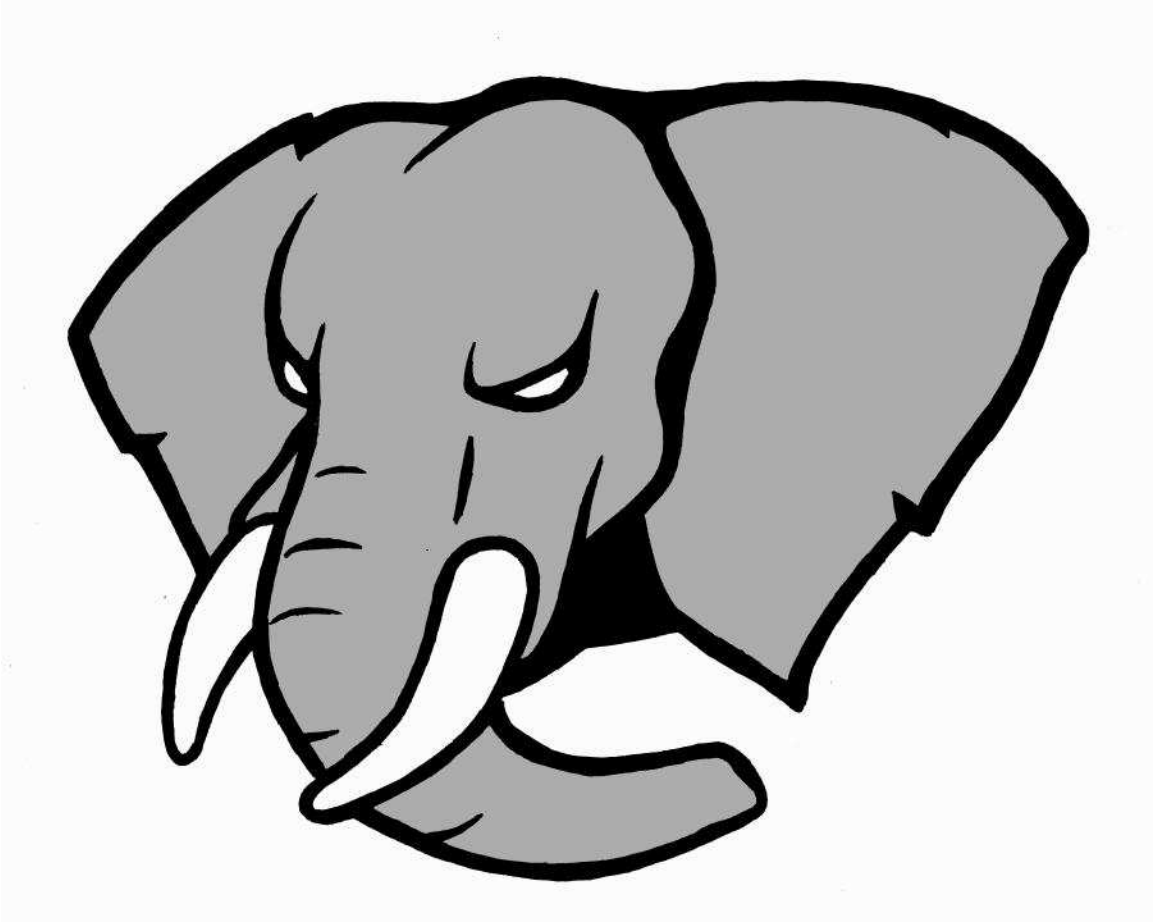


Slony-I
A replication system for PostgreSQL

Concept

Jan Wieck

Afilias USA INC.
Horsham, Pennsylvania, USA

ABSTRACT

This document describes the design goals and technical outline of the implementation of Slony-I, the first member of a new replication solutions family for the PostgreSQL ORDBMS.

Table of Contents

dummy
. 1

1. Design goals

This chapter gives a brief overview about the principle design goals that will be met in final product.

The *big picture* for the development of Slony-I is to build a master-slave system that includes all features and capabilities needed to replicate large databases to a reasonably limited number of slave systems. The analysis of existing replication systems for PostgreSQL has shown that it is literally impossible to add a fundamental feature to an existing replication system if that feature was not planned in the initial design.

The core capabilites defined in this chapter might not all get fully implemented in the first release. They however need to be an integral part of the metadata and administrative structures of the system to be added later with minimal impact to a running system.

The number of different replication solutions available supports the theory that *"one size fits all"* is not true when it comes to database replication. Slony-I is planned as a system for data centers and backup sites, where the normal mode of operation is that all nodes are available. Extended periods of downtime will require to remove or deactivate the node in question in the configuration. Neither offline nodes that only become available sporadic for synchronization (the salesman on the road) nor multimaster or synchronous replication will be supported and are subject to a future member of the Slony family.

