows event logger rather than standard error.

- Make Kerberos authentication work on Windows (Magnus)
- Allow `ALTER DATABASE RENAME` by superusers who aren't flagged as having CREATEDB privilege (Tom)
- Modify WAL log entries for `CREATE` and `DROP DATABASE` to not specify absolute paths (Tom)

  This allows point-in-time recovery on a different machine with possibly different database location. Note that `CREATE TABLESPACE` still poses a hazard in such situations.

- Fix crash from a backend exiting with an open transaction that created a table and opened a cursor on it (Tom)
- Fix `array_map()` so it can call PL functions (Tom)
- Several `contrib/tsearch2` and `contrib/btree_gist` fixes (Teodor)
- Fix crash of some `contrib/pgcrypto` functions on some platforms (Marko Kreen)
- Fix `contrib/intagg` for 64-bit platforms (Tom)
- Fix ecpg bugs in parsing of `CREATE` statement (Michael)
- Work around gcc bug on powerpc and amd64 causing problems in ecpg (Christof Petig)
- Do not use locale-aware versions of `upper()`, `lower()`, and `initcap()` when the locale is `C` (Bruce)

This allows these functions to work on platforms that generate errors for non-7-bit data when the locale is C.

- Fix quote_ident() to quote names that match keywords (Tom)
- Fix to_date() to behave reasonably when CC and YY fields are both used (Karel)
- Prevent to_char(interval) from failing when given a zero-month interval (Tom)
- Fix wrong week returned by date_trunc('week') (Bruce)

  date_trunc('week') returned the wrong year for the first few days of January in some years.
- Use the correct default mask length for class D addresses in INET data types (Tom)

E.34. Release 8.0.1

Release date: 2005-01-31

This release contains a variety of fixes from 8.0.0, including several security-related issues.

E.34.1. Migration to Version 8.0.1

A dump/restore is not required for those running 8.0.0.

E.34.2. Changes

- Disallow LOAD to non-superusers

  On platforms that will automatically execute initialization functions of a shared library (this includes at least Windows and ELF-based Unixen), LOAD can be used to make the server execute arbitrary code. Thanks to NGS Software for reporting this.
- Check that creator of an aggregate function has the right to execute the specified transition functions

  This oversight made it possible to bypass denial of EXECUTE permission on a function.
- Fix security and 64-bit issues in contrib/intagg
- Add needed STRICT marking to some contrib functions (Kris Jurka)
- Avoid buffer overrun when plpgsql cursor declaration has too many parameters (Neil)
- Make ALTER TABLE ADD COLUMN enforce domain constraints in all cases
- Fix planning error for FULL and RIGHT outer joins

  The result of the join was mistakenly supposed to be sorted the same as the left input. This could not only deliver mis-sorted output to the user, but in case of nested merge joins could give outright wrong answers.

- Improve planning of grouped aggregate queries
- `ROLLBACK TO` *`savepoint`* closes cursors created since the savepoint
- Fix inadequate backend stack size on Windows
- Avoid SHGetSpecialFolderPath() on Windows (Magnus)
- Fix some problems in running pg_autovacuum as a Windows service (Dave Page)
- Multiple minor bug fixes in pg_dump/pg_restore
- Fix ecpg segfault with named structs used in typedefs (Michael)

E.35. Release 8.0

Release date: 2005-01-19

E.35.1. Overview

Major changes in this release:

Microsoft Windows Native Server

This is the first PostgreSQL release to run natively on Microsoft Windows® as a server. It can run as a Windows service. This release supports NT-based Windows releases like Windows 2000 SP4, Windows XP, and Windows 2003. Older releases like Windows 95, Windows 98, and Windows ME are not supported because these operating systems do not have the infrastructure to support PostgreSQL. A separate installer project has been created to ease installation on Windows — see http://www.postgresql.org/ftp/win32/.

Although tested throughout our release cycle, the Windows port does not have the benefit of years of use in production environments that PostgreSQL has on Unix platforms. Therefore it should be treated with the same level of caution as you would a new product.

Previous releases required the Unix emulation toolkit Cygwin in order to run the server on Windows operating systems. PostgreSQL has supported native clients on Windows for many years.

Savepoints

Savepoints allow specific parts of a transaction to be aborted without affecting the remainder of the transaction. Prior releases had no such capability; there was no way to recover from a statement failure within a transaction except by aborting the whole transaction. This feature is valuable for application writers who require error recovery within a complex transaction.

Point-In-Time Recovery

In previous releases there was no way to recover from disk drive failure except to restore from a previous backup or use a standby replication server. Point-in-time recovery allows continuous backup of the server. You can recover either to the point of failure or to some transaction in the past.

Tablespaces

Tablespaces allow administrators to select different file systems for storage of individual tables, indexes, and databases. This improves performance and control over disk space usage. Prior releases used initlocation and manual symlink management for such tasks.

Improved Buffer Management, CHECKPOINT, VACUUM

This release has a more intelligent buffer replacement strategy, which will make better use of available shared buffers and improve performance. The performance impact of vacuum and checkpoints is also lessened.

Change Column Types

A column's data type can now be changed with ALTER TABLE.

New Perl Server-Side Language

A new version of the plperl server-side language now supports a persistent shared storage area, triggers, returning records and arrays of records, and SPI calls to access the database.

Comma-separated-value (CSV) support in COPY

COPY can now read and write comma-separated-value files. It has the flexibility to interpret nonstandard quoting and separation characters too.

E.35.2. Migration to Version 8.0

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release.

Observe the following incompatibilities:

- In READ COMMITTED serialization mode, volatile functions now see the results of concurrent transactions committed up to the beginning of each statement within the function, rather than up to the beginning of the interactive command that called the function.
- Functions declared STABLE or IMMUTABLE always use the snapshot of the calling query, and therefore do not see the effects of actions taken after the calling query starts, whether in their own transaction or other transactions. Such a function must be read-only, too, meaning that it cannot use any SQL commands other than SELECT.
- Nondeferred AFTER triggers are now fired immediately after completion of the triggering query, rather than upon finishing the current interactive command. This makes a difference when the triggering query occurred within a function: the trigger is invoked before the function proceeds to its next operation.
- Server configuration parameters virtual_host and tcpip_socket have been replaced with a more general parameter listen_addresses. Also, the server now listens on localhost by default, which eliminates the need for the -i postmaster switch in many scenarios.
- Server configuration parameters SortMem and VacuumMem have been renamed to work_mem and maintenance_work_mem to better reflect their use. The original names are still supported in SET and SHOW.
- Server configuration parameters log_pid, log_timestamp, and log_source_port have been replaced with a more general parameter log_line_prefix.

- Server configuration parameter `syslog` has been replaced with a more logical `log_destination` variable to control the log output destination.
- Server configuration parameter `log_statement` has been changed so it can selectively log just database modification or data definition statements. Server configuration parameter `log_duration` now prints only when `log_statement` prints the query.
- Server configuration parameter `max_expr_depth` parameter has been replaced with `max_stack_depth` which measures the physical stack size rather than the expression nesting depth. This helps prevent session termination due to stack overflow caused by recursive functions.
- The `length()` function no longer counts trailing spaces in `CHAR(n)` values.
- Casting an integer to `BIT(N)` selects the rightmost N bits of the integer, not the leftmost N bits as before.
- Updating an element or slice of a NULL array value now produces a nonnull array result, namely an array containing just the assigned-to positions.
- Syntax checking of array input values has been tightened up considerably. Junk that was previously allowed in odd places with odd results now causes an error. Empty-string element values must now be written as `""`, rather than writing nothing. Also changed behavior with respect to whitespace surrounding array elements: trailing whitespace is now ignored, for symmetry with leading whitespace (which has always been ignored).
- Overflow in integer arithmetic operations is now detected and reported as an error.
- The arithmetic operators associated with the single-byte `"char"` data type have been removed.
- The `extract()` function (also called `date_part`) now returns the proper year for BC dates. It previously returned one less than the correct year. The function now also returns the proper values for millennium and century.
- `CIDR` values now must have their nonmasked bits be zero. For example, we no longer allow `204.248.199.1/31` as a `CIDR` value. Such values should never have been accepted by PostgreSQL and will now be rejected.
- `EXECUTE` now returns a completion tag that matches the executed statement.
- psql's `\copy` command now reads or writes to the query's `stdin/stdout`, rather than psql's `stdin/stdout`. The previous behavior can be accessed via new `pstdin`/`pstdout` parameters.
- The JDBC client interface has been removed from the core distribution, and is now hosted at http://jdbc.postgresql.org.
- The Tcl client interface has also been removed. There are several Tcl interfaces now hosted at http://gborg.postgresql.org.
- The server now uses its own time zone database, rather than the one supplied by the operating system. This will provide consistent behavior across all platforms. In most cases, there should be little noticeable difference in time zone behavior, except that the time zone names used by `SET`/`SHOW TimeZone` might be different from what your platform provides.
- Configure's threading option no longer requires users to run tests or edit configuration files; threading options are now detected automatically.
- Now that tablespaces have been implemented, initlocation has been removed.

- The API for user-defined GiST indexes has been changed. The Union and PickSplit methods are now passed a pointer to a special `GistEntryVector` structure, rather than a `bytea`.

E.35.3. Deprecated Features

Some aspects of PostgreSQL's behavior have been determined to be suboptimal. For the sake of backward compatibility these have not been removed in 8.0, but they are considered deprecated and will be removed in the next major release.

- The 8.1 release will remove the `to_char()` function for intervals.
- The server now warns of empty strings passed to `oid`/`float4`/`float8` data types, but continues to interpret them as zeroes as before. In the next major release, empty strings will be considered invalid input for these data types.
- By default, tables in PostgreSQL 8.0 and earlier are created with `OID`s. In the next release, this will *not* be the case: to create a table that contains `OID`s, the `WITH OIDS` clause must be specified or the `default_with_oids` configuration parameter must be set. Users are encouraged to explicitly specify `WITH OIDS` if their tables require OIDs for compatibility with future releases of PostgreSQL.

E.35.4. Changes

Below you will find a detailed account of the changes between release 8.0 and the previous major release.

E.35.4.1. Performance Improvements

- Support cross-data-type index usage (Tom)

  Before this change, many queries would not use an index if the data types did not match exactly. This improvement makes index usage more intuitive and consistent.

- New buffer replacement strategy that improves caching (Jan)

  Prior releases used a least-recently-used (LRU) cache to keep recently referenced pages in memory. The LRU algorithm did not consider the number of times a specific cache entry was accessed, so large table scans could force out useful cache pages. The new cache algorithm uses four separate lists to track most recently used and most frequently used cache pages and dynamically optimize their replacement based on the work load. This should lead to much more efficient use of the shared buffer cache. Administrators who have tested shared buffer sizes in the past should retest with this new cache replacement policy.

- Add subprocess to write dirty buffers periodically to reduce checkpoint writes (Jan)

  In previous releases, the checkpoint process, which runs every few minutes, would write all dirty buffers to the operating system's buffer cache then flush all dirty operating system buffers to disk. This resulted in a periodic spike in disk usage that often hurt performance. The new code uses a background writer to trickle disk writes at a steady pace so checkpoints have far fewer dirty pages to write to disk. Also, the new code does not issue a global `sync()` call, but instead `fsync()`s just the files written since the last checkpoint. This should improve performance and minimize degradation during checkpoints.

- Add ability to prolong vacuum to reduce performance impact (Jan)

  On busy systems, `VACUUM` performs many I/O requests which can hurt performance for other users. This release allows you to slow down `VACUUM` to reduce its impact on other users, though this increases the total duration of `VACUUM`.

- Improve B-tree index performance for duplicate keys (Dmitry Tkach, Tom)

  This improves the way indexes are scanned when many duplicate values exist in the index.

- Use dynamically-generated table size estimates while planning (Tom)

  Formerly the planner estimated table sizes using the values seen by the last `VACUUM` or `ANALYZE`, both as to physical table size (number of pages) and number of rows. Now, the current physical table size is obtained from the kernel, and the number of rows is estimated by multiplying the table size by the row density (rows per page) seen by the last `VACUUM` or `ANALYZE`. This should produce more reliable estimates in cases where the table size has changed significantly since the last housekeeping command.

- Improved index usage with `OR` clauses (Tom)

  This allows the optimizer to use indexes in statements with many OR clauses that would not have been indexed in the past. It can also use multi-column indexes where the first column is specified and the second column is part of an `OR` clause.

- Improve matching of partial index clauses (Tom)

  The server is now smarter about using partial indexes in queries involving complex `WHERE` clauses.

- Improve performance of the GEQO optimizer (Tom)

  The GEQO optimizer is used to plan queries involving many tables (by default, twelve or more). This release speeds up the way queries are analyzed to decrease time spent in optimization.

- Miscellaneous optimizer improvements

  There is not room here to list all the minor improvements made, but numerous special cases work better than in prior releases.

- Improve lookup speed for C functions (Tom)

  This release uses a hash table to lookup information for dynamically loaded C functions. This improves their speed so they perform nearly as quickly as functions that are built into the server executable.

- Add type-specific `ANALYZE` statistics capability (Mark Cave-Ayland)

  This feature allows more flexibility in generating statistics for nonstandard data types.

- `ANALYZE` now collects statistics for expression indexes (Tom)

  Expression indexes (also called functional indexes) allow users to index not just columns but the results of expressions and function calls. With this release, the optimizer can gather and use statistics about the contents of expression indexes. This will greatly improve the quality of planning for queries in which an expression index is relevant.

- New two-stage sampling method for `ANALYZE` (Manfred Koizar)

  This gives better statistics when the density of valid rows is very different in different regions of a table.

- Speed up `TRUNCATE` (Tom)

  This buys back some of the performance loss observed in 7.4, while still keeping `TRUNCATE` transaction-safe.

E.35.4.2. Server Changes

- Add WAL file archiving and point-in-time recovery (Simon Riggs)
- Add tablespaces so admins can control disk layout (Gavin)
- Add a built-in log rotation program (Andreas Pflug)

  It is now possible to log server messages conveniently without relying on either syslog or an external log rotation program.
- Add new read-only server configuration parameters to show server compile-time settings: `block_size`, `integer_datetimes`, `max_function_args`, `max_identifier_length`, `max_index_keys` (Joe)
- Make quoting of `sameuser`, `samegroup`, and `all` remove special meaning of these terms in `pg_hba.conf` (Andrew)
- Use clearer IPv6 name `::1/128` for `localhost` in default `pg_hba.conf` (Andrew)
- Use CIDR format in `pg_hba.conf` examples (Andrew)
- Rename server configuration parameters `SortMem` and `VacuumMem` to `work_mem` and `maintenance_work_mem` (Old names still supported) (Tom)

  This change was made to clarify that bulk operations such as index and foreign key creation use `maintenance_work_mem`, while `work_mem` is for workspaces used during query execution.
- Allow logging of session disconnections using server configuration `log_disconnections` (Andrew)
- Add new server configuration parameter `log_line_prefix` to allow control of information emitted in each log line (Andrew)

  Available information includes user name, database name, remote IP address, and session start time.
- Remove server configuration parameters `log_pid`, `log_timestamp`, `log_source_port`; functionality superseded by `log_line_prefix` (Andrew)
- Replace the `virtual_host` and `tcpip_socket` parameters with a unified `listen_addresses` parameter (Andrew, Tom)

  `virtual_host` could only specify a single IP address to listen on. `listen_addresses` allows multiple addresses to be specified.
- Listen on localhost by default, which eliminates the need for the `-i` postmaster switch in many scenarios (Andrew)

  Listening on localhost (`127.0.0.1`) opens no new security holes but allows configurations like Windows and JDBC, which do not support local sockets, to work without special adjustments.
- Remove `syslog` server configuration parameter, and add more logical `log_destination` variable to control log output location (Magnus)
- Change server configuration parameter `log_statement` to take values `all`, `mod`, `ddl`, or `none` to select which queries are logged (Bruce)

  This allows administrators to log only data definition changes or only data modification statements.
- Some logging-related configuration parameters could formerly be adjusted by ordinary users, but only in the "more verbose" direction. They are now treated more strictly: only superusers can set them. How-

ever, a superuser can use ALTER USER to provide per-user settings of these values for non-superusers. Also, it is now possible for superusers to set values of superuser-only configuration parameters via PGOPTIONS.

- Allow configuration files to be placed outside the data directory (mlw)

  By default, configuration files are kept in the cluster's top directory. With this addition, configuration files can be placed outside the data directory, easing administration.

- Plan prepared queries only when first executed so constants can be used for statistics (Oliver Jowett)

  Prepared statements plan queries once and execute them many times. While prepared queries avoid the overhead of re-planning on each use, the quality of the plan suffers from not knowing the exact parameters to be used in the query. In this release, planning of unnamed prepared statements is delayed until the first execution, and the actual parameter values of that execution are used as optimization hints. This allows use of out-of-line parameter passing without incurring a performance penalty.

- Allow DECLARE CURSOR to take parameters (Oliver Jowett)

  It is now useful to issue DECLARE CURSOR in a Parse message with parameters. The parameter values sent at Bind time will be substituted into the execution of the cursor's query.

- Fix hash joins and aggregates of inet and cidr data types (Tom)

  Release 7.4 handled hashing of mixed inet and cidr values incorrectly. (This bug did not exist in prior releases because they wouldn't try to hash either data type.)

- Make log_duration print only when log_statement prints the query (Ed L.)

E.35.4.3. Query Changes

- Add savepoints (nested transactions) (Alvaro)
- Unsupported isolation levels are now accepted and promoted to the nearest supported level (Peter)

  The SQL specification states that if a database doesn't support a specific isolation level, it should use the next more restrictive level. This change complies with that recommendation.

- Allow BEGIN WORK to specify transaction isolation levels like START TRANSACTION does (Bruce)
- Fix table permission checking for cases in which rules generate a query type different from the originally submitted query (Tom)
- Implement dollar quoting to simplify single-quote usage (Andrew, Tom, David Fetter)

  In previous releases, because single quotes had to be used to quote a function's body, the use of single quotes inside the function text required use of two single quotes or other error-prone notations. With this release we add the ability to use "dollar quoting" to quote a block of text. The ability to use different quoting delimiters at different nesting levels greatly simplifies the task of quoting correctly, especially in complex functions. Dollar quoting can be used anywhere quoted text is needed.

- Make CASE val WHEN compval1 THEN ... evaluate val only once (Tom)

  CASE no longer evaluates the tested expression multiple times. This has benefits when the expression is complex or is volatile.

- Test HAVING before computing target list of an aggregate query (Tom)

Fixes improper failure of cases such as `SELECT SUM(win)/SUM(lose) ... GROUP BY ... HAVING SUM(lose) > 0`. This should work but formerly could fail with divide-by-zero.

- Replace `max_expr_depth` parameter with `max_stack_depth` parameter, measured in kilobytes of stack size (Tom)

  This gives us a fairly bulletproof defense against crashing due to runaway recursive functions. Instead of measuring the depth of expression nesting, we now directly measure the size of the execution stack.

- Allow arbitrary row expressions (Tom)

  This release allows SQL expressions to contain arbitrary composite types, that is, row values. It also allows functions to more easily take rows as arguments and return row values.

- Allow `LIKE`/`ILIKE` to be used as the operator in row and subselect comparisons (Fabien Coelho)

- Avoid locale-specific case conversion of basic ASCII letters in identifiers and keywords (Tom)

  This solves the "Turkish problem" with mangling of words containing `I` and `i`. Folding of characters outside the 7-bit-ASCII set is still locale-aware.

- Improve syntax error reporting (Fabien, Tom)

  Syntax error reports are more useful than before.

- Change `EXECUTE` to return a completion tag matching the executed statement (Kris Jurka)

  Previous releases return an `EXECUTE` tag for any `EXECUTE` call. In this release, the tag returned will reflect the command executed.

- Avoid emitting `NATURAL CROSS JOIN` in rule listings (Tom)

  Such a clause makes no logical sense, but in some cases the rule decompiler formerly produced this syntax.

E.35.4.4. Object Manipulation Changes

- Add `COMMENT ON` for casts, conversions, languages, operator classes, and large objects (Christopher)
- Add new server configuration parameter `default_with_oids` to control whether tables are created with `OID`s by default (Neil)

  This allows administrators to control whether `CREATE TABLE` commands create tables with or without `OID` columns by default. (Note: the current factory default setting for `default_with_oids` is `TRUE`, but the default will become `FALSE` in future releases.)

- Add `WITH` / `WITHOUT OIDS` clause to `CREATE TABLE AS` (Neil)
- Allow `ALTER TABLE DROP COLUMN` to drop an `OID` column (`ALTER TABLE SET WITHOUT OIDS` still works) (Tom)
- Allow composite types as table columns (Tom)
- Allow `ALTER ... ADD COLUMN` with defaults and `NOT NULL` constraints; works per SQL spec (Rod)

  It is now possible for `ADD COLUMN` to create a column that is not initially filled with NULLs, but with a specified default value.

- Add `ALTER COLUMN TYPE` to change column's type (Rod)

It is now possible to alter a column's data type without dropping and re-adding the column.

- Allow multiple `ALTER` actions in a single `ALTER TABLE` command (Rod)

  This is particularly useful for `ALTER` commands that rewrite the table (which include `ALTER COLUMN TYPE` and `ADD COLUMN` with a default). By grouping `ALTER` commands together, the table need be rewritten only once.

- Allow `ALTER TABLE` to add `SERIAL` columns (Tom)

  This falls out from the new capability of specifying defaults for new columns.

- Allow changing the owners of aggregates, conversions, databases, functions, operators, operator classes, schemas, types, and tablespaces (Christopher, Euler Taveira de Oliveira)

  Previously this required modifying the system tables directly.

- Allow temporary object creation to be limited to `SECURITY DEFINER` functions (Sean Chittenden)

- Add `ALTER TABLE ... SET WITHOUT CLUSTER` (Christopher)

  Prior to this release, there was no way to clear an auto-cluster specification except to modify the system tables.

- Constraint/Index/`SERIAL` names are now *`table_column_type`* with numbers appended to guarantee uniqueness within the schema (Tom)

  The SQL specification states that such names should be unique within a schema.

- Add `pg_get_serial_sequence()` to return a `SERIAL` column's sequence name (Christopher)

  This allows automated scripts to reliably find the `SERIAL` sequence name.

- Warn when primary/foreign key data type mismatch requires costly lookup

- New `ALTER INDEX` command to allow moving of indexes between tablespaces (Gavin)

- Make `ALTER TABLE OWNER` change dependent sequence ownership too (Alvaro)

E.35.4.5. Utility Command Changes

- Allow `CREATE SCHEMA` to create triggers, indexes, and sequences (Neil)

- Add `ALSO` keyword to `CREATE RULE` (Fabien Coelho)

  This allows `ALSO` to be added to rule creation to contrast it with `INSTEAD` rules.

- Add `NOWAIT` option to `LOCK` (Tatsuo)

  This allows the `LOCK` command to fail if it would have to wait for the requested lock.

- Allow `COPY` to read and write comma-separated-value (CSV) files (Andrew, Bruce)

- Generate error if the `COPY` delimiter and NULL string conflict (Bruce)

- `GRANT`/`REVOKE` behavior follows the SQL spec more closely

- Avoid locking conflict between `CREATE INDEX` and `CHECKPOINT` (Tom)

  In 7.3 and 7.4, a long-running B-tree index build could block concurrent `CHECKPOINT`s from completing, thereby causing WAL bloat because the WAL log could not be recycled.

- Database-wide `ANALYZE` does not hold locks across tables (Tom)

  This reduces the potential for deadlocks against other backends that want exclusive locks on tables. To get the benefit of this change, do not execute database-wide `ANALYZE` inside a transaction block (`BEGIN` block); it must be able to commit and start a new transaction for each table.

- `REINDEX` does not exclusively lock the index's parent table anymore

  The index itself is still exclusively locked, but readers of the table can continue if they are not using the particular index being rebuilt.

- Erase MD5 user passwords when a user is renamed (Bruce)

  PostgreSQL uses the user name as salt when encrypting passwords via MD5. When a user's name is changed, the salt will no longer match the stored MD5 password, so the stored password becomes useless. In this release a notice is generated and the password is cleared. A new password must then be assigned if the user is to be able to log in with a password.

- New pg_ctl `kill` option for Windows (Andrew)

  Windows does not have a `kill` command to send signals to backends so this capability was added to pg_ctl.

- Information schema improvements

- Add `--pwfile` option to initdb so the initial password can be set by GUI tools (Magnus)

- Detect locale/encoding mismatch in initdb (Peter)

- Add `register` command to pg_ctl to register Windows operating system service (Dave Page)

E.35.4.6. Data Type and Function Changes

- More complete support for composite types (row types) (Tom)

  Composite values can be used in many places where only scalar values worked before.

- Reject nonrectangular array values as erroneous (Joe)

  Formerly, `array_in` would silently build a surprising result.

- Overflow in integer arithmetic operations is now detected (Tom)

- The arithmetic operators associated with the single-byte `"char"` data type have been removed.

  Formerly, the parser would select these operators in many situations where an "unable to select an operator" error would be more appropriate, such as `null * null`. If you actually want to do arithmetic on a `"char"` column, you can cast it to integer explicitly.

- Syntax checking of array input values considerably tightened up (Joe)

  Junk that was previously allowed in odd places with odd results now causes an `ERROR`, for example, non-whitespace after the closing right brace.

- Empty-string array element values must now be written as `""`, rather than writing nothing (Joe)

  Formerly, both ways of writing an empty-string element value were allowed, but now a quoted empty string is required. The case where nothing at all appears will probably be considered to be a NULL element value in some future release.

- Array element trailing whitespace is now ignored (Joe)

  Formerly leading whitespace was ignored, but trailing whitespace between an element value and the delimiter or right brace was significant. Now trailing whitespace is also ignored.

- Emit array values with explicit array bounds when lower bound is not one (Joe)
- Accept `YYYY-monthname-DD` as a date string (Tom)
- Make `netmask` and `hostmask` functions return maximum-length mask length (Tom)
- Change factorial function to return `numeric` (Gavin)

  Returning `numeric` allows the factorial function to work for a wider range of input values.

- `to_char`/`to_date()` date conversion improvements (Kurt Roeckx, Fabien Coelho)
- Make `length()` disregard trailing spaces in `CHAR(n)` (Gavin)

  This change was made to improve consistency: trailing spaces are semantically insignificant in `CHAR(n)` data, so they should not be counted by `length()`.

- Warn about empty string being passed to `OID`/`float4`/`float8` data types (Neil)

  8.1 will throw an error instead.

- Allow leading or trailing whitespace in `int2`/`int4`/`int8`/`float4`/`float8` input routines (Neil)
- Better support for IEEE `Infinity` and `NaN` values in `float4`/`float8` (Neil)

  These should now work on all platforms that support IEEE-compliant floating point arithmetic.

- Add `week` option to `date_trunc()` (Robert Creager)
- Fix `to_char` for `1 BC` (previously it returned `1 AD`) (Bruce)
- Fix `date_part(year)` for BC dates (previously it returned one less than the correct year) (Bruce)
- Fix `date_part()` to return the proper millennium and century (Fabien Coelho)

  In previous versions, the century and millennium results had a wrong number and started in the wrong year, as compared to standard reckoning of such things.

- Add `ceiling()` as an alias for `ceil()`, and `power()` as an alias for `pow()` for standards compliance (Neil)
- Change `ln()`, `log()`, `power()`, and `sqrt()` to emit the correct `SQLSTATE` error codes for certain error conditions, as specified by SQL:2003 (Neil)
- Add `width_bucket()` function as defined by SQL:2003 (Neil)
- Add `generate_series()` functions to simplify working with numeric sets (Joe)
- Fix `upper/lower/initcap()` functions to work with multibyte encodings (Tom)
- Add boolean and bitwise integer `AND`/`OR` aggregates (Fabien Coelho)
- New session information functions to return network addresses for client and server (Sean Chittenden)
- Add function to determine the area of a closed path (Sean Chittenden)
- Add function to send cancel request to other backends (Magnus)
- Add `interval` plus `datetime` operators (Tom)

The reverse ordering, `datetime` plus `interval`, was already supported, but both are required by the SQL standard.

- Casting an integer to `BIT(N)` selects the rightmost N bits of the integer (Tom)

  In prior releases, the leftmost N bits were selected, but this was deemed unhelpful, not to mention inconsistent with casting from bit to int.

- Require `CIDR` values to have all nonmasked bits be zero (Kevin Brintnall)

E.35.4.7. Server-Side Language Changes

- In `READ COMMITTED` serialization mode, volatile functions now see the results of concurrent transactions committed up to the beginning of each statement within the function, rather than up to the beginning of the interactive command that called the function.
- Functions declared `STABLE` or `IMMUTABLE` always use the snapshot of the calling query, and therefore do not see the effects of actions taken after the calling query starts, whether in their own transaction or other transactions. Such a function must be read-only, too, meaning that it cannot use any SQL commands other than `SELECT`. There is a considerable performance gain from declaring a function `STABLE` or `IMMUTABLE` rather than `VOLATILE`.
- Nondeferred `AFTER` triggers are now fired immediately after completion of the triggering query, rather than upon finishing the current interactive command. This makes a difference when the triggering query occurred within a function: the trigger is invoked before the function proceeds to its next operation. For example, if a function inserts a new row into a table, any nondeferred foreign key checks occur before proceeding with the function.
- Allow function parameters to be declared with names (Dennis Björklund)

  This allows better documentation of functions. Whether the names actually do anything depends on the specific function language being used.

- Allow PL/pgSQL parameter names to be referenced in the function (Dennis Björklund)

  This basically creates an automatic alias for each named parameter.

- Do minimal syntax checking of PL/pgSQL functions at creation time (Tom)

  This allows us to catch simple syntax errors sooner.

- More support for composite types (row and record variables) in PL/pgSQL

  For example, it now works to pass a rowtype variable to another function as a single variable.

- Default values for PL/pgSQL variables can now reference previously declared variables
- Improve parsing of PL/pgSQL FOR loops (Tom)

  Parsing is now driven by presence of `".."` rather than data type of `FOR` variable. This makes no difference for correct functions, but should result in more understandable error messages when a mistake is made.

- Major overhaul of PL/Perl server-side language (Command Prompt, Andrew Dunstan)

- In PL/Tcl, SPI commands are now run in subtransactions. If an error occurs, the subtransaction is cleaned up and the error is reported as an ordinary Tcl error, which can be trapped with `catch`. Formerly, it was not possible to catch such errors.
- Accept `ELSEIF` in PL/pgSQL (Neil)

  Previously PL/pgSQL only allowed `ELSIF`, but many people are accustomed to spelling this keyword `ELSEIF`.

E.35.4.8. psql Changes

- Improve psql information display about database objects (Christopher)
- Allow psql to display group membership in `\du` and `\dg` (Markus Bertheau)
- Prevent psql `\dn` from showing temporary schemas (Bruce)
- Allow psql to handle tilde user expansion for file names (Zach Irmen)
- Allow psql to display fancy prompts, including color, via readline (Reece Hart, Chet Ramey)
- Make psql `\copy` match `COPY` command syntax fully (Tom)
- Show the location of syntax errors (Fabien Coelho, Tom)
- Add `CLUSTER` information to psql `\d` display (Bruce)
- Change psql `\copy stdin/stdout` to read from command input/output (Bruce)
- Add `pstdin`/`pstdout` to read from psql's `stdin`/`stdout` (Mark Feit)
- Add global psql configuration file, `psqlrc.sample` (Bruce)

  This allows a central file where global psql startup commands can be stored.
- Have psql `\d+` indicate if the table has an `OID` column (Neil)
- On Windows, use binary mode in psql when reading files so control-Z is not seen as end-of-file
- Have `\dn+` show permissions and description for schemas (Dennis Björklund)
- Improve tab completion support (Stefan Kaltenbrunn, Greg Sabino Mullane)
- Allow boolean settings to be set using upper or lower case (Michael Paesold)

E.35.4.9. pg_dump Changes

- Use dependency information to improve the reliability of pg_dump (Tom)

  This should solve the longstanding problems with related objects sometimes being dumped in the wrong order.
- Have pg_dump output objects in alphabetical order if possible (Tom)

  This should make it easier to identify changes between dump files.
- Allow pg_restore to ignore some SQL errors (Fabien Coelho)

This makes pg_restore's behavior similar to the results of feeding a pg_dump output script to psql. In most cases, ignoring errors and plowing ahead is the most useful thing to do. Also added was a pg_restore option to give the old behavior of exiting on an error.

- pg_restore `-l` display now includes objects' schema names
- New begin/end markers in pg_dump text output (Bruce)
- Add start/stop times for pg_dump/pg_dumpall in verbose mode (Bruce)
- Allow most pg_dump options in pg_dumpall (Christopher)
- Have pg_dump use `ALTER OWNER` rather than `SET SESSION AUTHORIZATION` by default (Christopher)

E.35.4.10. libpq Changes

- Make libpq's `SIGPIPE` handling thread-safe (Bruce)
- Add `PQmbdsplen()` which returns the display length of a character (Tatsuo)
- Add thread locking to SSL and Kerberos connections (Manfred Spraul)
- Allow `PQoidValue()`, `PQcmdTuples()`, and `PQoidStatus()` to work on `EXECUTE` commands (Neil)
- Add `PQserverVersion()` to provide more convenient access to the server version number (Greg Sabino Mullane)
- Add `PQprepare/PQsendPrepared()` functions to support preparing statements without necessarily specifying the data types of their parameters (Abhijit Menon-Sen)
- Many ECPG improvements, including `SET DESCRIPTOR` (Michael)

E.35.4.11. Source Code Changes

- Allow the database server to run natively on Windows (Claudio, Magnus, Andrew)
- Shell script commands converted to C versions for Windows support (Andrew)
- Create an extension makefile framework (Fabien Coelho, Peter)

  This simplifies the task of building extensions outside the original source tree.
- Support relocatable installations (Bruce)

  Directory paths for installed files (such as the `/share` directory) are now computed relative to the actual location of the executables, so that an installation tree can be moved to another place without reconfiguring and rebuilding.
- Use `--with-docdir` to choose installation location of documentation; also allow `--infodir` (Peter)
- Add `--without-docdir` to prevent installation of documentation (Peter)
- Upgrade to DocBook V4.2 SGML (Peter)
- New `PostgreSQL` CVS tag (Marc)

This was done to make it easier for organizations to manage their own copies of the PostgreSQL CVS repository. File version stamps from the master repository will not get munged by checking into or out of a copied repository.

- Clarify locking code (Manfred Koizar)
- Buffer manager cleanup (Neil)
- Decouple platform tests from CPU spinlock code (Bruce, Tom)
- Add inlined test-and-set code on PA-RISC for gcc (ViSolve, Tom)
- Improve i386 spinlock code (Manfred Spraul)
- Clean up spinlock assembly code to avoid warnings from newer gcc releases (Tom)
- Remove JDBC from source tree; now a separate project
- Remove the libpgtcl client interface; now a separate project
- More accurately estimate memory and file descriptor usage (Tom)
- Improvements to the Mac OS X startup scripts (Ray A.)
- New `fsync()` test program (Bruce)
- Major documentation improvements (Neil, Peter)
- Remove pg_encoding; not needed anymore
- Remove pg_id; not needed anymore
- Remove initlocation; not needed anymore
- Auto-detect thread flags (no more manual testing) (Bruce)
- Use Olson's public domain timezone library (Magnus)
- With threading enabled, use thread flags on Unixware for backend executables too (Bruce)

  Unixware cannot mix threaded and nonthreaded object files in the same executable, so everything must be compiled as threaded.
- psql now uses a flex-generated lexical analyzer to process command strings
- Reimplement the linked list data structure used throughout the backend (Neil)

  This improves performance by allowing list append and length operations to be more efficient.
- Allow dynamically loaded modules to create their own server configuration parameters (Thomas Hallgren)
- New Brazilian version of FAQ (Euler Taveira de Oliveira)
- Add French FAQ (Guillaume Lelarge)
- New pgevent for Windows logging
- Make libpq and ECPG build as proper shared libraries on OS X (Tom)

E.35.4.12. Contrib Changes

- Overhaul of `contrib/dblink` (Joe)

- contrib/dbmirror improvements (Steven Singer)
- New contrib/xml2 (John Gray, Torchbox)
- Updated contrib/mysql
- New version of contrib/btree_gist (Teodor)
- New contrib/trgm, trigram matching for PostgreSQL (Teodor)
- Many contrib/tsearch2 improvements (Teodor)
- Add double metaphone to contrib/fuzzystrmatch (Andrew)
- Allow contrib/pg_autovacuum to run as a Windows service (Dave Page)
- Add functions to contrib/dbsize (Andreas Pflug)
- Removed contrib/pg_logger: obsoleted by integrated logging subprocess
- Removed contrib/rserv: obsoleted by various separate projects

E.36. Release 7.4.19

Release date: 2008-01-07

This release contains a variety of fixes from 7.4.18, including fixes for significant security issues.

E.36.1. Migration to Version 7.4.19

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see the release notes for 7.4.11.

E.36.2. Changes

- Prevent functions in indexes from executing with the privileges of the user running VACUUM, ANALYZE, etc (Tom)

  Functions used in index expressions and partial-index predicates are evaluated whenever a new table entry is made. It has long been understood that this poses a risk of trojan-horse code execution if one modifies a table owned by an untrustworthy user. (Note that triggers, defaults, check constraints, etc. pose the same type of risk.) But functions in indexes pose extra danger because they will be executed by routine maintenance operations such as VACUUM FULL, which are commonly performed automatically under a superuser account. For example, a nefarious user can execute code with superuser privileges by setting up a trojan-horse index definition and waiting for the next routine vacuum. The fix arranges for standard maintenance operations (including VACUUM, ANALYZE, REINDEX, and CLUSTER) to execute as the table owner rather than the calling user, using the same privilege-switching mechanism already

used for SECURITY DEFINER functions. To prevent bypassing this security measure, execution of SET SESSION AUTHORIZATION and SET ROLE is now forbidden within a SECURITY DEFINER context. (CVE-2007-6600)

- Repair assorted bugs in the regular-expression package (Tom, Will Drewry)

  Suitably crafted regular-expression patterns could cause crashes, infinite or near-infinite looping, and/or massive memory consumption, all of which pose denial-of-service hazards for applications that accept regex search patterns from untrustworthy sources. (CVE-2007-4769, CVE-2007-4772, CVE-2007-6067)

- Require non-superusers who use /contrib/dblink to use only password authentication, as a security measure (Joe)

  The fix that appeared for this in 7.4.18 was incomplete, as it plugged the hole for only some dblink functions. (CVE-2007-6601, CVE-2007-3278)

- Fix planner failure in some cases of WHERE false AND var IN (SELECT ...) (Tom)
- Fix potential crash in translate() when using a multibyte database encoding (Tom)
- Fix PL/Python to not crash on long exception messages (Alvaro)
- ecpg parser fixes (Michael)
- Make contrib/tablefunc's crosstab() handle NULL rowid as a category in its own right, rather than crashing (Joe)
- Fix tsvector and tsquery output routines to escape backslashes correctly (Teodor, Bruce)
- Fix crash of to_tsvector() on huge input strings (Teodor)
- Require a specific version of Autoconf to be used when re-generating the configure script (Peter)

  This affects developers and packagers only. The change was made to prevent accidental use of untested combinations of Autoconf and PostgreSQL versions. You can remove the version check if you really want to use a different Autoconf version, but it's your responsibility whether the result works or not.

E.37. Release 7.4.18

Release date: 2007-09-17

This release contains fixes from 7.4.17.

E.37.1. Migration to Version 7.4.18

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see the release notes for 7.4.11.

E.37.2. Changes

- Prevent index corruption when a transaction inserts rows and then aborts close to the end of a concurrent `VACUUM` on the same table (Tom)
- Make `CREATE DOMAIN ... DEFAULT NULL` work properly (Tom)
- Fix excessive logging of SSL error messages (Tom)
- Fix crash when `log_min_error_statement` logging runs out of memory (Tom)
- Prevent `CLUSTER` from failing due to attempting to process temporary tables of other sessions (Alvaro)
- Require non-superusers who use `/contrib/dblink` to use only password authentication, as a security measure (Joe)

E.38. Release 7.4.17

Release date: 2007-04-23

This release contains fixes from 7.4.16, including a security fix.

E.38.1. Migration to Version 7.4.17

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see the release notes for 7.4.11.

E.38.2. Changes

- Support explicit placement of the temporary-table schema within `search_path`, and disable searching it for functions and operators (Tom)

  This is needed to allow a security-definer function to set a truly secure value of `search_path`. Without it, an unprivileged SQL user can use temporary objects to execute code with the privileges of the security-definer function (CVE-2007-2138). See `CREATE FUNCTION` for more information.
- `/contrib/tsearch2` crash fixes (Teodor)
- Fix potential-data-corruption bug in how `VACUUM FULL` handles `UPDATE` chains (Tom, Pavan Deolasee)
- Fix PANIC during enlargement of a hash index (bug introduced in 7.4.15) (Tom)

E.39. Release 7.4.16

Release date: 2007-02-05

This release contains a variety of fixes from 7.4.15, including a security fix.

E.39.1. Migration to Version 7.4.16

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see the release notes for 7.4.11.

E.39.2. Changes

- Remove security vulnerability that allowed connected users to read backend memory (Tom)

  The vulnerability involves suppressing the normal check that a SQL function returns the data type it's declared to, or changing the data type of a table column used in a SQL function (CVE-2007-0555). This error can easily be exploited to cause a backend crash, and in principle might be used to read database content that the user should not be able to access.
- Fix rare bug wherein btree index page splits could fail due to choosing an infeasible split point (Heikki Linnakangas)
- Fix for rare Assert() crash triggered by `UNION` (Tom)
- Tighten security of multi-byte character processing for UTF8 sequences over three bytes long (Tom)

E.40. Release 7.4.15

Release date: 2007-01-08

This release contains a variety of fixes from 7.4.14.

