Slony-I 2.1.1 Documentation

Legal Notice

COLLABORATORS			
	TITLE : Slony-I 2.1.1 Documentation		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Christopher Browne	January 25, 2012	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

List of Figures

List of Tables

Chapter 1

Preface

1.1 Introduction to Slony-I

1.1.1 What Is Slony-I

Slony-I is a 'master to multiple slaves' replication system for PostgreSQL supporting cascading and slave promotion. Key features of Slony-I include:

- Slony-I can replicate data between different PostgreSQL major versions
- Slony-I can replicate data between different hardware or operating systems
- Slony-I allows you to only replicate some of the tables to slave
- Slony-I allows you to replicate some tables to one slave and other tables to another slave
- Slony-I allows different database servers to be the origin(master) for different tables

1.1.2 About This Book

This book is intended as an administration guide and reference for Slony-I version 2.1.1 If you are using a different version of Slony-I then you should refer to the administration guide for that version.

1.2 System Requirements

To run Slony-I you will need

- PostgreSQL 8.3 or above (this version of Slony-I is known to work with 8.3.x, 8.4.x and 9.0.x). Earlier versions of PostgreSQL require Slony-I 1.2.x
- The Slony-I binary files either compiled from source or from a binary package

The following are recommended for running Slony-I

- A method of keeping the clocks on your replicas reasonably in sync such as NTP

  Also, it is preferable to use an consistent, stable time zone such as UTC or GMT.

  Users have run into problems with slon(1) functioning properly when their system uses a time zone that PostgreSQL was unable to recognize such as CUT0 or WST. It is necessary that you use a timezone that PostgreSQL can recognize correctly. It is furthermore preferable to use a time zone where times do not have discontinuities due to Daylight Savings Time.

  Most of Slony-I does not directly reference or use times, but if clocks are out of sync between servers running Slony-I components, confusion may be expected in the following places:

- The monitoring view `sl_status` uses timestamps sourced from multiple servers.
- Monitoring table sl_components captures timestamps based on the clock time on the host running slon(1).
- slon(1) logs are likely to contain timestamps.

Figuring out what is going on is likely to be made rather confusing if the database servers and servers where slon(1) instances run do not agree on what time it its.

- A reliable network between nodes

  slon(1) processes should run in the same 'network context' as the node that each is responsible for managing so that the connection to that node is a 'local' one. Do *not* run such links across a WAN. Thus, if you have nodes in London and nodes in New York, the slon(1)s managing London nodes should run in London, and the slon(1)s managing New York nodes should run in New York.

  A WAN outage (or flakiness of the WAN in general) can leave database connections 'zombied', and typical TCP/IP behaviour will allow those connections to persist, preventing a slon restart for around two hours.

  It is not difficult to remedy this; you need only **kill SIGINT** the offending backend connection. But by running the slon(1) locally, you will generally not be vulnerable to this condition.

- All of your databases should be using the same database encoding. (ie LATIN1 or UTF8). Replicating from a database in one encoding to a database with a different encoding might lead to replication errors.

1.2.1 Requirements for compiling Slony-I

In order to compile Slony-I, you need to have the following tools:

- GNU make. Other make programs will not work. GNU make is often installed under the name **gmake**; this document will therefore always refer to it by that name. (On Linux-based systems GNU make is typically the default make, and is called **make**)
- You need an ISO/ANSI C compiler. Recent versions of GCC work.
- You also need a recent version of PostgreSQL including the server headers. You must have PostgreSQL version 8.3 or newer to be able to build and use Slony-I.
- This documentation is written in SGML using DocBook , and may be processed into numerous formats including HTML, RTF, and PDF using tools in the DocBook Open Repository along with OpenJade.
- On Windows™ you will also need the same MinGW/Msys Toolset used to build PostgreSQL 8.3 and above. In addition you will need to install pthreads-win32 2.x.