1.1. Master to multiple cascaded slaves

The basic structure of the systems combined in a Slony-I installation is a master with one or more slaves nodes. Not all slave nodes must receive the replication data directly from the master. Every node that receives the data from a valid source can be configured to be able to forward that data to other nodes.

There are three distinct ideas behind this capability. The first is scalability. One database, especially the master that receives all the update transactions from the client applications, has only a limited capability to satisfy the slave nodes queries during the replication process. In order to satisfy the need for a big number of read-only slave systems it must be possible to cascade.

The second idea is to limit the required network bandwidth for a backup site while keeping the ability to have multiple slaves at the remote location.

The third idea is to be able to configure failover scenarios. In a master to multiple slave configuration, it is unlikely that all slave nodes are exactly in the same synchronization status when the master fails. To ensure that one slave can be promoted to the master it is necessary that all remaining systems can agree on the status of the data. Since a committed transaction cannot be rolled back, this status is undoubtly the most recent sync status of all remaining slave nodes. The delta between this one and every other node must be easily and fast generated and applied at least to the new master (if that's not the same system) before the promotion can occur.

1.2. Hot installation and configuration

It must be possible to install and uninstall the entire replication system on a running production database system without stopping the client application. This includes creating the initial configuration on the master system, configuring one or more slaves, copying the data and catching up to a full running master-slave status.

Changing the configuration also includes that a cascaded slave node can change its data provider on the fly. Especially for the failover scenario mentioned in the former section it is important to have the ability to promote one of the first level slaves to the master, redirect the other first level slaves to replicate from the new master and lower the workload on the new master by redirecting some or all of its cascaded slaves to replicate from another first level slave.

Hot installation and configuration change is further the only way to guarantee the ability to upgrade the replication software itself to a new version that is incompatible with the existing one in its metadata.

Even if this is given, upgrading the slaves will not work without interrupting the slave. What will be provided at least is the ability to install a new version in parallel to the old one, so that a new slave can be created and started before an existing one gets removed from the system.

1.3. Database schema changes

Replicating schema changes is an often discussed problem and only very few database systems provide the necessary hooks to implement it. PostgreSQL does not provide the ability to define triggers called on schema changes, so a transparent way to replicate schema changes is not possible without substantial work in the core PostgreSQL system.

Moreover, very often database schema changes are not single, isolated DDL statements that can occur at any time within a running system. Instead they tend to be groups of DDL and DML statements that modify multiple database objects and do mass data manipulation like updating a new column to its initial value.

The Slony-I replication system will have a mechanism to execute SQL scripts in a controlled fashion as part of the replication process.

1.4. Multiple database versions

To aid in the process of upgrading from one database version to another, the system must be able to replicate between different PostgreSQL versions.

A database upgrade of the master must be doable by failing over to a slave. A pure asynchronous master slave system like Slony-I will never be able to provide the ability to failover with zero transaction loss. True failover with zero loss of committed transactions is only possible with synchronous replication and will not be supported by Slony-I. Therefore, this administrative forced failover for the purpose of changing the master will need brief interruption of the client application to let the slave system catch up and become the master before the client resumes

work, now against the promoted new master.

1.5. Backup and point in time recovery

It is not necessarily obvious why backup and recovery is a topic for a replication system. The reason why it is subject to the design of Slony-I is that the PostgreSQL database system lacks any point in time recovery and a system design that covers failover would be incomplete without covering an application fault corrupting the data.

The technical design presented later in this document will make it relatively easy to use one or more slave systems for backup purposes. In addition it will be possible to configure single slaves with or without cascaded slaves to apply replication data after a delay. In high availability scenarios there is usually no time to restore a backup and do a point in time recovery. The affordable backup media are just not fast enough. A slave that applies the replication data with a 1 hour delay can be promoted to the master at logically any point in time within the past 60 minutes. Provided at least one other node (the master or any other node that does not replicate with a delay) has the log information for the last hour and is available, the backup node can be instructed to catchup until a specific point in time and then be promoted to the master. Assuming that the node can replicate faster than the master was able to work (how does it keep up otherwise), this would take less time than the delay it had.

2. Technical overview

This chapter explains the components and the logical operation of Slony-I.

2.1. Nodes, Sets and forwarding

The Slony-I replication system can replicate tables and sequence numbers. Replicating sequence numbers is not unproblematic and is discussed in more detail in section

Table and sequence objects are logically grouped into sets. Every set should contain a group of objects that is independant from other objects originating from the same master. In short, all tables that have relationships that could be expressed as foreign key constraints and all the sequences used to generate any serial numbers in these tables should be contained in one and the same set.