E.40.1. Migration to Version 7.4.15

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see the release notes for 7.4.11.

E.40.2. Changes

- Improve handling of `getaddrinfo()` on AIX (Tom)

  This fixes a problem with starting the statistics collector, among other things.
- Fix "failed to re-find parent key" errors in `VACUUM` (Tom)
- Fix bugs affecting multi-gigabyte hash indexes (Tom)
- Fix error when constructing an `ARRAY[]` made up of multiple empty elements (Tom)
- `to_number()` and `to_char(numeric)` are now `STABLE`, not `IMMUTABLE`, for new initdb installs (Tom)

  This is because `lc_numeric` can potentially change the output of these functions.
- Improve index usage of regular expressions that use parentheses (Tom)

  This improves psql `\d` performance also.

E.41. Release 7.4.14

Release date: 2006-10-16

This release contains a variety of fixes from 7.4.13.

E.41.1. Migration to Version 7.4.14

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see the release notes for 7.4.11.

E.41.2. Changes

- Fix core dump when an untyped literal is taken as ANYARRAY
- Fix `string_to_array()` to handle overlapping matches for the separator string

  For example, `string_to_array('123xx456xxx789', 'xx')`.
- Fix corner cases in pattern matching for psql's `\d` commands
- Fix index-corrupting bugs in /contrib/ltree (Teodor)
- Fix backslash escaping in /contrib/dbmirror
- Adjust regression tests for recent changes in US DST laws

E.42. Release 7.4.13

Release date: 2006-05-23

This release contains a variety of fixes from 7.4.12, including patches for extremely serious security issues.

E.42.1. Migration to Version 7.4.13

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see the release notes for 7.4.11.

Full security against the SQL-injection attacks described in CVE-2006-2313 and CVE-2006-2314 might require changes in application code. If you have applications that embed untrustworthy strings into SQL commands, you should examine them as soon as possible to ensure that they are using recommended escaping techniques. In most cases, applications should be using subroutines provided by libraries or drivers (such as libpq's `PQescapeStringConn()`) to perform string escaping, rather than relying on *ad hoc* code to do it.

E.42.2. Changes

- Change the server to reject invalidly-encoded multibyte characters in all cases (Tatsuo, Tom)

  While PostgreSQL has been moving in this direction for some time, the checks are now applied uniformly to all encodings and all textual input, and are now always errors not merely warnings. This change defends against SQL-injection attacks of the type described in CVE-2006-2313.

- Reject unsafe uses of `\'` in string literals

  As a server-side defense against SQL-injection attacks of the type described in CVE-2006-2314, the server now only accepts `''` and not `\'` as a representation of ASCII single quote in SQL string literals. By default, `\'` is rejected only when `client_encoding` is set to a client-only encoding (SJIS, BIG5, GBK, GB18030, or UHC), which is the scenario in which SQL injection is possible. A new configuration parameter `backslash_quote` is available to adjust this behavior when needed. Note that full security against CVE-2006-2314 might require client-side changes; the purpose of `backslash_quote` is in part to make it obvious that insecure clients are insecure.

- Modify libpq's string-escaping routines to be aware of encoding considerations and `standard_conforming_strings`

  This fixes libpq-using applications for the security issues described in CVE-2006-2313 and CVE-2006-2314, and also future-proofs them against the planned changeover to SQL-standard string literal syntax. Applications that use multiple PostgreSQL connections concurrently should migrate to `PQescapeStringConn()` and `PQescapeByteaConn()` to ensure that escaping is done correctly for the settings in use in each database connection. Applications that do string escaping "by hand" should be modified to rely on library routines instead.

- Fix some incorrect encoding conversion functions

`win1251_to_iso`, `alt_to_iso`, `euc_tw_to_big5`, `euc_tw_to_mic`, `mic_to_euc_tw` were all broken to varying extents.

- Clean up stray remaining uses of `\'` in strings (Bruce, Jan)
- Fix bug that sometimes caused OR'd index scans to miss rows they should have returned
- Fix WAL replay for case where a btree index has been truncated
- Fix `SIMILAR TO` for patterns involving `|` (Tom)
- Fix server to use custom DH SSL parameters correctly (Michael Fuhr)
- Fix for Bonjour on Intel Macs (Ashley Clark)
- Fix various minor memory leaks

E.43. Release 7.4.12

Release date: 2006-02-14

This release contains a variety of fixes from 7.4.11.

E.43.1. Migration to Version 7.4.12

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see the release notes for 7.4.11.

E.43.2. Changes

- Fix potential crash in `SET SESSION AUTHORIZATION` (CVE-2006-0553)

  An unprivileged user could crash the server process, resulting in momentary denial of service to other users, if the server has been compiled with Asserts enabled (which is not the default). Thanks to Akio Ishida for reporting this problem.

- Fix bug with row visibility logic in self-inserted rows (Tom)

  Under rare circumstances a row inserted by the current command could be seen as already valid, when it should not be. Repairs bug created in 7.4.9 and 7.3.11 releases.

- Fix race condition that could lead to "file already exists" errors during pg_clog file creation (Tom)
- Properly check `DOMAIN` constraints for `UNKNOWN` parameters in prepared statements (Neil)
- Fix to allow restoring dumps that have cross-schema references to custom operators (Tom)
- Portability fix for testing presence of `finite` and `isinf` during configure (Tom)

E.44. Release 7.4.11

Release date: 2006-01-09

This release contains a variety of fixes from 7.4.10.

E.44.1. Migration to Version 7.4.11

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.8, see the release notes for 7.4.8. Also, you might need to `REINDEX` indexes on textual columns after updating, if you are affected by the locale or plperl issues described below.

E.44.2. Changes

- Fix for protocol-level Describe messages issued outside a transaction or in a failed transaction (Tom)
- Fix character string comparison for locales that consider different character combinations as equal, such as Hungarian (Tom)

  This might require `REINDEX` to fix existing indexes on textual columns.
- Set locale environment variables during postmaster startup to ensure that plperl won't change the locale later

  This fixes a problem that occurred if the postmaster was started with environment variables specifying a different locale than what initdb had been told. Under these conditions, any use of plperl was likely to lead to corrupt indexes. You might need `REINDEX` to fix existing indexes on textual columns if this has happened to you.
- Fix longstanding bug in strpos() and regular expression handling in certain rarely used Asian multi-byte character sets (Tatsuo)
- Fix bug in `/contrib/pgcrypto` gen_salt, which caused it not to use all available salt space for MD5 and XDES algorithms (Marko Kreen, Solar Designer)

  Salts for Blowfish and standard DES are unaffected.
- Fix `/contrib/dblink` to throw an error, rather than crashing, when the number of columns specified is different from what's actually returned by the query (Joe)

E.45. Release 7.4.10

Release date: 2005-12-12

This release contains a variety of fixes from 7.4.9.

E.45.1. Migration to Version 7.4.10

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.8, see the release notes for 7.4.8.

E.45.2. Changes

- Fix race condition in transaction log management

  There was a narrow window in which an I/O operation could be initiated for the wrong page, leading to an Assert failure or data corruption.
- Prevent failure if client sends Bind protocol message when current transaction is already aborted
- `/contrib/ltree` fixes (Teodor)
- AIX and HPUX compile fixes (Tom)
- Fix longstanding planning error for outer joins

  This bug sometimes caused a bogus error "RIGHT JOIN is only supported with merge-joinable join conditions".
- Prevent core dump in pg_autovacuum when a table has been dropped

E.46. Release 7.4.9

Release date: 2005-10-04

This release contains a variety of fixes from 7.4.8.

E.46.1. Migration to Version 7.4.9

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.8, see the release notes for 7.4.8.

E.46.2. Changes

- Fix error that allowed `VACUUM` to remove `ctid` chains too soon, and add more checking in code that follows `ctid` links

This fixes a long-standing problem that could cause crashes in very rare circumstances.

- Fix `CHAR()` to properly pad spaces to the specified length when using a multiple-byte character set (Yoshiyuki Asaba)

  In prior releases, the padding of `CHAR()` was incorrect because it only padded to the specified number of bytes without considering how many characters were stored.

- Fix the sense of the test for read-only transaction in `COPY`

  The code formerly prohibited `COPY TO`, where it should prohibit `COPY FROM`.

- Fix planning problem with outer-join ON clauses that reference only the inner-side relation
- Further fixes for `x FULL JOIN y ON true` corner cases
- Make `array_in` and `array_recv` more paranoid about validating their OID parameter
- Fix missing rows in queries like `UPDATE a=... WHERE a...` with GiST index on column `a`
- Improve robustness of datetime parsing
- Improve checking for partially-written WAL pages
- Improve robustness of signal handling when SSL is enabled
- Don't try to open more than `max_files_per_process` files during postmaster startup
- Various memory leakage fixes
- Various portability improvements
- Fix PL/PgSQL to handle `var := var` correctly when the variable is of pass-by-reference type
- Update `contrib/tsearch2` to use current Snowball code

E.47. Release 7.4.8

Release date: 2005-05-09

This release contains a variety of fixes from 7.4.7, including several security-related issues.

E.47.1. Migration to Version 7.4.8

A dump/restore is not required for those running 7.4.X. However, it is one possible way of handling two significant security problems that have been found in the initial contents of 7.4.X system catalogs. A dump/initdb/reload sequence using 7.4.8's initdb will automatically correct these problems.

The larger security problem is that the built-in character set encoding conversion functions can be invoked from SQL commands by unprivileged users, but the functions were not designed for such use and are not secure against malicious choices of arguments. The fix involves changing the declared parameter list of these functions so that they can no longer be invoked from SQL commands. (This does not affect their normal use by the encoding conversion machinery.)

The lesser problem is that the `contrib/tsearch2` module creates several functions that are misdeclared to return `internal` when they do not accept `internal` arguments. This breaks type safety for all functions using `internal` arguments.

It is strongly recommended that all installations repair these errors, either by initdb or by following the manual repair procedures given below. The errors at least allow unprivileged database users to crash their server process, and might allow unprivileged users to gain the privileges of a database superuser.

If you wish not to do an initdb, perform the following procedures instead. As the database superuser, do:

```
BEGIN;
UPDATE pg_proc SET proargtypes[3] = 'internal'::regtype
WHERE pronamespace = 11 AND pronargs = 5
     AND proargtypes[2] = 'cstring'::regtype;
-- The command should report having updated 90 rows;
-- if not, rollback and investigate instead of committing!
COMMIT;
```

Next, if you have installed `contrib/tsearch2`, do:

```
BEGIN;
UPDATE pg_proc SET proargtypes[0] = 'internal'::regtype
WHERE oid IN (
   'dex_init(text)'::regprocedure,
   'snb_en_init(text)'::regprocedure,
   'snb_ru_init(text)'::regprocedure,
   'spell_init(text)'::regprocedure,
   'syn_init(text)'::regprocedure
);
-- The command should report having updated 5 rows;
-- if not, rollback and investigate instead of committing!
COMMIT;
```

If this command fails with a message like "function "dex_init(text)" does not exist", then either `tsearch2` is not installed in this database, or you already did the update.

The above procedures must be carried out in *each* database of an installation, including `template1`, and ideally including `template0` as well. If you do not fix the template databases then any subsequently created databases will contain the same errors. `template1` can be fixed in the same way as any other database, but fixing `template0` requires additional steps. First, from any database issue:

```
UPDATE pg_database SET datallowconn = true WHERE datname = 'template0';
```

Next connect to `template0` and perform the above repair procedures. Finally, do:

```
-- re-freeze template0:
VACUUM FREEZE;
-- and protect it against future alterations:
UPDATE pg_database SET datallowconn = false WHERE datname = 'template0';
```

E.47.2. Changes

- Change encoding function signature to prevent misuse
- Change `contrib/tsearch2` to avoid unsafe use of `INTERNAL` function results
- Repair ancient race condition that allowed a transaction to be seen as committed for some purposes (eg SELECT FOR UPDATE) slightly sooner than for other purposes

  This is an extremely serious bug since it could lead to apparent data inconsistencies being briefly visible to applications.
- Repair race condition between relation extension and VACUUM

  This could theoretically have caused loss of a page's worth of freshly-inserted data, although the scenario seems of very low probability. There are no known cases of it having caused more than an Assert failure.
- Fix comparisons of `TIME WITH TIME ZONE` values

  The comparison code was wrong in the case where the `--enable-integer-datetimes` configuration switch had been used. NOTE: if you have an index on a `TIME WITH TIME ZONE` column, it will need to be `REINDEX`ed after installing this update, because the fix corrects the sort order of column values.
- Fix `EXTRACT(EPOCH)` for `TIME WITH TIME ZONE` values
- Fix mis-display of negative fractional seconds in `INTERVAL` values

  This error only occurred when the `--enable-integer-datetimes` configuration switch had been used.
- Ensure operations done during backend shutdown are counted by statistics collector

  This is expected to resolve reports of pg_autovacuum not vacuuming the system catalogs often enough — it was not being told about catalog deletions caused by temporary table removal during backend exit.
- Additional buffer overrun checks in plpgsql (Neil)
- Fix pg_dump to dump trigger names containing `%` correctly (Neil)
- Fix `contrib/pgcrypto` for newer OpenSSL builds (Marko Kreen)
- Still more 64-bit fixes for `contrib/intagg`
- Prevent incorrect optimization of functions returning `RECORD`
- Prevent `to_char(interval)` from dumping core for month-related formats
- Prevent crash on `COALESCE(NULL,NULL)`
- Fix `array_map` to call PL functions correctly
- Fix permission checking in `ALTER DATABASE RENAME`
- Fix `ALTER LANGUAGE RENAME`
- Make `RemoveFromWaitQueue` clean up after itself

  This fixes a lock management error that would only be visible if a transaction was kicked out of a wait for a lock (typically by query cancel) and then the holder of the lock released it within a very narrow window.

- Fix problem with untyped parameter appearing in `INSERT ... SELECT`
- Fix `CLUSTER` failure after `ALTER TABLE SET WITHOUT OIDS`

E.48. Release 7.4.7

Release date: 2005-01-31

This release contains a variety of fixes from 7.4.6, including several security-related issues.

E.48.1. Migration to Version 7.4.7

A dump/restore is not required for those running 7.4.X.

E.48.2. Changes

- Disallow `LOAD` to non-superusers

  On platforms that will automatically execute initialization functions of a shared library (this includes at least Windows and ELF-based Unixen), `LOAD` can be used to make the server execute arbitrary code. Thanks to NGS Software for reporting this.
- Check that creator of an aggregate function has the right to execute the specified transition functions

  This oversight made it possible to bypass denial of EXECUTE permission on a function.
- Fix security and 64-bit issues in contrib/intagg
- Add needed STRICT marking to some contrib functions (Kris Jurka)
- Avoid buffer overrun when plpgsql cursor declaration has too many parameters (Neil)
- Fix planning error for FULL and RIGHT outer joins

  The result of the join was mistakenly supposed to be sorted the same as the left input. This could not only deliver mis-sorted output to the user, but in case of nested merge joins could give outright wrong answers.
- Fix plperl for quote marks in tuple fields
- Fix display of negative intervals in SQL and GERMAN datestyles
- Make age(timestamptz) do calculation in local timezone not GMT

E.49. Release 7.4.6

Release date: 2004-10-22

This release contains a variety of fixes from 7.4.5.

E.49.1. Migration to Version 7.4.6

A dump/restore is not required for those running 7.4.X.

E.49.2. Changes

- Repair possible failure to update hint bits on disk

  Under rare circumstances this oversight could lead to "could not access transaction status" failures, which qualifies it as a potential-data-loss bug.

- Ensure that hashed outer join does not miss tuples

  Very large left joins using a hash join plan could fail to output unmatched left-side rows given just the right data distribution.

- Disallow running pg_ctl as root

  This is to guard against any possible security issues.

- Avoid using temp files in `/tmp` in `make_oidjoins_check`

  This has been reported as a security issue, though it's hardly worthy of concern since there is no reason for non-developers to use this script anyway.

- Prevent forced backend shutdown from re-emitting prior command result

  In rare cases, a client might think that its last command had succeeded when it really had been aborted by forced database shutdown.

- Repair bug in `pg_stat_get_backend_idset`

  This could lead to misbehavior in some of the system-statistics views.

- Fix small memory leak in postmaster

- Fix "expected both swapped tables to have TOAST tables" bug

  This could arise in cases such as CLUSTER after ALTER TABLE DROP COLUMN.

- Prevent `pg_ctl restart` from adding `-D` multiple times

- Fix problem with NULL values in GiST indexes

- `::` is no longer interpreted as a variable in an ECPG prepare statement

E.50. Release 7.4.5

Release date: 2004-08-18

This release contains one serious bug fix over 7.4.4.

E.50.1. Migration to Version 7.4.5

A dump/restore is not required for those running 7.4.X.

E.50.2. Changes

- Repair possible crash during concurrent B-tree index insertions

  This patch fixes a rare case in which concurrent insertions into a B-tree index could result in a server panic. No permanent damage would result, but it's still worth a re-release. The bug does not exist in pre-7.4 releases.

E.51. Release 7.4.4

Release date: 2004-08-16

This release contains a variety of fixes from 7.4.3.

E.51.1. Migration to Version 7.4.4

A dump/restore is not required for those running 7.4.X.

E.51.2. Changes

- Prevent possible loss of committed transactions during crash

  Due to insufficient interlocking between transaction commit and checkpointing, it was possible for transactions committed just before the most recent checkpoint to be lost, in whole or in part, following a database crash and restart. This is a serious bug that has existed since PostgreSQL 7.1.
- Check HAVING restriction before evaluating result list of an aggregate plan
- Avoid crash when session's current user ID is deleted

- Fix hashed crosstab for zero-rows case (Joe)
- Force cache update after renaming a column in a foreign key
- Pretty-print UNION queries correctly
- Make psql handle `\r\n` newlines properly in COPY IN
- pg_dump handled ACLs with grant options incorrectly
- Fix thread support for OS X and Solaris
- Updated JDBC driver (build 215) with various fixes
- ECPG fixes
- Translation updates (various contributors)

E.52. Release 7.4.3

Release date: 2004-06-14

This release contains a variety of fixes from 7.4.2.

E.52.1. Migration to Version 7.4.3

A dump/restore is not required for those running 7.4.X.

E.52.2. Changes

- Fix temporary memory leak when using non-hashed aggregates (Tom)
- ECPG fixes, including some for Informix compatibility (Michael)
- Fixes for compiling with thread-safety, particularly Solaris (Bruce)
- Fix error in COPY IN termination when using the old network protocol (ljb)
- Several important fixes in pg_autovacuum, including fixes for large tables, unsigned oids, stability, temp tables, and debug mode (Matthew T. O'Connor)
- Fix problem with reading tar-format dumps on NetBSD and BSD/OS (Bruce)
- Several JDBC fixes
- Fix ALTER SEQUENCE RESTART where last_value equals the restart value (Tom)
- Repair failure to recalculate nested sub-selects (Tom)
- Fix problems with non-constant expressions in LIMIT/OFFSET
- Support FULL JOIN with no join clause, such as X FULL JOIN Y ON TRUE (Tom)

- Fix another zero-column table bug (Tom)
- Improve handling of non-qualified identifiers in GROUP BY clauses in sub-selects (Tom)

  Select-list aliases within the sub-select will now take precedence over names from outer query levels.
- Do not generate "NATURAL CROSS JOIN" when decompiling rules (Tom)
- Add checks for invalid field length in binary COPY (Tom)

  This fixes a difficult-to-exploit security hole.
- Avoid locking conflict between `ANALYZE` and `LISTEN`/`NOTIFY`
- Numerous translation updates (various contributors)

E.53. Release 7.4.2

Release date: 2004-03-08

This release contains a variety of fixes from 7.4.1.

E.53.1. Migration to Version 7.4.2

A dump/restore is not required for those running 7.4.X. However, it might be advisable as the easiest method of incorporating fixes for two errors that have been found in the initial contents of 7.4.X system catalogs. A dump/initdb/reload sequence using 7.4.2's initdb will automatically correct these problems.

The more severe of the two errors is that data type `anyarray` has the wrong alignment label; this is a problem because the `pg_statistic` system catalog uses `anyarray` columns. The mislabeling can cause planner misestimations and even crashes when planning queries that involve `WHERE` clauses on double-aligned columns (such as `float8` and `timestamp`). It is strongly recommended that all installations repair this error, either by initdb or by following the manual repair procedure given below.

The lesser error is that the system view `pg_settings` ought to be marked as having public update access, to allow `UPDATE pg_settings` to be used as a substitute for `SET`. This can also be fixed either by initdb or manually, but it is not necessary to fix unless you want to use `UPDATE pg_settings`.

If you wish not to do an initdb, the following procedure will work for fixing `pg_statistic`. As the database superuser, do:

```
-- clear out old data in pg_statistic:
DELETE FROM pg_statistic;
VACUUM pg_statistic;
-- this should update 1 row:
UPDATE pg_type SET typalign = 'd' WHERE oid = 2277;
-- this should update 6 rows:
UPDATE pg_attribute SET attalign = 'd' WHERE atttypid = 2277;
--
-- At this point you MUST start a fresh backend to avoid a crash!
```

```
--
-- repopulate pg_statistic:
ANALYZE;
```

This can be done in a live database, but beware that all backends running in the altered database must be restarted before it is safe to repopulate `pg_statistic`.

To repair the `pg_settings` error, simply do:

```
GRANT SELECT, UPDATE ON pg_settings TO PUBLIC;
```

The above procedures must be carried out in *each* database of an installation, including `template1`, and ideally including `template0` as well. If you do not fix the template databases then any subsequently created databases will contain the same errors. `template1` can be fixed in the same way as any other database, but fixing `template0` requires additional steps. First, from any database issue:

```
UPDATE pg_database SET datallowconn = true WHERE datname = 'template0';
```

Next connect to `template0` and perform the above repair procedures. Finally, do:

```
-- re-freeze template0:
VACUUM FREEZE;
-- and protect it against future alterations:
UPDATE pg_database SET datallowconn = false WHERE datname = 'template0';
```

E.53.2. Changes

Release 7.4.2 incorporates all the fixes included in release 7.3.6, plus the following fixes:

- Fix `pg_statistics` alignment bug that could crash optimizer

  See above for details about this problem.
- Allow non-super users to update `pg_settings`
- Fix several optimizer bugs, most of which led to "variable not found in subplan target lists" errors
- Avoid out-of-memory failure during startup of large multiple index scan
- Fix multibyte problem that could lead to "out of memory" error during `COPY IN`
- Fix problems with `SELECT INTO` / `CREATE TABLE AS` from tables without OIDs
- Fix problems with `alter_table` regression test during parallel testing
- Fix problems with hitting open file limit, especially on OS X (Tom)
- Partial fix for Turkish-locale issues

  initdb will succeed now in Turkish locale, but there are still some inconveniences associated with the `i/I` problem.
- Make pg_dump set client encoding on restore

- Other minor pg_dump fixes
- Allow ecpg to again use C keywords as column names (Michael)
- Added ecpg `WHENEVER NOT_FOUND` to `SELECT/INSERT/UPDATE/DELETE` (Michael)
- Fix ecpg crash for queries calling set-returning functions (Michael)
- Various other ecpg fixes (Michael)
- Fixes for Borland compiler
- Thread build improvements (Bruce)
- Various other build fixes
- Various JDBC fixes

E.54. Release 7.4.1

Release date: 2003-12-22

This release contains a variety of fixes from 7.4.

E.54.1. Migration to Version 7.4.1

A dump/restore is *not* required for those running 7.4.

If you want to install the fixes in the information schema you need to reload it into the database. This is either accomplished by initializing a new cluster by running `initdb`, or by running the following sequence of SQL commands in each database (ideally including `template1`) as a superuser in psql, after installing the new release:

```
DROP SCHEMA information_schema CASCADE;
\i /usr/local/pgsql/share/information_schema.sql
```

Substitute your installation path in the second command.

E.54.2. Changes

- Fixed bug in `CREATE SCHEMA` parsing in ECPG (Michael)
- Fix compile error when `--enable-thread-safety` and `--with-perl` are used together (Peter)
- Fix for subqueries that used hash joins (Tom)

  Certain subqueries that used hash joins would crash because of improperly shared structures.
- Fix free space map compaction bug (Tom)

This fixes a bug where compaction of the free space map could lead to a database server shutdown.

- Fix for Borland compiler build of libpq (Bruce)
- Fix `netmask()` and `hostmask()` to return the maximum-length masklen (Tom)

  Fix these functions to return values consistent with pre-7.4 releases.
- Several `contrib/pg_autovacuum` fixes

  Fixes include improper variable initialization, missing vacuum after `TRUNCATE`, and duration computation overflow for long vacuums.
- Allow compile of `contrib/cube` under Cygwin (Jason Tishler)
- Fix Solaris use of password file when no passwords are defined (Tom)

  Fix crash on Solaris caused by use of any type of password authentication when no passwords were defined.
- JDBC fix for thread problems, other fixes
- Fix for `bytea` index lookups (Joe)
- Fix information schema for bit data types (Peter)
- Force zero_damaged_pages to be on during recovery from WAL
- Prevent some obscure cases of "variable not in subplan target lists"
- Make `PQescapeBytea` and `byteaout` consistent with each other (Joe)
- Escape `bytea` output for bytes $>$ 0x7e(Joe)

  If different client encodings are used for `bytea` output and input, it is possible for `bytea` values to be corrupted by the differing encodings. This fix escapes all bytes that might be affected.
- Added missing `SPI_finish()` calls to dblink's `get_tuple_of_interest()` (Joe)
- New Czech FAQ
- Fix information schema view `constraint_column_usage` for foreign keys (Peter)
- ECPG fixes (Michael)
- Fix bug with multiple `IN` subqueries and joins in the subqueries (Tom)
- Allow `COUNT('x')` to work (Tom)
- Install ECPG include files for Informix compatibility into separate directory (Peter)

  Some names of ECPG include files for Informix compatibility conflicted with operating system include files. By installing them in their own directory, name conflicts have been reduced.
- Fix SSL memory leak (Neil)

  This release fixes a bug in 7.4 where SSL didn't free all memory it allocated.
- Prevent `pg_service.conf` from using service name as default dbname (Bruce)
- Fix local ident authentication on FreeBSD (Tom)

E.55. Release 7.4

Release date: 2003-11-17

E.55.1. Overview

Major changes in this release:

`IN` / `NOT IN` subqueries are now much more efficient

In previous releases, `IN`/`NOT IN` subqueries were joined to the upper query by sequentially scanning the subquery looking for a match. The 7.4 code uses the same sophisticated techniques used by ordinary joins and so is much faster. An `IN` will now usually be as fast as or faster than an equivalent `EXISTS` subquery; this reverses the conventional wisdom that applied to previous releases.

Improved `GROUP BY` processing by using hash buckets

In previous releases, rows to be grouped had to be sorted first. The 7.4 code can do `GROUP BY` without sorting, by accumulating results into a hash table with one entry per group. It will still use the sort technique, however, if the hash table is estimated to be too large to fit in `sort_mem`.

New multikey hash join capability

In previous releases, hash joins could only occur on single keys. This release allows multicolumn hash joins.

Queries using the explicit `JOIN` syntax are now better optimized

Prior releases evaluated queries using the explicit `JOIN` syntax only in the order implied by the syntax. 7.4 allows full optimization of these queries, meaning the optimizer considers all possible join orderings and chooses the most efficient. Outer joins, however, must still follow the declared ordering.

Faster and more powerful regular expression code

The entire regular expression module has been replaced with a new version by Henry Spencer, originally written for Tcl. The code greatly improves performance and supports several flavors of regular expressions.

Function-inlining for simple SQL functions

Simple SQL functions can now be inlined by including their SQL in the main query. This improves performance by eliminating per-call overhead. That means simple SQL functions now behave like macros.

Full support for IPv6 connections and IPv6 address data types

Previous releases allowed only IPv4 connections, and the IP data types only supported IPv4 addresses. This release adds full IPv6 support in both of these areas.

Major improvements in SSL performance and reliability

Several people very familiar with the SSL API have overhauled our SSL code to improve SSL key negotiation and error recovery.

Make free space map efficiently reuse empty index pages, and other free space management improvements

In previous releases, B-tree index pages that were left empty because of deleted rows could only be reused by rows with index values similar to the rows originally indexed on that page. In 7.4, `VACUUM` records empty index pages and allows them to be reused for any future index rows.

SQL-standard information schema

The information schema provides a standardized and stable way to access information about the schema objects defined in a database.

Cursors conform more closely to the SQL standard

The commands `FETCH` and `MOVE` have been overhauled to conform more closely to the SQL standard.

Cursors can exist outside transactions

These cursors are also called holdable cursors.

New client-to-server protocol

The new protocol adds error codes, more status information, faster startup, better support for binary data transmission, parameter values separated from SQL commands, prepared statements available at the protocol level, and cleaner recovery from `COPY` failures. The older protocol is still supported by both server and clients.

libpq and ECPG applications are now fully thread-safe

While previous libpq releases already supported threads, this release improves thread safety by fixing some non-thread-safe code that was used during database connection startup. The `configure` option `--enable-thread-safety` must be used to enable this feature.

New version of full-text indexing

A new full-text indexing suite is available in `contrib/tsearch2`.

New autovacuum tool

The new autovacuum tool in `contrib/autovacuum` monitors the database statistics tables for `INSERT`/`UPDATE`/`DELETE` activity and automatically vacuums tables when needed.

Array handling has been improved and moved into the server core

Many array limitations have been removed, and arrays behave more like fully-supported data types.

E.55.2. Migration to Version 7.4

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release.

Observe the following incompatibilities:

- The server-side autocommit setting was removed and reimplemented in client applications and languages. Server-side autocommit was causing too many problems with languages and applications that wanted to control their own autocommit behavior, so autocommit was removed from the server and added to individual client APIs as appropriate.
- Error message wording has changed substantially in this release. Significant effort was invested to make the messages more consistent and user-oriented. If your applications try to detect different error

conditions by parsing the error message, you are strongly encouraged to use the new error code facility instead.

- Inner joins using the explicit `JOIN` syntax might behave differently because they are now better optimized.
- A number of server configuration parameters have been renamed for clarity, primarily those related to logging.
- `FETCH 0` or `MOVE 0` now does nothing. In prior releases, `FETCH 0` would fetch all remaining rows, and `MOVE 0` would move to the end of the cursor.
- `FETCH` and `MOVE` now return the actual number of rows fetched/moved, or zero if at the beginning/end of the cursor. Prior releases would return the row count passed to the command, not the number of rows actually fetched or moved.
- `COPY` now can process files that use carriage-return or carriage-return/line-feed end-of-line sequences. Literal carriage-returns and line-feeds are no longer accepted in data values; use `\r` and `\n` instead.
- Trailing spaces are now trimmed when converting from type `char(`*n*`)` to `varchar(`*n*`)` or `text`. This is what most people always expected to happen anyway.
- The data type `float(`*p*`)` now measures *p* in binary digits, not decimal digits. The new behavior follows the SQL standard.
- Ambiguous date values now must match the ordering specified by the `datestyle` setting. In prior releases, a date specification of `10/20/03` was interpreted as a date in October even if `datestyle` specified that the day should be first. 7.4 will throw an error if a date specification is invalid for the current setting of `datestyle`.
- The functions `oidrand`, `oidsrand`, and `userfntest` have been removed. These functions were determined to be no longer useful.
- String literals specifying time-varying date/time values, such as `'now'` or `'today'` will no longer work as expected in column default expressions; they now cause the time of the table creation to be the default, not the time of the insertion. Functions such as `now()`, `current_timestamp`, or `current_date` should be used instead.

  In previous releases, there was special code so that strings such as `'now'` were interpreted at `INSERT` time and not at table creation time, but this work around didn't cover all cases. Release 7.4 now requires that defaults be defined properly using functions such as `now()` or `current_timestamp`. These will work in all situations.
- The dollar sign (`$`) is no longer allowed in operator names. It can instead be a non-first character in identifiers. This was done to improve compatibility with other database systems, and to avoid syntax problems when parameter placeholders (`$`*n*) are written adjacent to operators.

E.55.3. Changes

Below you will find a detailed account of the changes between release 7.4 and the previous major release.

E.55.3.1. Server Operation Changes

- Allow IPv6 server connections (Nigel Kukard, Johan Jordaan, Bruce, Tom, Kurt Roeckx, Andrew Dunstan)
- Fix SSL to handle errors cleanly (Nathan Mueller)

  In prior releases, certain SSL API error reports were not handled correctly. This release fixes those problems.
- SSL protocol security and performance improvements (Sean Chittenden)

  SSL key renegotiation was happening too frequently, causing poor SSL performance. Also, initial key handling was improved.
- Print lock information when a deadlock is detected (Tom)

  This allows easier debugging of deadlock situations.
- Update `/tmp` socket modification times regularly to avoid their removal (Tom)

  This should help prevent `/tmp` directory cleaner administration scripts from removing server socket files.
- Enable PAM for Mac OS X (Aaron Hillegass)
- Make B-tree indexes fully WAL-safe (Tom)

  In prior releases, under certain rare cases, a server crash could cause B-tree indexes to become corrupt. This release removes those last few rare cases.
- Allow B-tree index compaction and empty page reuse (Tom)
- Fix inconsistent index lookups during split of first root page (Tom)

  In prior releases, when a single-page index split into two pages, there was a brief period when another database session could miss seeing an index entry. This release fixes that rare failure case.
- Improve free space map allocation logic (Tom)
- Preserve free space information between server restarts (Tom)

  In prior releases, the free space map was not saved when the postmaster was stopped, so newly started servers had no free space information. This release saves the free space map, and reloads it when the server is restarted.
- Add start time to `pg_stat_activity` (Neil)
- New code to detect corrupt disk pages; erase with `zero_damaged_pages` (Tom)
- New client/server protocol: faster, no username length limit, allow clean exit from `COPY` (Tom)
- Add transaction status, table ID, column ID to client/server protocol (Tom)
- Add binary I/O to client/server protocol (Tom)
- Remove autocommit server setting; move to client applications (Tom)
- New error message wording, error codes, and three levels of error detail (Tom, Joe, Peter)

E.55.3.2. Performance Improvements

- Add hashing for `GROUP BY` aggregates (Tom)
- Make nested-loop joins be smarter about multicolumn indexes (Tom)
- Allow multikey hash joins (Tom)
- Improve constant folding (Tom)
- Add ability to inline simple SQL functions (Tom)
- Reduce memory usage for queries using complex functions (Tom)

  In prior releases, functions returning allocated memory would not free it until the query completed. This release allows the freeing of function-allocated memory when the function call completes, reducing the total memory used by functions.
- Improve GEQO optimizer performance (Tom)

  This release fixes several inefficiencies in the way the GEQO optimizer manages potential query paths.
- Allow `IN`/`NOT IN` to be handled via hash tables (Tom)
- Improve `NOT IN (`*`subquery`*`)` performance (Tom)
- Allow most `IN` subqueries to be processed as joins (Tom)
- Pattern matching operations can use indexes regardless of locale (Peter)

  There is no way for non-ASCII locales to use the standard indexes for `LIKE` comparisons. This release adds a way to create a special index for `LIKE`.
- Allow the postmaster to preload libraries using `preload_libraries` (Joe)

  For shared libraries that require a long time to load, this option is available so the library can be preloaded in the postmaster and inherited by all database sessions.
- Improve optimizer cost computations, particularly for subqueries (Tom)
- Avoid sort when subquery `ORDER BY` matches upper query (Tom)
- Deduce that `WHERE a.x = b.y AND b.y = 42` also means `a.x = 42` (Tom)
- Allow hash/merge joins on complex joins (Tom)
- Allow hash joins for more data types (Tom)
- Allow join optimization of explicit inner joins, disable with `join_collapse_limit` (Tom)
- Add parameter `from_collapse_limit` to control conversion of subqueries to joins (Tom)
- Use faster and more powerful regular expression code from Tcl (Henry Spencer, Tom)
- Use bit-mapped relation sets in the optimizer (Tom)
- Improve connection startup time (Tom)

  The new client/server protocol requires fewer network packets to start a database session.
- Improve trigger/constraint performance (Stephan)
- Improve speed of `col IN (const, const, const, ...)` (Tom)
- Fix hash indexes which were broken in rare cases (Tom)

- Improve hash index concurrency and speed (Tom)

  Prior releases suffered from poor hash index performance, particularly for high concurrency situations. This release fixes that, and the development group is interested in reports comparing B-tree and hash index performance.

- Align shared buffers on 32-byte boundary for copy speed improvement (Manfred Spraul)

  Certain CPU's perform faster data copies when addresses are 32-byte aligned.

- Data type `numeric` reimplemented for better performance (Tom)

  `numeric` used to be stored in base 100. The new code uses base 10000, for significantly better performance.

E.55.3.3. Server Configuration Changes

- Rename server parameter `server_min_messages` to `log_min_messages` (Bruce)

  This was done so most parameters that control the server logs begin with `log_`.

- Rename `show_*_stats` to `log_*_stats` (Bruce)
- Rename `show_source_port` to `log_source_port` (Bruce)
- Rename `hostname_lookup` to `log_hostname` (Bruce)
- Add `checkpoint_warning` to warn of excessive checkpointing (Bruce)

  In prior releases, it was difficult to determine if checkpoint was happening too frequently. This feature adds a warning to the server logs when excessive checkpointing happens.

- New read-only server parameters for localization (Tom)
- Change debug server log messages to output as `DEBUG` rather than `LOG` (Bruce)
- Prevent server log variables from being turned off by non-superusers (Bruce)

  This is a security feature so non-superusers cannot disable logging that was enabled by the administrator.

- `log_min_messages`/`client_min_messages` now controls `debug_*` output (Bruce)

  This centralizes client debug information so all debug output can be sent to either the client or server logs.

- Add Mac OS X Rendezvous server support (Chris Campbell)

  This allows Mac OS X hosts to query the network for available PostgreSQL servers.

- Add ability to print only slow statements using `log_min_duration_statement` (Christopher)

  This is an often requested debugging feature that allows administrators to see only slow queries in their server logs.

- Allow `pg_hba.conf` to accept netmasks in CIDR format (Andrew Dunstan)

  This allows administrators to merge the host IP address and netmask fields into a single CIDR field in `pg_hba.conf`.

- New read-only parameter `is_superuser` (Tom)

- New parameter `log_error_verbosity` to control error detail (Tom)

  This works with the new error reporting feature to supply additional error information like hints, file names and line numbers.

- `postgres --describe-config` now dumps server config variables (Aizaz Ahmed, Peter)

  This option is useful for administration tools that need to know the configuration variable names and their minimums, maximums, defaults, and descriptions.

- Add new columns in `pg_settings`: `context`, `type`, `source`, `min_val`, `max_val` (Joe)

- Make default `shared_buffers` 1000 and `max_connections` 100, if possible (Tom)

  Prior versions defaulted to 64 shared buffers so PostgreSQL would start on even very old systems. This release tests the amount of shared memory allowed by the platform and selects more reasonable default values if possible. Of course, users are still encouraged to evaluate their resource load and size `shared_buffers` accordingly.

- New `pg_hba.conf` record type `hostnossl` to prevent SSL connections (Jon Jensen)

  In prior releases, there was no way to prevent SSL connections if both the client and server supported SSL. This option allows that capability.

- Remove parameter `geqo_random_seed` (Tom)

- Add server parameter `regex_flavor` to control regular expression processing (Tom)

- Make `pg_ctl` better handle nonstandard ports (Greg)

E.55.3.4. Query Changes

- New SQL-standard information schema (Peter)

- Add read-only transactions (Peter)

- Print key name and value in foreign-key violation messages (Dmitry Tkach)

- Allow users to see their own queries in `pg_stat_activity` (Kevin Brown)

  In prior releases, only the superuser could see query strings using `pg_stat_activity`. Now ordinary users can see their own query strings.

- Fix aggregates in subqueries to match SQL standard (Tom)

  The SQL standard says that an aggregate function appearing within a nested subquery belongs to the outer query if its argument contains only outer-query variables. Prior PostgreSQL releases did not handle this fine point correctly.

- Add option to prevent auto-addition of tables referenced in query (Nigel J. Andrews)

  By default, tables mentioned in the query are automatically added to the `FROM` clause if they are not already there. This is compatible with historic POSTGRES behavior but is contrary to the SQL standard. This option allows selecting standard-compatible behavior.

- Allow `UPDATE ... SET col = DEFAULT` (Rod)

  This allows `UPDATE` to set a column to its declared default value.

- Allow expressions to be used in `LIMIT`/`OFFSET` (Tom)

In prior releases, LIMIT/OFFSET could only use constants, not expressions.

- Implement CREATE TABLE AS EXECUTE (Neil, Peter)

E.55.3.5. Object Manipulation Changes

- Make CREATE SEQUENCE grammar more conforming to SQL:2003 (Neil)
- Add statement-level triggers (Neil)

  While this allows a trigger to fire at the end of a statement, it does not allow the trigger to access all rows modified by the statement. This capability is planned for a future release.
- Add check constraints for domains (Rod)

  This greatly increases the usefulness of domains by allowing them to use check constraints.
- Add ALTER DOMAIN (Rod)

  This allows manipulation of existing domains.
- Fix several zero-column table bugs (Tom)

  PostgreSQL supports zero-column tables. This fixes various bugs that occur when using such tables.
- Have ALTER TABLE ... ADD PRIMARY KEY add not-null constraint (Rod)

  In prior releases, ALTER TABLE ... ADD PRIMARY would add a unique index, but not a not-null constraint. That is fixed in this release.
- Add ALTER TABLE ... WITHOUT OIDS (Rod)

  This allows control over whether new and updated rows will have an OID column. This is most useful for saving storage space.
- Add ALTER SEQUENCE to modify minimum, maximum, increment, cache, cycle values (Rod)
- Add ALTER TABLE ... CLUSTER ON (Alvaro Herrera)

  This command is used by pg_dump to record the cluster column for each table previously clustered. This information is used by database-wide cluster to cluster all previously clustered tables.
- Improve automatic type casting for domains (Rod, Tom)
- Allow dollar signs in identifiers, except as first character (Tom)
- Disallow dollar signs in operator names, so x=$1 works (Tom)
- Allow copying table schema using LIKE *subtable*, also SQL:2003 feature INCLUDING DEFAULTS (Rod)
- Add WITH GRANT OPTION clause to GRANT (Peter)

  This enabled GRANT to give other users the ability to grant privileges on a object.