1.2.2 Getting Slony-I Source

You can get the Slony-I source from http://main.slony.info/downloads/

1.3 Slony-I Concepts

In order to set up a set of Slony-I replicas, it is necessary to understand the following major abstractions that it uses:

- Cluster
- Node
- Replication Set
- Origin, Providers and Subscribers
- slon daemons

- slonik configuration processor

It is also worth knowing the meanings of certain Russian words:

- slon is Russian for ‘elephant’
- slony is the plural of slon, and therefore refers to a group of elephants
- slonik is Russian for ‘little elephant’

The use of these terms in Slony-I is a ‘tip of the hat’ to Vadim Mikheev, who was responsible for the rserv prototype which inspired some of the algorithms used in Slony-I.

1.3.1 Cluster

In Slony-I terms, a ‘cluster’ is a named set of PostgreSQL database instances; replication takes place between those databases.

The cluster name is specified in each and every Slonik script via the directive:

```
cluster name = something;
```

If the Cluster name is `something`, then Slony-I will create, in each database instance in the cluster, the namespace/schema `_something`.

1.3.2 Node

A Slony-I Node is a named PostgreSQL database that will be participating in replication.

It is defined, near the beginning of each Slonik script, using the directive:

```
NODE 1 ADMIN CONNINFO = 'dbname=testdb host=server1 user=slony';
```

The SLONIK ADMIN CONNINFO(7) information indicates database connection information that will ultimately be passed to the `PQconnectdb()` libpq function.

Thus, a Slony-I cluster consists of:

- A cluster name
- A set of Slony-I nodes, each of which has a namespace based on that cluster name

1.3.3 Replication Set

A replication set is defined as a set of tables and sequences that are to be replicated between nodes in a Slony-I cluster.

You may have several sets, and the ‘flow’ of replication does not need to be identical between those sets.

1.3.4 Origin, Providers and Subscribers

Each replication set has some origin node, which is the *only* place where user applications are permitted to modify data in the tables that are being replicated. This might also be termed the ‘master provider’; it is the main place from which data is provided.

Other nodes in the cluster subscribe to the replication set, indicating that they want to receive the data.

The origin node will never be considered a ‘subscriber.’ (Ignoring the case where the cluster is reshaped, and the origin is expressly shifted to another node.) But Slony-I supports the notion of cascaded subscriptions, that is, a node that is subscribed to some set may also behave as a ‘provider’ to other nodes in the cluster for that replication set.

1.3.5 slon Daemon

For each node in the cluster, there will be a slon(1) process to manage replication activity for that node.

slon(1) is a program implemented in C that processes replication events. There are two main sorts of events:

- Configuration events

  These normally occur when a slonik(1) script is run, and submit updates to the configuration of the cluster.

- **SYNC** events

  Updates to the tables that are replicated are grouped together into **SYNC**s; these groups of transactions are applied together to the subscriber nodes.

1.3.6 slonik Configuration Processor

The slonik(1) command processor processes scripts in a 'little language' that are used to submit events to update the configuration of a Slony-I cluster. This includes such things as adding and removing nodes, modifying communications paths, adding or removing subscriptions.

1.3.7 Slony-I Path Communications

Slony-I uses PostgreSQL DSNs in three contexts to establish access to databases:

- SLONIK ADMIN CONNINFO(7) - controlling how a slonik(1) script accesses the various nodes.

  These connections are the ones that go from your 'administrative workstation' to all of the nodes in a Slony-I cluster.

  It is *vital* that you have connections from the central location where you run slonik(1) to each and every node in the network. These connections are only used briefly, to submit the few SQL requests required to control the administration of the cluster.

  Since these communications paths are only used briefly, it may be quite reasonable to 'hack together' temporary connections using SSH tunnelling.