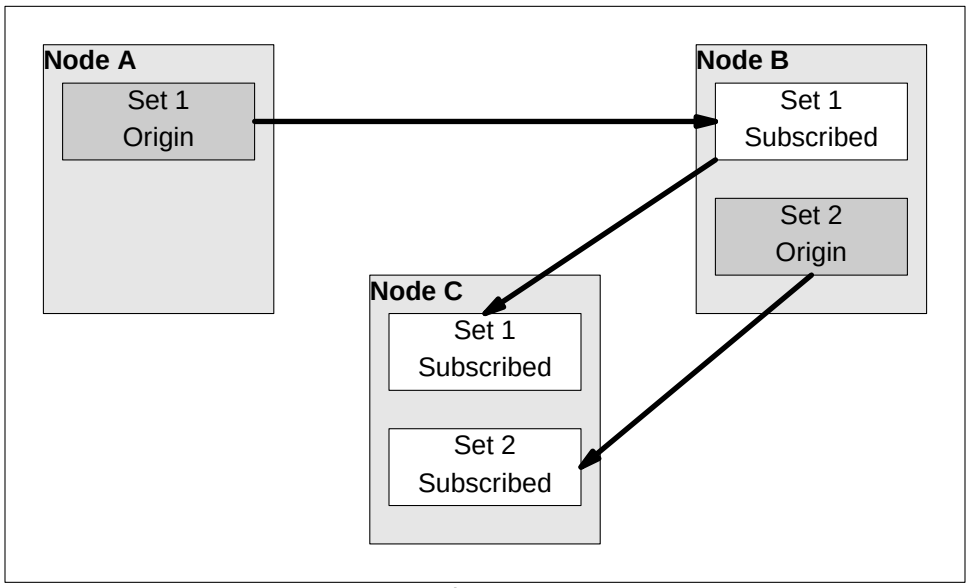


Figure 1

Figure 1 illustrates a replication configuration that has 2 data sets with different origins. To replicate both data sets to NodeC it is not required that Node C really communicates with the origin of Set 1. This scenario has full redundancy for every node. Obviously if Node C fails, the masters of Set 1 and Set2 are still alive, no problem. If Node A fails, Node B can get promoted to the master of both sets. The tricky situation is if Node B fails.

In the case Node B fails, Node C needs to get promoted to the master of Set 2 and it must continue replicating Set 1 from Node A. For that to be possible, Node A must have knowledge about Node C and its subscription to Set 1. Generally speaking, every node that stores replication log information must keep it until all subscribers of the affected set are known to have replicated that data.

To simplify the logic, the configuration of the whole network with all nodes, sets and subscriptions will be forwarded to and stored on all nodes. Because the sets, a node is not subscribed to must not even exist in its database, this does not include the information about what tables and sequences are included in any specific set.

2.2. Logging database activity

Slony-I will be an AFTER ROW trigger based replication system that analyses the NEW and OLD rows to reconstruct the meaningful pieces of an SQL statement representing the change to the actual data row. To identify a row in the log, the table must have some UNIQUE constraint. This can be a compound key of any data types. If there is none at all, the Slony-I installation process needs to add an int8 column to the table. Unmodified fields in an UPDATE event will not be included in the statement. Some analysis of existing replication methods has shown that despite the increase of log information that must be stored during replication cycles, this technology has several advantages over a system that holds information about which application tables need to be replicated, but will

fetch the latest value at the time of replication from the current row.

Stability: There are possible duplicate key conflicts that are not easy solvable when losing history information. The simplest case to demonstrate is a unique field where two rows swap their value like

```
UPDATE table SET col = 'temp' WHERE col = 'A';
UPDATE table SET col = 'A' WHERE col = 'B';
UPDATE table SET col = 'B' WHERE col = 'temp';
```

Without doing the extra step over the 'temp' value, there is no order in which the replication engine can replicate these updates.

Splitting: Slony-I will split the entire amount of replication activity into smaller units covering a few seconds of workload as described in section

This will be done on the visibility boundaries of two serializable transactions. So the slave systems will leap from one consistent state to another as if multiple master transactions would have been done at once. Without history information this is not possible and the slave only has the chance to jump from its last sync point to now. If it was stopped for a while for whatever reason, it must catch up in one big transaction covering the whole work done on the master in the meantime, increasing the duplicate key risk mentioned above.

The point in time standby capability via delayed application of replication data, described in 1.5., needs this splitting as well.