E.55.3.6. Utility Command Changes

- Add ON COMMIT clause to CREATE TABLE for temporary tables (Gavin)

This adds the ability for a table to be dropped or all rows deleted on transaction commit.

- Allow cursors outside transactions using `WITH HOLD` (Neil)

  In previous releases, cursors were removed at the end of the transaction that created them. Cursors can now be created with the `WITH HOLD` option, which allows them to continue to be accessed after the creating transaction has committed.

- `FETCH 0` and `MOVE 0` now do nothing (Bruce)

  In previous releases, `FETCH 0` fetched all remaining rows, and `MOVE 0` moved to the end of the cursor.

- Cause `FETCH` and `MOVE` to return the number of rows fetched/moved, or zero if at the beginning/end of cursor, per SQL standard (Bruce)

  In prior releases, the row count returned by `FETCH` and `MOVE` did not accurately reflect the number of rows processed.

- Properly handle `SCROLL` with cursors, or report an error (Neil)

  Allowing random access (both forward and backward scrolling) to some kinds of queries cannot be done without some additional work. If `SCROLL` is specified when the cursor is created, this additional work will be performed. Furthermore, if the cursor has been created with `NO SCROLL`, no random access is allowed.

- Implement SQL-compatible options `FIRST`, `LAST`, `ABSOLUTE` *n*, `RELATIVE` *n* for `FETCH` and `MOVE` (Tom)

- Allow `EXPLAIN` on `DECLARE CURSOR` (Tom)

- Allow `CLUSTER` to use index marked as pre-clustered by default (Alvaro Herrera)

- Allow `CLUSTER` to cluster all tables (Alvaro Herrera)

  This allows all previously clustered tables in a database to be reclustered with a single command.

- Prevent `CLUSTER` on partial indexes (Tom)

- Allow DOS and Mac line-endings in `COPY` files (Bruce)

- Disallow literal carriage return as a data value, backslash-carriage-return and `\r` are still allowed (Bruce)

- `COPY` changes (binary, `\.`) (Tom)

- Recover from `COPY` failure cleanly (Tom)

- Prevent possible memory leaks in `COPY` (Tom)

- Make `TRUNCATE` transaction-safe (Rod)

  `TRUNCATE` can now be used inside a transaction. If the transaction aborts, the changes made by the `TRUNCATE` are automatically rolled back.

- Allow prepare/bind of utility commands like `FETCH` and `EXPLAIN` (Tom)

- Add `EXPLAIN EXECUTE` (Neil)

- Improve `VACUUM` performance on indexes by reducing WAL traffic (Tom)

- Functional indexes have been generalized into indexes on expressions (Tom)

  In prior releases, functional indexes only supported a simple function applied to one or more column names. This release allows any type of scalar expression.

- Have `SHOW TRANSACTION ISOLATION` match input to `SET TRANSACTION ISOLATION` (Tom)
- Have `COMMENT ON DATABASE` on nonlocal database generate a warning, rather than an error (Rod)

  Database comments are stored in database-local tables so comments on a database have to be stored in each database.
- Improve reliability of `LISTEN`/`NOTIFY` (Tom)
- Allow `REINDEX` to reliably reindex nonshared system catalog indexes (Tom)

  This allows system tables to be reindexed without the requirement of a standalone session, which was necessary in previous releases. The only tables that now require a standalone session for reindexing are the global system tables `pg_database`, `pg_shadow`, and `pg_group`.

E.55.3.7. Data Type and Function Changes

- New server parameter `extra_float_digits` to control precision display of floating-point numbers (Pedro Ferreira, Tom)

  This controls output precision which was causing regression testing problems.
- Allow `+1300` as a numeric time-zone specifier, for FJST (Tom)
- Remove rarely used functions `oidrand`, `oidsrand`, and `userfntest` functions (Neil)
- Add `md5()` function to main server, already in `contrib/pgcrypto` (Joe)

  An MD5 function was frequently requested. For more complex encryption capabilities, use `contrib/pgcrypto`.
- Increase date range of `timestamp` (John Cochran)
- Change `EXTRACT(EPOCH FROM timestamp)` so `timestamp without time zone` is assumed to be in local time, not GMT (Tom)
- Trap division by zero in case the operating system doesn't prevent it (Tom)
- Change the `numeric` data type internally to base 10000 (Tom)
- New `hostmask()` function (Greg Wickham)
- Fixes for `to_char()` and `to_timestamp()` (Karel)
- Allow functions that can take any argument data type and return any data type, using `anyelement` and `anyarray` (Joe)

  This allows the creation of functions that can work with any data type.
- Arrays can now be specified as `ARRAY[1,2,3]`, `ARRAY[['a','b'],['c','d']]`, or `ARRAY[ARRAY[ARRAY[2]]]` (Joe)
- Allow proper comparisons for arrays, including `ORDER BY` and `DISTINCT` support (Joe)
- Allow indexes on array columns (Joe)
- Allow array concatenation with `||` (Joe)
- Allow `WHERE` qualification *`expr op`* `ANY/SOME/ALL (`*`array_expr`*`)` (Joe)

This allows arrays to behave like a list of values, for purposes like `SELECT * FROM tab WHERE col IN (array_val)`.

- New array functions `array_append`, `array_cat`, `array_lower`, `array_prepend`, `array_to_string`, `array_upper`, `string_to_array` (Joe)
- Allow user defined aggregates to use polymorphic functions (Joe)
- Allow assignments to empty arrays (Joe)
- Allow 60 in seconds fields of `time`, `timestamp`, and `interval` input values (Tom)

  Sixty-second values are needed for leap seconds.
- Allow `cidr` data type to be cast to `text` (Tom)
- Disallow invalid time zone names in SET TIMEZONE
- Trim trailing spaces when `char` is cast to `varchar` or `text` (Tom)
- Make `float(`*`p`*`)` measure the precision *`p`* in binary digits, not decimal digits (Tom)
- Add IPv6 support to the `inet` and `cidr` data types (Michael Graff)
- Add `family()` function to report whether address is IPv4 or IPv6 (Michael Graff)
- Have `SHOW datestyle` generate output similar to that used by `SET datestyle` (Tom)
- Make `EXTRACT(TIMEZONE)` and `SET/SHOW TIME ZONE` follow the SQL convention for the sign of time zone offsets, i.e., positive is east from UTC (Tom)
- Fix `date_trunc('quarter', ...)` (Böjthe Zoltán)

  Prior releases returned an incorrect value for this function call.
- Make `initcap()` more compatible with Oracle (Mike Nolan)

  `initcap()` now uppercases a letter appearing after any non-alphanumeric character, rather than only after whitespace.
- Allow only `datestyle` field order for date values not in ISO-8601 format (Greg)
- Add new `datestyle` values `MDY`, `DMY`, and `YMD` to set input field order; honor `US` and `European` for backward compatibility (Tom)
- String literals like `'now'` or `'today'` will no longer work as a column default. Use functions such as `now()`, `current_timestamp` instead. (change required for prepared statements) (Tom)
- Treat NaN as larger than any other value in `min()`/`max()` (Tom)

  NaN was already sorted after ordinary numeric values for most purposes, but `min()` and `max()` didn't get this right.
- Prevent interval from suppressing `:00` seconds display
- New functions `pg_get_triggerdef(prettyprint)` and `pg_conversion_is_visible()` (Christopher)
- Allow time to be specified as `040506` or `0405` (Tom)
- Input date order must now be `YYYY-MM-DD` (with 4-digit year) or match `datestyle`
- Make `pg_get_constraintdef` support unique, primary-key, and check constraints (Christopher)

E.55.3.8. Server-Side Language Changes

- Prevent PL/pgSQL crash when `RETURN NEXT` is used on a zero-row record variable (Tom)
- Make PL/Python's `spi_execute` interface handle null values properly (Andrew Bosma)
- Allow PL/pgSQL to declare variables of composite types without `%ROWTYPE` (Tom)
- Fix PL/Python's `_quote()` function to handle big integers
- Make PL/Python an untrusted language, now called `plpythonu` (Kevin Jacobs, Tom)

  The Python language no longer supports a restricted execution environment, so the trusted version of PL/Python was removed. If this situation changes, a version of PL/Python that can be used by non-superusers will be readded.
- Allow polymorphic PL/pgSQL functions (Joe, Tom)
- Allow polymorphic SQL functions (Joe)
- Improved compiled function caching mechanism in PL/pgSQL with full support for polymorphism (Joe)
- Add new parameter `$0` in PL/pgSQL representing the function's actual return type (Joe)
- Allow PL/Tcl and PL/Python to use the same trigger on multiple tables (Tom)
- Fixed PL/Tcl's `spi_prepare` to accept fully qualified type names in the parameter type list (Jan)

E.55.3.9. psql Changes

- Add `\pset pager always` to always use pager (Greg)

  This forces the pager to be used even if the number of rows is less than the screen height. This is valuable for rows that wrap across several screen rows.
- Improve tab completion (Rod, Ross Reedstrom, Ian Barwick)
- Reorder `\?` help into groupings (Harald Armin Massa, Bruce)
- Add backslash commands for listing schemas, casts, and conversions (Christopher)
- `\encoding` now changes based on the server parameter `client_encoding` (Tom)

  In previous versions, `\encoding` was not aware of encoding changes made using `SET client_encoding`.
- Save editor buffer into readline history (Ross)

  When `\e` is used to edit a query, the result is saved in the readline history for retrieval using the up arrow.
- Improve `\d` display (Christopher)
- Enhance HTML mode to be more standards-conforming (Greg)
- New `\set AUTOCOMMIT off` capability (Tom)

  This takes the place of the removed server parameter `autocommit`.
- New `\set VERBOSITY` to control error detail (Tom)

This controls the new error reporting details.

- New prompt escape sequence `%x` to show transaction status (Tom)
- Long options for psql are now available on all platforms

E.55.3.10. pg_dump Changes

- Multiple pg_dump fixes, including tar format and large objects
- Allow pg_dump to dump specific schemas (Neil)
- Make pg_dump preserve column storage characteristics (Christopher)

  This preserves `ALTER TABLE ... SET STORAGE` information.
- Make pg_dump preserve `CLUSTER` characteristics (Christopher)
- Have pg_dumpall use `GRANT`/`REVOKE` to dump database-level privileges (Tom)
- Allow pg_dumpall to support the options `-a`, `-s`, `-x` of pg_dump (Tom)
- Prevent pg_dump from lowercasing identifiers specified on the command line (Tom)
- pg_dump options `--use-set-session-authorization` and `--no-reconnect` now do nothing, all dumps use `SET SESSION AUTHORIZATION`

  pg_dump no longer reconnects to switch users, but instead always uses `SET SESSION AUTHORIZATION`. This will reduce password prompting during restores.
- Long options for pg_dump are now available on all platforms

  PostgreSQL now includes its own long-option processing routines.

E.55.3.11. libpq Changes

- Add function `PQfreemem` for freeing memory on Windows, suggested for `NOTIFY` (Bruce)

  Windows requires that memory allocated in a library be freed by a function in the same library, hence `free()` doesn't work for freeing memory allocated by libpq. `PQfreemem` is the proper way to free libpq memory, especially on Windows, and is recommended for other platforms as well.
- Document service capability, and add sample file (Bruce)

  This allows clients to look up connection information in a central file on the client machine.
- Make `PQsetdbLogin` have the same defaults as `PQconnectdb` (Tom)
- Allow libpq to cleanly fail when result sets are too large (Tom)
- Improve performance of function `PQunescapeBytea` (Ben Lamb)
- Allow thread-safe libpq with `configure` option `--enable-thread-safety` (Lee Kindness, Philip Yarra)
- Allow function `pqInternalNotice` to accept a format string and arguments instead of just a preformatted message (Tom, Sean Chittenden)

- Control SSL negotiation with `sslmode` values `disable`, `allow`, `prefer`, and `require` (Jon Jensen)
- Allow new error codes and levels of text (Tom)
- Allow access to the underlying table and column of a query result (Tom)

  This is helpful for query-builder applications that want to know the underlying table and column names associated with a specific result set.
- Allow access to the current transaction status (Tom)
- Add ability to pass binary data directly to the server (Tom)
- Add function `PQexecPrepared` and `PQsendQueryPrepared` functions which perform bind/execute of previously prepared statements (Tom)

E.55.3.12. JDBC Changes

- Allow `setNull` on updateable result sets
- Allow `executeBatch` on a prepared statement (Barry)
- Support SSL connections (Barry)
- Handle schema names in result sets (Paul Sorenson)
- Add refcursor support (Nic Ferrier)

E.55.3.13. Miscellaneous Interface Changes

- Prevent possible memory leak or core dump during libpgtcl shutdown (Tom)
- Add Informix compatibility to ECPG (Michael)

  This allows ECPG to process embedded C programs that were written using certain Informix extensions.
- Add type `decimal` to ECPG that is fixed length, for Informix (Michael)
- Allow thread-safe embedded SQL programs with `configure` option `--enable-thread-safety` (Lee Kindness, Bruce)

  This allows multiple threads to access the database at the same time.
- Moved Python client PyGreSQL to http://www.pygresql.org (Marc)

E.55.3.14. Source Code Changes

- Prevent need for separate platform geometry regression result files (Tom)
- Improved PPC locking primitive (Reinhard Max)
- New function `palloc0` to allocate and clear memory (Bruce)
- Fix locking code for s390x CPU (64-bit) (Tom)

- Allow OpenBSD to use local ident credentials (William Ahern)
- Make query plan trees read-only to executor (Tom)
- Add Darwin startup scripts (David Wheeler)
- Allow libpq to compile with Borland C++ compiler (Lester Godwin, Karl Waclawek)
- Use our own version of `getopt_long()` if needed (Peter)
- Convert administration scripts to C (Peter)
- Bison >= 1.85 is now required to build the PostgreSQL grammar, if building from CVS
- Merge documentation into one book (Peter)
- Add Windows compatibility functions (Bruce)
- Allow client interfaces to compile under MinGW (Bruce)
- New `ereport()` function for error reporting (Tom)
- Support Intel compiler on Linux (Peter)
- Improve Linux startup scripts (Slawomir Sudnik, Darko Prenosil)
- Add support for AMD Opteron and Itanium (Jeffrey W. Baker, Bruce)
- Remove `--enable-recode` option from `configure`

  This was no longer needed now that we have `CREATE CONVERSION`.
- Generate a compile error if spinlock code is not found (Bruce)

  Platforms without spinlock code will now fail to compile, rather than silently using semaphores. This failure can be disabled with a new `configure` option.

E.55.3.15. Contrib Changes

- Change dbmirror license to BSD
- Improve earthdistance (Bruno Wolff III)
- Portability improvements to pgcrypto (Marko Kreen)
- Prevent crash in xml (John Gray, Michael Richards)
- Update oracle
- Update mysql
- Update cube (Bruno Wolff III)
- Update earthdistance to use cube (Bruno Wolff III)
- Update btree_gist (Oleg)
- New tsearch2 full-text search module (Oleg, Teodor)
- Add hash-based crosstab function to tablefuncs (Joe)
- Add serial column to order `connectby()` siblings in tablefuncs (Nabil Sayegh,Joe)
- Add named persistent connections to dblink (Shridhar Daithanka)

- New pg_autovacuum allows automatic `VACUUM` (Matthew T. O'Connor)
- Make pgbench honor environment variables `PGHOST`, `PGPORT`, `PGUSER` (Tatsuo)
- Improve intarray (Teodor Sigaev)
- Improve pgstattuple (Rod)
- Fix bug in `metaphone()` in fuzzystrmatch
- Improve adddepend (Rod)
- Update spi/timetravel (Böjthe Zoltán)
- Fix dbase `-s` option and improve non-ASCII handling (Thomas Behr, Márcio Smiderle)
- Remove array module because features now included by default (Joe)

E.56. Release 7.3.21

Release date: 2008-01-07

This release contains a variety of fixes from 7.3.20, including fixes for significant security issues.

This is expected to be the last PostgreSQL release in the 7.3.X series. Users are encouraged to update to a newer release branch soon.

E.56.1. Migration to Version 7.3.21

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.13, see the release notes for 7.3.13.

E.56.2. Changes

- Prevent functions in indexes from executing with the privileges of the user running `VACUUM`, `ANALYZE`, etc (Tom)

  Functions used in index expressions and partial-index predicates are evaluated whenever a new table entry is made. It has long been understood that this poses a risk of trojan-horse code execution if one modifies a table owned by an untrustworthy user. (Note that triggers, defaults, check constraints, etc. pose the same type of risk.) But functions in indexes pose extra danger because they will be executed by routine maintenance operations such as `VACUUM FULL`, which are commonly performed automatically under a superuser account. For example, a nefarious user can execute code with superuser privileges by setting up a trojan-horse index definition and waiting for the next routine vacuum. The fix arranges for standard maintenance operations (including `VACUUM`, `ANALYZE`, `REINDEX`, and `CLUSTER`) to execute as the table owner rather than the calling user, using the same privilege-switching mechanism already

used for SECURITY DEFINER functions. To prevent bypassing this security measure, execution of SET SESSION AUTHORIZATION and SET ROLE is now forbidden within a SECURITY DEFINER context. (CVE-2007-6600)

- Require non-superusers who use /contrib/dblink to use only password authentication, as a security measure (Joe)

  The fix that appeared for this in 7.3.20 was incomplete, as it plugged the hole for only some dblink functions. (CVE-2007-6601, CVE-2007-3278)

- Fix potential crash in translate() when using a multibyte database encoding (Tom)
- Make contrib/tablefunc's crosstab() handle NULL rowid as a category in its own right, rather than crashing (Joe)
- Require a specific version of Autoconf to be used when re-generating the configure script (Peter)

  This affects developers and packagers only. The change was made to prevent accidental use of untested combinations of Autoconf and PostgreSQL versions. You can remove the version check if you really want to use a different Autoconf version, but it's your responsibility whether the result works or not.

E.57. Release 7.3.20

Release date: 2007-09-17

This release contains fixes from 7.3.19.

E.57.1. Migration to Version 7.3.20

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.13, see the release notes for 7.3.13.

E.57.2. Changes

- Prevent index corruption when a transaction inserts rows and then aborts close to the end of a concurrent VACUUM on the same table (Tom)
- Make CREATE DOMAIN ... DEFAULT NULL work properly (Tom)
- Fix crash when log_min_error_statement logging runs out of memory (Tom)
- Require non-superusers who use /contrib/dblink to use only password authentication, as a security measure (Joe)

E.58. Release 7.3.19

Release date: 2007-04-23

This release contains fixes from 7.3.18, including a security fix.

E.58.1. Migration to Version 7.3.19

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.13, see the release notes for 7.3.13.

E.58.2. Changes

- Support explicit placement of the temporary-table schema within `search_path`, and disable searching it for functions and operators (Tom)

  This is needed to allow a security-definer function to set a truly secure value of `search_path`. Without it, an unprivileged SQL user can use temporary objects to execute code with the privileges of the security-definer function (CVE-2007-2138). See `CREATE FUNCTION` for more information.
- Fix potential-data-corruption bug in how `VACUUM FULL` handles `UPDATE` chains (Tom, Pavan Deolasee)

E.59. Release 7.3.18

Release date: 2007-02-05

This release contains a variety of fixes from 7.3.17, including a security fix.

E.59.1. Migration to Version 7.3.18

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.13, see the release notes for 7.3.13.

E.59.2. Changes

- Remove security vulnerability that allowed connected users to read backend memory (Tom)

The vulnerability involves changing the data type of a table column used in a SQL function (CVE-2007-0555). This error can easily be exploited to cause a backend crash, and in principle might be used to read database content that the user should not be able to access.

- Fix rare bug wherein btree index page splits could fail due to choosing an infeasible split point (Heikki Linnakangas)
- Tighten security of multi-byte character processing for UTF8 sequences over three bytes long (Tom)

E.60. Release 7.3.17

Release date: 2007-01-08

This release contains a variety of fixes from 7.3.16.

E.60.1. Migration to Version 7.3.17

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.13, see the release notes for 7.3.13.

E.60.2. Changes

- `to_number()` and `to_char(numeric)` are now `STABLE`, not `IMMUTABLE`, for new initdb installs (Tom)

  This is because `lc_numeric` can potentially change the output of these functions.
- Improve index usage of regular expressions that use parentheses (Tom)

  This improves psql `\d` performance also.

E.61. Release 7.3.16

Release date: 2006-10-16

This release contains a variety of fixes from 7.3.15.

E.61.1. Migration to Version 7.3.16

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.13, see the release notes for 7.3.13.

E.61.2. Changes

- Fix corner cases in pattern matching for psql's `\d` commands
- Fix index-corrupting bugs in /contrib/ltree (Teodor)
- Back-port 7.4 spinlock code to improve performance and support 64-bit architectures better
- Fix SSL-related memory leak in libpq
- Fix backslash escaping in /contrib/dbmirror
- Adjust regression tests for recent changes in US DST laws

E.62. Release 7.3.15

Release date: 2006-05-23

This release contains a variety of fixes from 7.3.14, including patches for extremely serious security issues.

E.62.1. Migration to Version 7.3.15

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.13, see the release notes for 7.3.13.

Full security against the SQL-injection attacks described in CVE-2006-2313 and CVE-2006-2314 might require changes in application code. If you have applications that embed untrustworthy strings into SQL commands, you should examine them as soon as possible to ensure that they are using recommended escaping techniques. In most cases, applications should be using subroutines provided by libraries or drivers (such as libpq's `PQescapeStringConn()`) to perform string escaping, rather than relying on *ad hoc* code to do it.

E.62.2. Changes

- Change the server to reject invalidly-encoded multibyte characters in all cases (Tatsuo, Tom)

  While PostgreSQL has been moving in this direction for some time, the checks are now applied uniformly to all encodings and all textual input, and are now always errors not merely warnings. This change defends against SQL-injection attacks of the type described in CVE-2006-2313.

- Reject unsafe uses of `\'` in string literals

  As a server-side defense against SQL-injection attacks of the type described in CVE-2006-2314, the server now only accepts `''` and not `\'` as a representation of ASCII single quote in SQL string literals. By default, `\'` is rejected only when `client_encoding` is set to a client-only encoding (SJIS, BIG5, GBK, GB18030, or UHC), which is the scenario in which SQL injection is possible. A new configuration parameter `backslash_quote` is available to adjust this behavior when needed. Note that full security against CVE-2006-2314 might require client-side changes; the purpose of `backslash_quote` is in part to make it obvious that insecure clients are insecure.

- Modify libpq's string-escaping routines to be aware of encoding considerations

  This fixes libpq-using applications for the security issues described in CVE-2006-2313 and CVE-2006-2314. Applications that use multiple PostgreSQL connections concurrently should migrate to `PQescapeStringConn()` and `PQescapeByteaConn()` to ensure that escaping is done correctly for the settings in use in each database connection. Applications that do string escaping "by hand" should be modified to rely on library routines instead.

- Fix some incorrect encoding conversion functions

  `win1251_to_iso`, `alt_to_iso`, `euc_tw_to_big5`, `euc_tw_to_mic`, `mic_to_euc_tw` were all broken to varying extents.

- Clean up stray remaining uses of `\'` in strings (Bruce, Jan)

- Fix server to use custom DH SSL parameters correctly (Michael Fuhr)

- Fix various minor memory leaks

E.63. Release 7.3.14

Release date: 2006-02-14

This release contains a variety of fixes from 7.3.13.

E.63.1. Migration to Version 7.3.14

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.13, see the release notes for 7.3.13.

E.63.2. Changes

- Fix potential crash in `SET SESSION AUTHORIZATION` (CVE-2006-0553)

An unprivileged user could crash the server process, resulting in momentary denial of service to other users, if the server has been compiled with Asserts enabled (which is not the default). Thanks to Akio Ishida for reporting this problem.

- Fix bug with row visibility logic in self-inserted rows (Tom)

  Under rare circumstances a row inserted by the current command could be seen as already valid, when it should not be. Repairs bug created in 7.3.11 release.

- Fix race condition that could lead to "file already exists" errors during pg_clog file creation (Tom)
- Fix to allow restoring dumps that have cross-schema references to custom operators (Tom)
- Portability fix for testing presence of `finite` and `isinf` during configure (Tom)

E.64. Release 7.3.13

Release date: 2006-01-09

This release contains a variety of fixes from 7.3.12.

E.64.1. Migration to Version 7.3.13

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.10, see the release notes for 7.3.10. Also, you might need to `REINDEX` indexes on textual columns after updating, if you are affected by the locale or plperl issues described below.

E.64.2. Changes

- Fix character string comparison for locales that consider different character combinations as equal, such as Hungarian (Tom)

  This might require `REINDEX` to fix existing indexes on textual columns.

- Set locale environment variables during postmaster startup to ensure that plperl won't change the locale later

  This fixes a problem that occurred if the postmaster was started with environment variables specifying a different locale than what initdb had been told. Under these conditions, any use of plperl was likely to lead to corrupt indexes. You might need `REINDEX` to fix existing indexes on textual columns if this has happened to you.

- Fix longstanding bug in strpos() and regular expression handling in certain rarely used Asian multi-byte character sets (Tatsuo)
- Fix bug in `/contrib/pgcrypto` gen_salt, which caused it not to use all available salt space for MD5 and XDES algorithms (Marko Kreen, Solar Designer)

Salts for Blowfish and standard DES are unaffected.

- Fix `/contrib/dblink` to throw an error, rather than crashing, when the number of columns specified is different from what's actually returned by the query (Joe)

E.65. Release 7.3.12

Release date: 2005-12-12

This release contains a variety of fixes from 7.3.11.

E.65.1. Migration to Version 7.3.12

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.10, see the release notes for 7.3.10.

E.65.2. Changes

- Fix race condition in transaction log management

  There was a narrow window in which an I/O operation could be initiated for the wrong page, leading to an Assert failure or data corruption.

- `/contrib/ltree` fixes (Teodor)
- Fix longstanding planning error for outer joins

  This bug sometimes caused a bogus error "RIGHT JOIN is only supported with merge-joinable join conditions".

- Prevent core dump in pg_autovacuum when a table has been dropped

E.66. Release 7.3.11

Release date: 2005-10-04

This release contains a variety of fixes from 7.3.10.

E.66.1. Migration to Version 7.3.11

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.10, see the release notes for 7.3.10.

E.66.2. Changes

- Fix error that allowed `VACUUM` to remove `ctid` chains too soon, and add more checking in code that follows `ctid` links

  This fixes a long-standing problem that could cause crashes in very rare circumstances.

- Fix `CHAR()` to properly pad spaces to the specified length when using a multiple-byte character set (Yoshiyuki Asaba)

  In prior releases, the padding of `CHAR()` was incorrect because it only padded to the specified number of bytes without considering how many characters were stored.

- Fix missing rows in queries like `UPDATE a=... WHERE a...` with GiST index on column `a`
- Improve checking for partially-written WAL pages
- Improve robustness of signal handling when SSL is enabled
- Various memory leakage fixes
- Various portability improvements
- Fix PL/PgSQL to handle `var := var` correctly when the variable is of pass-by-reference type

E.67. Release 7.3.10

Release date: 2005-05-09

This release contains a variety of fixes from 7.3.9, including several security-related issues.

E.67.1. Migration to Version 7.3.10

A dump/restore is not required for those running 7.3.X. However, it is one possible way of handling a significant security problem that has been found in the initial contents of 7.3.X system catalogs. A dump/initdb/reload sequence using 7.3.10's initdb will automatically correct this problem.

The security problem is that the built-in character set encoding conversion functions can be invoked from SQL commands by unprivileged users, but the functions were not designed for such use and are not secure against malicious choices of arguments. The fix involves changing the declared parameter list of these functions so that they can no longer be invoked from SQL commands. (This does not affect their normal use by the encoding conversion machinery.) It is strongly recommended that all installations repair this

error, either by initdb or by following the manual repair procedure given below. The error at least allows unprivileged database users to crash their server process, and might allow unprivileged users to gain the privileges of a database superuser.

If you wish not to do an initdb, perform the following procedure instead. As the database superuser, do:

```
BEGIN;
UPDATE pg_proc SET proargtypes[3] = 'internal'::regtype
WHERE pronamespace = 11 AND pronargs = 5
     AND proargtypes[2] = 'cstring'::regtype;
-- The command should report having updated 90 rows;
-- if not, rollback and investigate instead of committing!
COMMIT;
```

The above procedure must be carried out in *each* database of an installation, including `template1`, and ideally including `template0` as well. If you do not fix the template databases then any subsequently created databases will contain the same error. `template1` can be fixed in the same way as any other database, but fixing `template0` requires additional steps. First, from any database issue:

```
UPDATE pg_database SET datallowconn = true WHERE datname = 'template0';
```

Next connect to `template0` and perform the above repair procedure. Finally, do:

```
-- re-freeze template0:
VACUUM FREEZE;
-- and protect it against future alterations:
UPDATE pg_database SET datallowconn = false WHERE datname = 'template0';
```

E.67.2. Changes

- Change encoding function signature to prevent misuse
- Repair ancient race condition that allowed a transaction to be seen as committed for some purposes (eg SELECT FOR UPDATE) slightly sooner than for other purposes

  This is an extremely serious bug since it could lead to apparent data inconsistencies being briefly visible to applications.
- Repair race condition between relation extension and VACUUM

  This could theoretically have caused loss of a page's worth of freshly-inserted data, although the scenario seems of very low probability. There are no known cases of it having caused more than an Assert failure.
- Fix comparisons of `TIME WITH TIME ZONE` values

  The comparison code was wrong in the case where the `--enable-integer-datetimes` configuration switch had been used. NOTE: if you have an index on a `TIME WITH TIME ZONE` column, it will

need to be REINDEXed after installing this update, because the fix corrects the sort order of column values.

- Fix EXTRACT(EPOCH) for TIME WITH TIME ZONE values
- Fix mis-display of negative fractional seconds in INTERVAL values

  This error only occurred when the --enable-integer-datetimes configuration switch had been used.
- Additional buffer overrun checks in plpgsql (Neil)
- Fix pg_dump to dump trigger names containing % correctly (Neil)
- Prevent to_char(interval) from dumping core for month-related formats
- Fix contrib/pgcrypto for newer OpenSSL builds (Marko Kreen)
- Still more 64-bit fixes for contrib/intagg
- Prevent incorrect optimization of functions returning RECORD

E.68. Release 7.3.9

Release date: 2005-01-31

This release contains a variety of fixes from 7.3.8, including several security-related issues.

E.68.1. Migration to Version 7.3.9

A dump/restore is not required for those running 7.3.X.

E.68.2. Changes

- Disallow LOAD to non-superusers

  On platforms that will automatically execute initialization functions of a shared library (this includes at least Windows and ELF-based Unixen), LOAD can be used to make the server execute arbitrary code. Thanks to NGS Software for reporting this.
- Check that creator of an aggregate function has the right to execute the specified transition functions

  This oversight made it possible to bypass denial of EXECUTE permission on a function.
- Fix security and 64-bit issues in contrib/intagg
- Add needed STRICT marking to some contrib functions (Kris Jurka)
- Avoid buffer overrun when plpgsql cursor declaration has too many parameters (Neil)
- Fix planning error for FULL and RIGHT outer joins

The result of the join was mistakenly supposed to be sorted the same as the left input. This could not only deliver mis-sorted output to the user, but in case of nested merge joins could give outright wrong answers.

- Fix plperl for quote marks in tuple fields
- Fix display of negative intervals in SQL and GERMAN datestyles

E.69. Release 7.3.8

Release date: 2004-10-22

This release contains a variety of fixes from 7.3.7.

E.69.1. Migration to Version 7.3.8

A dump/restore is not required for those running 7.3.X.

E.69.2. Changes

- Repair possible failure to update hint bits on disk

  Under rare circumstances this oversight could lead to "could not access transaction status" failures, which qualifies it as a potential-data-loss bug.

- Ensure that hashed outer join does not miss tuples

  Very large left joins using a hash join plan could fail to output unmatched left-side rows given just the right data distribution.

- Disallow running pg_ctl as root

  This is to guard against any possible security issues.

- Avoid using temp files in /tmp in make_oidjoins_check

  This has been reported as a security issue, though it's hardly worthy of concern since there is no reason for non-developers to use this script anyway.

E.70. Release 7.3.7

Release date: 2004-08-16

This release contains one critical fix over 7.3.6, and some minor items.

E.70.1. Migration to Version 7.3.7

A dump/restore is not required for those running 7.3.X.

E.70.2. Changes

- Prevent possible loss of committed transactions during crash

  Due to insufficient interlocking between transaction commit and checkpointing, it was possible for transactions committed just before the most recent checkpoint to be lost, in whole or in part, following a database crash and restart. This is a serious bug that has existed since PostgreSQL 7.1.
- Remove asymmetrical word processing in tsearch (Teodor)
- Properly schema-qualify function names when pg_dump'ing a CAST

E.71. Release 7.3.6

Release date: 2004-03-02

This release contains a variety of fixes from 7.3.5.

E.71.1. Migration to Version 7.3.6

A dump/restore is *not* required for those running 7.3.*.

E.71.2. Changes

- Revert erroneous changes in rule permissions checking

  A patch applied in 7.3.3 to fix a corner case in rule permissions checks turns out to have disabled rule-related permissions checks in many not-so-corner cases. This would for example allow users to insert into views they weren't supposed to have permission to insert into. We have therefore reverted the 7.3.3 patch. The original bug will be fixed in 8.0.
- Repair incorrect order of operations in GetNewTransactionId()

  This bug could result in failure under out-of-disk-space conditions, including inability to restart even after disk space is freed.
- Ensure configure selects -fno-strict-aliasing even when an external value for CFLAGS is supplied

On some platforms, building with -fstrict-aliasing causes bugs.

- Make pg_restore handle 64-bit off_t correctly

  This bug prevented proper restoration from archive files exceeding 4 GB.

- Make contrib/dblink not assume that local and remote type OIDs match (Joe)
- Quote connectby()'s start_with argument properly (Joe)
- Don't crash when a rowtype argument to a plpgsql function is NULL
- Avoid generating invalid character encoding sequences in corner cases when planning LIKE operations
- Ensure text_position() cannot scan past end of source string in multibyte cases (Korea PostgreSQL Users' Group)
- Fix index optimization and selectivity estimates for LIKE operations on bytea columns (Joe)

E.72. Release 7.3.5

Release date: 2003-12-03

This has a variety of fixes from 7.3.4.

E.72.1. Migration to Version 7.3.5

A dump/restore is *not* required for those running 7.3.*.

E.72.2. Changes

- Force zero_damaged_pages to be on during recovery from WAL
- Prevent some obscure cases of "variable not in subplan target lists"
- Force stats processes to detach from shared memory, ensuring cleaner shutdown
- Make PQescapeBytea and byteaout consistent with each other (Joe)
- Added missing SPI_finish() calls to dblink's get_tuple_of_interest() (Joe)
- Fix for possible foreign key violation when rule rewrites INSERT (Jan)
- Support qualified type names in PL/Tcl's spi_prepare command (Jan)
- Make pg_dump handle a procedural language handler located in pg_catalog
- Make pg_dump handle cases where a custom opclass is in another schema
- Make pg_dump dump binary-compatible casts correctly (Jan)
- Fix insertion of expressions containing subqueries into rule bodies

- Fix incorrect argument processing in clusterdb script (Anand Ranganathan)
- Fix problems with dropped columns in plpython triggers
- Repair problems with to_char() reading past end of its input string (Karel)
- Fix GB18030 mapping errors (Tatsuo)
- Fix several problems with SSL error handling and asynchronous SSL I/O
- Remove ability to bind a list of values to a single parameter in JDBC (prevents possible SQL-injection attacks)
- Fix some errors in HAVE_INT64_TIMESTAMP code paths
- Fix corner case for btree search in parallel with first root page split

E.73. Release 7.3.4

Release date: 2003-07-24

This has a variety of fixes from 7.3.3.

E.73.1. Migration to Version 7.3.4

A dump/restore is *not* required for those running 7.3.*.

E.73.2. Changes

- Repair breakage in timestamp-to-date conversion for dates before 2000
- Prevent rare possibility of server startup failure (Tom)
- Fix bugs in interval-to-time conversion (Tom)
- Add constraint names in a few places in pg_dump (Rod)
- Improve performance of functions with many parameters (Tom)
- Fix to_ascii() buffer overruns (Tom)
- Prevent restore of database comments from throwing an error (Tom)
- Work around buggy strxfrm() present in some Solaris releases (Tom)
- Properly escape jdbc setObject() strings to improve security (Barry)

E.74. Release 7.3.3

Release date: 2003-05-22

This release contains a variety of fixes for version 7.3.2.

E.74.1. Migration to Version 7.3.3

A dump/restore is *not* required for those running version 7.3.*.

E.74.2. Changes

- Repair sometimes-incorrect computation of StartUpID after a crash
- Avoid slowness with lots of deferred triggers in one transaction (Stephan)
- Don't lock referenced row when `UPDATE` doesn't change foreign key's value (Jan)
- Use `-fPIC` not `-fpic` on Sparc (Tom Callaway)
- Repair lack of schema-awareness in contrib/reindexdb
- Fix contrib/intarray error for zero-element result array (Teodor)
- Ensure createuser script will exit on control-C (Oliver)
- Fix errors when the type of a dropped column has itself been dropped
- `CHECKPOINT` does not cause database panic on failure in noncritical steps
- Accept 60 in seconds fields of timestamp, time, interval input values
- Issue notice, not error, if `TIMESTAMP(`, `TIME`, or `INTERVAL` precision too large
- Fix `abstime-to-time` cast function (fix is not applied unless you initdb)
- Fix pg_proc entry for `timestampt_izone` (fix is not applied unless you initdb)
- Make `EXTRACT(EPOCH FROM timestamp without time zone)` treat input as local time
- `'now'::timestamptz` gave wrong answer if timezone changed earlier in transaction
- `HAVE_INT64_TIMESTAMP` code for time with timezone overwrote its input
- Accept `GLOBAL TEMP/TEMPORARY` as a synonym for `TEMPORARY`
- Avoid improper schema-privilege-check failure in foreign-key triggers
- Fix bugs in foreign-key triggers for `SET DEFAULT` action
- Fix incorrect time-qual check in row fetch for `UPDATE` and `DELETE` triggers
- Foreign-key clauses were parsed but ignored in `ALTER TABLE ADD COLUMN`
- Fix createlang script breakage for case where handler function already exists
- Fix misbehavior on zero-column tables in pg_dump, COPY, ANALYZE, other places

- Fix misbehavior of `func_error()` on type names containing '%'
- Fix misbehavior of `replace()` on strings containing '%'
- Regular-expression patterns containing certain multibyte characters failed
- Account correctly for `NULL`s in more cases in join size estimation
- Avoid conflict with system definition of `isblank()` function or macro
- Fix failure to convert large code point values in EUC_TW conversions (Tatsuo)
- Fix error recovery for `SSL_read`/`SSL_write` calls
- Don't do early constant-folding of type coercion expressions
- Validate page header fields immediately after reading in any page
- Repair incorrect check for ungrouped variables in unnamed joins
- Fix buffer overrun in `to_ascii` (Guido Notari)
- contrib/ltree fixes (Teodor)
- Fix core dump in deadlock detection on machines where char is unsigned
- Avoid running out of buffers in many-way indexscan (bug introduced in 7.3)
- Fix planner's selectivity estimation functions to handle domains properly
- Fix dbmirror memory-allocation bug (Steven Singer)
- Prevent infinite loop in `ln(numeric)` due to roundoff error
- `GROUP BY` got confused if there were multiple equal GROUP BY items
- Fix bad plan when inherited `UPDATE`/`DELETE` references another inherited table
- Prevent clustering on incomplete (partial or non-NULL-storing) indexes
- Service shutdown request at proper time if it arrives while still starting up
- Fix left-links in temporary indexes (could make backwards scans miss entries)
- Fix incorrect handling of client_encoding setting in postgresql.conf (Tatsuo)
- Fix failure to respond to `pg_ctl stop -m fast` after Async_NotifyHandler runs
- Fix SPI for case where rule contains multiple statements of the same type
- Fix problem with checking for wrong type of access privilege in rule query
- Fix problem with `EXCEPT` in `CREATE RULE`
- Prevent problem with dropping temp tables having serial columns
- Fix replace_vars_with_subplan_refs failure in complex views
- Fix regexp slowness in single-byte encodings (Tatsuo)
- Allow qualified type names in `CREATE CAST` and `DROP CAST`
- Accept `SETOF type[]`, which formerly had to be written `SETOF _type`
- Fix pg_dump core dump in some cases with procedural languages
- Force ISO datestyle in pg_dump output, for portability (Oliver)
- pg_dump failed to handle error return from `lo_read` (Oleg Drokin)

- pg_dumpall failed with groups having no members (Nick Eskelinen)
- pg_dumpall failed to recognize --globals-only switch
- pg_restore failed to restore blobs if -X disable-triggers is specified
- Repair intrafunction memory leak in plpgsql
- pltcl's `elog` command dumped core if given wrong parameters (Ian Harding)
- plpython used wrong value of `atttypmod` (Brad McLean)
- Fix improper quoting of boolean values in Python interface (D'Arcy)
- Added `addDataType()` method to PGConnection interface for JDBC
- Fixed various problems with updateable ResultSets for JDBC (Shawn Green)
- Fixed various problems with DatabaseMetaData for JDBC (Kris Jurka, Peter Royal)
- Fixed problem with parsing table ACLs in JDBC
- Better error message for character set conversion problems in JDBC

E.75. Release 7.3.2

Release date: 2003-02-04

This release contains a variety of fixes for version 7.3.1.