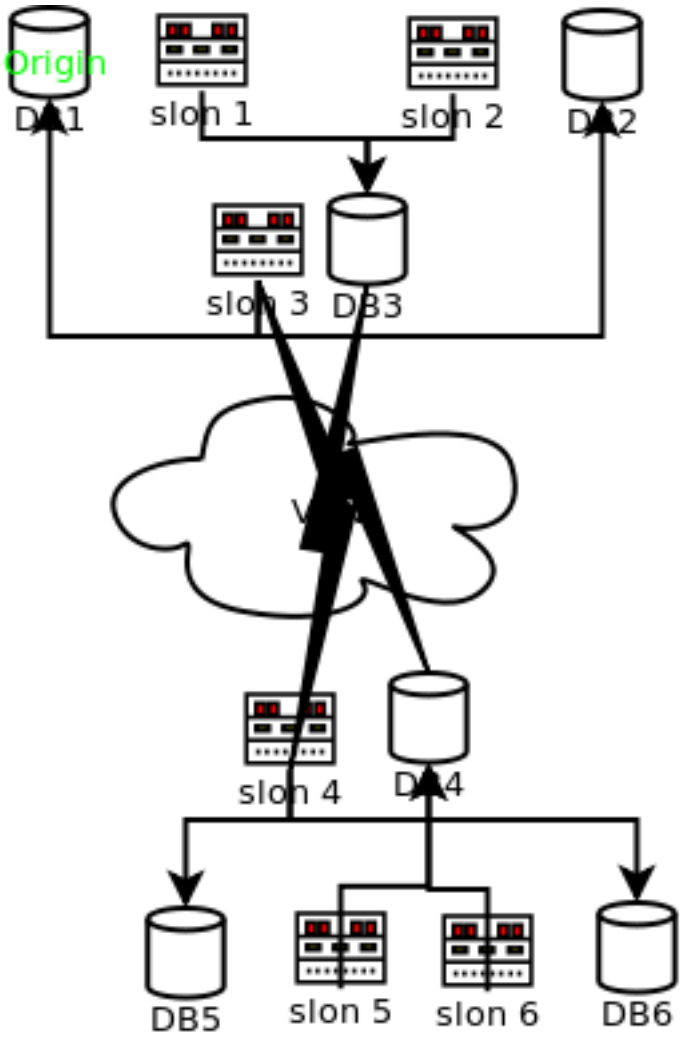
- The slon(1) DSN parameter.

  The DSN parameter passed to each slon(1) indicates what network path should be used to get from the slon(1) process to the database that it manages.

- SLONIK STORE PATH(7) - controlling how slon(1) daemons communicate with remote nodes. These paths are stored in sl_path.

  You forcibly *need* to have a path between each subscriber node and its provider; other paths are optional, and will not be used unless a listen path in sl_listen. is needed that uses that particular path.

The distinctions and possible complexities of paths are not normally an issue for people with simple networks where all the hosts can see one another via a comparatively 'global' set of network addresses. In contrast, it matters rather a lot for those with complex firewall configurations, nodes at multiple locations, and the issue where nodes may not be able to all talk to one another via a uniform set of network addresses.



Consider the attached diagram, which describes a set of six nodes

- DB1 and DB2 are databases residing in a secure 'database layer,' firewalled against outside access except from specifically controlled locations.

  Let's suppose that DB1 is the origin node for the replication system.

- DB3 resides in a 'DMZ' at the same site; it is intended to be used as a Slony-I 'provider' for remote locations.

- DB4 is a counterpart to DB3 in a 'DMZ' at a secondary/failover site. Its job, in the present configuration, is to 'feed' servers in the secure database layers at the secondary site.

- DB5 and DB6 are counterparts to DB1 and DB2, but are, at present, configured as subscribers.

  Supposing disaster were to strike at the 'primary' site, the secondary site would be well-equipped to take over servicing the applications that use this data.

- The symmetry of the configuration means that if you had *two* transactional applications needing protection from failure, it would be straightforward to have additional replication sets so that each site is normally 'primary' for one application, and where destruction of one site could be addressed by consolidating services at the remaining site.