Failover: While it is relatively easy to tell in a master to multiple slave scenario which of the slaves is most recent at the time the master fails, it is nearly impossible to tell the actual row delta between two slaves. So in the case of a failing master, one slave can be promoted to the master, but all other slaves need to be re-synchronized with the new master.

Performance:

Storing the logging information in one or very few rotating log tables means that the replication engine can retrieve the actual data for one replication step with very few queries that select from one table only. In contrast to that a system that fetches the current values from the application tables at replication time needs to issue the same number of queries **per replicated table** and these queries will be joining the log table(s) with the application data table. It is obvious that this systems performance will be reverse proportional to the number of replicated tables. At some time the complete delta to be applied, which can not be split as pointed out already, will cause the PostgreSQL database system to require less optimal than in memory hash join query plans to deal with the number of rows returned by these queries and the replication system will be unable to ever catch up unless the workload on the master drops significantly.

The log will under normal circumstances be collected in one log table, deleted from there periodically and the table vacuumed (see section). A

reasonably large table with sufficient freespace has a better performance on INSERT operations than an empty table that gets only extended at the end. This is because the free space handling in PostgreSQL allows multiple backends to simultaneously add new tuples to different blocks. Also extending a table at the end is more expensive than reusing existing blocks as those blocks can never be found in the cache and need filesystem metadata changes in the OS due to increasing the file size. A log switching mechanism to another table will be provided for the case that a log table had once grown out of reasonable size, so that it is possible to shrink it without doing a VACUUM FULL which would cause an exclusive lock on the table, effectively stopping the client application.

Each log row will contain the current transaction ID, the local node ID, the affected table ID, a log action sequence number and the information required to reconstruct the SQL statement that can cause the same modification on a slave system. Since the action sequence is allocated in an AFTER ROW trigger, its ascending order is automatically an order that is not in conflict with the order in which concurrent updates happened to the base tables. It is not necessarily the exact same order in which the updates really occured, and it is for sure not the order in which those updates became visible or in other words their transactions committed. But statements executed in this order within logically ascending groups of transactions, grouped by the order in which they became visible, will lead to the exact same result. This order is called agreeable order.

2.3. Replicating sequences

Sequence number generators in PostgreSQL are highly optimized for concurrency. Because of that they only guarantee not to generate duplicate ID's. They do not roll back and can therefore generate gaps. Another problem is that triggers cannot be defined on sequence numbers.

Since sequences in PostgreSQL are 64 bit integers, it would be quite possible to split the entire available number range into multiple segments and assign each node that will eventually be promoted to the master its own unique range. This way, sequences can be simply ignored during the replication process. The drawback is that they cannot be ignored in the backup/restore process and the risk of restoring the wrong backup without re- adjusting the sequences is high.

Another possibility is to use a user defined function and effectively replace sequences by a row held in a replicated table, destroying thus the concurrency and making sequences a major bottleneck in the entire client application.

Yet another approach seen is not to replicate sequences, but to adjust them at the time a slave would be promoted to master. This requires at least one full table scan on every table that contains sequence generated values and can mean a significant delay in the failover process.

The approach Slony-I will take is a different one. The standard function that generates sequence numbers, *nextval()*, as well as *setval()*, will be moved out of the way by creating a new pg_proc catalog entry with another name and Oid for it. Their places will be taken by new custom functions that will call the original nextval() or setval() function and then check the configuration table if the

sequence is replicated. In the case of sequence replication, the function will insert a replication action row into the log table. Since no updates are ever done to the log table and the cleanup process only removes log entries that are in the past, this will not block concurrent transactions from allocating sequences. The fact that an aborted transaction will loose the allocated sequence can be ignored because it will be skipped on the next allocation anyway.

The slave must be carefull during the replication not to adjust the sequence number backwards, because the side effect that guarantees the agreeable order of action record sequences, the row lock on the applications table, does not exist for sequences. The allocation of sequence numbers happens logically at a time even before a BEFORE ROW trigger would fire and inside of our replacement nextval() function there is a race condition (the gap between calling the original nextval() and inserting the log record) that we do not want to serialize for concurrency reasons.

2.4. The node daemon

In Slony-I every database that participates in a replication system is a node. Databases need not necessarily reside on different servers or even be served by different postmasters. Two different databases are two different nodes.

For each database in the replication system, a node daemon called **Slon** is started. This daemon is the replication engine itself and consists of one hybrid program with master and slave functionality. The differentiation between master and slave is not really appropriate in Slony-I anyway since the role of a node is only defined on the set level, not on the database level. Slon has the following duties.