E.75.1. Migration to Version 7.3.2

A dump/restore is *not* required for those running version 7.3.*.

E.75.2. Changes

- Restore creation of OID column in CREATE TABLE AS / SELECT INTO
- Fix pg_dump core dump when dumping views having comments
- Dump DEFERRABLE/INITIALLY DEFERRED constraints properly
- Fix UPDATE when child table's column numbering differs from parent
- Increase default value of max_fsm_relations
- Fix problem when fetching backwards in a cursor for a single-row query
- Make backward fetch work properly with cursor on SELECT DISTINCT query
- Fix problems with loading pg_dump files containing contrib/lo usage
- Fix problem with all-numeric user names

- Fix possible memory leak and core dump during disconnect in libpgtcl
- Make plpython's spi_execute command handle nulls properly (Andrew Bosma)
- Adjust plpython error reporting so that its regression test passes again
- Work with bison 1.875
- Handle mixed-case names properly in plpgsql's %type (Neil)
- Fix core dump in pltcl when executing a query rewritten by a rule
- Repair array subscript overruns (per report from Yichen Xie)
- Reduce MAX_TIME_PRECISION from 13 to 10 in floating-point case
- Correctly case-fold variable names in per-database and per-user settings
- Fix coredump in plpgsql's RETURN NEXT when SELECT into record returns no rows
- Fix outdated use of pg_type.typprtlen in python client interface
- Correctly handle fractional seconds in timestamps in JDBC driver
- Improve performance of getImportedKeys() in JDBC
- Make shared-library symlinks work standardly on HPUX (Giles)
- Repair inconsistent rounding behavior for timestamp, time, interval
- SSL negotiation fixes (Nathan Mueller)
- Make libpq's ~/.pgpass feature work when connecting with PQconnectDB
- Update my2pg, ora2pg
- Translation updates
- Add casts between types lo and oid in contrib/lo
- fastpath code now checks for privilege to call function

E.76. Release 7.3.1

Release date: 2002-12-18

This release contains a variety of fixes for version 7.3.

E.76.1. Migration to Version 7.3.1

A dump/restore is *not* required for those running version 7.3. However, it should be noted that the main PostgreSQL interface library, libpq, has a new major version number for this release, which might require recompilation of client code in certain cases.

E.76.2. Changes

- Fix a core dump of COPY TO when client/server encodings don't match (Tom)
- Allow pg_dump to work with pre-7.2 servers (Philip)
- contrib/adddepend fixes (Tom)
- Fix problem with deletion of per-user/per-database config settings (Tom)
- contrib/vacuumlo fix (Tom)
- Allow 'password' encryption even when pg_shadow contains MD5 passwords (Bruce)
- contrib/dbmirror fix (Steven Singer)
- Optimizer fixes (Tom)
- contrib/tsearch fixes (Teodor Sigaev, Magnus)
- Allow locale names to be mixed case (Nicolai Tufar)
- Increment libpq library's major version number (Bruce)
- pg_hba.conf error reporting fixes (Bruce, Neil)
- Add SCO Openserver 5.0.4 as a supported platform (Bruce)
- Prevent EXPLAIN from crashing server (Tom)
- SSL fixes (Nathan Mueller)
- Prevent composite column creation via ALTER TABLE (Tom)

E.77. Release 7.3

Release date: 2002-11-27

E.77.1. Overview

Major changes in this release:

Schemas

Schemas allow users to create objects in separate namespaces, so two people or applications can have tables with the same name. There is also a public schema for shared tables. Table/index creation can be restricted by removing privileges on the public schema.

Drop Column

PostgreSQL now supports the `ALTER TABLE ... DROP COLUMN` functionality.

Table Functions

Functions returning multiple rows and/or multiple columns are now much easier to use than before. You can call such a "table function" in the `SELECT FROM` clause, treating its output like a table. Also, PL/pgSQL functions can now return sets.

Prepared Queries

PostgreSQL now supports prepared queries, for improved performance.

Dependency Tracking

PostgreSQL now records object dependencies, which allows improvements in many areas. `DROP` statements now take either `CASCADE` or `RESTRICT` to control whether dependent objects are also dropped.

Privileges

Functions and procedural languages now have privileges, and functions can be defined to run with the privileges of their creator.

Internationalization

Both multibyte and locale support are now always enabled.

Logging

A variety of logging options have been enhanced.

Interfaces

A large number of interfaces have been moved to http://gborg.postgresql.org where they can be developed and released independently.

Functions/Identifiers

By default, functions can now take up to 32 parameters, and identifiers can be up to 63 bytes long. Also, `OPAQUE` is now deprecated: there are specific "pseudo-datatypes" to represent each of the former meanings of `OPAQUE` in function argument and result types.

E.77.2. Migration to Version 7.3

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release. If your application examines the system catalogs, additional changes will be required due to the introduction of schemas in 7.3; for more information, see: http://developer.postgresql.org/~momjian/upgrade_tips_7.3.

Observe the following incompatibilities:

- Pre-6.3 clients are no longer supported.
- `pg_hba.conf` now has a column for the user name and additional features. Existing files need to be adjusted.
- Several `postgresql.conf` logging parameters have been renamed.
- `LIMIT #,#` has been disabled; use `LIMIT # OFFSET #`.

- `INSERT` statements with column lists must specify a value for each specified column. For example, `INSERT INTO tab (col1, col2) VALUES ('val1')` is now invalid. It's still allowed to supply fewer columns than expected if the `INSERT` does not have a column list.
- `serial` columns are no longer automatically `UNIQUE`; thus, an index will not automatically be created.
- A `SET` command inside an aborted transaction is now rolled back.
- `COPY` no longer considers missing trailing columns to be null. All columns need to be specified. (However, one can achieve a similar effect by specifying a column list in the `COPY` command.)
- The data type `timestamp` is now equivalent to `timestamp without time zone`, instead of `timestamp with time zone`.
- Pre-7.3 databases loaded into 7.3 will not have the new object dependencies for `serial` columns, unique constraints, and foreign keys. See the directory `contrib/adddepend/` for a detailed description and a script that will add such dependencies.
- An empty string (″) is no longer allowed as the input into an integer field. Formerly, it was silently interpreted as 0.

E.77.3. Changes

E.77.3.1. Server Operation

- Add pg_locks view to show locks (Neil)
- Security fixes for password negotiation memory allocation (Neil)
- Remove support for version 0 FE/BE protocol (PostgreSQL 6.2 and earlier) (Tom)
- Reserve the last few backend slots for superusers, add parameter superuser_reserved_connections to control this (Nigel J. Andrews)

E.77.3.2. Performance

- Improve startup by calling localtime() only once (Tom)
- Cache system catalog information in flat files for faster startup (Tom)
- Improve caching of index information (Tom)
- Optimizer improvements (Tom, Fernando Nasser)
- Catalog caches now store failed lookups (Tom)
- Hash function improvements (Neil)
- Improve performance of query tokenization and network handling (Peter)
- Speed improvement for large object restore (Mario Weilguni)
- Mark expired index entries on first lookup, saving later heap fetches (Tom)
- Avoid excessive NULL bitmap padding (Manfred Koizar)

- Add BSD-licensed qsort() for Solaris, for performance (Bruce)
- Reduce per-row overhead by four bytes (Manfred Koizar)
- Fix GEQO optimizer bug (Neil Conway)
- Make WITHOUT OID actually save four bytes per row (Manfred Koizar)
- Add default_statistics_target variable to specify ANALYZE buckets (Neil)
- Use local buffer cache for temporary tables so no WAL overhead (Tom)
- Improve free space map performance on large tables (Stephen Marshall, Tom)
- Improved WAL write concurrency (Tom)

E.77.3.3. Privileges

- Add privileges on functions and procedural languages (Peter)
- Add OWNER to CREATE DATABASE so superusers can create databases on behalf of unprivileged users (Gavin Sherry, Tom)
- Add new object privilege bits EXECUTE and USAGE (Tom)
- Add SET SESSION AUTHORIZATION DEFAULT and RESET SESSION AUTHORIZATION (Tom)
- Allow functions to be executed with the privilege of the function owner (Peter)

E.77.3.4. Server Configuration

- Server log messages now tagged with LOG, not DEBUG (Bruce)
- Add user column to pg_hba.conf (Bruce)
- Have log_connections output two lines in log file (Tom)
- Remove debug_level from postgresql.conf, now server_min_messages (Bruce)
- New ALTER DATABASE/USER ... SET command for per-user/database initialization (Peter)
- New parameters server_min_messages and client_min_messages to control which messages are sent to the server logs or client applications (Bruce)
- Allow pg_hba.conf to specify lists of users/databases separated by commas, group names prepended with +, and file names prepended with @ (Bruce)
- Remove secondary password file capability and pg_password utility (Bruce)
- Add variable db_user_namespace for database-local user names (Bruce)
- SSL improvements (Bear Giles)
- Make encryption of stored passwords the default (Bruce)
- Allow pg_statistics to be reset by calling pg_stat_reset() (Christopher)
- Add log_duration parameter (Bruce)

- Rename debug_print_query to log_statement (Bruce)
- Rename show_query_stats to show_statement_stats (Bruce)
- Add param log_min_error_statement to print commands to logs on error (Gavin)

E.77.3.5. Queries

- Make cursors insensitive, meaning their contents do not change (Tom)
- Disable LIMIT #,# syntax; now only LIMIT # OFFSET # supported (Bruce)
- Increase identifier length to 63 (Neil, Bruce)
- UNION fixes for merging >= 3 columns of different lengths (Tom)
- Add DEFAULT key word to INSERT, e.g., INSERT ... (..., DEFAULT, ...) (Rod)
- Allow views to have default values using ALTER COLUMN ... SET DEFAULT (Neil)
- Fail on INSERTs with column lists that don't supply all column values, e.g., INSERT INTO tab (col1, col2) VALUES ('val1'); (Rod)
- Fix for join aliases (Tom)
- Fix for FULL OUTER JOINs (Tom)
- Improve reporting of invalid identifier and location (Tom, Gavin)
- Fix OPEN cursor(args) (Tom)
- Allow 'ctid' to be used in a view and currtid(viewname) (Hiroshi)
- Fix for CREATE TABLE AS with UNION (Tom)
- SQL99 syntax improvements (Thomas)
- Add statement_timeout variable to cancel queries (Bruce)
- Allow prepared queries with PREPARE/EXECUTE (Neil)
- Allow FOR UPDATE to appear after LIMIT/OFFSET (Bruce)
- Add variable autocommit (Tom, David Van Wie)

E.77.3.6. Object Manipulation

- Make equals signs optional in CREATE DATABASE (Gavin Sherry)
- Make ALTER TABLE OWNER change index ownership too (Neil)
- New ALTER TABLE tabname ALTER COLUMN colname SET STORAGE controls TOAST storage, compression (John Gray)
- Add schema support, CREATE/DROP SCHEMA (Tom)
- Create schema for temporary tables (Tom)
- Add variable search_path for schema search (Tom)

- Add ALTER TABLE SET/DROP NOT NULL (Christopher)
- New CREATE FUNCTION volatility levels (Tom)
- Make rule names unique only per table (Tom)
- Add 'ON tablename' clause to DROP RULE and COMMENT ON RULE (Tom)
- Add ALTER TRIGGER RENAME (Joe)
- New current_schema() and current_schemas() inquiry functions (Tom)
- Allow functions to return multiple rows (table functions) (Joe)
- Make WITH optional in CREATE DATABASE, for consistency (Bruce)
- Add object dependency tracking (Rod, Tom)
- Add RESTRICT/CASCADE to DROP commands (Rod)
- Add ALTER TABLE DROP for non-CHECK CONSTRAINT (Rod)
- Autodestroy sequence on DROP of table with SERIAL (Rod)
- Prevent column dropping if column is used by foreign key (Rod)
- Automatically drop constraints/functions when object is dropped (Rod)
- Add CREATE/DROP OPERATOR CLASS (Bill Studenmund, Tom)
- Add ALTER TABLE DROP COLUMN (Christopher, Tom, Hiroshi)
- Prevent inherited columns from being removed or renamed (Alvaro Herrera)
- Fix foreign key constraints to not error on intermediate database states (Stephan)
- Propagate column or table renaming to foreign key constraints
- Add CREATE OR REPLACE VIEW (Gavin, Neil, Tom)
- Add CREATE OR REPLACE RULE (Gavin, Neil, Tom)
- Have rules execute alphabetically, returning more predictable values (Tom)
- Triggers are now fired in alphabetical order (Tom)
- Add /contrib/adddepend to handle pre-7.3 object dependencies (Rod)
- Allow better casting when inserting/updating values (Tom)

E.77.3.7. Utility Commands

- Have COPY TO output embedded carriage returns and newlines as \r and \n (Tom)
- Allow DELIMITER in COPY FROM to be 8-bit clean (Tatsuo)
- Make pg_dump use ALTER TABLE ADD PRIMARY KEY, for performance (Neil)
- Disable brackets in multistatement rules (Bruce)
- Disable VACUUM from being called inside a function (Bruce)
- Allow dropdb and other scripts to use identifiers with spaces (Bruce)
- Restrict database comment changes to the current database

- Allow comments on operators, independent of the underlying function (Rod)
- Rollback SET commands in aborted transactions (Tom)
- EXPLAIN now outputs as a query (Tom)
- Display condition expressions and sort keys in EXPLAIN (Tom)
- Add 'SET LOCAL var = value' to set configuration variables for a single transaction (Tom)
- Allow ANALYZE to run in a transaction (Bruce)
- Improve COPY syntax using new WITH clauses, keep backward compatibility (Bruce)
- Fix pg_dump to consistently output tags in non-ASCII dumps (Bruce)
- Make foreign key constraints clearer in dump file (Rod)
- Add COMMENT ON CONSTRAINT (Rod)
- Allow COPY TO/FROM to specify column names (Brent Verner)
- Dump UNIQUE and PRIMARY KEY constraints as ALTER TABLE (Rod)
- Have SHOW output a query result (Joe)
- Generate failure on short COPY lines rather than pad NULLs (Neil)
- Fix CLUSTER to preserve all table attributes (Alvaro Herrera)
- New pg_settings table to view/modify GUC settings (Joe)
- Add smart quoting, portability improvements to pg_dump output (Peter)
- Dump serial columns out as SERIAL (Tom)
- Enable large file support, >2G for pg_dump (Peter, Philip Warner, Bruce)
- Disallow TRUNCATE on tables that are involved in referential constraints (Rod)
- Have TRUNCATE also auto-truncate the toast table of the relation (Tom)
- Add clusterdb utility that will auto-cluster an entire database based on previous CLUSTER operations (Alvaro Herrera)
- Overhaul pg_dumpall (Peter)
- Allow REINDEX of TOAST tables (Tom)
- Implemented START TRANSACTION, per SQL99 (Neil)
- Fix rare index corruption when a page split affects bulk delete (Tom)
- Fix ALTER TABLE ... ADD COLUMN for inheritance (Alvaro Herrera)

E.77.3.8. Data Types and Functions

- Fix factorial(0) to return 1 (Bruce)
- Date/time/timezone improvements (Thomas)
- Fix for array slice extraction (Tom)
- Fix extract/date_part to report proper microseconds for timestamp (Tatsuo)

- Allow text_substr() and bytea_substr() to read TOAST values more efficiently (John Gray)
- Add domain support (Rod)
- Make WITHOUT TIME ZONE the default for TIMESTAMP and TIME data types (Thomas)
- Allow alternate storage scheme of 64-bit integers for date/time types using --enable-integer-datetimes in configure (Thomas)
- Make timezone(timestamptz) return timestamp rather than a string (Thomas)
- Allow fractional seconds in date/time types for dates prior to 1BC (Thomas)
- Limit timestamp data types to 6 decimal places of precision (Thomas)
- Change timezone conversion functions from timetz() to timezone() (Thomas)
- Add configuration variables datestyle and timezone (Tom)
- Add OVERLAY(), which allows substitution of a substring in a string (Thomas)
- Add SIMILAR TO (Thomas, Tom)
- Add regular expression SUBSTRING(string FROM pat FOR escape) (Thomas)
- Add LOCALTIME and LOCALTIMESTAMP functions (Thomas)
- Add named composite types using CREATE TYPE typename AS (column) (Joe)
- Allow composite type definition in the table alias clause (Joe)
- Add new API to simplify creation of C language table functions (Joe)
- Remove ODBC-compatible empty parentheses from calls to SQL99 functions for which these parentheses do not match the standard (Thomas)
- Allow macaddr data type to accept 12 hex digits with no separators (Mike Wyer)
- Add CREATE/DROP CAST (Peter)
- Add IS DISTINCT FROM operator (Thomas)
- Add SQL99 TREAT() function, synonym for CAST() (Thomas)
- Add pg_backend_pid() to output backend pid (Bruce)
- Add IS OF / IS NOT OF type predicate (Thomas)
- Allow bit string constants without fully-specified length (Thomas)
- Allow conversion between 8-byte integers and bit strings (Thomas)
- Implement hex literal conversion to bit string literal (Thomas)
- Allow table functions to appear in the FROM clause (Joe)
- Increase maximum number of function parameters to 32 (Bruce)
- No longer automatically create index for SERIAL column (Tom)
- Add current_database() (Rod)
- Fix cash_words() to not overflow buffer (Tom)
- Add functions replace(), split_part(), to_hex() (Joe)
- Fix LIKE for bytea as a right-hand argument (Joe)

- Prevent crashes caused by SELECT cash_out(2) (Tom)
- Fix to_char(1,'FM999.99') to return a period (Karel)
- Fix trigger/type/language functions returning OPAQUE to return proper type (Tom)

E.77.3.9. Internationalization

- Add additional encodings: Korean (JOHAB), Thai (WIN874), Vietnamese (TCVN), Arabic (WIN1256), Simplified Chinese (GBK), Korean (UHC) (Eiji Tokuya)
- Enable locale support by default (Peter)
- Add locale variables (Peter)
- Escape byes >= 0x7f for multibyte in PQescapeBytea/PQunescapeBytea (Tatsuo)
- Add locale awareness to regular expression character classes
- Enable multibyte support by default (Tatsuo)
- Add GB18030 multibyte support (Bill Huang)
- Add CREATE/DROP CONVERSION, allowing loadable encodings (Tatsuo, Kaori)
- Add pg_conversion table (Tatsuo)
- Add SQL99 CONVERT() function (Tatsuo)
- pg_dumpall, pg_controldata, and pg_resetxlog now national-language aware (Peter)
- New and updated translations

E.77.3.10. Server-side Languages

- Allow recursive SQL function (Peter)
- Change PL/Tcl build to use configured compiler and Makefile.shlib (Peter)
- Overhaul the PL/pgSQL FOUND variable to be more Oracle-compatible (Neil, Tom)
- Allow PL/pgSQL to handle quoted identifiers (Tom)
- Allow set-returning PL/pgSQL functions (Neil)
- Make PL/pgSQL schema-aware (Joe)
- Remove some memory leaks (Nigel J. Andrews, Tom)

E.77.3.11. psql

- Don't lowercase psql \connect database name for 7.2.0 compatibility (Tom)
- Add psql \timing to time user queries (Greg Sabino Mullane)
- Have psql \d show index information (Greg Sabino Mullane)

- New psql \dD shows domains (Jonathan Eisler)
- Allow psql to show rules on views (Paul ?)
- Fix for psql variable substitution (Tom)
- Allow psql \d to show temporary table structure (Tom)
- Allow psql \d to show foreign keys (Rod)
- Fix \? to honor \pset pager (Bruce)
- Have psql reports its version number on startup (Tom)
- Allow \copy to specify column names (Tom)

E.77.3.12. libpq

- Add ~/.pgpass to store host/user password combinations (Alvaro Herrera)
- Add PQunescapeBytea() function to libpq (Patrick Welche)
- Fix for sending large queries over non-blocking connections (Bernhard Herzog)
- Fix for libpq using timers on Win9X (David Ford)
- Allow libpq notify to handle servers with different-length identifiers (Tom)
- Add libpq PQescapeString() and PQescapeBytea() to Windows (Bruce)
- Fix for SSL with non-blocking connections (Jack Bates)
- Add libpq connection timeout parameter (Denis A Ustimenko)

E.77.3.13. JDBC

- Allow JDBC to compile with JDK 1.4 (Dave)
- Add JDBC 3 support (Barry)
- Allows JDBC to set loglevel by adding ?loglevel=X to the connection URL (Barry)
- Add Driver.info() message that prints out the version number (Barry)
- Add updateable result sets (Raghu Nidagal, Dave)
- Add support for callable statements (Paul Bethe)
- Add query cancel capability
- Add refresh row (Dave)
- Fix MD5 encryption handling for multibyte servers (Jun Kawai)
- Add support for prepared statements (Barry)

E.77.3.14. Miscellaneous Interfaces

- Fixed ECPG bug concerning octal numbers in single quotes (Michael)
- Move src/interfaces/libpgeasy to http://gborg.postgresql.org (Marc, Bruce)
- Improve Python interface (Elliot Lee, Andrew Johnson, Greg Copeland)
- Add libpgtcl connection close event (Gerhard Hintermayer)
- Move src/interfaces/libpq++ to http://gborg.postgresql.org (Marc, Bruce)
- Move src/interfaces/odbc to http://gborg.postgresql.org (Marc)
- Move src/interfaces/libpgeasy to http://gborg.postgresql.org (Marc, Bruce)
- Move src/interfaces/perl5 to http://gborg.postgresql.org (Marc, Bruce)
- Remove src/bin/pgaccess from main tree, now at http://www.pgaccess.org (Bruce)
- Add pg_on_connection_loss command to libpgtcl (Gerhard Hintermayer, Tom)

E.77.3.15. Source Code

- Fix for parallel make (Peter)
- AIX fixes for linking Tcl (Andreas Zeugswetter)
- Allow PL/Perl to build under Cygwin (Jason Tishler)
- Improve MIPS compiles (Peter, Oliver Elphick)
- Require Autoconf version 2.53 (Peter)
- Require readline and zlib by default in configure (Peter)
- Allow Solaris to use Intimate Shared Memory (ISM), for performance (Scott Brunza, P.J. Josh Rovero)
- Always enable syslog in compile, remove --enable-syslog option (Tatsuo)
- Always enable multibyte in compile, remove --enable-multibyte option (Tatsuo)
- Always enable locale in compile, remove --enable-locale option (Peter)
- Fix for Win9x DLL creation (Magnus Naeslund)
- Fix for link() usage by WAL code on Windows, BeOS (Jason Tishler)
- Add sys/types.h to c.h, remove from main files (Peter, Bruce)
- Fix AIX hang on SMP machines (Tomoyuki Niijima)
- AIX SMP hang fix (Tomoyuki Niijima)
- Fix pre-1970 date handling on newer glibc libraries (Tom)
- Fix PowerPC SMP locking (Tom)
- Prevent gcc -ffast-math from being used (Peter, Tom)
- Bison >= 1.50 now required for developer builds
- Kerberos 5 support now builds with Heimdal (Peter)

- Add appendix in the User's Guide which lists SQL features (Thomas)
- Improve loadable module linking to use RTLD_NOW (Tom)
- New error levels WARNING, INFO, LOG, DEBUG[1-5] (Bruce)
- New src/port directory holds replaced libc functions (Peter, Bruce)
- New pg_namespace system catalog for schemas (Tom)
- Add pg_class.relnamespace for schemas (Tom)
- Add pg_type.typnamespace for schemas (Tom)
- Add pg_proc.pronamespace for schemas (Tom)
- Restructure aggregates to have pg_proc entries (Tom)
- System relations now have their own namespace, pg_* test not required (Fernando Nasser)
- Rename TOAST index names to be *_index rather than *_idx (Neil)
- Add namespaces for operators, opclasses (Tom)
- Add additional checks to server control file (Thomas)
- New Polish FAQ (Marcin Mazurek)
- Add Posix semaphore support (Tom)
- Document need for reindex (Bruce)
- Rename some internal identifiers to simplify Windows compile (Jan, Katherine Ward)
- Add documentation on computing disk space (Bruce)
- Remove KSQO from GUC (Bruce)
- Fix memory leak in rtree (Kenneth Been)
- Modify a few error messages for consistency (Bruce)
- Remove unused system table columns (Peter)
- Make system columns NOT NULL where appropriate (Tom)
- Clean up use of sprintf in favor of snprintf() (Neil, Jukka Holappa)
- Remove OPAQUE and create specific subtypes (Tom)
- Cleanups in array internal handling (Joe, Tom)
- Disallow pg_atoi('') (Bruce)
- Remove parameter wal_files because WAL files are now recycled (Bruce)
- Add version numbers to heap pages (Tom)

E.77.3.16. Contrib

- Allow inet arrays in /contrib/array (Neil)
- GiST fixes (Teodor Sigaev, Neil)
- Upgrade /contrib/mysql

- Add /contrib/dbsize which shows table sizes without vacuum (Peter)
- Add /contrib/intagg, integer aggregator routines (mlw)
- Improve /contrib/oid2name (Neil, Bruce)
- Improve /contrib/tsearch (Oleg, Teodor Sigaev)
- Cleanups of /contrib/rserver (Alexey V. Borzov)
- Update /contrib/oracle conversion utility (Gilles Darold)
- Update /contrib/dblink (Joe)
- Improve options supported by /contrib/vacuumlo (Mario Weilguni)
- Improvements to /contrib/intarray (Oleg, Teodor Sigaev, Andrey Oktyabrski)
- Add /contrib/reindexdb utility (Shaun Thomas)
- Add indexing to /contrib/isbn_issn (Dan Weston)
- Add /contrib/dbmirror (Steven Singer)
- Improve /contrib/pgbench (Neil)
- Add /contrib/tablefunc table function examples (Joe)
- Add /contrib/ltree data type for tree structures (Teodor Sigaev, Oleg Bartunov)
- Move /contrib/pg_controldata, pg_resetxlog into main tree (Bruce)
- Fixes to /contrib/cube (Bruno Wolff)
- Improve /contrib/fulltextindex (Christopher)

E.78. Release 7.2.8

Release date: 2005-05-09

This release contains a variety of fixes from 7.2.7, including one security-related issue.

E.78.1. Migration to Version 7.2.8

A dump/restore is not required for those running 7.2.X.

E.78.2. Changes

- Repair ancient race condition that allowed a transaction to be seen as committed for some purposes (eg SELECT FOR UPDATE) slightly sooner than for other purposes

This is an extremely serious bug since it could lead to apparent data inconsistencies being briefly visible to applications.

- Repair race condition between relation extension and VACUUM

  This could theoretically have caused loss of a page's worth of freshly-inserted data, although the scenario seems of very low probability. There are no known cases of it having caused more than an Assert failure.

- Fix `EXTRACT(EPOCH)` for `TIME WITH TIME ZONE` values
- Additional buffer overrun checks in plpgsql (Neil)
- Fix pg_dump to dump index names and trigger names containing `%` correctly (Neil)
- Prevent `to_char(interval)` from dumping core for month-related formats
- Fix `contrib/pgcrypto` for newer OpenSSL builds (Marko Kreen)

E.79. Release 7.2.7

Release date: 2005-01-31

This release contains a variety of fixes from 7.2.6, including several security-related issues.

E.79.1. Migration to Version 7.2.7

A dump/restore is not required for those running 7.2.X.

E.79.2. Changes

- Disallow `LOAD` to non-superusers

  On platforms that will automatically execute initialization functions of a shared library (this includes at least Windows and ELF-based Unixen), `LOAD` can be used to make the server execute arbitrary code. Thanks to NGS Software for reporting this.

- Add needed STRICT marking to some contrib functions (Kris Jurka)
- Avoid buffer overrun when plpgsql cursor declaration has too many parameters (Neil)
- Fix planning error for FULL and RIGHT outer joins

  The result of the join was mistakenly supposed to be sorted the same as the left input. This could not only deliver mis-sorted output to the user, but in case of nested merge joins could give outright wrong answers.

- Fix display of negative intervals in SQL and GERMAN datestyles

E.80. Release 7.2.6

Release date: 2004-10-22

This release contains a variety of fixes from 7.2.5.

E.80.1. Migration to Version 7.2.6

A dump/restore is not required for those running 7.2.X.

E.80.2. Changes

- Repair possible failure to update hint bits on disk

  Under rare circumstances this oversight could lead to "could not access transaction status" failures, which qualifies it as a potential-data-loss bug.

- Ensure that hashed outer join does not miss tuples

  Very large left joins using a hash join plan could fail to output unmatched left-side rows given just the right data distribution.

- Disallow running pg_ctl as root

  This is to guard against any possible security issues.

- Avoid using temp files in /tmp in make_oidjoins_check

  This has been reported as a security issue, though it's hardly worthy of concern since there is no reason for non-developers to use this script anyway.

- Update to newer versions of Bison

E.81. Release 7.2.5

Release date: 2004-08-16

This release contains a variety of fixes from 7.2.4.

E.81.1. Migration to Version 7.2.5

A dump/restore is not required for those running 7.2.X.

E.81.2. Changes

- Prevent possible loss of committed transactions during crash

  Due to insufficient interlocking between transaction commit and checkpointing, it was possible for transactions committed just before the most recent checkpoint to be lost, in whole or in part, following a database crash and restart. This is a serious bug that has existed since PostgreSQL 7.1.
- Fix corner case for btree search in parallel with first root page split
- Fix buffer overrun in `to_ascii` (Guido Notari)
- Fix core dump in deadlock detection on machines where char is unsigned
- Fix failure to respond to `pg_ctl stop -m fast` after Async_NotifyHandler runs
- Repair memory leaks in pg_dump
- Avoid conflict with system definition of `isblank()` function or macro

E.82. Release 7.2.4

Release date: 2003-01-30

This release contains a variety of fixes for version 7.2.3, including fixes to prevent possible data loss.

E.82.1. Migration to Version 7.2.4

A dump/restore is *not* required for those running version 7.2.*.

E.82.2. Changes

- Fix some additional cases of VACUUM "No one parent tuple was found" error
- Prevent VACUUM from being called inside a function (Bruce)
- Ensure pg_clog updates are sync'd to disk before marking checkpoint complete
- Avoid integer overflow during large hash joins
- Make GROUP commands work when pg_group.grolist is large enough to be toasted
- Fix errors in datetime tables; some timezone names weren't being recognized
- Fix integer overflows in circle_poly(), path_encode(), path_add() (Neil)
- Repair long-standing logic errors in lseg_eq(), lseg_ne(), lseg_center()

E.83. Release 7.2.3

Release date: 2002-10-01

This release contains a variety of fixes for version 7.2.2, including fixes to prevent possible data loss.

E.83.1. Migration to Version 7.2.3

A dump/restore is *not* required for those running version 7.2.*.

E.83.2. Changes

- Prevent possible compressed transaction log loss (Tom)
- Prevent non-superuser from increasing most recent vacuum info (Tom)
- Handle pre-1970 date values in newer versions of glibc (Tom)
- Fix possible hang during server shutdown
- Prevent spinlock hangs on SMP PPC machines (Tomoyuki Niijima)
- Fix pg_dump to properly dump FULL JOIN USING (Tom)

E.84. Release 7.2.2

Release date: 2002-08-23

This release contains a variety of fixes for version 7.2.1.

E.84.1. Migration to Version 7.2.2

A dump/restore is *not* required for those running version 7.2.*.

E.84.2. Changes

- Allow EXECUTE of "CREATE TABLE AS ... SELECT" in PL/pgSQL (Tom)
- Fix for compressed transaction log id wraparound (Tom)
- Fix PQescapeBytea/PQunescapeBytea so that they handle bytes > 0x7f (Tatsuo)

- Fix for psql and pg_dump crashing when invoked with non-existent long options (Tatsuo)
- Fix crash when invoking geometric operators (Tom)
- Allow OPEN cursor(args) (Tom)
- Fix for rtree_gist index build (Teodor)
- Fix for dumping user-defined aggregates (Tom)
- contrib/intarray fixes (Oleg)
- Fix for complex UNION/EXCEPT/INTERSECT queries using parens (Tom)
- Fix to pg_convert (Tatsuo)
- Fix for crash with long DATA strings (Thomas, Neil)
- Fix for repeat(), lpad(), rpad() and long strings (Neil)

E.85. Release 7.2.1

Release date: 2002-03-21

This release contains a variety of fixes for version 7.2.

E.85.1. Migration to Version 7.2.1

A dump/restore is *not* required for those running version 7.2.

E.85.2. Changes

- Ensure that sequence counters do not go backwards after a crash (Tom)
- Fix pgaccess kanji-conversion key binding (Tatsuo)
- Optimizer improvements (Tom)
- Cash I/O improvements (Tom)
- New Russian FAQ
- Compile fix for missing AuthBlockSig (Heiko)
- Additional time zones and time zone fixes (Thomas)
- Allow psql \connect to handle mixed case database and user names (Tom)
- Return proper OID on command completion even with ON INSERT rules (Tom)
- Allow COPY FROM to use 8-bit DELIMITERS (Tatsuo)
- Fix bug in extract/date_part for milliseconds/microseconds (Tatsuo)

- Improve handling of multiple UNIONs with different lengths (Tom)
- contrib/btree_gist improvements (Teodor Sigaev)
- contrib/tsearch dictionary improvements, see README.tsearch for an additional installation step (Thomas T. Thai, Teodor Sigaev)
- Fix for array subscripts handling (Tom)
- Allow EXECUTE of "CREATE TABLE AS ... SELECT" in PL/pgSQL (Tom)

E.86. Release 7.2

Release date: 2002-02-04

E.86.1. Overview

This release improves PostgreSQL for use in high-volume applications.

Major changes in this release:

VACUUM

Vacuuming no longer locks tables, thus allowing normal user access during the vacuum. A new `VACUUM FULL` command does old-style vacuum by locking the table and shrinking the on-disk copy of the table.

Transactions

There is no longer a problem with installations that exceed four billion transactions.

OIDs

OIDs are now optional. Users can now create tables without OIDs for cases where OID usage is excessive.

Optimizer

The system now computes histogram column statistics during `ANALYZE`, allowing much better optimizer choices.

Security

A new MD5 encryption option allows more secure storage and transfer of passwords. A new Unix-domain socket authentication option is available on Linux and BSD systems.

Statistics

Administrators can use the new table access statistics module to get fine-grained information about table and index usage.

Internationalization

Program and library messages can now be displayed in several languages.

E.86.2. Migration to Version 7.2

A dump/restore using `pg_dump` is required for those wishing to migrate data from any previous release.

Observe the following incompatibilities:

- The semantics of the `VACUUM` command have changed in this release. You might wish to update your maintenance procedures accordingly.
- In this release, comparisons using `= NULL` will always return false (or NULL, more precisely). Previous releases automatically transformed this syntax to `IS NULL`. The old behavior can be re-enabled using a `postgresql.conf` parameter.
- The `pg_hba.conf` and `pg_ident.conf` configuration is now only reloaded after receiving a SIGHUP signal, not with each connection.
- The function `octet_length()` now returns the uncompressed data length.
- The date/time value `'current'` is no longer available. You will need to rewrite your applications.
- The `timestamp()`, `time()`, and `interval()` functions are no longer available. Instead of `timestamp()`, use `timestamp 'string'` or `CAST`.

The `SELECT ... LIMIT #,#` syntax will be removed in the next release. You should change your queries to use separate LIMIT and OFFSET clauses, e.g. `LIMIT 10 OFFSET 20`.