1.3.8 SSH tunnelling

If a direct connection to PostgreSQL can not be established because of a firewall then you can establish an ssh tunnel that Slony-I can operate over.

SSH tunnels can be configured by passing the `w` to SSH. This enables forwarding PostgreSQL traffic where a local port is forwarded across a connection, encrypted and compressed, using SSH

See the ssh documentation for details on how to configure and use SSH tunnels.

1.4 Current Limitations

Slony-I does not automatically replicate

- Changes to large objects (BLOBS)
- Changes made by DDL commands
- Changes to users and roles

The main reason for these limitations is that Slony-I collects updates using triggers, and neither schema changes nor large object operations are captured by triggers

There is a capability for Slony-I to propagate notably DDL changes if you submit them as scripts via the slonik SLONIK EXECUTE SCRIPT(7) operation. That is not handled 'automatically;' you, as a database administrator, will have to construct an SQL DDL script and submit it, via SLONIK EXECUTE SCRIPT(7).

If you have these sorts of requirements, it may be worth exploring the use of PostgreSQL 8.0+ PITR (Point In Time Recovery), where WAL logs are replicated to remote nodes.

Chapter 2

Tutorial

2.1 Replicating Your First Database

In this example, we will be replicating a brand new pgbench database. The mechanics of replicating an existing database are covered here, however we recommend that you learn how Slony-I functions by using a fresh new non-production database.

Note that pgbench is a 'benchmark' tool that is in the PostgreSQL set of `contrib` tools. If you build PostgreSQL from source, you can readily head to `contrib/pgbench` and do a **make install** to build and install it; you may discover that included in packaged binary PostgreSQL installations.

The Slony-I replication engine is trigger-based, allowing us to replicate databases (or portions thereof) running under the same postmaster.

This example will show how to replicate the pgbench database running on localhost (master) to the pgbench slave database also running on localhost (slave). We make a couple of assumptions about your PostgreSQL configuration:

- You have `tcpip_socket=true` in your `postgresql.conf`;

> **Note**
> This is no longer needed for PostgreSQL 8.0 and later versions.

- You have enabled access in your cluster(s) via `pg_hba.conf`

The `REPLICATIONUSER` is commonly set up to be a PostgreSQL superuser, perhaps an existing one such as `postgres`, `pgsql`, or one created for this purpose such as `slony` or `replication`. Traditionally, people have used a database superuser for this, but that is not necessary as discussed Section 5.7.2. If you set up a non-superuser user for this, there is more of a configuration burden in granting the specifically-required permissions.

You should also set the following shell variables:

- `CLUSTERNAME`=slony_example
- `MASTERDBNAME`=pgbench
- `SLAVEDBNAME`=pgbenchslave
- `MASTERHOST`=localhost
- `SLAVEHOST`=localhost
- `REPLICATIONUSER`=pgsql
- `PGBENCHUSER`=pgbench

Here are a couple of examples for setting variables in common shells:

- bash, sh, ksh **export CLUSTERNAME=slony_example**
- (t)csh: **setenv CLUSTERNAME slony_example**

Warning
If you're changing these variables to use different hosts for `MASTERHOST` and `SLAVEHOST`, be sure *not* to use localhost for either of them. This will result in an error similar to the following:
ERROR remoteListenThread_1: db_getLocalNodeId() returned 2 - wrong database?

2.1.1 Creating the pgbench User

createuser -SRD $PGBENCHUSER

2.1.2 Preparing the Databases

```
createdb -O $PGBENCHUSER -h $MASTERHOST $MASTERDBNAME
createdb -O $PGBENCHUSER -h $SLAVEHOST $SLAVEDBNAME
pgbench -i -s 1 -U $PGBENCHUSER -h $MASTERHOST $MASTERDBNAME
```

One of the tables created by pgbench, `pgbench_history`, does not have a primary key. Slony-I *requires* that there is a suitable candidate primary key.