2.4.1. Splitting the logdata

Splitting the logdata into groups of logically ascending transactions is much easier than someone might imagine. The Slony-I daemon will check in a configurable timeout if the log action sequence number of the local node has changed and if so, it will generate a SYNC event. All events generated by a system are generated in a serializable transaction and lock one object. It is thus guaranteed that their event sequence is the exact order in which they are generated and committed.

An event contains among the message code and its payload information the entire serializable snapshot information of the transaction, that created this event. All transactions that committed between any two ascending SYNC events can thus be defined as

```
SELECT xid FROM logtable
    WHERE (xid > sync1_maxxid OR
          (xid >= sync1_minxid AND xid IN (sync1_xip)))
    AND   (xid < sync2_minxid OR
          (xid <= sync2_maxxid AND xid NOT IN (sync2_xip)));
```

The real query used in the activity described in section

is far more complicated. Yet the general principle is this simple and after all, the daemon on the local node only checks the local log action sequence, inserts a row and generates a notification if the sequence has changed.

2.4.2. Exchanging messages

All configuration changes like adding nodes, subscribing or unsubscribing sets, adding a table to a set and so forth are communicated through the system as events. An event is generated by inserting the event information into a table and notifying all listeners on the same. SYNC messages are communicated with the same mechanism.

The Slony-I system configuration contains information for every node which other it will query for which events.

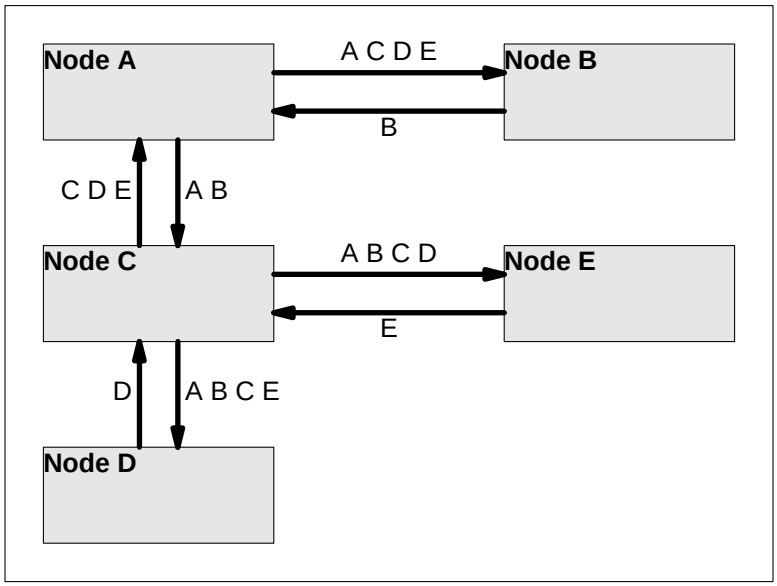


Figure 2

Figure 2 illustrates the event flow in a configuration with 5 nodes, where direct connections only exist between the following combinations of nodes.

NodeA <-> NodeB
NodeA <-> NodeC
NodeC <-> NodeD
NodeC <-> NodeE

Every daemon establishes remote database connections to the nodes, from where it receives events (which as shown in figure 2 is not necessarily the event origin). The daemons use the PostgreSQL LISTEN/NOTIFY mechanism to inform each other about event generation.

When receiving a new event, the daemon processes it and in the same transaction, inserts it into its own event table. This way the event gets forwarded and it is guaranteed, that all required data is stored and available on the forwarding node when the event arrives on the next receiver in the chain.

The fact that an event generated on node D or E will travel a while before it is seen by node B is good. Events including SYNC messages are only important for any node if it is subscribed to any set that originates on the same node, the event originates from.

We assume a data set originating on node A that is currently subscribed on nodes B and C, both with forwarding enabled. This data set now should be subscribed by node D. The actual subscribe event must be generated on node A, the origin of the data set, and travel within the flow of SYNC events to all subscribers of the set. Otherwise, node B and C would not know at which logical point in time node D subscribed the set and would not know that they need to keep replication data for possible forwarding to D. When node D receives the event by looking at node C's event queue, it is guaranteed that C has processed all replication deltas until the SYNC event prior to this subscribe event and that C currently knows that D possibly needs all following delta's resulting from future SYNC events.

Likewise will node B receive the subscribe message at the same logical point in time within the event flow and know, that it from this moment on has to keep delta information for the case that node C might fail at any time, even before it would be able to provide the current data snapshot or even the subscribe message itself to D and D would be reconfigured to talk to B as a substitute provider.