E.86.3. Changes

E.86.3.1. Server Operation

- Create temporary files in a separate directory (Bruce)
- Delete orphaned temporary files on postmaster startup (Bruce)
- Added unique indexes to some system tables (Tom)
- System table operator reorganization (Oleg Bartunov, Teodor Sigaev, Tom)
- Renamed pg_log to pg_clog (Tom)
- Enable SIGTERM, SIGQUIT to kill backends (Jan)
- Removed compile-time limit on number of backends (Tom)
- Better cleanup for semaphore resource failure (Tatsuo, Tom)
- Allow safe transaction ID wraparound (Tom)
- Removed OIDs from some system tables (Tom)
- Removed "triggered data change violation" error check (Tom)

- SPI portal creation of prepared/saved plans (Jan)
- Allow SPI column functions to work for system columns (Tom)
- Long value compression improvement (Tom)
- Statistics collector for table, index access (Jan)
- Truncate extra-long sequence names to a reasonable value (Tom)
- Measure transaction times in milliseconds (Thomas)
- Fix TID sequential scans (Hiroshi)
- Superuser ID now fixed at 1 (Peter E)
- New pg_ctl "reload" option (Tom)

E.86.3.2. Performance

- Optimizer improvements (Tom)
- New histogram column statistics for optimizer (Tom)
- Reuse write-ahead log files rather than discarding them (Tom)
- Cache improvements (Tom)
- IS NULL, IS NOT NULL optimizer improvement (Tom)
- Improve lock manager to reduce lock contention (Tom)
- Keep relcache entries for index access support functions (Tom)
- Allow better selectivity with NaN and infinities in NUMERIC (Tom)
- R-tree performance improvements (Kenneth Been)
- B-tree splits more efficient (Tom)

E.86.3.3. Privileges

- Change UPDATE, DELETE privileges to be distinct (Peter E)
- New REFERENCES, TRIGGER privileges (Peter E)
- Allow GRANT/REVOKE to/from more than one user at a time (Peter E)
- New has_table_privilege() function (Joe Conway)
- Allow non-superuser to vacuum database (Tom)
- New SET SESSION AUTHORIZATION command (Peter E)
- Fix bug in privilege modifications on newly created tables (Tom)
- Disallow access to pg_statistic for non-superuser, add user-accessible views (Tom)

E.86.3.4. Client Authentication

- Fork postmaster before doing authentication to prevent hangs (Peter E)
- Add ident authentication over Unix domain sockets on Linux, *BSD (Helge Bahmann, Oliver Elphick, Teodor Sigaev, Bruce)
- Add a password authentication method that uses MD5 encryption (Bruce)
- Allow encryption of stored passwords using MD5 (Bruce)
- PAM authentication (Dominic J. Eidson)
- Load pg_hba.conf and pg_ident.conf only on startup and SIGHUP (Bruce)

E.86.3.5. Server Configuration

- Interpretation of some time zone abbreviations as Australian rather than North American now settable at run time (Bruce)
- New parameter to set default transaction isolation level (Peter E)
- New parameter to enable conversion of "expr = NULL" into "expr IS NULL", off by default (Peter E)
- New parameter to control memory usage by VACUUM (Tom)
- New parameter to set client authentication timeout (Tom)
- New parameter to set maximum number of open files (Tom)

E.86.3.6. Queries

- Statements added by INSERT rules now execute after the INSERT (Jan)
- Prevent unadorned relation names in target list (Bruce)
- NULLs now sort after all normal values in ORDER BY (Tom)
- New IS UNKNOWN, IS NOT UNKNOWN Boolean tests (Tom)
- New SHARE UPDATE EXCLUSIVE lock mode (Tom)
- New EXPLAIN ANALYZE command that shows run times and row counts (Martijn van Oosterhout)
- Fix problem with LIMIT and subqueries (Tom)
- Fix for LIMIT, DISTINCT ON pushed into subqueries (Tom)
- Fix nested EXCEPT/INTERSECT (Tom)

E.86.3.7. Schema Manipulation

- Fix SERIAL in temporary tables (Bruce)
- Allow temporary sequences (Bruce)

- Sequences now use int8 internally (Tom)
- New SERIAL8 creates int8 columns with sequences, default still SERIAL4 (Tom)
- Make OIDs optional using WITHOUT OIDS (Tom)
- Add %TYPE syntax to CREATE TYPE (Ian Lance Taylor)
- Add ALTER TABLE / DROP CONSTRAINT for CHECK constraints (Christopher Kings-Lynne)
- New CREATE OR REPLACE FUNCTION to alter existing function (preserving the function OID) (Gavin Sherry)
- Add ALTER TABLE / ADD [UNIQUE | PRIMARY] (Christopher Kings-Lynne)
- Allow column renaming in views
- Make ALTER TABLE / RENAME COLUMN update column names of indexes (Brent Verner)
- Fix for ALTER TABLE / ADD CONSTRAINT ... CHECK with inherited tables (Stephan Szabo)
- ALTER TABLE RENAME update foreign-key trigger arguments correctly (Brent Verner)
- DROP AGGREGATE and COMMENT ON AGGREGATE now accept an aggtype (Tom)
- Add automatic return type data casting for SQL functions (Tom)
- Allow GiST indexes to handle NULLs and multikey indexes (Oleg Bartunov, Teodor Sigaev, Tom)
- Enable partial indexes (Martijn van Oosterhout)

E.86.3.8. Utility Commands

- Add RESET ALL, SHOW ALL (Marko Kreen)
- CREATE/ALTER USER/GROUP now allow options in any order (Vince)
- Add LOCK A, B, C functionality (Neil Padgett)
- New ENCRYPTED/UNENCRYPTED option to CREATE/ALTER USER (Bruce)
- New light-weight VACUUM does not lock table; old semantics are available as VACUUM FULL (Tom)
- Disable COPY TO/FROM on views (Bruce)
- COPY DELIMITERS string must be exactly one character (Tom)
- VACUUM warning about index tuples fewer than heap now only appears when appropriate (Martijn van Oosterhout)
- Fix privilege checks for CREATE INDEX (Tom)
- Disallow inappropriate use of CREATE/DROP INDEX/TRIGGER/VIEW (Tom)

E.86.3.9. Data Types and Functions

- SUM(), AVG(), COUNT() now uses int8 internally for speed (Tom)
- Add convert(), convert2() (Tatsuo)

- New function bit_length() (Peter E)
- Make the "n" in CHAR(n)/VARCHAR(n) represents letters, not bytes (Tatsuo)
- CHAR(), VARCHAR() now reject strings that are too long (Peter E)
- BIT VARYING now rejects bit strings that are too long (Peter E)
- BIT now rejects bit strings that do not match declared size (Peter E)
- INET, CIDR text conversion functions (Alex Pilosov)
- INET, CIDR operators << and <<= indexable (Alex Pilosov)
- Bytea \### now requires valid three digit octal number
- Bytea comparison improvements, now supports =, <>, >, >=, <, and <=
- Bytea now supports B-tree indexes
- Bytea now supports LIKE, LIKE...ESCAPE, NOT LIKE, NOT LIKE...ESCAPE
- Bytea now supports concatenation
- New bytea functions: position, substring, trim, btrim, and length
- New encode() function mode, "escaped", converts minimally escaped bytea to/from text
- Add pg_database_encoding_max_length() (Tatsuo)
- Add pg_client_encoding() function (Tatsuo)
- now() returns time with millisecond precision (Thomas)
- New TIMESTAMP WITHOUT TIMEZONE data type (Thomas)
- Add ISO date/time specification with "T", yyyy-mm-ddThh:mm:ss (Thomas)
- New xid/int comparison functions (Hiroshi)
- Add precision to TIME, TIMESTAMP, and INTERVAL data types (Thomas)
- Modify type coercion logic to attempt binary-compatible functions first (Tom)
- New encode() function installed by default (Marko Kreen)
- Improved to_*() conversion functions (Karel Zak)
- Optimize LIKE/ILIKE when using single-byte encodings (Tatsuo)
- New functions in contrib/pgcrypto: crypt(), hmac(), encrypt(), gen_salt() (Marko Kreen)
- Correct description of translate() function (Bruce)
- Add INTERVAL argument for SET TIME ZONE (Thomas)
- Add INTERVAL YEAR TO MONTH (etc.) syntax (Thomas)
- Optimize length functions when using single-byte encodings (Tatsuo)
- Fix path_inter, path_distance, path_length, dist_ppath to handle closed paths (Curtis Barrett, Tom)
- octet_length(text) now returns non-compressed length (Tatsuo, Bruce)
- Handle "July" full name in date/time literals (Greg Sabino Mullane)
- Some datatype() function calls now evaluated differently
- Add support for Julian and ISO time specifications (Thomas)

E.86.3.10. Internationalization

- National language support in psql, pg_dump, libpq, and server (Peter E)
- Message translations in Chinese (simplified, traditional), Czech, French, German, Hungarian, Russian, Swedish (Peter E, Serguei A. Mokhov, Karel Zak, Weiping He, Zhenbang Wei, Kovacs Zoltan)
- Make trim, ltrim, rtrim, btrim, lpad, rpad, translate multibyte aware (Tatsuo)
- Add LATIN5,6,7,8,9,10 support (Tatsuo)
- Add ISO 8859-5,6,7,8 support (Tatsuo)
- Correct LATIN5 to mean ISO-8859-9, not ISO-8859-5 (Tatsuo)
- Make mic2ascii() non-ASCII aware (Tatsuo)
- Reject invalid multibyte character sequences (Tatsuo)

E.86.3.11. PL/pgSQL

- Now uses portals for SELECT loops, allowing huge result sets (Jan)
- CURSOR and REFCURSOR support (Jan)
- Can now return open cursors (Jan)
- Add ELSEIF (Klaus Reger)
- Improve PL/pgSQL error reporting, including location of error (Tom)
- Allow IS or FOR key words in cursor declaration, for compatibility (Bruce)
- Fix for SELECT ... FOR UPDATE (Tom)
- Fix for PERFORM returning multiple rows (Tom)
- Make PL/pgSQL use the server's type coercion code (Tom)
- Memory leak fix (Jan, Tom)
- Make trailing semicolon optional (Tom)

E.86.3.12. PL/Perl

- New untrusted PL/Perl (Alex Pilosov)
- PL/Perl is now built on some platforms even if libperl is not shared (Peter E)

E.86.3.13. PL/Tcl

- Now reports errorInfo (Vsevolod Lobko)
- Add spi_lastoid function (bob@redivi.com)

E.86.3.14. PL/Python

- ...is new (Andrew Bosma)

E.86.3.15. psql

- \d displays indexes in unique, primary groupings (Christopher Kings-Lynne)
- Allow trailing semicolons in backslash commands (Greg Sabino Mullane)
- Read password from /dev/tty if possible
- Force new password prompt when changing user and database (Tatsuo, Tom)
- Format the correct number of columns for Unicode (Patrice)

E.86.3.16. libpq

- New function PQescapeString() to escape quotes in command strings (Florian Weimer)
- New function PQescapeBytea() escapes binary strings for use as SQL string literals

E.86.3.17. JDBC

- Return OID of INSERT (Ken K)
- Handle more data types (Ken K)
- Handle single quotes and newlines in strings (Ken K)
- Handle NULL variables (Ken K)
- Fix for time zone handling (Barry Lind)
- Improved Druid support
- Allow eight-bit characters with non-multibyte server (Barry Lind)
- Support BIT, BINARY types (Ned Wolpert)
- Reduce memory usage (Michael Stephens, Dave Cramer)
- Update DatabaseMetaData (Peter E)
- Add DatabaseMetaData.getCatalogs() (Peter E)
- Encoding fixes (Anders Bengtsson)
- Get/setCatalog methods (Jason Davies)
- DatabaseMetaData.getColumns() now returns column defaults (Jason Davies)
- DatabaseMetaData.getColumns() performance improvement (Jeroen van Vianen)
- Some JDBC1 and JDBC2 merging (Anders Bengtsson)

- Transaction performance improvements (Barry Lind)
- Array fixes (Greg Zoller)
- Serialize addition
- Fix batch processing (Rene Pijlman)
- ExecSQL method reorganization (Anders Bengtsson)
- GetColumn() fixes (Jeroen van Vianen)
- Fix isWriteable() function (Rene Pijlman)
- Improved passage of JDBC2 conformance tests (Rene Pijlman)
- Add bytea type capability (Barry Lind)
- Add isNullable() (Rene Pijlman)
- JDBC date/time test suite fixes (Liam Stewart)
- Fix for SELECT 'id' AS xxx FROM table (Dave Cramer)
- Fix DatabaseMetaData to show precision properly (Mark Lillywhite)
- New getImported/getExported keys (Jason Davies)
- MD5 password encryption support (Jeremy Wohl)
- Fix to actually use type cache (Ned Wolpert)

E.86.3.18. ODBC

- Remove query size limit (Hiroshi)
- Remove text field size limit (Hiroshi)
- Fix for SQLPrimaryKeys in multibyte mode (Hiroshi)
- Allow ODBC procedure calls (Hiroshi)
- Improve boolean handing (Aidan Mountford)
- Most configuration options now settable via DSN (Hiroshi)
- Multibyte, performance fixes (Hiroshi)
- Allow driver to be used with iODBC or unixODBC (Peter E)
- MD5 password encryption support (Bruce)
- Add more compatibility functions to odbc.sql (Peter E)

E.86.3.19. ECPG

- EXECUTE ... INTO implemented (Christof Petig)
- Multiple row descriptor support (e.g. CARDINALITY) (Christof Petig)
- Fix for GRANT parameters (Lee Kindness)

- Fix INITIALLY DEFERRED bug
- Various bug fixes (Michael, Christof Petig)
- Auto allocation for indicator variable arrays (int *ind_p=NULL)
- Auto allocation for string arrays (char **foo_pp=NULL)
- ECPGfree_auto_mem fixed
- All function names with external linkage are now prefixed by ECPG
- Fixes for arrays of structures (Michael)

E.86.3.20. Misc. Interfaces

- Python fix fetchone() (Gerhard Haring)
- Use UTF, Unicode in Tcl where appropriate (Vsevolod Lobko, Reinhard Max)
- Add Tcl COPY TO/FROM (ljb)
- Prevent output of default index op class in pg_dump (Tom)
- Fix libpgeasy memory leak (Bruce)

E.86.3.21. Build and Install

- Configure, dynamic loader, and shared library fixes (Peter E)
- Fixes in QNX 4 port (Bernd Tegge)
- Fixes in Cygwin and Windows ports (Jason Tishler, Gerhard Haring, Dmitry Yurtaev, Darko Prenosil, Mikhail Terekhov)
- Fix for Windows socket communication failures (Magnus, Mikhail Terekhov)
- Hurd compile fix (Oliver Elphick)
- BeOS fixes (Cyril Velter)
- Remove configure --enable-unicode-conversion, now enabled by multibyte (Tatsuo)
- AIX fixes (Tatsuo, Andreas)
- Fix parallel make (Peter E)
- Install SQL language manual pages into OS-specific directories (Peter E)
- Rename config.h to pg_config.h (Peter E)
- Reorganize installation layout of header files (Peter E)

E.86.3.22. Source Code

- Remove SEP_CHAR (Bruce)

- New GUC hooks (Tom)
- Merge GUC and command line handling (Marko Kreen)
- Remove EXTEND INDEX (Martijn van Oosterhout, Tom)
- New pgjindent utility to indent java code (Bruce)
- Remove define of true/false when compiling under C++ (Leandro Fanzone, Tom)
- pgindent fixes (Bruce, Tom)
- Replace strcasecmp() with strcmp() where appropriate (Peter E)
- Dynahash portability improvements (Tom)
- Add 'volatile' usage in spinlock structures
- Improve signal handling logic (Tom)

E.86.3.23. Contrib

- New contrib/rtree_gist (Oleg Bartunov, Teodor Sigaev)
- New contrib/tsearch full-text indexing (Oleg, Teodor Sigaev)
- Add contrib/dblink for remote database access (Joe Conway)
- contrib/ora2pg Oracle conversion utility (Gilles Darold)
- contrib/xml XML conversion utility (John Gray)
- contrib/fulltextindex fixes (Christopher Kings-Lynne)
- New contrib/fuzzystrmatch with levenshtein and metaphone, soundex merged (Joe Conway)
- Add contrib/intarray boolean queries, binary search, fixes (Oleg Bartunov)
- New pg_upgrade utility (Bruce)
- Add new pg_resetxlog options (Bruce, Tom)

E.87. Release 7.1.3

Release date: 2001-08-15

E.87.1. Migration to Version 7.1.3

A dump/restore is *not* required for those running 7.1.X.

E.87.2. Changes

```
Remove unused WAL segments of large transactions (Tom)
Multiaction rule fix (Tom)
PL/pgSQL memory allocation fix (Jan)
VACUUM buffer fix (Tom)
Regression test fixes (Tom)
pg_dump fixes for GRANT/REVOKE/comments on views, user-defined types (Tom)
Fix subselects with DISTINCT ON or LIMIT (Tom)
BeOS fix
Disable COPY TO/FROM a view (Tom)
Cygwin build (Jason Tishler)
```

E.88. Release 7.1.2

Release date: 2001-05-11

This has one fix from 7.1.1.

E.88.1. Migration to Version 7.1.2

A dump/restore is *not* required for those running 7.1.X.

E.88.2. Changes

```
Fix PL/pgSQL SELECTs when returning no rows
Fix for psql backslash core dump
Referential integrity privilege fix
Optimizer fixes
pg_dump cleanups
```

E.89. Release 7.1.1

Release date: 2001-05-05

This has a variety of fixes from 7.1.

E.89.1. Migration to Version 7.1.1

A dump/restore is *not* required for those running 7.1.

E.89.2. Changes

```
Fix for numeric MODULO operator (Tom)
pg_dump fixes (Philip)
pg_dump can dump 7.0 databases (Philip)
readline 4.2 fixes (Peter E)
JOIN fixes (Tom)
AIX, MSWIN, VAX, N32K fixes (Tom)
Multibytes fixes (Tom)
Unicode fixes (Tatsuo)
Optimizer improvements (Tom)
Fix for whole rows in functions (Tom)
Fix for pg_ctl and option strings with spaces (Peter E)
ODBC fixes (Hiroshi)
EXTRACT can now take string argument (Thomas)
Python fixes (Darcy)
```

E.90. Release 7.1

Release date: 2001-04-13

This release focuses on removing limitations that have existed in the PostgreSQL code for many years.

Major changes in this release:

Write-ahead Log (WAL)

To maintain database consistency in case of an operating system crash, previous releases of PostgreSQL have forced all data modifications to disk before each transaction commit. With WAL, only one log file must be flushed to disk, greatly improving performance. If you have been using -F in previous releases to disable disk flushes, you might want to consider discontinuing its use.

TOAST

TOAST - Previous releases had a compiled-in row length limit, typically 8k - 32k. This limit made storage of long text fields difficult. With TOAST, long rows of any length can be stored with good performance.

Outer Joins

We now support outer joins. The UNION/NOT IN workaround for outer joins is no longer required. We use the SQL92 outer join syntax.

Function Manager

The previous C function manager did not handle null values properly, nor did it support 64-bit CPU's (Alpha). The new function manager does. You can continue using your old custom functions, but you might want to rewrite them in the future to use the new function manager call interface.

Complex Queries

A large number of complex queries that were unsupported in previous releases now work. Many combinations of views, aggregates, UNION, LIMIT, cursors, subqueries, and inherited tables now work properly. Inherited tables are now accessed by default. Subqueries in FROM are now supported.

E.90.1. Migration to Version 7.1

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release.

E.90.2. Changes

```
Bug Fixes
---------
Many multibyte/Unicode/locale fixes (Tatsuo and others)
More reliable ALTER TABLE RENAME (Tom)
Kerberos V fixes (David Wragg)
Fix for INSERT INTO...SELECT where targetlist has subqueries (Tom)
Prompt username/password on standard error (Bruce)
Large objects inv_read/inv_write fixes (Tom)
Fixes for to_char(), to_date(), to_ascii(), and to_timestamp() (Karel,
   Daniel Baldoni)
Prevent query expressions from leaking memory (Tom)
Allow UPDATE of arrays elements (Tom)
Wake up lock waiters during cancel (Hiroshi)
Fix rare cursor crash when using hash join (Tom)
Fix for DROP TABLE/INDEX in rolled-back transaction (Hiroshi)
Fix psql crash from \l+ if MULTIBYTE enabled (Peter E)
Fix truncation of rule names during CREATE VIEW (Ross Reedstrom)
Fix PL/perl (Alex Kapranoff)
Disallow LOCK on views (Mark Hollomon)
Disallow INSERT/UPDATE/DELETE on views (Mark Hollomon)
Disallow DROP RULE, CREATE INDEX, TRUNCATE on views (Mark Hollomon)
Allow PL/pgSQL accept non-ASCII identifiers (Tatsuo)
Allow views to proper handle GROUP BY, aggregates, DISTINCT (Tom)
```

```
Fix rare failure with TRUNCATE command (Tom)
Allow UNION/INTERSECT/EXCEPT to be used with ALL, subqueries, views,
   DISTINCT, ORDER BY, SELECT...INTO (Tom)
Fix parser failures during aborted transactions (Tom)
Allow temporary relations to properly clean up indexes (Bruce)
Fix VACUUM problem with moving rows in same page (Tom)
Modify pg_dump to better handle user-defined items in template1 (Philip)
Allow LIMIT in VIEW (Tom)
Require cursor FETCH to honor LIMIT (Tom)
Allow PRIMARY/FOREIGN Key definitions on inherited columns (Stephan)
Allow ORDER BY, LIMIT in subqueries (Tom)
Allow UNION in CREATE RULE (Tom)
Make ALTER/DROP TABLE rollback-able (Vadim, Tom)
Store initdb collation in pg_control so collation cannot be changed (Tom)
Fix INSERT...SELECT with rules (Tom)
Fix FOR UPDATE inside views and subselects (Tom)
Fix OVERLAPS operators conform to SQL92 spec regarding NULLs (Tom)
Fix lpad() and rpad() to handle length less than input string (Tom)
Fix use of NOTIFY in some rules (Tom)
Overhaul btree code (Tom)
Fix NOT NULL use in Pl/pgSQL variables (Tom)
Overhaul GIST code (Oleg)
Fix CLUSTER to preserve constraints and column default (Tom)
Improved deadlock detection handling (Tom)
Allow multiple SERIAL columns in a table (Tom)
Prevent occasional index corruption (Vadim)

Enhancements
------------
Add OUTER JOINs (Tom)
Function manager overhaul (Tom)
Allow ALTER TABLE RENAME on indexes (Tom)
Improve CLUSTER (Tom)
Improve ps status display for more platforms (Peter E, Marc)
Improve CREATE FUNCTION failure message (Ross)
JDBC improvements (Peter, Travis Bauer, Christopher Cain, William Webber,
   Gunnar)
Grand Unified Configuration scheme/GUC.  Many options can now be set in
   data/postgresql.conf, postmaster/postgres flags, or SET commands (Peter E)
Improved handling of file descriptor cache (Tom)
New warning code about auto-created table alias entries (Bruce)
Overhaul initdb process (Tom, Peter E)
Overhaul of inherited tables; inherited tables now accessed by default;
  new ONLY key word prevents it (Chris Bitmead, Tom)
ODBC cleanups/improvements (Nick Gorham, Stephan Szabo, Zoltan Kovacs,
   Michael Fork)
Allow renaming of temp tables (Tom)
Overhaul memory manager contexts (Tom)
pg_dumpall uses CREATE USER or CREATE GROUP rather using COPY (Peter E)
Overhaul pg_dump (Philip Warner)
Allow pg_hba.conf secondary password file to specify only username (Peter E)
Allow TEMPORARY or TEMP key word when creating temporary tables (Bruce)
New memory leak checker (Karel)
```

```
New SET SESSION CHARACTERISTICS (Thomas)
Allow nested block comments (Thomas)
Add WITHOUT TIME ZONE type qualifier (Thomas)
New ALTER TABLE ADD CONSTRAINT (Stephan)
Use NUMERIC accumulators for INTEGER aggregates (Tom)
Overhaul aggregate code (Tom)
New VARIANCE and STDDEV() aggregates
Improve dependency ordering of pg_dump (Philip)
New pg_restore command (Philip)
New pg_dump tar output option (Philip)
New pg_dump of large objects  (Philip)
New ESCAPE option to LIKE (Thomas)
New case-insensitive LIKE - ILIKE (Thomas)
Allow functional indexes to use binary-compatible type (Tom)
Allow SQL functions to be used in more contexts (Tom)
New pg_config utility (Peter E)
New PL/pgSQL EXECUTE command which allows dynamic SQL and utility statements
   (Jan)
New PL/pgSQL GET DIAGNOSTICS statement for SPI value access (Jan)
New quote_identifiers() and quote_literal() functions (Jan)
New ALTER TABLE table OWNER TO user command (Mark Hollomon)
Allow subselects in FROM, i.e. FROM (SELECT ...) [AS] alias (Tom)
Update PyGreSQL to version 3.1 (D'Arcy)
Store tables as files named by OID (Vadim)
New SQL function setval(seq,val,bool) for use in pg_dump (Philip)
Require DROP VIEW to remove views, no DROP TABLE (Mark)
Allow DROP VIEW view1, view2 (Mark)
Allow multiple objects in DROP INDEX, DROP RULE, and DROP TYPE (Tom)
Allow automatic conversion to/from Unicode (Tatsuo, Eiji)
New /contrib/pgcrypto hashing functions (Marko Kreen)
New pg_dumpall --globals-only option (Peter E)
New CHECKPOINT command for WAL which creates new WAL log file (Vadim)
New AT TIME ZONE syntax (Thomas)
Allow location of Unix domain socket to be configurable (David J. MacKenzie)
Allow postmaster to listen on a specific IP address (David J. MacKenzie)
Allow socket path name to be specified in hostname by using leading slash
   (David J. MacKenzie)
Allow CREATE DATABASE to specify template database (Tom)
New utility to convert MySQL schema dumps to SQL92 and PostgreSQL (Thomas)
New /contrib/rserv replication toolkit (Vadim)
New file format for COPY BINARY (Tom)
New /contrib/oid2name to map numeric files to table names (B Palmer)
New "idle in transaction" ps status message (Marc)
Update to pgaccess 0.98.7 (Constantin Teodorescu)
pg_ctl now defaults to -w (wait) on shutdown, new -l (log) option
Add rudimentary dependency checking to pg_dump (Philip)

Types
-----
Fix INET/CIDR type ordering and add new functions (Tom)
Make OID behave as an unsigned type (Tom)
Allow BIGINT as synonym for INT8 (Peter E)
New int2 and int8 comparison operators (Tom)
```

```
New BIT and BIT VARYING types (Adriaan Joubert, Tom, Peter E)
CHAR() no longer faster than VARCHAR() because of TOAST (Tom)
New GIST seg/cube examples (Gene Selkov)
Improved round(numeric) handling (Tom)
Fix CIDR output formatting (Tom)
New CIDR abbrev() function (Tom)

Performance
-----------
Write-Ahead Log (WAL) to provide crash recovery with less performance
   overhead (Vadim)
ANALYZE stage of VACUUM no longer exclusively locks table (Bruce)
Reduced file seeks (Denis Perchine)
Improve BTREE code for duplicate keys (Tom)
Store all large objects in a single table (Denis Perchine, Tom)
Improve memory allocation performance (Karel, Tom)

Source Code
-----------
New function manager call conventions (Tom)
SGI portability fixes (David Kaelbling)
New configure --enable-syslog option (Peter E)
New BSDI README (Bruce)
configure script moved to top level, not /src (Peter E)
Makefile/configuration/compilation overhaul (Peter E)
New configure --with-python option (Peter E)
Solaris cleanups (Peter E)
Overhaul /contrib Makefiles (Karel)
New OpenSSL configuration option (Magnus, Peter E)
AIX fixes (Andreas)
QNX fixes (Maurizio)
New heap_open(), heap_openr() API (Tom)
Remove colon and semi-colon operators (Thomas)
New pg_class.relkind value for views (Mark Hollomon)
Rename ichar() to chr() (Karel)
New documentation for btrim(), ascii(), chr(), repeat() (Karel)
Fixes for NT/Cygwin (Pete Forman)
AIX port fixes (Andreas)
New BeOS port (David Reid, Cyril Velter)
Add proofreader's changes to docs (Addison-Wesley, Bruce)
New Alpha spinlock code (Adriaan Joubert, Compaq)
UnixWare port overhaul (Peter E)
New Darwin/MacOS X port (Peter Bierman, Bruce Hartzler)
New FreeBSD Alpha port (Alfred)
Overhaul shared memory segments (Tom)
Add IBM S/390 support (Neale Ferguson)
Moved macmanuf to /contrib (Larry Rosenman)
Syslog improvements (Larry Rosenman)
New template0 database that contains no user additions (Tom)
New /contrib/cube and /contrib/seg GIST sample code (Gene Selkov)
Allow NetBSD's libedit instead of readline (Peter)
Improved assembly language source code format (Bruce)
New contrib/pg_logger
```

```
New --template option to createdb
New contrib/pg_control utility (Oliver)
New FreeBSD tools ipc_check, start-scripts/freebsd
```

E.91. Release 7.0.3

Release date: 2000-11-11

This has a variety of fixes from 7.0.2.

E.91.1. Migration to Version 7.0.3

A dump/restore is *not* required for those running 7.0.*.

E.91.2. Changes

```
Jdbc fixes (Peter)
Large object fix (Tom)
Fix lean in COPY WITH OIDS leak (Tom)
Fix backwards-index-scan (Tom)
Fix SELECT ... FOR UPDATE so it checks for duplicate keys (Hiroshi)
Add --enable-syslog to configure (Marc)
Fix abort transaction at backend exit in rare cases (Tom)
Fix for psql \l+ when multibyte enabled (Tatsuo)
Allow PL/pgSQL to accept non ascii identifiers (Tatsuo)
Make vacuum always flush buffers (Tom)
Fix to allow cancel while waiting for a lock (Hiroshi)
Fix for memory allocation problem in user authentication code (Tom)
Remove bogus use of int4out() (Tom)
Fixes for multiple subqueries in COALESCE or BETWEEN (Tom)
Fix for failure of triggers on heap open in certain cases (Jeroen van
   Vianen)
Fix for erroneous selectivity of not-equals (Tom)
Fix for erroneous use of strcmp() (Tom)
Fix for bug where storage manager accesses items beyond end of file
   (Tom)
Fix to include kernel errno message in all smgr elog messages (Tom)
Fix for '.' not in PATH at build time (SL Baur)
Fix for out-of-file-descriptors error (Tom)
Fix to make pg_dump dump 'iscachable' flag for functions (Tom)
Fix for subselect in targetlist of Append node (Tom)
```

```
Fix for mergejoin plans (Tom)
Fix TRUNCATE failure on relations with indexes (Tom)
Avoid database-wide restart on write error (Hiroshi)
Fix nodeMaterial to honor chgParam by recomputing its output (Tom)
Fix VACUUM problem with moving chain of update row versions when source
   and destination of a row version lie on the same page (Tom)
Fix user.c CommandCounterIncrement (Tom)
Fix for AM/PM boundary problem in to_char() (Karel Zak)
Fix TIME aggregate handling (Tom)
Fix to_char() to avoid coredump on NULL input (Tom)
Buffer fix (Tom)
Fix for inserting/copying longer multibyte strings into char() data
   types (Tatsuo)
Fix for crash of backend, on abort (Tom)
```

E.92. Release 7.0.2

Release date: 2000-06-05

This is a repackaging of 7.0.1 with added documentation.

E.92.1. Migration to Version 7.0.2

A dump/restore is *not* required for those running 7.*.

E.92.2. Changes

```
Added documentation to tarball.
```

E.93. Release 7.0.1

Release date: 2000-06-01

This is a cleanup release for 7.0.

E.93.1. Migration to Version 7.0.1

A dump/restore is *not* required for those running 7.0.

E.93.2. Changes

```
Fix many CLUSTER failures (Tom)
Allow ALTER TABLE RENAME works on indexes (Tom)
Fix plpgsql to handle datetime->timestamp and timespan->interval (Bruce)
New configure --with-setproctitle switch to use setproctitle() (Marc, Bruce)
Fix the off by one errors in ResultSet from 6.5.3, and more.
jdbc ResultSet fixes (Joseph Shraibman)
optimizer tunings (Tom)
Fix create user for pgaccess
Fix for UNLISTEN failure
IRIX fixes (David Kaelbling)
QNX fixes (Andreas Kardos)
Reduce COPY IN lock level (Tom)
Change libpqeasy to use PQconnectdb() style parameters (Bruce)
Fix pg_dump to handle OID indexes (Tom)
Fix small memory leak (Tom)
Solaris fix for createdb/dropdb (Tatsuo)
Fix for non-blocking connections (Alfred Perlstein)
Fix improper recovery after RENAME TABLE failures (Tom)
Copy pg_ident.conf.sample into /lib directory in install (Bruce)
Add SJIS UDC (NEC selection IBM kanji) support (Eiji Tokuya)
Fix too long syslog message (Tatsuo)
Fix problem with quoted indexes that are too long (Tom)
JDBC ResultSet.getTimestamp() fix (Gregory Krasnow & Floyd Marinescu)
ecpg changes (Michael)
```

E.94. Release 7.0

Release date: 2000-05-08

This release contains improvements in many areas, demonstrating the continued growth of PostgreSQL. There are more improvements and fixes in 7.0 than in any previous release. The developers have confidence that this is the best release yet; we do our best to put out only solid releases, and this one is no exception.

Major changes in this release:

Foreign Keys

Foreign keys are now implemented, with the exception of PARTIAL MATCH foreign keys. Many users have been asking for this feature, and we are pleased to offer it.

Optimizer Overhaul

Continuing on work started a year ago, the optimizer has been improved, allowing better query plan selection and faster performance with less memory usage.

Updated psql

psql, our interactive terminal monitor, has been updated with a variety of new features. See the psql manual page for details.

Join Syntax

SQL92 join syntax is now supported, though only as `INNER JOIN` for this release. `JOIN`, `NATURAL JOIN`, `JOIN/USING`, and `JOIN/ON` are available, as are column correlation names.

E.94.1. Migration to Version 7.0

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release of PostgreSQL. For those upgrading from 6.5.*, you can instead use pg_upgrade to upgrade to this release; however, a full dump/reload installation is always the most robust method for upgrades.

Interface and compatibility issues to consider for the new release include:

- The date/time types `datetime` and `timespan` have been superseded by the SQL92-defined types `timestamp` and `interval`. Although there has been some effort to ease the transition by allowing PostgreSQL to recognize the deprecated type names and translate them to the new type names, this mechanism cannot be completely transparent to your existing application.
- The optimizer has been substantially improved in the area of query cost estimation. In some cases, this will result in decreased query times as the optimizer makes a better choice for the preferred plan. However, in a small number of cases, usually involving pathological distributions of data, your query times might go up. If you are dealing with large amounts of data, you might want to check your queries to verify performance.
- The JDBC and ODBC interfaces have been upgraded and extended.
- The string function `CHAR_LENGTH` is now a native function. Previous versions translated this into a call to `LENGTH`, which could result in ambiguity with other types implementing `LENGTH` such as the geometric types.

E.94.2. Changes

```
Bug Fixes
---------
Prevent function calls exceeding maximum number of arguments (Tom)
Improve CASE construct (Tom)
```

```
Fix SELECT coalesce(f1,0) FROM int4_tbl GROUP BY f1 (Tom)
Fix SELECT sentence.words[0] FROM sentence GROUP BY sentence.words[0] (Tom)
Fix GROUP BY scan bug (Tom)
Improvements in SQL grammar processing (Tom)
Fix for views involved in INSERT ... SELECT ... (Tom)
Fix for SELECT a/2, a/2 FROM test_missing_target GROUP BY a/2 (Tom)
Fix for subselects in INSERT ... SELECT (Tom)
Prevent INSERT ... SELECT ... ORDER BY (Tom)
Fixes for relations greater than 2GB, including vacuum
Improve propagating system table changes to other backends (Tom)
Improve propagating user table changes to other backends (Tom)
Fix handling of temp tables in complex situations (Bruce, Tom)
Allow table locking at table open, improving concurrent reliability (Tom)
Properly quote sequence names in pg_dump (Ross J. Reedstrom)
Prevent DROP DATABASE while others accessing
Prevent any rows from being returned by GROUP BY if no rows processed (Tom)
Fix SELECT COUNT(1) FROM table WHERE ...' if no rows matching WHERE (Tom)
Fix pg_upgrade so it works for MVCC (Tom)
Fix for SELECT ... WHERE x IN (SELECT ... HAVING SUM(x) > 1) (Tom)
Fix for "f1 datetime DEFAULT 'now'"  (Tom)
Fix problems with CURRENT_DATE used in DEFAULT (Tom)
Allow comment-only lines, and ;;; lines too. (Tom)
Improve recovery after failed disk writes, disk full (Hiroshi)
Fix cases where table is mentioned in FROM but not joined (Tom)
Allow HAVING clause without aggregate functions (Tom)
Fix for "--" comment and no trailing newline, as seen in perl interface
Improve pg_dump failure error reports (Bruce)
Allow sorts and hashes to exceed 2GB file sizes (Tom)
Fix for pg_dump dumping of inherited rules (Tom)
Fix for NULL handling comparisons (Tom)
Fix inconsistent state caused by failed CREATE/DROP commands (Hiroshi)
Fix for dbname with dash
Prevent DROP INDEX from interfering with other backends (Tom)
Fix file descriptor leak in verify_password()
Fix for "Unable to identify an operator =$" problem
Fix ODBC so no segfault if CommLog and Debug enabled (Dirk Niggemann)
Fix for recursive exit call (Massimo)
Fix for extra-long timezones (Jeroen van Vianen)
Make pg_dump preserve primary key information (Peter E)
Prevent databases with single quotes (Peter E)
Prevent DROP DATABASE inside  transaction (Peter E)
ecpg memory leak fixes (Stephen Birch)
Fix for SELECT null::text, SELECT int4fac(null) and SELECT 2 + (null) (Tom)
Y2K timestamp fix (Massimo)
Fix for VACUUM 'HEAP_MOVED_IN was not expected' errors (Tom)
Fix for views with tables/columns containing spaces  (Tom)
Prevent privileges on indexes (Peter E)
Fix for spinlock stuck problem when error is generated (Hiroshi)
Fix ipcclean on Linux
Fix handling of NULL constraint conditions (Tom)
Fix memory leak in odbc driver (Nick Gorham)
Fix for privilege check on UNION tables (Tom)
Fix to allow SELECT 'a' LIKE 'a' (Tom)
```

```
Fix for SELECT 1 + NULL (Tom)
Fixes to CHAR
Fix log() on numeric type (Tom)
Deprecate ':' and ';' operators
Allow vacuum of temporary tables
Disallow inherited columns with the same name as new columns
Recover or force failure when disk space is exhausted (Hiroshi)
Fix INSERT INTO ... SELECT with AS columns matching result columns
Fix INSERT ... SELECT ... GROUP BY groups by target columns not source columns (Tom)
Fix CREATE TABLE test (a char(5) DEFAULT text '', b int4) with INSERT (Tom)
Fix UNION with LIMIT
Fix CREATE TABLE x AS SELECT 1 UNION SELECT 2
Fix CREATE TABLE test(col char(2) DEFAULT user)
Fix mismatched types in CREATE TABLE ... DEFAULT
Fix SELECT * FROM pg_class where oid in (0,-1)
Fix SELECT COUNT('asdf') FROM pg_class WHERE oid=12
Prevent user who can create databases can modifying pg_database table (Peter E)
Fix btree to give a useful elog when key > 1/2 (page - overhead) (Tom)
Fix INSERT of 0.0 into DECIMAL(4,4) field (Tom)

Enhancements
------------
New CLI interface include file sqlcli.h, based on SQL3/SQL98
Remove all limits on query length, row length limit still exists (Tom)
Update jdbc protocol to 2.0 (Jens Glaser <jens@jens.de>)
Add TRUNCATE command to quickly truncate relation (Mike Mascari)
Fix to give super user and createdb user proper update catalog rights (Peter E)
Allow ecpg bool variables to have NULL values (Christof)
Issue ecpg error if NULL value for variable with no NULL indicator (Christof)
Allow ^C to cancel COPY command (Massimo)
Add SET FSYNC and SHOW PG_OPTIONS commands(Massimo)
Function name overloading for dynamically-loaded C functions (Frankpitt)
Add CmdTuples() to libpq++(Vince)
New CREATE CONSTRAINT TRIGGER and SET CONSTRAINTS commands(Jan)
Allow CREATE FUNCTION/WITH clause to be used for all language types
configure --enable-debug adds -g (Peter E)
configure --disable-debug removes -g (Peter E)
Allow more complex default expressions (Tom)
First real FOREIGN KEY constraint trigger functionality (Jan)
Add FOREIGN KEY ... MATCH FULL ... ON DELETE CASCADE (Jan)
Add FOREIGN KEY ... MATCH <unspecified> referential actions (Don Baccus)
Allow WHERE restriction on ctid (physical heap location) (Hiroshi)
Move pginterface from contrib to interface directory, rename to pgeasy (Bruce)
Change pgeasy connectdb() parameter ordering (Bruce)
Require SELECT DISTINCT target list to have all ORDER BY columns (Tom)
Add Oracle's COMMENT ON command (Mike Mascari <mascarim@yahoo.com>)
libpq's PQsetNoticeProcessor function now returns previous hook(Peter E)
Prevent PQsetNoticeProcessor from being set to NULL (Peter E)
Make USING in COPY optional (Bruce)
Allow subselects in the target list (Tom)
Allow subselects on the left side of comparison operators (Tom)
New parallel regression test (Jan)
Change backend-side COPY to write files with permissions 644 not 666 (Tom)
```

```
Force permissions on PGDATA directory to be secure, even if it exists (Tom)
Added psql LASTOID variable to return last inserted oid (Peter E)
Allow concurrent vacuum and remove pg_vlock vacuum lock file (Tom)
Add privilege check for vacuum (Peter E)
New libpq functions to allow asynchronous connections: PQconnectStart(),
  PQconnectPoll(), PQresetStart(), PQresetPoll(), PQsetenvStart(),
  PQsetenvPoll(), PQsetenvAbort (Ewan Mellor)
New libpq PQsetenv() function (Ewan Mellor)
create/alter user extension (Peter E)
New postmaster.pid and postmaster.opts under $PGDATA (Tatsuo)
New scripts for create/drop user/db (Peter E)
Major psql overhaul (Peter E)
Add const to libpq interface (Peter E)
New libpq function PQoidValue (Peter E)
Show specific non-aggregate causing problem with GROUP BY (Tom)
Make changes to pg_shadow recreate pg_pwd file (Peter E)
Add aggregate(DISTINCT ...) (Tom)
Allow flag to control COPY input/output of NULLs (Peter E)
Make postgres user have a password by default (Peter E)
Add CREATE/ALTER/DROP GROUP (Peter E)
All administration scripts now support --long options (Peter E, Karel)
Vacuumdb script now supports --all option (Peter E)
ecpg new portable FETCH syntax
Add ecpg EXEC SQL IFDEF, EXEC SQL IFNDEF, EXEC SQL ELSE, EXEC SQL ELIF
       and EXEC SQL ENDIF directives
Add pg_ctl script to control backend start-up (Tatsuo)
Add postmaster.opts.default file to store start-up flags (Tatsuo)
Allow --with-mb=SQL_ASCII
Increase maximum number of index keys to 16 (Bruce)
Increase maximum number of function arguments to 16 (Bruce)
Allow configuration of maximum number of index keys and arguments (Bruce)
Allow unprivileged users to change their passwords (Peter E)
Password authentication enabled; required for new users (Peter E)
Disallow dropping a user who owns a database (Peter E)
Change initdb option --with-mb to --enable-multibyte
Add option for initdb to prompts for superuser password (Peter E)
Allow complex type casts like col::numeric(9,2) and col::int2::float8 (Tom)
Updated user interfaces on initdb, initlocation, pg_dump, ipcclean (Peter E)
New pg_char_to_encoding() and pg_encoding_to_char() functions (Tatsuo)
libpq non-blocking mode (Alfred Perlstein)
Improve conversion of types in casts that don't specify a length
New plperl internal programming language (Mark Hollomon)
Allow COPY IN to read file that do not end with a newline (Tom)
Indicate when long identifiers are truncated (Tom)
Allow aggregates to use type equivalency (Peter E)
Add Oracle's to_char(), to_date(), to_datetime(), to_timestamp(), to_number()
       conversion functions (Karel Zak <zakkr@zf.jcu.cz>)
Add SELECT DISTINCT ON (expr [, expr ...]) targetlist ... (Tom)
Check to be sure ORDER BY is compatible with the DISTINCT operation (Tom)
Add NUMERIC and int8 types to ODBC
Improve EXPLAIN results for Append, Group, Agg, Unique (Tom)
Add ALTER TABLE ... ADD FOREIGN KEY (Stephan Szabo)
Allow SELECT .. FOR UPDATE in PL/pgSQL (Hiroshi)
```

```
Enable backward sequential scan even after reaching EOF (Hiroshi)
Add btree indexing of boolean values, >= and <= (Don Baccus)
Print current line number when COPY FROM fails (Massimo)
Recognize POSIX time zone e.g. "PST+8" and "GMT-8" (Thomas)
Add DEC as synonym for DECIMAL (Thomas)
Add SESSION_USER as SQL92 key word, same as CURRENT_USER (Thomas)
Implement SQL92 column aliases (aka correlation names) (Thomas)
Implement SQL92 join syntax (Thomas)
Make INTERVAL reserved word allowed as a column identifier (Thomas)
Implement REINDEX command (Hiroshi)
Accept ALL in aggregate function SUM(ALL col) (Tom)
Prevent GROUP BY from using column aliases (Tom)
New psql \encoding option (Tatsuo)
Allow PQrequestCancel() to terminate when in waiting-for-lock state (Hiroshi)
Allow negation of a negative number in all cases
Add ecpg descriptors (Christof, Michael)
Allow CREATE VIEW v AS SELECT f1::char(8) FROM tbl
Allow casts with length, like foo::char(8)
New libpq functions PQsetClientEncoding(), PQclientEncoding() (Tatsuo)
Add support for SJIS user defined characters (Tatsuo)
Larger views/rules supported
Make libpq's PQconndefaults() thread-safe (Tom)
Disable // as comment to be ANSI conforming, should use -- (Tom)
Allow column aliases on views CREATE VIEW name (collist)
Fixes for views with subqueries (Tom)
Allow UPDATE table SET fld = (SELECT ...) (Tom)
SET command options no longer require quotes
Update pgaccess to 0.98.6
New SET SEED command
New pg_options.sample file
New SET FSYNC command (Massimo)
Allow pg_descriptions when creating tables
Allow pg_descriptions when creating types, columns, and functions
Allow psql \copy to allow delimiters (Peter E)
Allow psql to print nulls as distinct from "" [null] (Peter E)

Types
-----
Many array fixes (Tom)
Allow bare column names to be subscripted as arrays (Tom)
Improve type casting of int and float constants (Tom)
Cleanups for int8 inputs, range checking, and type conversion (Tom)
Fix for SELECT timespan('21:11:26'::time) (Tom)
netmask('x.x.x.x/0') is 255.255.255.255 instead of 0.0.0.0 (Oleg Sharoiko)
Add btree index on NUMERIC (Jan)
Perl fix for large objects containing NUL characters (Douglas Thomson)
ODBC fix for large objects (free)
Fix indexing of cidr data type
Fix for Ethernet MAC addresses (macaddr type) comparisons
Fix for date/time types when overflows happened in computations (Tom)
Allow array on int8 (Peter E)
Fix for rounding/overflow of NUMERIC type, like NUMERIC(4,4) (Tom)
Allow NUMERIC arrays
```

```
Fix bugs in NUMERIC ceil() and floor() functions (Tom)
Make char_length()/octet_length including trailing blanks (Tom)
Made abstime/reltime use int4 instead of time_t (Peter E)
New lztext data type for compressed text fields
Revise code to handle coercion of int and float constants (Tom)
Start at new code to implement a BIT and BIT VARYING type (Adriaan Joubert)
NUMERIC now accepts scientific notation (Tom)
NUMERIC to int4 rounds (Tom)
Convert float4/8 to NUMERIC properly (Tom)
Allow type conversion with NUMERIC (Thomas)
Make ISO date style (2000-02-16 09:33) the default (Thomas)
Add NATIONAL CHAR [ VARYING ] (Thomas)
Allow NUMERIC round and trunc to accept negative scales (Tom)
New TIME WITH TIME ZONE type (Thomas)
Add MAX()/MIN() on time type (Thomas)
Add abs(), mod(), fac() for int8 (Thomas)
Rename functions to round(), sqrt(), cbrt(), pow() for float8 (Thomas)
Add transcendental math functions (e.g. sin(), acos()) for float8 (Thomas)
Add exp() and ln() for NUMERIC type
Rename NUMERIC power() to pow() (Thomas)
Improved TRANSLATE() function (Edwin Ramirez, Tom)
Allow X=-Y operators  (Tom)
Allow SELECT float8(COUNT(*))/(SELECT COUNT(*) FROM t) FROM t GROUP BY f1; (Tom)
Allow LOCALE to use indexes in regular expression searches (Tom)
Allow creation of functional indexes to use default types

Performance
-----------
Prevent exponential space consumption with many AND's and OR's (Tom)
Collect attribute selectivity values for system columns (Tom)
Reduce memory usage of aggregates (Tom)
Fix for LIKE optimization to use indexes with multibyte encodings (Tom)
Fix r-tree index optimizer selectivity (Thomas)
Improve optimizer selectivity computations and functions (Tom)
Optimize btree searching for cases where many equal keys exist (Tom)
Enable fast LIKE index processing only if index present (Tom)
Re-use free space on index pages with duplicates (Tom)
Improve hash join processing (Tom)
Prevent descending sort if result is already sorted(Hiroshi)
Allow commuting of index scan query qualifications (Tom)
Prefer index scans in cases where ORDER BY/GROUP BY is required (Tom)
Allocate large memory requests in fix-sized chunks for performance (Tom)
Fix vacuum's performance by reducing memory allocation requests (Tom)
Implement constant-expression simplification (Bernard Frankpitt, Tom)
Use secondary columns to be used to determine start of index scan (Hiroshi)
Prevent quadruple use of disk space when doing internal sorting (Tom)
Faster sorting by calling fewer functions (Tom)
Create system indexes to match all system caches (Bruce, Hiroshi)
Make system caches use system indexes (Bruce)
Make all system indexes unique (Bruce)
Improve pg_statistics management for VACUUM speed improvement (Tom)
Flush backend cache less frequently (Tom, Hiroshi)
COPY now reuses previous memory allocation, improving performance (Tom)
```

```
Improve optimization cost estimation (Tom)
Improve optimizer estimate of range queries x > lowbound AND x < highbound (Tom)
Use DNF instead of CNF where appropriate (Tom, Taral)
Further cleanup for OR-of-AND WHERE-clauses (Tom)
Make use of index in OR clauses (x = 1 AND y = 2) OR (x = 2 AND y = 4) (Tom)
Smarter optimizer computations for random index page access (Tom)
New SET variable to control optimizer costs (Tom)
Optimizer queries based on LIMIT, OFFSET, and EXISTS qualifications (Tom)
Reduce optimizer internal housekeeping of join paths for speedup (Tom)
Major subquery speedup (Tom)
Fewer fsync writes when fsync is not disabled (Tom)
Improved LIKE optimizer estimates (Tom)
Prevent fsync in SELECT-only queries (Vadim)
Make index creation use psort code, because it is now faster (Tom)
Allow creation of sort temp tables > 1 Gig

Source Tree Changes
-------------------
Fix for linux PPC compile
New generic expression-tree-walker subroutine (Tom)
Change form() to varargform() to prevent portability problems
Improved range checking for large integers on Alphas
Clean up #include in /include directory (Bruce)
Add scripts for checking includes (Bruce)
Remove un-needed #include's from *.c files (Bruce)
Change #include's to use <> and "" as appropriate (Bruce)
Enable Windows compilation of libpq
Alpha spinlock fix from Uncle George <gatgul@voicenet.com>
Overhaul of optimizer data structures (Tom)
Fix to cygipc library (Yutaka Tanida)
Allow pgsql to work on newer Cygwin snapshots (Dan)
New catalog version number (Tom)
Add Linux ARM
Rename heap_replace to heap_update
Update for QNX (Dr. Andreas Kardos)
New platform-specific regression handling (Tom)
Rename oid8 -> oidvector and int28 -> int2vector (Bruce)
Included all yacc and lex files into the distribution (Peter E.)
Remove lextest, no longer needed (Peter E)
Fix for libpq and psql on Windows (Magnus)
Internally change datetime and timespan into timestamp and interval (Thomas)
Fix for plpgsql on BSD/OS
Add SQL_ASCII test case to the regression test (Tatsuo)
configure --with-mb now deprecated (Tatsuo)
NT fixes
NetBSD fixes (Johnny C. Lam <lamj@stat.cmu.edu>)
Fixes for Alpha compiles
New multibyte encodings
```

E.95. Release 6.5.3

Release date: 1999-10-13

This is basically a cleanup release for 6.5.2. We have added a new PgAccess that was missing in 6.5.2, and installed an NT-specific fix.