The following SQL requests will establish a proper primary key on this table:

```
psql -U $PGBENCHUSER -h $HOST1 -d $MASTERDBNAME -c "begin; alter table
pgbench_history add column id serial; update history set id =
nextval('pgbench_history_id_seq'); alter table pgbench_history add primary key(id);
commit;"
```

Because Slony-I depends on the databases having the pl/pgSQL procedural language installed, we better install it now. It is possible that you have installed pl/pgSQL into the template1 database in which case you can skip this step because it's already installed into the `$MASTERDBNAME`.

```
createlang -h $MASTERHOST plpgsql $MASTERDBNAME
```

Slony-I does not automatically copy table definitions from a master when a slave subscribes to it, so we need to import this data. We do this with pg_dump.

```
pg_dump -s -U $REPLICATIONUSER -h $MASTERHOST $MASTERDBNAME | psql -U $REPLICATIONUSER -h  ↩
    $SLAVEHOST $SLAVEDBNAME
```

To illustrate how Slony-I allows for on the fly replication subscription, let's start up pgbench. If you run the pgbench application in the foreground of a separate terminal window, you can stop and restart it with different parameters at any time. You'll need to re-export the variables again so they are available in this session as well.

The typical command to run pgbench would look like:

```
pgbench -s 1 -c 5 -t 1000 -U $PGBENCHUSER -h $MASTERHOST $MASTERDBNAME
```

This will run pgbench with 5 concurrent clients each processing 1000 transactions against the pgbench database running on localhost as the pgbench user.

2.1.3 Configuring the Database For Replication.

Creating the configuration tables, stored procedures, triggers and configuration is all done through the slonik(1) tool. It is a specialized scripting aid that mostly calls stored procedures in the master/slave (node) databases.

The example that follows uses slonik(1) directly (or embedded directly into scripts). This is not necessarily the most pleasant way to get started; there exist tools for building slonik(1) scripts under the `tools` directory, including:

- Section 6.1.1 - a set of Perl scripts that build slonik(1) scripts based on a single `slon_tools.conf` file.
- Section 6.1.2 - a shell script (*e.g.* - works with Bash) which, based either on self-contained configuration or on shell environment variables, generates a set of slonik(1) scripts to configure a whole cluster.

2.1.3.1 Using slonik Command Directly

The traditional approach to administering slony is to craft slonik commands directly. An example of this given here.

The script to create the initial configuration for the simple master-slave setup of our pgbench database looks like this:

```
#!/bin/sh

slonik <<_EOF_
  #--
  # define the namespace the replication system uses in our example it is
  # slony_example
  #--
  cluster name = $CLUSTERNAME;

  #--
  # admin conninfo's are used by slonik to connect to the nodes one for each
  # node on each side of the cluster, the syntax is that of PQconnectdb in
  # the C-API
  # --
  node 1 admin conninfo = 'dbname=$MASTERDBNAME host=$MASTERHOST user=$REPLICATIONUSER';
  node 2 admin conninfo = 'dbname=$SLAVEDBNAME host=$SLAVEHOST user=$REPLICATIONUSER';

  #--
  # init the first node.  Its id MUST be 1.  This creates the schema
  # _$CLUSTERNAME containing all replication system specific database
  # objects.

  #--
  init cluster ( id=1, comment = 'Master Node');

  #--
  # Slony-I organizes tables into sets.  The smallest unit a node can
  # subscribe is a set.  The following commands create one set containing
  # all 4 pgbench tables.  The master or origin of the set is node 1.
  #--
  create set (id=1, origin=1, comment='All pgbench tables');
  set add table (set id=1, origin=1, id=1, fully qualified name = 'public.pgbench_accounts ↩
      ', comment='accounts table');
  set add table (set id=1, origin=1, id=2, fully qualified name = 'public.pgbench_branches ↩
      ', comment='branches table');
  set add table (set id=1, origin=1, id=3, fully qualified name = 'public.pgbench_tellers', ↩
       comment='tellers table');
  set add table (set id=1, origin=1, id=4, fully qualified name = 'public.pgbench_history', ↩
       comment='history table');

  #--
  # Create the second node (the slave) tell the 2 nodes how to connect to
```

```
  # each other and how they should listen for events.
  #--

  store node (id=2, comment = 'Slave node', event node=1);
  store path (server = 1, client = 2, conninfo='dbname=$MASTERDBNAME host=$MASTERHOST user= ↩
      $REPLICATIONUSER');
  store path (server = 2, client = 1, conninfo='dbname=$SLAVEDBNAME host=$SLAVEHOST user= ↩
      $REPLICATIONUSER');
_EOF_
```