As a side note, the configuration in figure 2 with a set originating on node A is the very setup the author used during the development of the prototype. The entire configuration can be installed and started while node A is constantly online and write accessed by an application.

2.4.3. Confirming events

The majority of event types are configuration changes. The only exceptions are SYNC and SUBSCRIBE events covered more detailed in sections

and

Configuration change events carry all necessary information to modify the local configuration information in the event data row. Processing consists more or less of storing or deleting a row in one of the Slony-I control tables.

In the same transaction the local node daemon processes the event, he will insert a confirmation row into a local table that matches the events origin, the event sequence number and the local node ID.

Reverse to the event delivery mechanism, the daemon will now insert the same confirmation row into the confirmation table of every remote node it is connected to, and NOTIFY on that table. The remote node daemon will LISTEN on that table, pick up any new confirmation rows and propagate them through the network. This way, all nodes in the cluster will get to know that the local node has successfully processed the event.

2.4.4. Cleaning up

So far we have generated may events, confirmations and (hopefully) even more transaction log data. Needless to say that we need to get rid of all that after a while. Periodically the node daemon will clean up the event, confirm and log tables. This is done in two steps.

1. The confirmation data is condensed. Since all nodes process all events per origin in ascending order, we only need the row with the highest event sequence number per <origin,receiver>.
2. Old event and log data is removed. As we will see in section

   we need to keep the last SYNC event per origin. Thus we select the SYNC event with the smallest event sequence per origin, that is not yet confirmed by all other nodes in the cluster and loop over that result set. Per SYNC found we remove all older events from that origin and all log data from that origin that would be visible according to the snapshot information in the SYNC.

For the case that large volumes of log data once accumulated a log switching mechanism will be provided on a per node base. This is required since the only other way to reclaim the disk space would be a full vacuum, which grabs an exclusive lock on the table, thus effectively stopping the client application. After entering the switching mode, the triggers and functions inserting into the log table will start using an alterate table. While the node is in the switching mode, the log data is logically the union between the two log tables. When the cleanup process detects that the old log table is empty, it ends the log switching mode, waits until all transactions that could possibly have seen the system in switching mode have ended and truncates the old log table.

2.4.5. Replicating data

Upon receiving a remote SYNC the node checks if it is actually subscribed to any set originating on the node that generated the event. If it is not, it simply confirms the event like any other and is done with it. All other nodes do not need to keep the log data (at least not for this node) because it will never ask for log information prior to this SYNC event.

If it is subscribed to one or more sets from that origin, the actual replication works in the following steps.

1. The node checks that it has connections to all remote nodes that provide forward information for any set that is subscribed from the SYNC events origin.

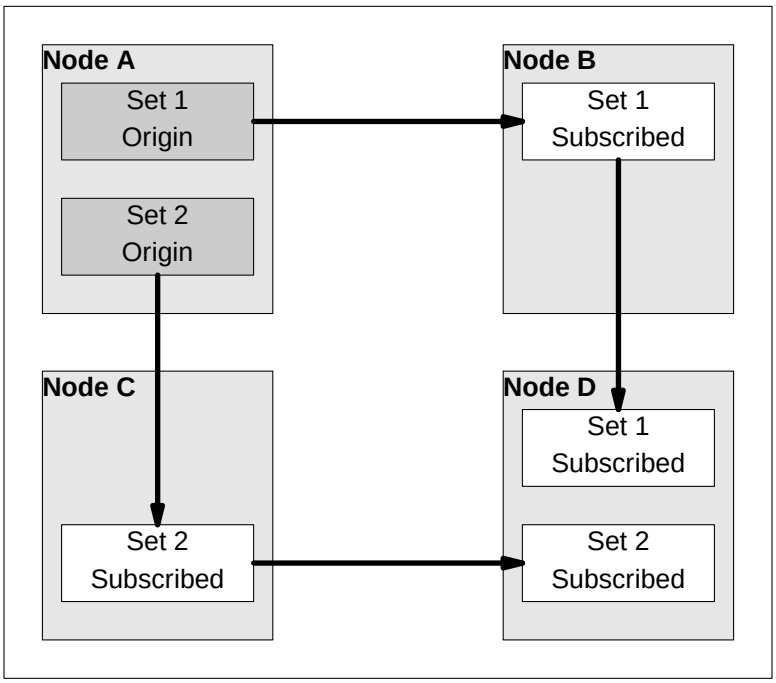


Figure 3

Figure 3 illustrates a scenario where node B is configured to replicate only set 1. Likewise is node C configured to replicate only set 2. For reporting purposes node D is subscribed to both sets, but to keep the workload on the primary node A as low as possible, it replicates set 1 from node B and set 2 from node C.