E.95.1. Migration to Version 6.5.3

A dump/restore is *not* required for those running 6.5.*.

E.95.2. Changes

```
Updated version of pgaccess 0.98
NT-specific patch
Fix dumping rules on inherited tables
```

E.96. Release 6.5.2

Release date: 1999-09-15

This is basically a cleanup release for 6.5.1. We have fixed a variety of problems reported by 6.5.1 users.

E.96.1. Migration to Version 6.5.2

A dump/restore is *not* required for those running 6.5.*.

E.96.2. Changes

```
subselect+CASE fixes(Tom)
Add SHLIB_LINK setting for solaris_i386 and solaris_sparc ports(Daren Sefcik)
Fixes for CASE in WHERE join clauses(Tom)
Fix BTScan abort(Tom)
Repair the check for redundant UNIQUE and PRIMARY KEY indexes(Thomas)
Improve it so that it checks for multicolumn constraints(Thomas)
Fix for Windows making problem with MB enabled(Hiroki Kataoka)
```

```
Allow BSD yacc and bison to compile pl code(Bruce)
Fix SET NAMES working
int8 fixes(Thomas)
Fix vacuum's memory consumption(Hiroshi,Tatsuo)
Reduce the total memory consumption of vacuum(Tom)
Fix for timestamp(datetime)
Rule deparsing bugfixes(Tom)
Fix quoting problems in mkMakefile.tcldefs.sh.in and mkMakefile.tkdefs.sh.in(Tom)
This is to re-use space on index pages freed by vacuum(Vadim)
document -x for pg_dump(Bruce)
Fix for unary operators in rule deparser(Tom)
Comment out FileUnlink of excess segments during mdtruncate()(Tom)
IRIX linking fix from Yu Cao >yucao@falcon.kla-tencor.com<
Repair logic error in LIKE: should not return LIKE_ABORT
  when reach end of pattern before end of text(Tom)
Repair incorrect cleanup of heap memory allocation during transaction abort(Tom)
Updated version of pgaccess 0.98
```

E.97. Release 6.5.1

Release date: 1999-07-15

This is basically a cleanup release for 6.5. We have fixed a variety of problems reported by 6.5 users.

E.97.1. Migration to Version 6.5.1

A dump/restore is *not* required for those running 6.5.

E.97.2. Changes

```
Add NT README file
Portability fixes for linux_ppc, IRIX, linux_alpha, OpenBSD, alpha
Remove QUERY_LIMIT, use SELECT...LIMIT
Fix for EXPLAIN on inheritance(Tom)
Patch to allow vacuum on multisegment tables(Hiroshi)
R-Tree optimizer selectivity fix(Tom)
ACL file descriptor leak fix(Atsushi Ogawa)
New expression subtree code(Tom)
Avoid disk writes for read-only transactions(Vadim)
Fix for removal of temp tables if last transaction was aborted(Bruce)
Fix to prevent too large row from being created(Bruce)
```

```
plpgsql fixes
Allow port numbers 32k - 64k(Bruce)
Add ^ precedence(Bruce)
Rename sort files called pg_temp to pg_sorttemp(Bruce)
Fix for microseconds in time values(Tom)
Tutorial source cleanup
New linux_m68k port
Fix for sorting of NULL's in some cases(Tom)
Shared library dependencies fixed (Tom)
Fixed glitches affecting GROUP BY in subselects(Tom)
Fix some compiler warnings (Tomoaki Nishiyama)
Add Win1250 (Czech) support (Pavel Behal)
```

E.98. Release 6.5

Release date: 1999-06-09

This release marks a major step in the development team's mastery of the source code we inherited from Berkeley. You will see we are now easily adding major features, thanks to the increasing size and experience of our world-wide development team.

Here is a brief summary of the more notable changes:

Multiversion concurrency control(MVCC)

This removes our old table-level locking, and replaces it with a locking system that is superior to most commercial database systems. In a traditional system, each row that is modified is locked until committed, preventing reads by other users. MVCC uses the natural multiversion nature of PostgreSQL to allow readers to continue reading consistent data during writer activity. Writers continue to use the compact pg_log transaction system. This is all performed without having to allocate a lock for every row like traditional database systems. So, basically, we no longer are restricted by simple table-level locking; we have something better than row-level locking.

Hot backups from pg_dump

pg_dump takes advantage of the new MVCC features to give a consistent database dump/backup while the database stays online and available for queries.

Numeric data type

We now have a true numeric data type, with user-specified precision.

Temporary tables

Temporary tables are guaranteed to have unique names within a database session, and are destroyed on session exit.

New SQL features

We now have CASE, INTERSECT, and EXCEPT statement support. We have new LIMIT/OFFSET, SET TRANSACTION ISOLATION LEVEL, SELECT ... FOR UPDATE, and an improved LOCK TABLE command.

Speedups

We continue to speed up PostgreSQL, thanks to the variety of talents within our team. We have sped up memory allocation, optimization, table joins, and row transfer routines.

Ports

We continue to expand our port list, this time including Windows NT/ix86 and NetBSD/arm32.

Interfaces

Most interfaces have new versions, and existing functionality has been improved.

Documentation

New and updated material is present throughout the documentation. New FAQs have been contributed for SGI and AIX platforms. The *Tutorial* has introductory information on SQL from Stefan Simkovics. For the *User's Guide*, there are reference pages covering the postmaster and more utility programs, and a new appendix contains details on date/time behavior. The *Administrator's Guide* has a new chapter on troubleshooting from Tom Lane. And the *Programmer's Guide* has a description of query processing, also from Stefan, and details on obtaining the PostgreSQL source tree via anonymous CVS and CVSup.

E.98.1. Migration to Version 6.5

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release of PostgreSQL. pg_upgrade can *not* be used to upgrade to this release because the on-disk structure of the tables has changed compared to previous releases.

The new Multiversion Concurrency Control (MVCC) features can give somewhat different behaviors in multiuser environments. *Read and understand the following section to ensure that your existing applications will give you the behavior you need.*

E.98.1.1. Multiversion Concurrency Control

Because readers in 6.5 don't lock data, regardless of transaction isolation level, data read by one transaction can be overwritten by another. In other words, if a row is returned by `SELECT` it doesn't mean that this row really exists at the time it is returned (i.e. sometime after the statement or transaction began) nor that the row is protected from being deleted or updated by concurrent transactions before the current transaction does a commit or rollback.

To ensure the actual existence of a row and protect it against concurrent updates one must use `SELECT FOR UPDATE` or an appropriate `LOCK TABLE` statement. This should be taken into account when porting applications from previous releases of PostgreSQL and other environments.

Keep the above in mind if you are using `contrib/refint.*` triggers for referential integrity. Additional techniques are required now. One way is to use `LOCK parent_table IN SHARE ROW EXCLUSIVE`

`MODE` command if a transaction is going to update/delete a primary key and use `LOCK parent_table IN SHARE MODE` command if a transaction is going to update/insert a foreign key.

> **Note:** Note that if you run a transaction in SERIALIZABLE mode then you must execute the `LOCK` commands above before execution of any DML statement (`SELECT/INSERT/DELETE/UPDATE/FETCH/COPY_TO`) in the transaction.

These inconveniences will disappear in the future when the ability to read dirty (uncommitted) data (regardless of isolation level) and true referential integrity will be implemented.

E.98.2. Changes

```
Bug Fixes
---------
Fix text<->float8 and text<->float4 conversion functions(Thomas)
Fix for creating tables with mixed-case constraints(Billy)
Change exp()/pow() behavior to generate error on underflow/overflow(Jan)
Fix bug in pg_dump -z
Memory overrun cleanups(Tatsuo)
Fix for lo_import crash(Tatsuo)
Adjust handling of data type names to suppress double quotes(Thomas)
Use type coercion for matching columns and DEFAULT(Thomas)
Fix deadlock so it only checks once after one second of sleep(Bruce)
Fixes for aggregates and PL/pgsql(Hiroshi)
Fix for subquery crash(Vadim)
Fix for libpq function PQfnumber and case-insensitive names(Bahman Rafatjoo)
Fix for large object write-in-middle, no extra block, memory consumption(Tatsuo)
Fix for pg_dump -d or -D and  quote special characters in INSERT
Repair serious problems with dynahash(Tom)
Fix INET/CIDR portability problems
Fix problem with selectivity error in ALTER TABLE ADD COLUMN(Bruce)
Fix executor so mergejoin of different column types works(Tom)
Fix for Alpha OR selectivity bug
Fix OR index selectivity problem(Bruce)
Fix so \d shows proper length for char()/varchar()(Ryan)
Fix tutorial code(Clark)
Improve destroyuser checking(Oliver)
Fix for Kerberos(Rodney McDuff)
Fix for dropping database while dirty buffers(Bruce)
Fix so sequence nextval() can be case-sensitive(Bruce)
Fix !!= operator
Drop buffers before destroying database files(Bruce)
Fix case where executor evaluates functions twice(Tatsuo)
Allow sequence nextval actions to be case-sensitive(Bruce)
Fix optimizer indexing not working for negative numbers(Bruce)
Fix for memory leak in executor with fjIsNull
```

```
Fix for aggregate memory leaks(Erik Riedel)
Allow user name containing a dash to grant privileges
Cleanup of NULL in inet types
Clean up system table bugs(Tom)
Fix problems of PAGER and \? command(Masaaki Sakaida)
Reduce default multisegment file size limit to 1GB(Peter)
Fix for dumping of CREATE OPERATOR(Tom)
Fix for backward scanning of cursors(Hiroshi Inoue)
Fix for COPY FROM STDIN when using \i(Tom)
Fix for subselect is compared inside an expression(Jan)
Fix handling of error reporting while returning rows(Tom)
Fix problems with reference to array types(Tom,Jan)
Prevent UPDATE SET oid(Jan)
Fix pg_dump so -t option can handle case-sensitive tablenames
Fixes for GROUP BY in special cases(Tom, Jan)
Fix for memory leak in failed queries(Tom)
DEFAULT now supports mixed-case identifiers(Tom)
Fix for multisegment uses of DROP/RENAME table, indexes(Ole Gjerde)
Disable use of pg_dump with both -o and -d options(Bruce)
Allow pg_dump to properly dump group privileges(Bruce)
Fix GROUP BY in INSERT INTO table SELECT * FROM table2(Jan)
Fix for computations in views(Jan)
Fix for aggregates on array indexes(Tom)
Fix for DEFAULT handles single quotes in value requiring too many quotes
Fix security problem with non-super users importing/exporting large objects(Tom)
Rollback of transaction that creates table cleaned up properly(Tom)
Fix to allow long table and column names to generate proper serial names(Tom)

Enhancements
------------
Add "vacuumdb" utility
Speed up libpq by allocating memory better(Tom)
EXPLAIN all indexes used(Tom)
Implement CASE, COALESCE, NULLIF  expression(Thomas)
New pg_dump table output format(Constantin)
Add string min()/max() functions(Thomas)
Extend new type coercion techniques to aggregates(Thomas)
New moddatetime contrib(Terry)
Update to pgaccess 0.96(Constantin)
Add routines for single-byte "char" type(Thomas)
Improved substr() function(Thomas)
Improved multibyte handling(Tatsuo)
Multiversion concurrency control/MVCC(Vadim)
New Serialized mode(Vadim)
Fix for tables over 2gigs(Peter)
New SET TRANSACTION ISOLATION LEVEL(Vadim)
New LOCK TABLE IN ... MODE(Vadim)
Update ODBC driver(Byron)
New NUMERIC data type(Jan)
New SELECT FOR UPDATE(Vadim)
Handle "NaN" and "Infinity" for input values(Jan)
Improved date/year handling(Thomas)
Improved handling of backend connections(Magnus)
```

New options ELOG_TIMESTAMPS and USE_SYSLOG options for log files(Massimo)
New TCL_ARRAYS option(Massimo)
New INTERSECT and EXCEPT(Stefan)
New pg_index.indisprimary for primary key tracking(D'Arcy)
New pg_dump option to allow dropping of tables before creation(Brook)
Speedup of row output routines(Tom)
New READ COMMITTED isolation level(Vadim)
New TEMP tables/indexes(Bruce)
Prevent sorting if result is already sorted(Jan)
New memory allocation optimization(Jan)
Allow psql to do \p\g(Bruce)
Allow multiple rule actions(Jan)
Added LIMIT/OFFSET functionality(Jan)
Improve optimizer when joining a large number of tables(Bruce)
New intro to SQL from S. Simkovics' Master's Thesis (Stefan, Thomas)
New intro to backend processing from S. Simkovics' Master's Thesis (Stefan)
Improved int8 support(Ryan Bradetich, Thomas, Tom)
New routines to convert between int8 and text/varchar types(Thomas)
New bushy plans, where meta-tables are joined(Bruce)
Enable right-hand queries by default(Bruce)
Allow reliable maximum number of backends to be set at configure time
     (--with-maxbackends and postmaster switch (-N backends))(Tom)
GEQO default now 10 tables because of optimizer speedups(Tom)
Allow NULL=Var for MS-SQL portability(Michael, Bruce)
Modify contrib check_primary_key() so either "automatic" or "dependent"(Anand)
Allow psql \d on a view show query(Ryan)
Speedup for LIKE(Bruce)
Ecpg fixes/features, see src/interfaces/ecpg/ChangeLog file(Michael)
JDBC fixes/features, see src/interfaces/jdbc/CHANGELOG(Peter)
Make % operator have precedence like /(Bruce)
Add new postgres -O option to allow system table structure changes(Bruce)
Update contrib/pginterface/findoidjoins script(Tom)
Major speedup in vacuum of deleted rows with indexes(Vadim)
Allow non-SQL functions to run different versions based on arguments(Tom)
Add -E option that shows actual queries sent by \dt and friends(Masaaki Sakaida)
Add version number in start-up banners for psql(Masaaki Sakaida)
New contrib/vacuumlo removes large objects not referenced(Peter)
New initialization for table sizes so non-vacuumed tables perform better(Tom)
Improve error messages when a connection is rejected(Tom)
Support for arrays of char() and varchar() fields(Massimo)
Overhaul of hash code to increase reliability and performance(Tom)
Update to PyGreSQL 2.4(D'Arcy)
Changed debug options so -d4 and -d5 produce different node displays(Jan)
New pg_options: pretty_plan, pretty_parse, pretty_rewritten(Jan)
Better optimization statistics for system table access(Tom)
Better handling of non-default block sizes(Massimo)
Improve GEQO optimizer memory consumption(Tom)
UNION now supports ORDER BY of columns not in target list(Jan)
Major libpq++ improvements(Vince Vielhaber)
pg_dump now uses -z(ACL's) as default(Bruce)
backend cache, memory speedups(Tom)
have pg_dump do everything in one snapshot transaction(Vadim)
fix for large object memory leakage, fix for pg_dumping(Tom)

```
INET type now respects netmask for comparisons
Make VACUUM ANALYZE only use a readlock(Vadim)
Allow VIEWs on UNIONS(Jan)
pg_dump now can generate consistent snapshots on active databases(Vadim)

Source Tree Changes
-------------------
Improve port matching(Tom)
Portability fixes for SunOS
Add Windows NT backend port and enable dynamic loading(Magnus and Daniel Horak)
New port to Cobalt Qube(Mips) running Linux(Tatsuo)
Port to NetBSD/m68k(Mr. Mutsuki Nakajima)
Port to NetBSD/sun3(Mr. Mutsuki Nakajima)
Port to NetBSD/macppc(Toshimi Aoki)
Fix for tcl/tk configuration(Vince)
Removed CURRENT key word for rule queries(Jan)
NT dynamic loading now works(Daniel Horak)
Add ARM32 support(Andrew McMurry)
Better support for HP-UX 11 and UnixWare
Improve file handling to be more uniform, prevent file descriptor leak(Tom)
New install commands for plpgsql(Jan)
```

E.99. Release 6.4.2

Release date: 1998-12-20

The 6.4.1 release was improperly packaged. This also has one additional bug fix.

E.99.1. Migration to Version 6.4.2

A dump/restore is *not* required for those running 6.4.*.

E.99.2. Changes

```
Fix for datetime constant problem on some platforms(Thomas)
```

E.100. Release 6.4.1

Release date: 1998-12-18

This is basically a cleanup release for 6.4. We have fixed a variety of problems reported by 6.4 users.

E.100.1. Migration to Version 6.4.1

A dump/restore is *not* required for those running 6.4.

E.100.2. Changes

```
Add pg_dump -N flag to force double quotes around identifiers.  This is
       the default(Thomas)
Fix for NOT in where clause causing crash(Bruce)
EXPLAIN VERBOSE coredump fix(Vadim)
Fix shared-library problems on Linux
Fix test for table existence to allow mixed-case and whitespace in
       the table name(Thomas)
Fix a couple of pg_dump bugs
Configure matches template/.similar entries better(Tom)
Change builtin function names from SPI_* to spi_*
OR WHERE clause fix(Vadim)
Fixes for mixed-case table names(Billy)
contrib/linux/postgres.init.csh/sh fix(Thomas)
libpq memory overrun fix
SunOS fixes(Tom)
Change exp() behavior to generate error on underflow(Thomas)
pg_dump fixes for memory leak, inheritance constraints, layout change
update pgaccess to 0.93
Fix prototype for 64-bit platforms
Multibyte fixes(Tatsuo)
New ecpg man page
Fix memory overruns(Tatsuo)
Fix for lo_import() crash(Bruce)
Better search for install program(Tom)
Timezone fixes(Tom)
HP-UX fixes(Tom)
Use implicit type coercion for matching DEFAULT values(Thomas)
Add routines to help with single-byte (internal) character type(Thomas)
Compilation of libpq for Windows fixes(Magnus)
Upgrade to PyGreSQL 2.2(D'Arcy)
```

E.101. Release 6.4

Release date: 1998-10-30

There are *many* new features and improvements in this release. Thanks to our developers and maintainers, nearly every aspect of the system has received some attention since the previous release. Here is a brief, incomplete summary:

- Views and rules are now functional thanks to extensive new code in the rewrite rules system from Jan Wieck. He also wrote a chapter on it for the *Programmer's Guide*.
- Jan also contributed a second procedural language, PL/pgSQL, to go with the original PL/pgTCL procedural language he contributed last release.
- We have optional multiple-byte character set support from Tatsuo Ishii to complement our existing locale support.
- Client/server communications has been cleaned up, with better support for asynchronous messages and interrupts thanks to Tom Lane.
- The parser will now perform automatic type coercion to match arguments to available operators and functions, and to match columns and expressions with target columns. This uses a generic mechanism which supports the type extensibility features of PostgreSQL. There is a new chapter in the *User's Guide* which covers this topic.
- Three new data types have been added. Two types, `inet` and `cidr`, support various forms of IP network, subnet, and machine addressing. There is now an 8-byte integer type available on some platforms. See the chapter on data types in the *User's Guide* for details. A fourth type, `serial`, is now supported by the parser as an amalgam of the `int4` type, a sequence, and a unique index.
- Several more SQL92-compatible syntax features have been added, including `INSERT DEFAULT VALUES`
- The automatic configuration and installation system has received some attention, and should be more robust for more platforms than it has ever been.

E.101.1. Migration to Version 6.4

A dump/restore using pg_dump or pg_dumpall is required for those wishing to migrate data from any previous release of PostgreSQL.

E.101.2. Changes

```
Bug Fixes
---------
Fix for a tiny memory leak in PQsetdb/PQfinish(Bryan)
```

```
Remove char2-16 data types, use char/varchar(Darren)
Pqfn not handles a NOTICE message(Anders)
Reduced busywaiting overhead for spinlocks with many backends (dg)
Stuck spinlock detection (dg)
Fix up "ISO-style" timespan decoding and encoding(Thomas)
Fix problem with table drop after rollback of transaction(Vadim)
Change error message and remove non-functional update message(Vadim)
Fix for COPY array checking
Fix for SELECT 1 UNION SELECT NULL
Fix for buffer leaks in large object calls(Pascal)
Change owner from oid to int4 type(Bruce)
Fix a bug in the oracle compatibility functions btrim() ltrim() and rtrim()
Fix for shared invalidation cache overflow(Massimo)
Prevent file descriptor leaks in failed COPY's(Bruce)
Fix memory leak in libpgtcl's pg_select(Constantin)
Fix problems with username/passwords over 8 characters(Tom)
Fix problems with handling of asynchronous NOTIFY in backend(Tom)
Fix of many bad system table entries(Tom)

Enhancements
------------
Upgrade ecpg and ecpglib,see src/interfaces/ecpc/ChangeLog(Michael)
Show the index used in an EXPLAIN(Zeugswetter)
EXPLAIN  invokes  rule system and shows plan(s) for rewritten queries(Jan)
Multibyte awareness of many data types and functions, via configure(Tatsuo)
New configure --with-mb option(Tatsuo)
New initdb --pgencoding option(Tatsuo)
New createdb -E multibyte option(Tatsuo)
Select version(); now returns PostgreSQL version(Jeroen)
libpq now allows asynchronous clients(Tom)
Allow cancel from client of backend query(Tom)
psql now cancels query with Control-C(Tom)
libpq users need not issue dummy queries to get NOTIFY messages(Tom)
NOTIFY now sends sender's PID, so you can tell whether it was your own(Tom)
PGresult struct now includes associated error message, if any(Tom)
Define "tz_hour" and "tz_minute" arguments to date_part()(Thomas)
Add routines to convert between varchar and bpchar(Thomas)
Add routines to allow sizing of varchar and bpchar into target columns(Thomas)
Add bit flags to support timezonehour and minute in data retrieval(Thomas)
Allow more variations on valid floating point numbers (e.g. ".1", "1e6")(Thomas)
Fixes for unary minus parsing with leading spaces(Thomas)
Implement TIMEZONE_HOUR, TIMEZONE_MINUTE per SQL92 specs(Thomas)
Check for and properly ignore FOREIGN KEY column constraints(Thomas)
Define USER as synonym for CURRENT_USER per SQL92 specs(Thomas)
Enable HAVING clause but no fixes elsewhere yet.
Make "char" type a synonym for "char(1)" (actually implemented as bpchar)(Thomas)
Save string type if specified for DEFAULT clause handling(Thomas)
Coerce operations involving different data types(Thomas)
Allow some index use for columns of different types(Thomas)
Add capabilities for automatic type conversion(Thomas)
Cleanups for large objects, so file is truncated on open(Peter)
Readline cleanups(Tom)
Allow psql  \f \ to make spaces as delimiter(Bruce)
```

Pass pg_attribute.atttypmod to the frontend for column field lengths(Tom,Bruce)
Msql compatibility library in /contrib(Aldrin)
Remove the requirement that ORDER/GROUP BY clause identifiers be
included in the target list(David)
Convert columns to match columns in UNION clauses(Thomas)
Remove fork()/exec() and only do fork()(Bruce)
Jdbc cleanups(Peter)
Show backend status on ps command line(only works on some platforms)(Bruce)
Pg_hba.conf now has a sameuser option in the database field
Make lo_unlink take oid param, not int4
New DISABLE_COMPLEX_MACRO for compilers that cannot handle our macros(Bruce)
Libpgtcl now handles NOTIFY as a Tcl event, need not send dummy queries(Tom)
libpgtcl cleanups(Tom)
Add -error option to libpgtcl's pg_result command(Tom)
New locale patch, see docs/README/locale(Oleg)
Fix for pg_dump so CONSTRAINT and CHECK syntax is correct(ccb)
New contrib/lo code for large object orphan removal(Peter)
New psql command "SET CLIENT_ENCODING TO 'encoding'" for multibytes
feature, see /doc/README.mb(Tatsuo)
contrib/noupdate code to revoke update permission on a column
libpq can now be compiled on Windows(Magnus)
Add PQsetdbLogin() in libpq
New 8-byte integer type, checked by configure for OS support(Thomas)
Better support for quoted table/column names(Thomas)
Surround table and column names with double-quotes in pg_dump(Thomas)
PQreset() now works with passwords(Tom)
Handle case of GROUP BY target list column number out of range(David)
Allow UNION in subselects
Add auto-size to screen to \d? commands(Bruce)
Use UNION to show all \d? results in one query(Bruce)
Add \d? field search feature(Bruce)
Pg_dump issues fewer \connect requests(Tom)
Make pg_dump -z flag work better, document it in manual page(Tom)
Add HAVING clause with full support for subselects and unions(Stephan)
Full text indexing routines in contrib/fulltextindex(Maarten)
Transaction ids now stored in shared memory(Vadim)
New PGCLIENTENCODING when issuing COPY command(Tatsuo)
Support for SQL92 syntax "SET NAMES"(Tatsuo)
Support for LATIN2-5(Tatsuo)
Add UNICODE regression test case(Tatsuo)
Lock manager cleanup, new locking modes for LLL(Vadim)
Allow index use with OR clauses(Bruce)
Allows "SELECT NULL ORDER BY 1;"
Explain VERBOSE prints the plan, and now pretty-prints the plan to
the postmaster log file(Bruce)
Add indexes display to \d command(Bruce)
Allow GROUP BY on functions(David)
New pg_class.relkind for large objects(Bruce)
New way to send libpq NOTICE messages to a different location(Tom)
New \w write command to psql(Bruce)
New /contrib/findoidjoins scans oid columns to find join relationships(Bruce)
Allow binary-compatible indexes to be considered when checking for valid
Indexes for restriction clauses containing a constant(Thomas)

```
New ISBN/ISSN code in /contrib/isbn_issn
Allow NOT LIKE, IN, NOT IN, BETWEEN, and NOT BETWEEN constraint(Thomas)
New rewrite system fixes many problems with rules and views(Jan)
       * Rules on relations work
       * Event qualifications on insert/update/delete work
       * New OLD variable to reference CURRENT, CURRENT will be remove in future
       * Update rules can reference NEW and OLD in rule qualifications/actions
       * Insert/update/delete rules on views work
       * Multiple rule actions are now supported, surrounded by parentheses
       * Regular users can create views/rules on tables they have RULE permits
       * Rules and views inherit the privileges of the creator
       * No rules at the column level
       * No UPDATE NEW/OLD rules
       * New pg_tables, pg_indexes, pg_rules and pg_views system views
       * Only a single action on SELECT rules
       * Total rewrite overhaul, perhaps for 6.5
       * handle subselects
       * handle aggregates on views
       * handle insert into select from view works
System indexes are now multikey(Bruce)
Oidint2, oidint4, and oidname types are removed(Bruce)
Use system cache for more system table lookups(Bruce)
New backend programming language PL/pgSQL in backend/pl(Jan)
New SERIAL data type, auto-creates sequence/index(Thomas)
Enable assert checking without a recompile(Massimo)
User lock enhancements(Massimo)
New setval() command to set sequence value(Massimo)
Auto-remove unix socket file on start-up if no postmaster running(Massimo)
Conditional trace package(Massimo)
New UNLISTEN command(Massimo)
psql and libpq now compile under Windows using win32.mak(Magnus)
Lo_read no longer stores trailing NULL(Bruce)
Identifiers are now truncated to 31 characters internally(Bruce)
Createuser options now available on the command line
Code for 64-bit integer supported added, configure tested, int8 type(Thomas)
Prevent file descriptor leaf from failed COPY(Bruce)
New pg_upgrade command(Bruce)
Updated /contrib directories(Massimo)
New CREATE TABLE DEFAULT VALUES statement available(Thomas)
New INSERT INTO TABLE DEFAULT VALUES statement available(Thomas)
New DECLARE and FETCH feature(Thomas)
libpq's internal structures now not exported(Tom)
Allow up to 8 key indexes(Bruce)
Remove ARCHIVE key word, that is no longer used(Thomas)
pg_dump -n flag to suppress quotes around indentifiers
disable system columns for views(Jan)
new INET and CIDR types for network addresses(TomH, Paul)
no more double quotes in psql output
pg_dump now dumps views(Terry)
new SET QUERY_LIMIT(Tatsuo,Jan)

Source Tree Changes
-------------------
```

```
/contrib cleanup(Jun)
Inline some small functions called for every row(Bruce)
Alpha/linux fixes
HP-UX cleanups(Tom)
Multibyte regression tests(Soonmyung.)
Remove --disabled options from configure
Define PGDOC to use POSTGRESDIR by default
Make regression optional
Remove extra braces code to pgindent(Bruce)
Add bsdi shared library support(Bruce)
New --without-CXX support configure option(Brook)
New FAQ_CVS
Update backend flowchart in tools/backend(Bruce)
Change atttypmod from int16 to int32(Bruce, Tom)
Getrusage() fix for platforms that do not have it(Tom)
Add PQconnectdb, PGUSER, PGPASSWORD to libpq man page
NS32K platform fixes(Phil Nelson, John Buller)
SCO 7/UnixWare 2.x fixes(Billy,others)
Sparc/Solaris 2.5 fixes(Ryan)
Pgbuiltin.3 is obsolete, move to doc files(Thomas)
Even more documentation(Thomas)
Nextstep support(Jacek)
Aix support(David)
pginterface manual page(Bruce)
shared libraries all have version numbers
merged all OS-specific shared library defines into one file
smarter TCL/TK configuration checking(Billy)
smarter perl configuration(Brook)
configure uses supplied install-sh if no install script found(Tom)
new Makefile.shlib for shared library configuration(Tom)
```

E.102. Release 6.3.2

Release date: 1998-04-07

This is a bug-fix release for 6.3.x. Refer to the release notes for version 6.3 for a more complete summary of new features.

Summary:

- Repairs automatic configuration support for some platforms, including Linux, from breakage inadvertently introduced in version 6.3.1.
- Correctly handles function calls on the left side of BETWEEN and LIKE clauses.

A dump/restore is NOT required for those running 6.3 or 6.3.1. A `make distclean`, `make`, and `make install` is all that is required. This last step should be performed while the postmaster is not running. You should re-link any custom applications that use PostgreSQL libraries.

For upgrades from pre-6.3 installations, refer to the installation and migration instructions for version 6.3.

E.102.1. Changes

```
Configure detection improvements for tcl/tk(Brook Milligan, Alvin)
Manual page improvements(Bruce)
BETWEEN and LIKE fix(Thomas)
fix for psql \connect used by pg_dump(Oliver Elphick)
New odbc driver
pgaccess, version 0.86
qsort removed, now uses libc version, cleanups(Jeroen)
fix for buffer over-runs detected(Maurice Gittens)
fix for buffer overrun in libpgtcl(Randy Kunkee)
fix for UNION with DISTINCT or ORDER BY(Bruce)
gettimeofday configure check(Doug Winterburn)
Fix "indexes not used" bug(Vadim)
docs additions(Thomas)
Fix for backend memory leak(Bruce)
libreadline cleanup(Erwan MAS)
Remove DISTDIR(Bruce)
Makefile dependency cleanup(Jeroen van Vianen)
ASSERT fixes(Bruce)
```

E.103. Release 6.3.1

Release date: 1998-03-23

Summary:

- Additional support for multibyte character sets.
- Repair byte ordering for mixed-endian clients and servers.
- Minor updates to allowed SQL syntax.
- Improvements to the configuration autodetection for installation.

A dump/restore is NOT required for those running 6.3. A `make distclean`, `make`, and `make install` is all that is required. This last step should be performed while the postmaster is not running. You should re-link any custom applications that use PostgreSQL libraries.

For upgrades from pre-6.3 installations, refer to the installation and migration instructions for version 6.3.

E.103.1. Changes

```
ecpg cleanup/fixes, now version 1.1(Michael Meskes)
pg_user cleanup(Bruce)
large object fix for pg_dump and tclsh (alvin)
LIKE fix for multiple adjacent underscores
fix for redefining builtin functions(Thomas)
ultrix4 cleanup
upgrade to pg_access 0.83
updated CLUSTER manual page
multibyte character set support, see doc/README.mb(Tatsuo)
configure --with-pgport fix
pg_ident fix
big-endian fix for backend communications(Kataoka)
SUBSTR() and substring() fix(Jan)
several jdbc fixes(Peter)
libpgtcl improvements, see libptcl/README(Randy Kunkee)
Fix for "Datasize = 0" error(Vadim)
Prevent \do from wrapping(Bruce)
Remove duplicate Russian character set entries
Sunos4 cleanup
Allow optional TABLE key word in LOCK and SELECT INTO(Thomas)
CREATE SEQUENCE options to allow a negative integer(Thomas)
Add "PASSWORD" as an allowed column identifier(Thomas)
Add checks for UNION target fields(Bruce)
Fix Alpha port(Dwayne Bailey)
Fix for text arrays containing quotes(Doug Gibson)
Solaris compile fix(Albert Chin-A-Young)
Better identify tcl and tk libs and includes(Bruce)
```

E.104. Release 6.3

Release date: 1998-03-01

There are *many* new features and improvements in this release. Here is a brief, incomplete summary:

- Many new SQL features, including full SQL92 subselect capability (everything is here but target-list subselects).
- Support for client-side environment variables to specify time zone and date style.
- Socket interface for client/server connection. This is the default now so you might need to start postmaster with the `-i` flag.
- Better password authorization mechanisms. Default table privileges have changed.
- Old-style *time travel* has been removed. Performance has been improved.

Note: Bruce Momjian wrote the following notes to introduce the new release.

There are some general 6.3 issues that I want to mention. These are only the big items that cannot be described in one sentence. A review of the detailed changes list is still needed.

First, we now have subselects. Now that we have them, I would like to mention that without subselects, SQL is a very limited language. Subselects are a major feature, and you should review your code for places where subselects provide a better solution for your queries. I think you will find that there are more uses for subselects than you might think. Vadim has put us on the big SQL map with subselects, and fully functional ones too. The only thing you cannot do with subselects is to use them in the target list.

Second, 6.3 uses Unix domain sockets rather than TCP/IP by default. To enable connections from other machines, you have to use the new postmaster -i option, and of course edit `pg_hba.conf`. Also, for this reason, the format of `pg_hba.conf` has changed.

Third, `char()` fields will now allow faster access than `varchar()` or `text`. Specifically, the `text` and `varchar()` have a penalty for access to any columns after the first column of this type. `char()` used to also have this access penalty, but it no longer does. This might suggest that you redesign some of your tables, especially if you have short character columns that you have defined as `varchar()` or `text`. This and other changes make 6.3 even faster than earlier releases.

We now have passwords definable independent of any Unix file. There are new SQL USER commands. See the *Administrator's Guide* for more information. There is a new table, pg_shadow, which is used to store user information and user passwords, and it by default only SELECT-able by the postgres super-user. pg_user is now a view of pg_shadow, and is SELECT-able by PUBLIC. You should keep using pg_user in your application without changes.

User-created tables now no longer have SELECT privilege to PUBLIC by default. This was done because the ANSI standard requires it. You can of course GRANT any privileges you want after the table is created. System tables continue to be SELECT-able by PUBLIC.

We also have real deadlock detection code. No more sixty-second timeouts. And the new locking code implements a FIFO better, so there should be less resource starvation during heavy use.

Many complaints have been made about inadequate documentation in previous releases. Thomas has put much effort into many new manuals for this release. Check out the doc/ directory.

For performance reasons, time travel is gone, but can be implemented using triggers (see `pgsql/contrib/spi/README`). Please check out the new \d command for types, operators, etc. Also, views have their own privileges now, not based on the underlying tables, so privileges on them have to be set separately. Check `/pgsql/interfaces` for some new ways to talk to PostgreSQL.

This is the first release that really required an explanation for existing users. In many ways, this was necessary because the new release removes many limitations, and the work-arounds people were using are no longer needed.

E.104.1. Migration to Version 6.3

A dump/restore using pg_dump or pg_dumpall is required for those wishing to migrate data from any previous release of PostgreSQL.

E.104.2. Changes

```
Bug Fixes
---------
Fix binary cursors broken by MOVE implementation(Vadim)
Fix for tcl library crash(Jan)
Fix for array handling, from Gerhard Hintermayer
Fix acl error, and remove duplicate pqtrace(Bruce)
Fix psql \e for empty file(Bruce)
Fix for textcat on varchar() fields(Bruce)
Fix for DBT Sendproc (Zeugswetter Andres)
Fix vacuum analyze syntax problem(Bruce)
Fix for international identifiers(Tatsuo)
Fix aggregates on inherited tables(Bruce)
Fix substr() for out-of-bounds data
Fix for select 1=1 or 2=2, select 1=1 and 2=2, and select sum(2+2)(Bruce)
Fix notty output to show status result.  -q option still turns it off(Bruce)
Fix for count(*), aggs with views and multiple tables and sum(3)(Bruce)
Fix cluster(Bruce)
Fix for PQtrace start/stop several times(Bruce)
Fix a variety of locking problems like newer lock waiters getting
       lock before older waiters, and having readlock people not share
       locks if a writer is waiting for a lock, and waiting writers not
       getting priority over waiting readers(Bruce)
Fix crashes in psql when executing queries from external files(James)
Fix problem with multiple order by columns, with the first one having
       NULL values(Jeroen)
Use correct hash table support functions for float8 and int4(Thomas)
Re-enable JOIN= option in CREATE OPERATOR statement (Thomas)
Change precedence for boolean operators to match expected behavior(Thomas)
Generate elog(ERROR) on over-large integer(Bruce)
Allow multiple-argument functions in constraint clauses(Thomas)
Check boolean input literals for 'true','false','yes','no','1','0'
       and throw elog(ERROR) if unrecognized(Thomas)
Major large objects fix
Fix for GROUP BY showing duplicates(Vadim)
Fix for index scans in MergeJoin(Vadim)

Enhancements
------------
```

```
Subselects with EXISTS, IN, ALL, ANY key words (Vadim, Bruce, Thomas)
New User Manual(Thomas, others)
Speedup by inlining some frequently-called functions
Real deadlock detection, no more timeouts(Bruce)
Add SQL92 "constants" CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP,
       CURRENT_USER(Thomas)
Modify constraint syntax to be SQL92-compliant(Thomas)
Implement SQL92 PRIMARY KEY and UNIQUE clauses using indexes(Thomas)
Recognize SQL92 syntax for FOREIGN KEY. Throw elog notice(Thomas)
Allow NOT NULL UNIQUE constraint clause (each allowed separately before)(Thomas)
Allow PostgreSQL-style casting ("::") of non-constants(Thomas)
Add support for SQL3 TRUE and FALSE boolean constants(Thomas)
Support SQL92 syntax for IS TRUE/IS FALSE/IS NOT TRUE/IS NOT FALSE(Thomas)
Allow shorter strings for boolean literals (e.g. "t", "tr", "tru")(Thomas)
Allow SQL92 delimited identifiers(Thomas)
Implement SQL92 binary and hexadecimal string decoding (b'10' and x'1F')(Thomas)
Support SQL92 syntax for type coercion of literal strings
       (e.g. "DATETIME 'now'")(Thomas)
Add conversions for int2, int4, and OID types to and from text(Thomas)
Use shared lock when building indexes(Vadim)
Free memory allocated for an user query inside transaction block after
       this query is done, was turned off in <= 6.2.1(Vadim)
New SQL statement CREATE PROCEDURAL LANGUAGE(Jan)
New PostgreSQL Procedural Language (PL) backend interface(Jan)
Rename pg_dump -H option to -h(Bruce)
Add Java support for passwords, European dates(Peter)
Use indexes for LIKE and ~, !~ operations(Bruce)
Add hash functions for datetime and timespan(Thomas)
Time Travel removed(Vadim, Bruce)
Add paging for \d and \z, and fix \i(Bruce)
Add Unix domain socket support to backend and to frontend library(Goran)
Implement CREATE DATABASE/WITH LOCATION and initlocation utility(Thomas)
Allow more SQL92 and/or PostgreSQL reserved words as column identifiers(Thomas)
Augment support for SQL92 SET TIME ZONE...(Thomas)
SET/SHOW/RESET TIME ZONE uses TZ backend environment variable(Thomas)
Implement SET keyword = DEFAULT and SET TIME ZONE DEFAULT(Thomas)
Enable SET TIME ZONE using TZ environment variable(Thomas)
Add PGDATESTYLE environment variable to frontend and backend initialization(Thomas)
Add PGTZ, PGCOSTHEAP, PGCOSTINDEX, PGRPLANS, PGGEQO
       frontend library initialization environment variables(Thomas)
Regression tests time zone automatically set with "setenv PGTZ PST8PDT"(Thomas)
Add pg_description table for info on tables, columns, operators, types, and
       aggregates(Bruce)
Increase 16 char limit on system table/index names to 32 characters(Bruce)
Rename system indexes(Bruce)
Add 'GERMAN' option to SET DATESTYLE(Thomas)
Define an "ISO-style" timespan output format with "hh:mm:ss" fields(Thomas)
Allow fractional values for delta times (e.g. '2.5 days')(Thomas)
Validate numeric input more carefully for delta times(Thomas)
Implement day of year as possible input to date_part()(Thomas)
Define timespan_finite() and text_timespan() functions(Thomas)
Remove archive stuff(Bruce)
Allow for a pg_password authentication database that is separate from
```

```
       the system password file(Todd)
Dump ACLs, GRANT, REVOKE privileges(Matt)
Define text, varchar, and bpchar string length functions(Thomas)
Fix Query handling for inheritance, and cost computations(Bruce)
Implement CREATE TABLE/AS SELECT (alternative to SELECT/INTO)(Thomas)
Allow NOT, IS NULL, IS NOT NULL in constraints(Thomas)
Implement UNIONs for SELECT(Bruce)
Add UNION, GROUP, DISTINCT to INSERT(Bruce)
varchar() stores only necessary bytes on disk(Bruce)
Fix for BLOBs(Peter)
Mega-Patch for JDBC...see README_6.3 for list of changes(Peter)
Remove unused "option" from PQconnectdb()
New LOCK command and lock manual page describing deadlocks(Bruce)
Add new psql \da, \dd, \df, \do, \dS, and \dT commands(Bruce)
Enhance psql \z to show sequences(Bruce)
Show NOT NULL and DEFAULT in psql \d table(Bruce)
New psql .psqlrc file start-up(Andrew)
Modify sample start-up script in contrib/linux to show syslog(Thomas)
New types for IP and MAC addresses in contrib/ip_and_mac(TomH)
Unix system time conversions with date/time types in contrib/unixdate(Thomas)
Update of contrib stuff(Massimo)
Add Unix socket support to DBD::Pg(Goran)
New python interface (PyGreSQL 2.0)(D'Arcy)
New frontend/backend protocol has a version number, network byte order(Phil)
Security features in pg_hba.conf enhanced and documented, many cleanups(Phil)
CHAR() now faster access than VARCHAR() or TEXT
ecpg embedded SQL preprocessor
Reduce system column overhead(Vadmin)
Remove pg_time table(Vadim)
Add pg_type attribute to identify types that need length (bpchar, varchar)
Add report of offending line when COPY command fails
Allow VIEW privileges to be set separately from the underlying tables.
       For security, use GRANT/REVOKE on views as appropriate(Jan)
Tables now have no default GRANT SELECT TO PUBLIC.  You must
       explicitly grant such privileges.
Clean up tutorial examples(Darren)

Source Tree Changes
-------------------
Add new html development tools, and flow chart in /tools/backend
Fix for SCO compiles
Stratus computer port Robert Gillies
Added support for shlib for BSD44_derived & i386_solaris
Make configure more automated(Brook)
Add script to check regression test results
Break parser functions into smaller files, group together(Bruce)
Rename heap_create to heap_create_and_catalog, rename heap_creatr
       to heap_create()(Bruce)
Sparc/Linux patch for locking(TomS)
Remove PORTNAME and reorganize port-specific stuff(Marc)
Add optimizer README file(Bruce)
Remove some recursion in optimizer and clean up some code there(Bruce)
Fix for NetBSD locking(Henry)
```

```
Fix for libptcl make(Tatsuo)
AIX patch(Darren)
Change IS TRUE, IS FALSE, ... to expressions using "=" rather than
       function calls to istrue() or isfalse() to allow optimization(Thomas)
Various fixes NetBSD/Sparc related(TomH)
Alpha linux locking(Travis,Ryan)
Change elog(WARN) to elog(ERROR)(Bruce)
FAQ for FreeBSD(Marc)
Bring in the PostODBC source tree as part of our standard distribution(Marc)
A minor patch for HP/UX 10 vs 9(Stan)
New pg_attribute.atttypmod for type-specific info like varchar length(Bruce)
UnixWare patches(Billy)
New i386 'lock' for spinlock asm(Billy)
Support for multiplexed backends is removed
Start an OpenBSD port
Start an AUX port
Start a Cygnus port
Add string functions to regression suite(Thomas)
Expand a few function names formerly truncated to 16 characters(Thomas)
Remove un-needed malloc() calls and replace with palloc()(Bruce)
```

E.105. Release 6.2.1

Release date: 1997-10-17

6.2.1 is a bug-fix and usability release on 6.2.

Summary:

- Allow strings to span lines, per SQL92.
- Include example trigger function for inserting user names on table updates.

This is a minor bug-fix release on 6.2. For upgrades from pre-6.2 systems, a full dump/reload is required. Refer to the 6.2 release notes for instructions.

E.105.1. Migration from version 6.2 to version 6.2.1

This is a minor bug-fix release. A dump/reload is not required from version 6.2, but is required from any release prior to 6.2.

In upgrading from version 6.2, if you choose to dump/reload you will find that avg(money) is now calculated correctly. All other bug fixes take effect upon updating the executables.

Another way to avoid dump/reload is to use the following SQL command from `psql` to update the existing system table:

```
update pg_aggregate set aggfinalfn = 'cash_div_flt8'
 where aggname = 'avg' and aggbasetype = 790;
```

This will need to be done to every existing database, including template1.

E.105.2. Changes

```
Allow TIME and TYPE column names(Thomas)
Allow larger range of true/false as boolean values(Thomas)
Support output of "now" and "current"(Thomas)
Handle DEFAULT with INSERT of NULL properly(Vadim)
Fix for relation reference counts problem in buffer manager(Vadim)
Allow strings to span lines, like ANSI(Thomas)
Fix for backward cursor with ORDER BY(Vadim)
Fix avg(cash) computation(Thomas)
Fix for specifying a column twice in ORDER/GROUP BY(Vadim)
Documented new libpq function to return affected rows, PQcmdTuples(Bruce)
Trigger function for inserting user names for INSERT/UPDATE(Brook Milligan)
```

E.106. Release 6.2

Release date: 1997-10-02

A dump/restore is required for those wishing to migrate data from previous releases of PostgreSQL.

E.106.1. Migration from version 6.1 to version 6.2

This migration requires a complete dump of the 6.1 database and a restore of the database in 6.2.

Note that the `pg_dump` and `pg_dumpall` utility from 6.2 should be used to dump the 6.1 database.

E.106.2. Migration from version 1.*x* to version 6.2

Those migrating from earlier 1.* releases should first upgrade to 1.09 because the COPY output format was improved from the 1.02 release.

E.106.3. Changes

```
Bug Fixes
---------
Fix problems with pg_dump for inheritance, sequences, archive tables(Bruce)
Fix compile errors on overflow due to shifts, unsigned, and bad prototypes
        from Solaris(Diab Jerius)
Fix bugs in geometric line arithmetic (bad intersection calculations)(Thomas)
Check for geometric intersections at endpoints to avoid rounding ugliness(Thomas)
Catch non-functional delete attempts(Vadim)
Change time function names to be more consistent(Michael Reifenberg)
Check for zero divides(Michael Reifenberg)
Fix very old bug which made rows changed/inserted by a command
       visible to the command itself (so we had multiple update of
       updated rows, etc.)(Vadim)
Fix for SELECT null, 'fail' FROM pg_am (Patrick)
SELECT NULL as EMPTY_FIELD now allowed(Patrick)
Remove un-needed signal stuff from contrib/pginterface
Fix OR (where x != 1 or x isnull didn't return rows with x NULL) (Vadim)
Fix time_cmp function (Vadim)
Fix handling of functions with non-attribute first argument in
       WHERE clauses (Vadim)
Fix GROUP BY when order of entries is different from order
       in target list (Vadim)
Fix pg_dump for aggregates without sfunc1 (Vadim)

Enhancements
------------
Default genetic optimizer GEQO parameter is now 8(Bruce)
Allow use parameters in target list having aggregates in functions(Vadim)
Added JDBC driver as an interface(Adrian & Peter)
pg_password utility
Return number of rows inserted/affected by INSERT/UPDATE/DELETE etc.(Vadim)
Triggers implemented with CREATE TRIGGER (SQL3)(Vadim)
SPI (Server Programming Interface) allows execution of queries inside
       C-functions (Vadim)
NOT NULL implemented (SQL92)(Robson Paniago de Miranda)
Include reserved words for string handling, outer joins, and unions(Thomas)
Implement extended comments ("/* ... */") using exclusive states(Thomas)
Add "//" single-line comments(Bruce)
Remove some restrictions on characters in operator names(Thomas)
DEFAULT and CONSTRAINT for tables implemented (SQL92)(Vadim & Thomas)
Add text concatenation operator and function (SQL92)(Thomas)
Support WITH TIME ZONE syntax (SQL92)(Thomas)
Support INTERVAL unit TO unit syntax (SQL92)(Thomas)
Define types DOUBLE PRECISION, INTERVAL, CHARACTER,
       and CHARACTER VARYING (SQL92)(Thomas)
Define type FLOAT(p) and rudimentary DECIMAL(p,s), NUMERIC(p,s) (SQL92)(Thomas)
Define EXTRACT(), POSITION(), SUBSTRING(), and TRIM() (SQL92)(Thomas)
Define CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP (SQL92)(Thomas)
Add syntax and warnings for UNION, HAVING, INNER and OUTER JOIN (SQL92)(Thomas)
Add more reserved words, mostly for SQL92 compliance(Thomas)
```

```
Allow hh:mm:ss time entry for timespan/reltime types(Thomas)
Add center() routines for lseg, path, polygon(Thomas)
Add distance() routines for circle-polygon, polygon-polygon(Thomas)
Check explicitly for points and polygons contained within polygons
       using an axis-crossing algorithm(Thomas)
Add routine to convert circle-box(Thomas)
Merge conflicting operators for different geometric data types(Thomas)
Replace distance operator "<===>" with "<->"(Thomas)
Replace "above" operator "!^" with ">^" and "below" operator "!|" with "<^"(Thomas)
Add routines for text trimming on both ends, substring, and string position(Thomas)
Added conversion routines circle(box) and poly(circle)(Thomas)
Allow internal sorts to be stored in memory rather than in files(Bruce & Vadim)
Allow functions and operators on internally-identical types to succeed(Bruce)
Speed up backend start-up after profiling analysis(Bruce)
Inline frequently called functions for performance(Bruce)
Reduce open() calls(Bruce)
psql:  Add PAGER for \h and \?,\C fix
Fix for psql pager when no tty(Bruce)
New entab utility(Bruce)
General trigger functions for referential integrity (Vadim)
General trigger functions for time travel (Vadim)
General trigger functions for AUTOINCREMENT/IDENTITY feature (Vadim)
MOVE implementation (Vadim)

Source Tree Changes
-------------------
HP-UX 10 patches (Vladimir Turin)
Added SCO support, (Daniel Harris)
MkLinux patches (Tatsuo Ishii)
Change geometric box terminology from "length" to "width"(Thomas)
Deprecate temporary unstored slope fields in geometric code(Thomas)
Remove restart instructions from INSTALL(Bruce)
Look in /usr/ucb first for install(Bruce)
Fix c++ copy example code(Thomas)
Add -o to psql manual page(Bruce)
Prevent relname unallocated string length from being copied into database(Bruce)
Cleanup for NAMEDATALEN use(Bruce)
Fix pg_proc names over 15 chars in output(Bruce)
Add strNcpy() function(Bruce)
remove some (void) casts that are unnecessary(Bruce)
new interfaces directory(Marc)
Replace fopen() calls with calls to fd.c functions(Bruce)
Make functions static where possible(Bruce)
enclose unused functions in #ifdef NOT_USED(Bruce)
Remove call to difftime() in timestamp support to fix SunOS(Bruce & Thomas)
Changes for Digital Unix
Portability fix for pg_dumpall(Bruce)
Rename pg_attribute.attnvals to attdispersion(Bruce)
"intro/unix" manual page now "pgintro"(Bruce)
"built-in" manual page now "pgbuiltin"(Bruce)
"drop" manual page now "drop_table"(Bruce)
Add "create_trigger", "drop_trigger" manual pages(Thomas)
Add constraints regression test(Vadim & Thomas)
```

```
Add comments syntax regression test(Thomas)
Add PGINDENT and support program(Bruce)
Massive commit to run PGINDENT on all *.c and *.h files(Bruce)
Files moved to /src/tools directory(Bruce)
SPI and Trigger programming guides (Vadim & D'Arcy)
```

E.107. Release 6.1.1

Release date: 1997-07-22

E.107.1. Migration from version 6.1 to version 6.1.1

This is a minor bug-fix release. A dump/reload is not required from version 6.1, but is required from any release prior to 6.1. Refer to the release notes for 6.1 for more details.

E.107.2. Changes

```
fix for SET with options (Thomas)
allow pg_dump/pg_dumpall to preserve ownership of all tables/objects(Bruce)
new psql \connect option allows changing usernames without changing databases
fix for initdb --debug option(Yoshihiko Ichikawa))
lextest cleanup(Bruce)
hash fixes(Vadim)
fix date/time month boundary arithmetic(Thomas)
fix timezone daylight handling for some ports(Thomas, Bruce, Tatsuo)
timestamp overhauled to use standard functions(Thomas)
other code cleanup in date/time routines(Thomas)
psql's \d now case-insensitive(Bruce)
psql's backslash commands can now have trailing semicolon(Bruce)
fix memory leak in psql when using \g(Bruce)
major fix for endian handling of communication to server(Thomas, Tatsuo)
Fix for Solaris assembler and include files(Yoshihiko Ichikawa)
allow underscores in usernames(Bruce)
pg_dumpall now returns proper status, portability fix(Bruce)
```

E.108. Release 6.1

Release date: 1997-06-08

The regression tests have been adapted and extensively modified for the 6.1 release of PostgreSQL.

Three new data types (`datetime`, `timespan`, and `circle`) have been added to the native set of PostgreSQL types. Points, boxes, paths, and polygons have had their output formats made consistent across the data types. The polygon output in misc.out has only been spot-checked for correctness relative to the original regression output.

PostgreSQL 6.1 introduces a new, alternate optimizer which uses *genetic* algorithms. These algorithms introduce a random behavior in the ordering of query results when the query contains multiple qualifiers or multiple tables (giving the optimizer a choice on order of evaluation). Several regression tests have been modified to explicitly order the results, and hence are insensitive to optimizer choices. A few regression tests are for data types which are inherently unordered (e.g. points and time intervals) and tests involving those types are explicitly bracketed with `set geqo to 'off'` and `reset geqo`.

The interpretation of array specifiers (the curly braces around atomic values) appears to have changed sometime after the original regression tests were generated. The current `./expected/*.out` files reflect this new interpretation, which might not be correct!

The float8 regression test fails on at least some platforms. This is due to differences in implementations of `pow()` and `exp()` and the signaling mechanisms used for overflow and underflow conditions.

The "random" results in the random test should cause the "random" test to be "failed", since the regression tests are evaluated using a simple diff. However, "random" does not seem to produce random results on my test machine (Linux/gcc/i686).

E.108.1. Migration to Version 6.1

This migration requires a complete dump of the 6.0 database and a restore of the database in 6.1.

Those migrating from earlier 1.* releases should first upgrade to 1.09 because the COPY output format was improved from the 1.02 release.

E.108.2. Changes

```
Bug Fixes
---------
packet length checking in library routines
lock manager priority patch
check for under/over flow of float8(Bruce)
multitable join fix(Vadim)
SIGPIPE crash fix(Darren)
large object fixes(Sven)
allow btree indexes to handle NULLs(Vadim)
timezone fixes(D'Arcy)
select SUM(x) can return NULL on no rows(Thomas)
```

```
internal optimizer, executor bug fixes(Vadim)
fix problem where inner loop in < or <= has no rows(Vadim)
prevent re-commuting join index clauses(Vadim)
fix join clauses for multiple tables(Vadim)
fix hash, hashjoin for arrays(Vadim)
fix btree for abstime type(Vadim)
large object fixes(Raymond)
fix buffer leak in hash indexes (Vadim)
fix rtree for use in inner scan (Vadim)
fix gist for use in inner scan, cleanups (Vadim, Andrea)
avoid unnecessary local buffers allocation (Vadim, Massimo)
fix local buffers leak in transaction aborts (Vadim)
fix file manager memmory leaks, cleanups (Vadim, Massimo)
fix storage manager memmory leaks (Vadim)
fix btree duplicates handling (Vadim)
fix deleted rows reincarnation caused by vacuum (Vadim)
fix SELECT varchar()/char() INTO TABLE made zero-length fields(Bruce)
many psql, pg_dump, and libpq memory leaks fixed using Purify (Igor)

Enhancements
------------
attribute optimization statistics(Bruce)
much faster new btree bulk load code(Paul)
BTREE UNIQUE added to bulk load code(Vadim)
new lock debug code(Massimo)
massive changes to libpg++(Leo)
new GEQO optimizer speeds table multitable optimization(Martin)
new WARN message for non-unique insert into unique key(Marc)
update x=-3, no spaces, now valid(Bruce)
remove case-sensitive identifier handling(Bruce,Thomas,Dan)
debug backend now pretty-prints tree(Darren)
new Oracle character functions(Edmund)
new plaintext password functions(Dan)
no such class or insufficient privilege changed to distinct messages(Dan)
new ANSI timestamp function(Dan)
new ANSI Time and Date types (Thomas)
move large chunks of data in backend(Martin)
multicolumn btree indexes(Vadim)
new SET var TO value command(Martin)
update transaction status on reads(Dan)
new locale settings for character types(Oleg)
new SEQUENCE serial number generator(Vadim)
GROUP BY function now possible(Vadim)
re-organize regression test(Thomas,Marc)
new optimizer operation weights(Vadim)
new psql \z grant/permit option(Marc)
new MONEY data type(D'Arcy,Thomas)
tcp socket communication speed improved(Vadim)
new VACUUM option for attribute statistics, and for certain columns (Vadim)
many geometric type improvements(Thomas,Keith)
additional regression tests(Thomas)
new datestyle variable(Thomas,Vadim,Martin)
more comparison operators for sorting types(Thomas)
```

```
new conversion functions(Thomas)
new more compact btree format(Vadim)
allow pg_dumpall to preserve database ownership(Bruce)
new SET GEQO=# and R_PLANS variable(Vadim)
old (!GEQO) optimizer can use right-sided plans (Vadim)
typechecking improvement in SQL parser(Bruce)
new SET, SHOW, RESET commands(Thomas,Vadim)
new \connect database USER option
new destroydb -i option (Igor)
new \dt and \di psql commands (Darren)
SELECT "\n" now escapes newline (A. Duursma)
new geometry conversion functions from old format (Thomas)

Source tree changes
-------------------
new configuration script(Marc)
readline configuration option added(Marc)
OS-specific configuration options removed(Marc)
new OS-specific template files(Marc)
no more need to edit Makefile.global(Marc)
re-arrange include files(Marc)
nextstep patches (Gregor Hoffleit)
removed Windows-specific code(Bruce)
removed postmaster -e option, now only postgres -e option (Bruce)
merge duplicate library code in front/backends(Martin)
now works with eBones, international Kerberos(Jun)
more shared library support
c++ include file cleanup(Bruce)
warn about buggy flex(Bruce)
DG/UX, Ultrix, IRIX, AIX portability fixes
```

E.109. Release 6.0

Release date: 1997-01-29

A dump/restore is required for those wishing to migrate data from previous releases of PostgreSQL.

E.109.1. Migration from version 1.09 to version 6.0

This migration requires a complete dump of the 1.09 database and a restore of the database in 6.0.

E.109.2. Migration from pre-1.09 to version 6.0

Those migrating from earlier 1.* releases should first upgrade to 1.09 because the COPY output format was improved from the 1.02 release.

E.109.3. Changes

```
Bug Fixes
---------
ALTER TABLE bug - running postgres process needs to re-read table definition
Allow vacuum to be run on one table or entire database(Bruce)
Array fixes
Fix array over-runs of memory writes(Kurt)
Fix elusive btree range/non-range bug(Dan)
Fix for hash indexes on some types like time and date
Fix for pg_log size explosion
Fix permissions on lo_export()(Bruce)
Fix uninitialized reads of memory(Kurt)
Fixed ALTER TABLE ... char(3) bug(Bruce)
Fixed a few small memory leaks
Fixed EXPLAIN handling of options and changed full_path option name
Fixed output of group acl privileges
Memory leaks (hunt and destroy with tools like Purify(Kurt)
Minor improvements to rules system
NOTIFY fixes
New asserts for run-checking
Overhauled parser/analyze code to properly report errors and increase speed
Pg_dump -d now handles NULL's properly(Bruce)
Prevent SELECT NULL from crashing server (Bruce)
Properly report errors when INSERT ... SELECT columns did not match
Properly report errors when insert column names were not correct
psql \g filename now works(Bruce)
psql fixed problem with multiple statements on one line with multiple outputs
Removed duplicate system OIDs
SELECT * INTO TABLE . GROUP/ORDER BY gives unlink error if table exists(Bruce)
Several fixes for queries that crashed the backend
Starting quote in insert string errors(Bruce)
Submitting an empty query now returns empty status, not just " " query(Bruce)

Enhancements
------------
Add EXPLAIN manual page(Bruce)
Add UNIQUE index capability(Dan)
Add hostname/user level access control rather than just hostname and user
Add synonym of != for <>(Bruce)
Allow "select oid,* from table"
Allow BY,ORDER BY to specify columns by number, or by non-alias table.column(Bruce)
Allow COPY from the frontend(Bryan)
Allow GROUP BY to use alias column name(Bruce)
Allow actual compression, not just reuse on the same page(Vadim)
Allow installation-configuration option to auto-add all local users(Bryan)
```

```
Allow libpq to distinguish between text value '' and null(Bruce)
Allow non-postgres users with createdb privs to destroydb's
Allow restriction on who can create C functions(Bryan)
Allow restriction on who can do backend COPY(Bryan)
Can shrink tables, pg_time and pg_log(Vadim & Erich)
Change debug level 2 to print queries only, changed debug heading layout(Bruce)
Change default decimal constant representation from float4 to float8(Bruce)
European date format now set when postmaster is started
Execute lowercase function names if not found with exact case
Fixes for aggregate/GROUP processing, allow 'select sum(func(x),sum(x+y) from z'
Gist now included in the distribution(Marc)
Idend authentication of local users(Bryan)
Implement BETWEEN qualifier(Bruce)
Implement IN qualifier(Bruce)
libpq has PQgetisnull()(Bruce)
libpq++ improvements
New options to initdb(Bryan)
Pg_dump allow dump of OIDs(Bruce)
Pg_dump create indexes after tables are loaded for speed(Bruce)
Pg_dumpall dumps all databases, and the user table
Pginterface additions for NULL values(Bruce)
Prevent postmaster from being run as root
psql \h and \? is now readable(Bruce)
psql allow backslashed, semicolons anywhere on the line(Bruce)
psql changed command prompt for lines in query or in quotes(Bruce)
psql char(3) now displays as (bp)char in \d output(Bruce)
psql return code now more accurate(Bryan?)
psql updated help syntax(Bruce)
Re-visit and fix vacuum(Vadim)
Reduce size of regression diffs, remove timezone name difference(Bruce)
Remove compile-time parameters to enable binary distributions(Bryan)
Reverse meaning of HBA masks(Bryan)
Secure Authentication of local users(Bryan)
Speed up vacuum(Vadim)
Vacuum now had VERBOSE option(Bruce)

Source tree changes
-------------------
All functions now have prototypes that are compared against the calls
Allow asserts to be disabled easily from Makefile.global(Bruce)
Change oid constants used in code to #define names
Decoupled sparc and solaris defines(Kurt)
Gcc -Wall compiles cleanly with warnings only from unfixable constructs
Major include file reorganization/reduction(Marc)
Make now stops on compile failure(Bryan)
Makefile restructuring(Bryan, Marc)
Merge bsdi_2_1 to bsdi(Bruce)
Monitor program removed
Name change from Postgres95 to PostgreSQL
New config.h file(Marc, Bryan)
PG_VERSION now set to 6.0 and used by postmaster
Portability additions, including Ultrix, DG/UX, AIX, and Solaris
Reduced the number of #define's, centralized #define's
```

```
Remove duplicate OIDS in system tables(Dan)
Remove duplicate system catalog info or report mismatches(Dan)
Removed many os-specific #define's
Restructured object file generation/location(Bryan, Marc)
Restructured port-specific file locations(Bryan, Marc)
Unused/uninitialized variables corrected
```

E.110. Release 1.09

Release date: 1996-11-04

Sorry, we didn't keep track of changes from 1.02 to 1.09. Some of the changes listed in 6.0 were actually included in the 1.02.1 to 1.09 releases.

E.111. Release 1.02

Release date: 1996-08-01

E.111.1. Migration from version 1.02 to version 1.02.1

Here is a new migration file for 1.02.1. It includes the 'copy' change and a script to convert old ASCII files.

> **Note:** The following notes are for the benefit of users who want to migrate databases from Postgres95 1.01 and 1.02 to Postgres95 1.02.1.
>
> If you are starting afresh with Postgres95 1.02.1 and do not need to migrate old databases, you do not need to read any further.

In order to upgrade older Postgres95 version 1.01 or 1.02 databases to version 1.02.1, the following steps are required:

1. Start up a new 1.02.1 postmaster
2. Add the new built-in functions and operators of 1.02.1 to 1.01 or 1.02 databases. This is done by running the new 1.02.1 server against your own 1.01 or 1.02 database and applying the queries attached at the end of the file. This can be done easily through `psql`. If your 1.01 or 1.02 database is named `testdb` and you have cut the commands from the end of this file and saved them in `addfunc.sql`:

```
% psql testdb -f addfunc.sql
```

Those upgrading 1.02 databases will get a warning when executing the last two statements in the file because they are already present in 1.02. This is not a cause for concern.

E.111.2. Dump/Reload Procedure

If you are trying to reload a pg_dump or text-mode, `copy tablename to stdout` generated with a previous version, you will need to run the attached `sed` script on the ASCII file before loading it into the database. The old format used '.' as end-of-data, while '\.' is now the end-of-data marker. Also, empty strings are now loaded in as '' rather than NULL. See the copy manual page for full details.

```
sed 's/^\.$/\\./g' <in_file >out_file
```

If you are loading an older binary copy or non-stdout copy, there is no end-of-data character, and hence no conversion necessary.

```
-- following lines added by agc to reflect the case-insensitive
-- regexp searching for varchar (in 1.02), and bpchar (in 1.02.1)
create operator ~* (leftarg = bpchar, rightarg = text, procedure = texticregexeq);
create operator !~* (leftarg = bpchar, rightarg = text, procedure = texticregexne);
create operator ~* (leftarg = varchar, rightarg = text, procedure = texticregexeq);
create operator !~* (leftarg = varchar, rightarg = text, procedure = texticregexne);
```

E.111.3. Changes

```
Source code maintenance and development
 * worldwide team of volunteers
 * the source tree now in CVS at ftp.ki.net

Enhancements
 * psql (and underlying libpq library) now has many more options for
   formatting output, including HTML
 * pg_dump now output the schema and/or the data, with many fixes to
   enhance completeness.
 * psql used in place of monitor in administration shell scripts.
   monitor to be deprecated in next release.
 * date/time functions enhanced
 * NULL insert/update/comparison fixed/enhanced
 * TCL/TK lib and shell fixed to work with both tck7.4/tk4.0 and tcl7.5/tk4.1

Bug Fixes (almost too numerous to mention)
 * indexes
 * storage management
 * check for NULL pointer before dereferencing
 * Makefile fixes
```

```
New Ports
 * added SolarisX86 port
 * added BSD/OS 2.1 port
 * added DG/UX port
```

E.112. Release 1.01

Release date: 1996-02-23

E.112.1. Migration from version 1.0 to version 1.01

The following notes are for the benefit of users who want to migrate databases from Postgres95 1.0 to Postgres95 1.01.

If you are starting afresh with Postgres95 1.01 and do not need to migrate old databases, you do not need to read any further.

In order to Postgres95 version 1.01 with databases created with Postgres95 version 1.0, the following steps are required:

1. Set the definition of `NAMEDATALEN` in `src/Makefile.global` to 16 and `OIDNAMELEN` to 20.
2. Decide whether you want to use Host based authentication.

    a. If you do, you must create a file name `pg_hba` in your top-level data directory (typically the value of your `$PGDATA`). `src/libpq/pg_hba` shows an example syntax.

    b. If you do not want host-based authentication, you can comment out the line:

    ```
    HBA = 1
    ```

    in `src/Makefile.global`

    Note that host-based authentication is turned on by default, and if you do not take steps A or B above, the out-of-the-box 1.01 will not allow you to connect to 1.0 databases.
3. Compile and install 1.01, but DO NOT do the `initdb` step.
4. Before doing anything else, terminate your 1.0 postmaster, and backup your existing `$PGDATA` directory.
5. Set your `PGDATA` environment variable to your 1.0 databases, but set up path up so that 1.01 binaries are being used.
6. Modify the file `$PGDATA/PG_VERSION` from 5.0 to 5.1
7. Start up a new 1.01 postmaster

8. Add the new built-in functions and operators of 1.01 to 1.0 databases. This is done by running the new 1.01 server against your own 1.0 database and applying the queries attached and saving in the file 1.0_to_1.01.sql. This can be done easily through `psql`. If your 1.0 database is name `testdb`:

   ```
   % psql testdb -f 1.0_to_1.01.sql
   ```

   and then execute the following commands (cut and paste from here):

   ```
   -- add builtin functions that are new to 1.01

   create function int4eqoid (int4, oid) returns bool as 'foo'
   language 'internal';
   create function oideqint4 (oid, int4) returns bool as 'foo'
   language 'internal';
   create function char2icregexeq (char2, text) returns bool as 'foo'
   language 'internal';
   create function char2icregexne (char2, text) returns bool as 'foo'
   language 'internal';
   create function char4icregexeq (char4, text) returns bool as 'foo'
   language 'internal';
   create function char4icregexne (char4, text) returns bool as 'foo'
   language 'internal';
   create function char8icregexeq (char8, text) returns bool as 'foo'
   language 'internal';
   create function char8icregexne (char8, text) returns bool as 'foo'
   language 'internal';
   create function char16icregexeq (char16, text) returns bool as 'foo'
   language 'internal';
   create function char16icregexne (char16, text) returns bool as 'foo'
   language 'internal';
   create function texticregexeq (text, text) returns bool as 'foo'
   language 'internal';
   create function texticregexne (text, text) returns bool as 'foo'
   language 'internal';

   -- add builtin functions that are new to 1.01

   create operator = (leftarg = int4, rightarg = oid, procedure = int4eqoid);
   create operator = (leftarg = oid, rightarg = int4, procedure = oideqint4);
   create operator ~* (leftarg = char2, rightarg = text, procedure = char2icregexeq);
   create operator !~* (leftarg = char2, rightarg = text, procedure = char2icregexne);
   create operator ~* (leftarg = char4, rightarg = text, procedure = char4icregexeq);
   create operator !~* (leftarg = char4, rightarg = text, procedure = char4icregexne);
   create operator ~* (leftarg = char8, rightarg = text, procedure = char8icregexeq);
   create operator !~* (leftarg = char8, rightarg = text, procedure = char8icregexne);
   create operator ~* (leftarg = char16, rightarg = text, procedure = char16icregexeq);
   create operator !~* (leftarg = char16, rightarg = text, procedure = char16icregexne);
   create operator ~* (leftarg = text, rightarg = text, procedure = texticregexeq);
   create operator !~* (leftarg = text, rightarg = text, procedure = texticregexne);
   ```

E.112.2. Changes

```
Incompatibilities:
 * 1.01 is backwards compatible with 1.0 database provided the user
   follow the steps outlined in the MIGRATION_from_1.0_to_1.01 file.
   If those steps are not taken, 1.01 is not compatible with 1.0 database.

Enhancements:
 * added PQdisplayTuples() to libpq and changed monitor and psql to use it
 * added NeXT port (requires SysVIPC implementation)
 * added CAST .. AS ... syntax
 * added ASC and DESC key words
 * added 'internal' as a possible language for CREATE FUNCTION
   internal functions are C functions which have been statically linked
   into the postgres backend.
 * a new type "name" has been added for system identifiers (table names,
   attribute names, etc.)  This replaces the old char16 type.   The
   of name is set by the NAMEDATALEN #define in src/Makefile.global
 * a readable reference manual that describes the query language.
 * added host-based access control.  A configuration file ($PGDATA/pg_hba)
   is used to hold the configuration data.  If host-based access control
   is not desired, comment out HBA=1 in src/Makefile.global.
 * changed regex handling to be uniform use of Henry Spencer's regex code
   regardless of platform.  The regex code is included in the distribution
 * added functions and operators for case-insensitive regular expressions.
   The operators are ~* and !~*.
 * pg_dump uses COPY instead of SELECT loop for better performance

Bug fixes:
 * fixed an optimizer bug that was causing core dumps when
   functions calls were used in comparisons in the WHERE clause
 * changed all uses of getuid to geteuid so that effective uids are used
 * psql now returns non-zero status on errors when using -c
 * applied public patches 1-14
```

E.113. Release 1.0

Release date: 1995-09-05

E.113.1. Changes

```
Copyright change:
 * The copyright of Postgres 1.0 has been loosened to be freely modifiable
```

```
   and modifiable for any purpose.  Please read the COPYRIGHT file.
   Thanks to Professor Michael Stonebraker for making this possible.

Incompatibilities:
 *  date formats have to be MM-DD-YYYY (or DD-MM-YYYY if you're using
   EUROPEAN STYLE).  This follows SQL-92 specs.
 *  "delimiters" is now a key word

Enhancements:
 *  sql LIKE syntax has been added
 *  copy command now takes an optional USING DELIMITER specification.
   delimiters can be any single-character string.
 *  IRIX 5.3 port has been added.
   Thanks to Paul Walmsley and others.
 *  updated pg_dump to work with new libpq
 *  \d has been added psql
   Thanks to Keith Parks
 *  regexp performance for architectures that use POSIX regex has been
   improved due to caching of precompiled patterns.
   Thanks to Alistair Crooks
 *  a new version of libpq++
   Thanks to William Wanders

Bug fixes:
 *  arbitrary userids can be specified in the createuser script
 *  \c to connect to other databases in psql now works.
 *  bad pg_proc entry for float4inc() is fixed
 *  users with usecreatedb field set can now create databases without
   having to be usesuper
 *  remove access control entries when the entry no longer has any
   privileges
 *  fixed non-portable datetimes implementation
 *  added kerberos flags to the src/backend/Makefile
 *  libpq now works with kerberos
 *  typographic errors in the user manual have been corrected.
 *  btrees with multiple index never worked, now we tell you they don't
   work when you try to use them
```

E.114. Postgres95 Release 0.03

Release date: 1995-07-21

E.114.1. Changes

```
Incompatible changes:
 * BETA-0.3 IS INCOMPATIBLE WITH DATABASES CREATED WITH PREVIOUS VERSIONS
   (due to system catalog changes and indexing structure changes).
 * double-quote (") is deprecated as a quoting character for string literals;
   you need to convert them to single quotes (').
 * name of aggregates (eg. int4sum) are renamed in accordance with the
   SQL standard (eg. sum).
 * CHANGE ACL syntax is replaced by GRANT/REVOKE syntax.
 * float literals (eg. 3.14) are now of type float4 (instead of float8 in
   previous releases); you might have to do typecasting if you depend on it
   being of type float8.  If you neglect to do the typecasting and you assign
   a float literal to a field of type float8, you might get incorrect values
   stored!
 * LIBPQ has been totally revamped so that frontend applications
   can connect to multiple backends
 * the usesysid field in pg_user has been changed from int2 to int4 to
   allow wider range of Unix user ids.
 * the netbsd/freebsd/bsd o/s ports have been consolidated into a
   single BSD44_derived port.  (thanks to Alistair Crooks)

SQL standard-compliance (the following details changes that makes postgres95
more compliant to the SQL-92 standard):
 * the following SQL types are now built-in: smallint, int(eger), float, real,
   char(N), varchar(N), date and time.

   The following are aliases to existing postgres types:
               smallint -> int2
               integer, int -> int4
               float, real  -> float4
   char(N) and varchar(N) are implemented as truncated text types. In
   addition, char(N) does blank-padding.
 * single-quote (') is used for quoting string literals; '' (in addition to
   \') is supported as means of inserting a single quote in a string
 * SQL standard aggregate names (MAX, MIN, AVG, SUM, COUNT) are used
   (Also, aggregates can now be overloaded, i.e. you can define your
   own MAX aggregate to take in a user-defined type.)
 * CHANGE ACL removed. GRANT/REVOKE syntax added.
   - Privileges can be given to a group using the "GROUP" key word.
        For example:
                GRANT SELECT ON foobar TO GROUP my_group;
        The key word 'PUBLIC' is also supported to mean all users.

        Privileges can only be granted or revoked to one user or group
        at a time.

        "WITH GRANT OPTION" is not supported.  Only class owners can change
        access control
   - The default access control is to grant users readonly access.
     You must explicitly grant insert/update access to users.  To change
     this, modify the line in
```

```
                src/backend/utils/acl.h
     that defines ACL_WORLD_DEFAULT

Bug fixes:
 * the bug where aggregates of empty tables were not run has been fixed. Now,
   aggregates run on empty tables will return the initial conditions of the
   aggregates. Thus, COUNT of an empty  table will now properly return 0.
   MAX/MIN of an empty table will return a row of value NULL.
 * allow the use of \; inside the monitor
 * the LISTEN/NOTIFY asynchronous notification mechanism now work
 * NOTIFY in rule action bodies now work
 * hash indexes work, and access methods in general should perform better.
   creation of large btree indexes should be much faster.  (thanks to Paul
   Aoki)

Other changes and enhancements:
 * addition of an EXPLAIN statement used for explaining the query execution
   plan (eg. "EXPLAIN SELECT * FROM EMP" prints out the execution plan for
   the query).
 * WARN and NOTICE messages no longer have timestamps on them. To turn on
   timestamps of error messages, uncomment the line in
   src/backend/utils/elog.h:
        /* define ELOG_TIMESTAMPS */
 * On an access control violation, the message
        "Either no such class or insufficient privilege"
   will be given.  This is the same message that is returned when
   a class is not found.  This dissuades non-privileged users from
   guessing the existence of privileged classes.
 * some additional system catalog changes have been made that are not
   visible to the user.

libpgtcl changes:
 * The -oid option has been added to the "pg_result" tcl command.
   pg_result -oid returns oid of the last row inserted.   If the
   last command was not an INSERT, then pg_result -oid returns "".
 * the large object interface is available as pg_lo* tcl commands:
   pg_lo_open, pg_lo_close, pg_lo_creat, etc.

Portability enhancements and New Ports:
 * flex/lex problems have been cleared up.  Now, you should be able to use
   flex instead of lex on any platforms.  We no longer make assumptions of
   what lexer you use based on the platform you use.
 * The Linux-ELF port is now supported.  Various configuration have been
   tested:  The following configuration is known to work:
        kernel 1.2.10, gcc 2.6.3, libc 4.7.2, flex 2.5.2, bison 1.24
   with everything in ELF format,

New utilities:
 * ipcclean added to the distribution
   ipcclean usually does not need to be run, but if your backend crashes
   and leaves shared memory segments hanging around, ipcclean will
   clean them up for you.
```

```
New documentation:
 * the user manual has been revised and libpq documentation added.
```

E.115. Postgres95 Release 0.02

Release date: 1995-05-25

E.115.1. Changes

```
Incompatible changes:
 * The SQL statement for creating a database is 'CREATE DATABASE' instead
   of 'CREATEDB'. Similarly, dropping a database is 'DROP DATABASE' instead
   of 'DESTROYDB'. However, the names of the executables 'createdb' and
   'destroydb' remain the same.

New tools:
 * pgperl - a Perl (4.036) interface to Postgres95
 * pg_dump - a utility for dumping out a postgres database into a
        script file containing query commands. The script files are in a ASCII
        format and can be used to reconstruct the database, even on other
        machines and other architectures. (Also good for converting
        a Postgres 4.2 database to Postgres95 database.)

The following ports have been incorporated into postgres95-beta-0.02:
 * the NetBSD port by Alistair Crooks
 * the AIX port by Mike Tung
 * the Windows NT port by Jon Forrest (more stuff but not done yet)
 * the Linux ELF port by Brian Gallew

The following bugs have been fixed in postgres95-beta-0.02:
 * new lines not escaped in COPY OUT and problem with COPY OUT when first
   attribute is a '.'
 * cannot type return to use the default user id in createuser
 * SELECT DISTINCT on big tables crashes
 * Linux installation problems
 * monitor doesn't allow use of 'localhost' as PGHOST
 * psql core dumps when doing \c or \l
 * the "pgtclsh" target missing from src/bin/pgtclsh/Makefile
 * libpgtcl has a hard-wired default port number
 * SELECT DISTINCT INTO TABLE hangs
 * CREATE TYPE doesn't accept 'variable' as the internallength
 * wrong result using more than 1 aggregate in a SELECT
```

E.116. Postgres95 Release 0.01

Release date: 1995-05-01

Initial release.