Despite of this distributed data path, the SYNC event generated on node A is meant for both sets and all the log data for both sets that has accumulated since the last SYNC event must be applied to node D in one transaction. Thus, node D can only proceed and start replicating if both nodes have already finished applying the SYNC event.

2. What the node daemon does now consists logically of selecting a union of the active log table of every remote node providing any set from the SYNC events origin in log action sequence order. The data selected is restricted to the tables contained in all the sets provided by the specific node and constrained to lay between the last and the actual SYNC event. In the example of figure 3, node D would query node B like

```
SELECT * FROM log
    WHERE log_origin = id_of_node A
    AND   log_tableid IN (list_of_tables_in_set_1)
    AND   (log_xid > last_maxxid OR
          (log_xid >= last_minxid
          AND log_xid IN (last_xip)))
    AND   (log_xid < sync_minxid OR
          (log_xid <= sync_maxxid
          AND log_xid NOT IN (sync_xip)))
    ORDER BY log_origin, log_actionseq;
```

Well, at least for theory starters. In practice because of the subscribe process it will be an OR'd list of those qualifications per set, and during the log switching of the queried node it will do this whole thing on a union between both log tables. Fortunately PostgreSQL has a sufficiently mature query optimizer to recognize that this is still an index scan along the origin and actionseq of the log table that does not need sorting.

3. All these remote result sets are now merged on the replicating node and applied to the local database. Since they are coming in correct sorted, the node can merge them on the fly with a one row lookahead. Triggers defined on any replicated table will be disabled during the entire SYNC processing. If there is a trigger defined on a table, it would be defined on the same table on the set origin as well. All the actions performed by that trigger, as long as they are actions that affect replicated tables, will get replicated as well. So there is no need to execute the trigger on the slave again and depending on the trigger code, it could even lead to inconsistencies between the master and the slave.
4. The SYNC event that caused all this trouble is stored as usual, the local transaction committed and the confirmation sent out as for all other events.

2.4.6. Subscribing a set

Subscribing to a set is an operation that must be initiated at the origin of the set. This is because Slony-I allows subscribing to sets that are actually in use on their origin, the application is concurrently modifying the sets data. For larger data sets it will take a while to create a snapshot copy of the data, and during that time all nodes that are possible replication providers for the set must know that there will be a new subscriber maybe asking for log data in the future. Generating the SUBSCRIBE event on the sets origin guarantees that every node will receive this event between the same two SYNC events coming from the origin of the set. So they will all start preserving the log data at the same point.

SUBSCRIBE events are a little special in that they must be received directly from the node that is the log data provider for the set. This is because the log data provider is the node from which the new subscriber will copy the initial snapshot as well.

When the SUBSCRIBE event is received from the correct node, the exact procedure how to subscribe depends on whether the log data provider is the sets origin so the new subscriber is a first level slave, or if is with respect to the set a forwarding slave and the new node cascades from that.

1. For all tables that are in the set, the slave will query the table configuration and store it locally. It will also create the replication trigger on all these tables.
2. All triggers on the tables in the set get disabled to speed up the data copy process and to avoid possible foreign key conflicts resulting from copying the data in the wrong order or because of circular dependencies.
3. For each table it will use the PostgreSQL command COPY on both sides and forward the data stream.
4. The triggers get restored.

5a. If the node we copied the data from is another slave (cascading), we have just copied the entire set in exactly the state at the last visible SYNC event from the sets origin inside of our current transaction. Whatever happened after we started copying the set is invisible to this transaction yet. So the local sets SYNC status is remembered as that and we are done.

5b. If the node we received the initial copy from is the sets origin, the problem is that the set data does not "leap" from one SYNC point to another. In this case we need to use the last SYNC event before the SUBSCRIBE event we are currently processing plus all action sequences that we already see after that last SYNC. We have copied the data rows with those actions applied already, so when later on processing the next SYNC event, we have to explicitly filter them out. This only applies to the first SYNC event that gets created after subscribing to a new set directly from its origin.

6. As usual, the SUBSCRIBE event is stored local, the transaction committed and the event processing confirmed.

2.4.7. Store and archive

In order to be able to cascade, the log data merged and applied in 2.4.5. must also be stored in the local log data table. Since this happens in the same transaction as inserting the SYNC event the log data was resulting from, every cascading slave that receives this data will be able to see it exactly when he receives the SYNC event, provided that the SYNC event was delivered by the provider. The log data will get cleaned up together with eventually local generated log data for sets originating on this node. The process described in 2.4.4. covers this already.