Appendix F. The CVS Repository

The PostgreSQL source code is stored and managed using the CVS version control system.

At least two methods, anonymous CVS and CVSup, are available to pull the CVS code tree from the PostgreSQL server to your local machine.

F.1. Getting The Source Via Anonymous CVS

If you would like to keep up with the current sources on a regular basis, you can fetch them from our CVS server and then use CVS to retrieve updates from time to time.

Anonymous CVS

1. You will need a local copy of CVS (Concurrent Version Control System), which you can get from http://www.nongnu.org/cvs/ (the official site with the latest version) or any GNU software archive site (often somewhat outdated). Many systems have a recent version of cvs installed by default.

2. Do an initial login to the CVS server:

   ```
   cvs -d :pserver:anoncvs@anoncvs.postgresql.org:/projects/cvsroot login
   ```
   You will be prompted for a password; you can enter anything except an empty string.

   You should only need to do this once, since the password will be saved in `.cvspass` in your home directory.

3. Fetch the PostgreSQL sources:

   ```
   cvs -z3 -d :pserver:anoncvs@anoncvs.postgresql.org:/projects/cvsroot co -P pgsql
   ```
   This installs the PostgreSQL sources into a subdirectory `pgsql` of the directory you are currently in.

   Note: If you have a fast link to the Internet, you may not need `-z3`, which instructs CVS to use `gzip` compression for transferred data. But on a modem-speed link, it's a very substantial win.

   This initial checkout is a little slower than simply downloading a `tar.gz` file; expect it to take 40 minutes or so if you have a 28.8K modem. The advantage of CVS doesn't show up until you want to update the file set later on.

4. Whenever you want to update to the latest CVS sources, `cd` into the `pgsql` subdirectory, and issue

   ```
   cvs -z3 update -d -P
   ```
   This will fetch only the changes since the last time you updated. You can update in just a couple of minutes, typically, even over a modem-speed line.

5. You can save yourself some typing by making a file `.cvsrc` in your home directory that contains

```
cvs -z3
update -d -P
```

This supplies the `-z3` option to all `cvs` commands, and the `-d` and `-P` options to `cvs update`. Then you just have to say

```
cvs update
```

to update your files.

CVS can do a lot of other things, such as fetching prior revisions of the PostgreSQL sources rather than the latest development version. For more info consult the manual that comes with CVS, or see the online documentation at http://www.nongnu.org/cvs/.

F.2. CVS Tree Organization

Author: Written by Marc G. Fournier (<`scrappy@hub.org`>) on 1998-11-05

The command `cvs checkout` has a flag, `-r`, that lets you check out a certain revision of a module. This flag makes it easy to, for example, retrieve the sources that make up release 6_4 of the module 'tc' at any time in the future:

```
cvs checkout -r REL6_4 tc
```

This is useful, for instance, if someone claims that there is a bug in that release, but you cannot find the bug in the current working copy.

Tip: You can also check out a module as it was at any given date using the `-D` option.

When you tag more than one file with the same tag you can think about the tag as "a curve drawn through a matrix of file name vs. revision number". Say we have 5 files with the following revisions:

```
file1   file2   file3   file4   file5

1.1     1.1     1.1     1.1  /--1.1*      <-*-  TAG
1.2*-   1.2     1.2    -1.2*-
1.3  \- 1.3*-   1.3   / 1.3
1.4          \  1.4  /  1.4
              \-1.5*-   1.5
                1.6
```

then the tag `TAG` will reference file1-1.2, file2-1.3, etc.

Note: For creating a release branch, other than a `-b` option added to the command, it's the same thing.

So, to create the 6.4 release I did the following:

```
cd pgsql
cvs tag -b REL6_4
```

which will create the tag and the branch for the RELEASE tree.

For those with CVS access, it's simple to create directories for different versions. First, create two subdirectories, RELEASE and CURRENT, so that you don't mix up the two. Then do:

```
cd RELEASE
cvs checkout -P -r REL6_4 pgsql
cd ../CURRENT
cvs checkout -P pgsql
```

which results in two directory trees, `RELEASE/pgsql` and `CURRENT/pgsql`. From that point on, CVS will keep track of which repository branch is in which directory tree, and will allow independent updates of either tree.

If you are *only* working on the `CURRENT` source tree, you just do everything as before we started tagging release branches.

After you've done the initial checkout on a branch

```
cvs checkout -r REL6_4
```

anything you do within that directory structure is restricted to that branch. If you apply a patch to that directory structure and do a

```
cvs commit
```

while inside of it, the patch is applied to the branch and *only* the branch.

F.3. Getting The Source Via CVSup

An alternative to using anonymous CVS for retrieving the PostgreSQL source tree is CVSup. CVSup was developed by John Polstra (<jdp@polstra.com>) to distribute CVS repositories and other file trees for the FreeBSD project[1].

A major advantage to using CVSup is that it can reliably replicate the *entire* CVS repository on your local system, allowing fast local access to `cvs` operations such as `log` and `diff`. Other advantages include fast synchronization to the PostgreSQL server due to an efficient streaming transfer protocol which only sends the changes since the last update.

1. http://www.freebsd.org

F.3.1. Preparing A CVSup Client System

Two directory areas are required for CVSup to do its job: a local CVS repository (or simply a directory area if you are fetching a snapshot rather than a repository; see below) and a local CVSup bookkeeping area. These can coexist in the same directory tree.

Decide where you want to keep your local copy of the CVS repository. On one of our systems we recently set up a repository in `/home/cvs/`, but had formerly kept it under a PostgreSQL development tree in `/opt/postgres/cvs/`. If you intend to keep your repository in `/home/cvs/`, then put

```
setenv CVSROOT /home/cvs
```

in your `.cshrc` file, or a similar line in your `.bashrc` or `.profile` file, depending on your shell.

The cvs repository area must be initialized. Once `CVSROOT` is set, then this can be done with a single command:

```
cvs init
```

after which you should see at least a directory named `CVSROOT` when listing the `CVSROOT` directory:

```
$ ls $CVSROOT
CVSROOT/
```

F.3.2. Running a CVSup Client

Verify that cvsup is in your path; on most systems you can do this by typing

```
which cvsup
```

Then, simply run cvsup using:

```
cvsup -L 2 postgres.cvsup
```

where `-L 2` enables some status messages so you can monitor the progress of the update, and *`postgres.cvsup`* is the path and name you have given to your CVSup configuration file.

Here is a CVSup configuration file modified for a specific installation, and which maintains a full local CVS repository:

```
# This file represents the standard CVSup distribution file
# for the PostgreSQL ORDBMS project
# Modified by lockhart@fourpalms.org 1997-08-28
# - Point to my local snapshot source tree
# - Pull the full CVS repository, not just the latest snapshot
#
# Defaults that apply to all the collections
*default host=cvsup.postgresql.org
*default compress
*default release=cvs
*default delete use-rel-suffix
```

```
# enable the following line to get the latest snapshot
#*default tag=.
# enable the following line to get whatever was specified above or by default
# at the date specified below
#*default date=97.08.29.00.00.00

# base directory where CVSup will store its 'bookmarks' file(s)
# will create subdirectory sup/
#*default base=/opt/postgres # /usr/local/pgsql
*default base=/home/cvs

# prefix directory where CVSup will store the actual distribution(s)
*default prefix=/home/cvs

# complete distribution, including all below
pgsql

# individual distributions vs 'the whole thing'
# pgsql-doc
# pgsql-perl5
# pgsql-src
```

If you specify `repository` instead of `pgsql` in the above setup, you will get a complete copy of the entire repository at cvsup.postgresql.org, including its `CVSROOT` directory. If you do that, you will probably want to exclude those files in that directory that you want to modify locally, using a refuse file. For example, for the above setup you might put this in `/home/cvs/sup/repository/refuse`:

```
CVSROOT/config*
CVSROOT/commitinfo*
CVSROOT/loginfo*
```

See the CVSup manual pages for how to use refuse files.

The following is a suggested CVSup configuration file from the PostgreSQL ftp site[2] which will fetch the current snapshot only:

```
# This file represents the standard CVSup distribution file
# for the PostgreSQL ORDBMS project
#
# Defaults that apply to all the collections
*default host=cvsup.postgresql.org
*default compress
*default release=cvs
*default delete use-rel-suffix
*default tag=.

# base directory where CVSup will store its 'bookmarks' file(s)
*default base=/usr/local/pgsql

# prefix directory where CVSup will store the actual distribution(s)
```

2. ftp://ftp.postgresql.org/pub/CVSup/README.cvsup

```
*default prefix=/usr/local/pgsql

# complete distribution, including all below
pgsql

# individual distributions vs 'the whole thing'
# pgsql-doc
# pgsql-perl5
# pgsql-src
```

Appendix G. Documentation

PostgreSQL has four primary documentation formats:

- Plain text, for pre-installation information
- HTML, for on-line browsing and reference
- PDF or Postscript, for printing
- man pages, for quick reference.

Additionally, a number of plain-text `README` files can be found throughout the PostgreSQL source tree, documenting various implementation issues.

HTML documentation and man pages are part of a standard distribution and are installed by default. PDF and Postscript format documentation is available separately for download.

G.1. DocBook

The documentation sources are written in *DocBook*, which is a markup language superficially similar to HTML. Both of these languages are applications of the *Standard Generalized Markup Language*, SGML, which is essentially a language for describing other languages. In what follows, the terms DocBook and SGML are both used, but technically they are not interchangeable.

DocBook allows an author to specify the structure and content of a technical document without worrying about presentation details. A document style defines how that content is rendered into one of several final forms. DocBook is maintained by the OASIS group[1]. The official DocBook site[2] has good introductory and reference documentation and a complete O'Reilly book for your online reading pleasure. The NewbieDoc Docbook Guide[3] is very helpful for beginners. The FreeBSD Documentation Project[4] also uses DocBook and has some good information, including a number of style guidelines that might be worth considering.

G.2. Tool Sets

The following tools are used to process the documentation. Some may be optional, as noted.

DocBook DTD[5]

This is the definition of DocBook itself. We currently use version 4.2; you cannot use later or earlier

1. http://www.oasis-open.org
2. http://www.oasis-open.org/docbook
3. http://newbiedoc.sourceforge.net/metadoc/docbook-guide.html
4. http://www.freebsd.org/docproj/docproj.html
5. http://www.oasis-open.org/docbook/sgml/

versions. Note that there is also an XML version of DocBook — do not use that.

ISO 8879 character entities[6]

These are required by DocBook but are distributed separately because they are maintained by ISO.

OpenJade[7]

This is the base package of SGML processing. It contains an SGML parser, a DSSSL processor (that is, a program to convert SGML to other formats using DSSSL stylesheets), as well as a number of related tools. Jade is now being maintained by the OpenJade group, no longer by James Clark.

DocBook DSSSL Stylesheets[8]

These contain the processing instructions for converting the DocBook sources to other formats, such as HTML.

DocBook2X tools[9]

This optional package is used to create man pages. It has a number of prerequisite packages of its own. Check the web site.

JadeTeX[10]

If you want to, you can also install JadeTeX to use TeX as a formatting backend for Jade. JadeTeX can create Postscript or PDF files (the latter with bookmarks).

However, the output from JadeTeX is inferior to what you get from the RTF backend. Particular problem areas are tables and various artifacts of vertical and horizontal spacing. Also, there is no opportunity to manually polish the results.

We have documented experience with several installation methods for the various tools that are needed to process the documentation. These will be described below. There may be some other packaged distributions for these tools. Please report package status to the documentation mailing list, and we will include that information here.

G.2.1. Linux RPM Installation

Most vendors provide a complete RPM set for DocBook processing in their distribution. Look for an "SGML" option while installing, or the following packages: `sgml-common`, `docbook`, `stylesheets`, `openjade` (or `jade`). Possibly `sgml-tools` will be needed as well. If your distributor does not provide these then you should be able to make use of the packages from some other, reasonably compatible vendor.

G.2.2. FreeBSD Installation

The FreeBSD Documentation Project is itself a heavy user of DocBook, so it comes as no surprise that there is a full set of "ports" of the documentation tools available on FreeBSD. The following ports need

6. http://www.oasis-open.org/cover/ISOEnts.zip
7. http://openjade.sourceforge.net
8. http://docbook.sourceforge.net/projects/dsssl/index.html
9. http://docbook2x.sourceforge.net
10. http://jadetex.sourceforge.net

to be installed to build the documentation on FreeBSD.

- `textproc/sp`
- `textproc/openjade`
- `textproc/iso8879`
- `textproc/dsssl-docbook-modular`

Apparently, there is no port for the DocBook V4.2 SGML DTD available right now. You will need to install it manually.

A number of things from `/usr/ports/print` (`tex`, `jadetex`) might also be of interest.

It's possible that the ports do not update the main catalog file in `/usr/local/share/sgml/catalog`. Be sure to have the following line in there:

```
CATALOG "/usr/local/share/sgml/docbook/4.2/docbook.cat"
```

If you do not want to edit the file you can also set the environment variable `SGML_CATALOG_FILES` to a colon-separated list of catalog files (such as the one above).

More information about the FreeBSD documentation tools can be found in the FreeBSD Documentation Project's instructions[11].

G.2.3. Debian Packages

There is a full set of packages of the documentation tools available for Debian GNU/Linux. To install, simply use:

```
apt-get install jade
apt-get install docbook
apt-get install docbook-stylesheets
```

G.2.4. Manual Installation from Source

The manual installation process of the DocBook tools is somewhat complex, so if you have pre-built packages available, use them. We describe here only a standard setup, with reasonably standard installation paths, and no "fancy" features. For details, you should study the documentation of the respective package, and read SGML introductory material.

11. http://www.freebsd.org/doc/en_US.ISO8859-1/books/fdp-primer/tools.html

G.2.4.1. Installing OpenJade

1. The installation of OpenJade offers a GNU-style `./configure; make; make install` build process. Details can be found in the OpenJade source distribution. In a nutshell:

   ```
   ./configure --enable-default-catalog=/usr/local/share/sgml/catalog
   make
   make install
   ```

   Be sure to remember where you put the "default catalog"; you will need it below. You can also leave it off, but then you will have to set the environment variable `SGML_CATALOG_FILES` to point to the file whenever you use jade later on. (This method is also an option if OpenJade is already installed and you want to install the rest of the tool chain locally.)

2. Additionally, you should install the files `dsssl.dtd`, `fot.dtd`, `style-sheet.dtd`, and `catalog` from the `dsssl` directory somewhere, perhaps into `/usr/local/share/sgml/dsssl`. It's probably easiest to copy the entire directory:

   ```
   cp -R dsssl /usr/local/share/sgml
   ```

3. Finally, create the file `/usr/local/share/sgml/catalog` and add this line to it:

   ```
   CATALOG "dsssl/catalog"
   ```

   (This is a relative path reference to the file installed in step 2. Be sure to adjust it if you chose your installation layout differently.)

G.2.4.2. Installing the DocBook DTD Kit

1. Obtain the DocBook V4.2 distribution[12].

2. Create the directory `/usr/local/share/sgml/docbook-4.2` and change to it. (The exact location is irrelevant, but this one is reasonable within the layout we are following here.)

   ```
   $ mkdir /usr/local/share/sgml/docbook-4.2
   $ cd /usr/local/share/sgml/docbook-4.2
   ```

3. Unpack the archive.

   ```
   $ unzip -a ...../docbook-4.2.zip
   ```

   (The archive will unpack its files into the current directory.)

4. Edit the file `/usr/local/share/sgml/catalog` (or whatever you told jade during installation) and put a line like this into it:

   ```
   CATALOG "docbook-4.2/docbook.cat"
   ```

5. Download the ISO 8879 character entities archive[13], unpack it, and put the files in the same directory you put the DocBook files in.

   ```
   $ cd /usr/local/share/sgml/docbook-4.2
   $ unzip ...../ISOEnts.zip
   ```

6. Run the following command in the directory with the DocBook and ISO files:

   ```
   perl -pi -e 's/iso-(.*).gml/ISO\1/g' docbook.cat
   ```

12. http://www.docbook.org/sgml/4.2/docbook-4.2.zip
13. http://www.oasis-open.org/cover/ISOEnts.zip

(This fixes a mixup between the names used in the DocBook catalog file and the actual names of the ISO character entity files.)

G.2.4.3. Installing the DocBook DSSSL Style Sheets

To install the style sheets, unzip and untar the distribution and move it to a suitable place, for example `/usr/local/share/sgml`. (The archive will automatically create a subdirectory.)

```
$ gunzip docbook-dsssl-1.xx.tar.gz
$ tar -C /usr/local/share/sgml -xf docbook-dsssl-1.xx.tar
```

The usual catalog entry in `/usr/local/share/sgml/catalog` can also be made:

```
CATALOG "docbook-dsssl-1.xx/catalog"
```

Because stylesheets change rather often, and it's sometimes beneficial to try out alternative versions, PostgreSQL doesn't use this catalog entry. See Section G.2.5 for information about how to select the stylesheets instead.

G.2.4.4. Installing JadeTeX

To install and use JadeTeX, you will need a working installation of TeX and LaTeX2e, including the supported tools and graphics packages, Babel, AMS fonts and AMS-LaTeX, the PSNFSS extension and companion kit of "the 35 fonts", the dvips program for generating PostScript, the macro packages fancyhdr, hyperref, minitoc, url and ot2enc. All of these can be found on your friendly neighborhood CTAN site[14]. The installation of the TeX base system is far beyond the scope of this introduction. Binary packages should be available for any system that can run TeX.

Before you can use JadeTeX with the PostgreSQL documentation sources, you will need to increase the size of TeX's internal data structures. Details on this can be found in the JadeTeX installation instructions.

Once that is finished you can install JadeTeX:

```
$ gunzip jadetex-xxx.tar.gz
$ tar xf jadetex-xxx.tar
$ cd jadetex
$ make install
$ mktexlsr
```

The last two need to be done as root.

14. http://www.ctan.org

G.2.5. Detection by `configure`

Before you can build the documentation you need to run the `configure` script as you would when building the PostgreSQL programs themselves. Check the output near the end of the run, it should look something like this:

```
checking for onsgmls... onsgmls
checking for openjade... openjade
checking for DocBook V4.2... yes
checking for DocBook stylesheets... /usr/lib/sgml/stylesheets/nwalsh-modular
checking for sgmlspl... sgmlspl
```

If neither `onsgmls` nor `nsgmls` were found then you will not see the remaining 4 lines. `nsgmls` is part of the Jade package. You can pass the environment variables `JADE` and `NSGMLS` to configure to point to the programs if they are not found automatically. If "DocBook V4.2" was not found then you did not install the DocBook DTD kit in a place where Jade can find it, or you have not set up the catalog files correctly. See the installation hints above. The DocBook stylesheets are looked for in a number of relatively standard places, but if you have them some other place then you should set the environment variable `DOCBOOKSTYLE` to the location and rerun `configure` afterwards.

G.3. Building The Documentation

Once you have everything set up, change to the directory `doc/src/sgml` and run one of the commands described in the following subsections to build the documentation. (Remember to use GNU make.)

G.3.1. HTML

To build the HTML version of the documentation:

```
doc/src/sgml$ gmake html
```

This is also the default target.

When the HTML documentation is built, the process also generates the linking information for the index entries. Thus, if you want your documentation to have a concept index at the end, you need to build the HTML documentation once, and then build the documentation again in whatever format you like.

To allow for easier handling in the final distribution, the files comprising the HTML documentation are stored in a tar archive that is unpacked at installation time. To create the HTML documentation package, use the commands

```
cd doc/src
gmake postgres.tar.gz
```

In the distribution, these archives live in the `doc` directory and are installed by default with `gmake install`.

G.3.2. Manpages

We use the docbook2man utility to convert DocBook `refentry` pages to *roff output suitable for man pages. The man pages are also distributed as a tar archive, similar to the HTML version. To create the man page package, use the commands

```
cd doc/src
gmake man.tar.gz
```

which will result in a tar file being generated in the `doc/src` directory.

To generate quality man pages, it might be necessary to use a hacked version of the conversion utility or do some manual postprocessing. All man pages should be manually inspected before distribution.

G.3.3. Print Output via JadeTex

If you want to use JadeTex to produce a printable rendition of the documentation, you can use one of the following commands:

- To make a DVI version:

  ```
  doc/src/sgml$ gmake postgres.dvi
  ```

- To generate Postscript from the DVI:

  ```
  doc/src/sgml$ gmake postgres.ps
  ```

- To make a PDF:

  ```
  doc/src/sgml$ gmake postgres.pdf
  ```

  (Of course you can also make a PDF version from the Postscript, but if you generate PDF directly, it will have hyperlinks and other enhanced features.)

G.3.4. Print Output via RTF

You can also create a printable version of the PostgreSQL documentation by converting it to RTF and applying minor formatting corrections using an office suite. Depending on the capabilities of the particular office suite, you can then convert the documentation to Postscript of PDF. The procedure below illustrates this process using Applixware.

> **Note:** It appears that current versions of the PostgreSQL documentation trigger some bug in or exceed the size limit of OpenJade. If the build process of the RTF version hangs for a long time and the output file still has size 0, then you may have hit that problem. (But keep in mind that a normal build takes 5 to 10 minutes, so don't abort too soon.)

Applixware RTF Cleanup

OpenJade omits specifying a default style for body text. In the past, this undiagnosed problem led to a long process of table of contents generation. However, with great help from the Applixware folks the symptom was diagnosed and a workaround is available.

1. Generate the RTF version by typing:

   ```
   doc/src/sgml$ gmake postgres.rtf
   ```

2. Repair the RTF file to correctly specify all styles, in particular the default style. If the document contains `refentry` sections, one must also replace formatting hints which tie a preceding paragraph to the current paragraph, and instead tie the current paragraph to the following one. A utility, `fixrtf`, is available in `doc/src/sgml` to accomplish these repairs:

   ```
   doc/src/sgml$ ./fixrtf --refentry postgres.rtf
   ```

   The script adds `{\s0 Normal;}` as the zeroth style in the document. According to Applixware, the RTF standard would prohibit adding an implicit zeroth style, though Microsoft Word happens to handle this case. For repairing `refentry` sections, the script replaces `\keepn` tags with `\keep`.

3. Open a new document in Applixware Words and then import the RTF file.

4. Generate a new table of contents (ToC) using Applixware.

   a. Select the existing ToC lines, from the beginning of the first character on the first line to the last character of the last line.

   b. Build a new ToC using Tools⟶Book Building⟶Create Table of Contents. Select the first three levels of headers for inclusion in the ToC. This will replace the existing lines imported in the RTF with a native Applixware ToC.

   c. Adjust the ToC formatting by using Format⟶Style, selecting each of the three ToC styles, and adjusting the indents for `First` and `Left`. Use the following values:

Style	First Indent (inches)	Left Indent (inches)
TOC-Heading 1	0.4	0.4
TOC-Heading 2	0.8	0.8
TOC-Heading 3	1.2	1.2

5. Work through the document to:

   - Adjust page breaks.
   - Adjust table column widths.

6. Replace the right-justified page numbers in the Examples and Figures portions of the ToC with correct values. This only takes a few minutes.

7. Delete the index section from the document if it is empty.

8. Regenerate and adjust the table of contents.

a. Select the ToC field.

b. Select Tools⟶Book Building⟶Create Table of Contents.

c. Unbind the ToC by selecting Tools⟶Field Editing⟶Unprotect.

d. Delete the first line in the ToC, which is an entry for the ToC itself.

9. Save the document as native Applixware Words format to allow easier last minute editing later.

10. "Print" the document to a file in Postscript format.

G.3.5. Plain Text Files

Several files are distributed as plain text, for reading during the installation process. The `INSTALL` file corresponds to Chapter 14, with some minor changes to account for the different context. To recreate the file, change to the directory `doc/src/sgml` and enter **`gmake INSTALL`**. This will create a file `INSTALL.html` that can be saved as text with Netscape Navigator and put into the place of the existing file. Netscape seems to offer the best quality for HTML to text conversions (over lynx and w3m).

The file `HISTORY` can be created similarly, using the command **`gmake HISTORY`**. For the file `src/test/regress/README` the command is **`gmake regress_README`**.

G.3.6. Syntax Check

Building the documentation can take very long. But there is a method to just check the correct syntax of the documentation files, which only takes a few seconds:

```
doc/src/sgml$ gmake check
```

G.4. Documentation Authoring

SGML and DocBook do not suffer from an oversupply of open-source authoring tools. The most common tool set is the Emacs/XEmacs editor with appropriate editing mode. On some systems these tools are provided in a typical full installation.

G.4.1. Emacs/PSGML

PSGML is the most common and most powerful mode for editing SGML documents. When properly configured, it will allow you to use Emacs to insert tags and check markup consistency. You could use it for HTML as well. Check the PSGML web site[15] for downloads, installation instructions, and detailed documentation.

15. http://www.lysator.liu.se/projects/about_psgml.html

There is one important thing to note with PSGML: its author assumed that your main SGML DTD directory would be `/usr/local/lib/sgml`. If, as in the examples in this chapter, you use `/usr/local/share/sgml`, you have to compensate for this, either by setting `SGML_CATALOG_FILES` environment variable, or you can customize your PSGML installation (its manual tells you how).

Put the following in your `~/.emacs` environment file (adjusting the path names to be appropriate for your system):

```
; ********** for SGML mode (psgml)

(setq sgml-omittag t)
(setq sgml-shorttag t)
(setq sgml-minimize-attributes nil)
(setq sgml-always-quote-attributes t)
(setq sgml-indent-step 1)
(setq sgml-indent-data t)
(setq sgml-parent-document nil)
(setq sgml-default-dtd-file "./reference.ced")
(setq sgml-exposed-tags nil)
(setq sgml-catalog-files '("/usr/local/share/sgml/catalog"))
(setq sgml-ecat-files nil)

(autoload 'sgml-mode "psgml" "Major mode to edit SGML files." t )
```

and in the same file add an entry for SGML into the (existing) definition for `auto-mode-alist`:

```
(setq
  auto-mode-alist
  '(("\\.sgml$" . sgml-mode)
   ))
```

The PostgreSQL distribution includes a parsed DTD definitions file `reference.ced`. You may find that when using PSGML, a comfortable way of working with these separate files of book parts is to insert a proper `DOCTYPE` declaration while you're editing them. If you are working on this source, for instance, it is an appendix chapter, so you would specify the document as an "appendix" instance of a DocBook document by making the first line look like this:

```
<!DOCTYPE appendix PUBLIC "-//OASIS//DTD DocBook V4.2//EN">
```

This means that anything and everything that reads SGML will get it right, and I can verify the document with `nsgmls -s docguide.sgml`. (But you need to take out that line before building the entire documentation set.)

G.4.2. Other Emacs modes

GNU Emacs ships with a different SGML mode, which is not quite as powerful as PSGML, but it's less confusing and lighter weight. Also, it offers syntax highlighting (font lock), which can be very helpful.

Norm Walsh offers a major mode[16] specifically for DocBook which also has font-lock and a number of

16. http://nwalsh.com/emacs/docbookide/index.html

features to reduce typing.

G.5. Style Guide

G.5.1. Reference Pages

Reference pages should follow a standard layout. This allows users to find the desired information more quickly, and it also encourages writers to document all relevant aspects of a command. Consistency is not only desired among PostgreSQL reference pages, but also with reference pages provided by the operating system and other packages. Hence the following guidelines have been developed. They are for the most part consistent with similar guidelines established by various operating systems.

Reference pages that describe executable commands should contain the following sections, in this order. Sections that do not apply may be omitted. Additional top-level sections should only be used in special circumstances; often that information belongs in the "Usage" section.

Name

This section is generated automatically. It contains the command name and a half-sentence summary of its functionality.

Synopsis

This section contains the syntax diagram of the command. The synopsis should normally not list each command-line option; that is done below. Instead, list the major components of the command line, such as where input and output files go.

Description

Several paragraphs explaining what the command does.

Options

A list describing each command-line option. If there are a lot of options, subsections may be used.

Exit Status

If the program uses 0 for success and non-zero for failure, then you do not need to document it. If there is a meaning behind the different non-zero exit codes, list them here.

Usage

Describe any sublanguage or run-time interface of the program. If the program is not interactive, this section can usually be omitted. Otherwise, this section is a catch-all for describing run-time features. Use subsections if appropriate.

Environment

List all environment variables that the program might use. Try to be complete; even seemingly trivial variables like `SHELL` might be of interest to the user.

Files

List any files that the program might access implicitly. That is, do not list input and output files that were specified on the command line, but list configuration files, etc.

Diagnostics

> Explain any unusual output that the program might create. Refrain from listing every possible error message. This is a lot of work and has little use in practice. But if, say, the error messages have a standard format that the user can parse, this would be the place to explain it.

Notes

> Anything that doesn't fit elsewhere, but in particular bugs, implementation flaws, security considerations, compatibility issues.

Examples

> Examples

History

> If there were some major milestones in the history of the program, they might be listed here. Usually, this section can be omitted.

See Also

> Cross-references, listed in the following order: other PostgreSQL command reference pages, PostgreSQL SQL command reference pages, citation of PostgreSQL manuals, other reference pages (e.g., operating system, other packages), other documentation. Items in the same group are listed alphabetically.

Reference pages describing SQL commands should contain the following sections: Name, Synopsis, Description, Parameters, Outputs, Notes, Examples, Compatibility, History, See Also. The Parameters section is like the Options section, but there is more freedom about which clauses of the command can be listed. The Outputs section is only needed if the command returns something other than a default command-completion tag. The Compatibility section should explain to what extent this command conforms to the SQL standard(s), or to which other database system it is compatible. The See Also section of SQL commands should list SQL commands before cross-references to programs.

Appendix H. External Projects

PostgreSQL is a complex software project, and managing the project is difficult. We have found that many enhancements to PostgreSQL can be more efficiently developed separately from the core project.

To help our community with the development of their external projects, we have created PgFoundry[1], a website that provides hosting for PostgreSQL-related projects that are maintained outside the core PostgreSQL distribution. PgFoundry is built using the GForge software project and is similar to SourceForge.net[2] in its feature set, providing mailing lists, forums, bug tracking, CVS, and web hosting. If you have a PostgreSQL-related open source project that you would like to have hosted at PgFoundy, please feel free to create a new project.

> **Note:** Many PostgreSQL-related projects are still hosted at GBorg[3]. GBorg is the original external community developer site, and while it is currently closed to new projects in favor of PgFoundry, it still contains many active and relevant projects. Other popular PostgreSQL-related projects are hosted independently, or on other project-hosting sites such as SourceForge.net[4]. You should search the web if you don't find the project you are looking for.

H.1. Client Interfaces

There are only two client interfaces included in the base PostgreSQL distribution:

- libpq is included because it is the primary C language interface, and because many other client interfaces are built on top of it.
- ecpg is included because it depends on the server-side SQL grammar, and is therefore sensitive to changes in PostgreSQL itself.

All other language interfaces are external projects and are distributed separately. Table H-1 includes a list of some of these projects. Note that some of these packages may not be released under the same license as PostgreSQL. For more information on each language interface, including licensing terms, refer to its website and documentation.

Table H-1. Externally Maintained Client Interfaces

Name	Language	Comments	Website
DBD::Pg	Perl	Perl DBI driver	http://search.cpan.org/dist/DBD-Pg/

1. http://www.pgfoundry.org/
2. http://sourceforge.net
3. http://gborg.postgresql.org/
4. http://sourceforge.net/

Name	Language	Comments	Website
JDBC	JDBC	Type 4 JDBC driver	http://jdbc.postgresql.org/
libpqxx	C++	New-style C++ interface	http://thaiopensource.org/development/libp
libpq++	C++	Old-style C++ interface	http://gborg.postgresql.org/project/libpqpp/
Npgsql	.NET	.NET data provider	http://pgfoundry.org/projects/npgsql/
ODBCng	ODBC	An alternative ODBC driver	http://projects.commandprompt.com/public
pgtclng	Tcl		http://pgfoundry.org/projects/pgtclng/
psqlODBC	ODBC	The most commonly-used ODBC driver	http://odbc.postgresql.org/
psycopg	Python	DB API 2.0-compliant	http://www.initd.org/

H.2. Procedural Languages

PostgreSQL includes several procedural languages with the base distribution: PL/PgSQL, PL/Tcl, PL/Perl, and PL/Python.

In addition, there are a number of procedural languages that are developed and maintained outside the core PostgreSQL distribution. Table H-2 lists some of these packages. Note that some of these projects may not be released under the same license as PostgreSQL. For more information on each procedural language, including licensing information, refer to its website and documentation.

Table H-2. Externally Maintained Procedural Languages

Name	Language	Website
PL/Java	Java	http://pljava.projects.postgresql.org/
PL/PHP	PHP	http://www.commandprompt.com/community/plphp
PL/Py	Python	http://python.projects.postgresql.org/
PL/R	R	http://www.joeconway.com/plr/
PL/Ruby	Ruby	http://raa.ruby-lang.org/project/pl-ruby/
PL/Scheme	Scheme	http://plscheme.projects.postgresql.org/

Name	Language	Website
PL/sh	Unix shell	http://plsh.projects.postgresql.org/

H.3. Extensions

PostgreSQL is designed to be easily extensible. For this reason, extensions loaded into the database can function just like features that are packaged with the database. The `contrib/` directory shipped with the source code contains a large number of extensions. The `README` file in that directory contains a summary. They include conversion tools, full-text indexing, XML tools, and additional data types and indexing methods. Other extensions are developed independently, like PostGIS[5]. Even PostgreSQL replication solutions are developed externally. For example, Slony-I[6] is a popular master/slave replication solution that is developed independently from the core project.

There are several administration tools available for PostgreSQL. The most popular is pgAdmin III[7], and there are several commercially available ones as well.

5. http://www.postgis.org/
6. http://www.slony.info
7. http://www.pgadmin.org/

Bibliography

Selected references and readings for SQL and PostgreSQL.

Some white papers and technical reports from the original POSTGRES development team are available at the University of California, Berkeley, Computer Science Department web site[1].

SQL Reference Books

Judith Bowman, Sandra Emerson, and Marcy Darnovsky, *The Practical SQL Handbook: Using SQL Variants*, Fourth Edition, Addison-Wesley Professional, ISBN 0-201-70309-2, 2001.

C. J. Date and Hugh Darwen, *A Guide to the SQL Standard: A user's guide to the standard database language SQL*, Fourth Edition, Addison-Wesley, ISBN 0-201-96426-0, 1997.

C. J. Date, *An Introduction to Database Systems*, Eighth Edition, Addison-Wesley, ISBN 0-321-19784-4, 2003.

Ramez Elmasri and Shamkant Navathe, *Fundamentals of Database Systems*, Fourth Edition, Addison-Wesley, ISBN 0-321-12226-7, 2003.

Jim Melton and Alan R. Simon, *Understanding the New SQL: A complete guide*, Morgan Kaufmann, ISBN 1-55860-245-3, 1993.

Jeffrey D. Ullman, *Principles of Database and Knowledge: Base Systems*, Volume 1, Computer Science Press, 1988.

PostgreSQL-Specific Documentation

Stefan Simkovics, *Enhancement of the ANSI SQL Implementation of PostgreSQL*, Department of Information Systems, Vienna University of Technology, November 29, 1998.

Discusses SQL history and syntax, and describes the addition of `INTERSECT` and `EXCEPT` constructs into PostgreSQL. Prepared as a Master's Thesis with the support of O. Univ. Prof. Dr. Georg Gottlob and Univ. Ass. Mag. Katrin Seyr at Vienna University of Technology.

A. Yu and J. Chen, The POSTGRES Group, *The Postgres95 User Manual*, University of California, Sept. 5, 1995.

1. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/

Zelaine Fong, *The design and implementation of the POSTGRES query optimizer* [2], University of California, Berkeley, Computer Science Department.

Proceedings and Articles

Nels Olson, *Partial indexing in POSTGRES: research project*, University of California, UCB Engin T7.49.1993 O676, 1993.

L. Ong and J. Goh, "A Unified Framework for Version Modeling Using Production Rules in a Database System", *ERL Technical Memorandum M90/33*, University of California, April, 1990.

L. Rowe and M. Stonebraker, " The POSTGRES data model [3]", Proc. VLDB Conference, Sept. 1987.

P. Seshadri and A. Swami, "Generalized Partial Indexes (cached version) [4] ", Proc. Eleventh International Conference on Data Engineering, 6-10 March 1995, IEEE Computer Society Press, Cat. No.95CH35724, 1995, 420-7.

M. Stonebraker and L. Rowe, " The design of POSTGRES [5]", Proc. ACM-SIGMOD Conference on Management of Data, May 1986.

M. Stonebraker, E. Hanson, and C. H. Hong, "The design of the POSTGRES rules system", Proc. IEEE Conference on Data Engineering, Feb. 1987.

M. Stonebraker, " The design of the POSTGRES storage system [6]", Proc. VLDB Conference, Sept. 1987.

M. Stonebraker, M. Hearst, and S. Potamianos, " A commentary on the POSTGRES rules system [7]", *SIGMOD Record 18(3)*, Sept. 1989.

M. Stonebraker, " The case for partial indexes [8]", *SIGMOD Record 18(4)*, Dec. 1989, 4-11.

M. Stonebraker, L. A. Rowe, and M. Hirohama, " The implementation of POSTGRES [9]", *Transactions on Knowledge and Data Engineering 2(1)*, IEEE, March 1990.

M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos, " On Rules, Procedures, Caching and Views in Database Systems [10]", Proc. ACM-SIGMOD Conference on Management of Data, June 1990.

2. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/UCB-MS-zfong.pdf
3. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M87-13.pdf
4. http://citeseer.ist.psu.edu/seshadri95generalized.html
5. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M85-95.pdf
6. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M87-06.pdf
7. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M89-82.pdf
8. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M89-17.pdf
9. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M90-34.pdf
10. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M90-36.pdf

Index