In addition to the cascading through store and forward, Slony-I will also be able to provide a backup and point in time recovery mechanism. The local node daemon knows exactly what the current SYNC status of its node is and it has the ability to delay the replication of the next SYNC status long enough to start a pg_dump and ensure that it has created its serializable transaction snapshot. The resulting dump will be an exact representation of the database at the time

the last SYNC event got committed locally. If it writes out files containing the same queries that get applied for all subsequent SYNC events, these files together will build a backup that can be restored with the same granularity as SYNC events are generated on the master.

2.4.8. Provider change and failover

To store the log data on a node so configured until all nodes that subscribe the set have confirmed the corresponding SYNC events is the basis for on-the-fly provider changes and failover.

Changing the log data provider means nothing else than starting at some arbitrary point in time (of course triggered and communicated with an event, what else) to select the log data in 2.4.5. from another node that is either the master or a slave that does store the data.

Failover is not much more than a logical sequence of syncing with other nodes, changing the origin of sets and finally a provider change with a twist.

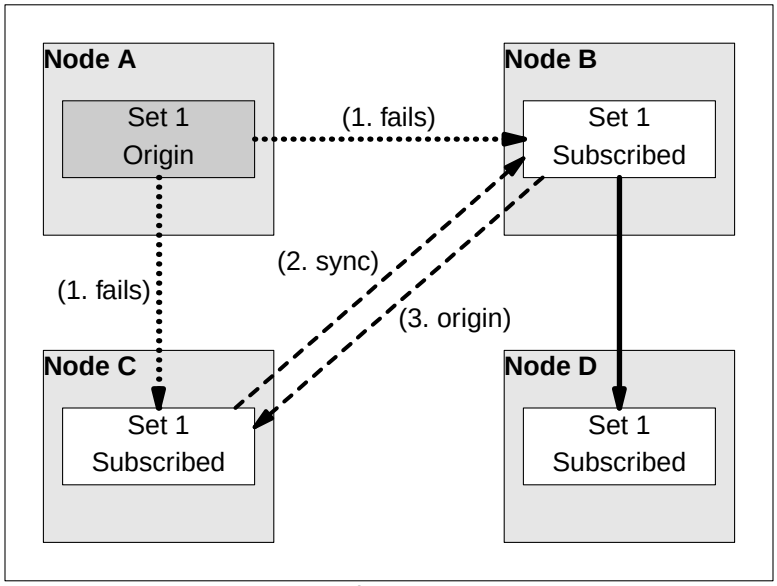


Figure 3

1. Node A in figure 4 fails. It is the current origin of the data set 1. The plan is to promote node B to the master and let node C continue to replicate against the new master.
2. Since it is possible that node C at that time is more advanced in the replication than node B, node B first asks for every event (and the corresponding log deltas for SYNC events) that it does not have itself yet. There is no real difference in this action than replicating against node A.
3. At the time Node B is for sure equally or more advanced than Node C, it takes over the set (becoming the origin). The twist in the provider change that node C now has to do is, that until now it is not guaranteed that node C has replicated all SYNC events from node A, that have been known to node B. Thus, the ORIGIN event from node B will contain the last node A event

known by node B at that time, which must be the last node A event known to the cluster at all. The twist in processing that ORIGIN event on node C is, that it cannot be confirmed until node C has replicated all events from node A until the one mentioned in the ORIGIN. At that time of course node C is free to either continue to replicate using node B or D as its provider.

The whole failover process looks relatively simple at this point because it is so simple. The entire Slony-I design pointed from the beginning into this direction, so it's no real surprise. However, this simplicity comes at a price. The price is, that if a (slave) node becomes unavailable, all other nodes in the cluster stop cleaning up and accumulate event information and possibly log data. So it is important that if a node becomes unavailable for a longer time, to change the configuration and let the system know that other techniques will be used to reactivate it. This can be done by suspending (deactivating) the node logically, or by removing it from the configuration completely.

For a deactivated node there is still hope to catch up with the rest of the cluster without re-joining from scratch. The point in time recovery delta files created in 2.4.7. can be used to feed it information that has been removed from the log tables long ago. When the node is finished replaying that it is reactivated, causing everyone else in the cluster to keep new log information again for the reactivated node. The reactivated node now again replays delta log files, eventually waiting for more to appear, until the one corresponding to the last known SYNC event before its reactivation appears. It is back online now.

3. Acknowledgements

Some of the core principles of Slony-I are taken from another replication solution that has been contributed to the PostgreSQL project. Namely the splitting of the continuous stream of log information at a transaction boundary compatible with the serializable isolation level and the idea to be able to switch log tables and how to do it exist very similar in eRServer, contributed by PostgreSQL INC.