Is the pgbench still running? If not, then start it again.

At this point we have 2 databases that are fully prepared. One is the master database in which pgbench is busy accessing and changing rows. It's now time to start the replication daemons.

On $MASTERHOST the command to start the replication engine is

```
slon $CLUSTERNAME "dbname=$MASTERDBNAME user=$REPLICATIONUSER host=$MASTERHOST"
```

Likewise we start the replication system on node 2 (the slave)

```
slon $CLUSTERNAME "dbname=$SLAVEDBNAME user=$REPLICATIONUSER host=$SLAVEHOST"
```

Even though we have the slon(1) running on both the master and slave, and they are both spitting out diagnostics and other messages, we aren't replicating any data yet. The notices you are seeing is the synchronization of cluster configurations between the 2 slon(1) processes.

To start replicating the 4 pgbench tables (set 1) from the master (node id 1) the the slave (node id 2), execute the following script.

```
#!/bin/sh
slonik <<_EOF_
   # ----
   # This defines which namespace the replication system uses
   # ----
   cluster name = $CLUSTERNAME;

   # ----
   # Admin conninfo's are used by the slonik program to connect
   # to the node databases.  So these are the PQconnectdb arguments
   # that connect from the administrators workstation (where
   # slonik is executed).
   # ----
   node 1 admin conninfo = 'dbname=$MASTERDBNAME host=$MASTERHOST user=$REPLICATIONUSER';
   node 2 admin conninfo = 'dbname=$SLAVEDBNAME host=$SLAVEHOST user=$REPLICATIONUSER';

   # ----
   # Node 2 subscribes set 1
   # ----
   subscribe set ( id = 1, provider = 1, receiver = 2, forward = no);
_EOF_
```

Any second now, the replication daemon on `$SLAVEHOST` will start to copy the current content of all 4 replicated tables. While doing so, of course, the pgbench application will continue to modify the database. When the copy process is finished, the replication daemon on `$SLAVEHOST` will start to catch up by applying the accumulated replication log. It will do this in little steps, initially doing about 10 seconds worth of application work at a time. Depending on the performance of the two systems involved, the sizing of the two databases, the actual transaction load and how well the two databases are tuned and maintained, this catchup process may be a matter of minutes, hours, or eons.

If you encounter problems getting this working, check over the logs for the slon(1) processes, as error messages are likely to be suggestive of the nature of the problem. The tool Section 5.1.1 is also useful for diagnosing problems with nearly-functioning replication clusters.

You have now successfully set up your first basic master/slave replication system, and the 2 databases should, once the slave has caught up, contain identical data. That's the theory, at least. In practice, it's good to build confidence by verifying that the datasets are in fact the same.

The following script will create ordered dumps of the 2 databases and compare them. Make sure that pgbench has completed, so that there are no new updates hitting the origin node, and that your slon sessions have caught up.

```
#!/bin/sh
echo -n "**** comparing sample1 ... "
psql -U $REPLICATIONUSER -h $MASTERHOST $MASTERDBNAME >dump.tmp.1.$$ <<_EOF_
   select 'accounts:'::text, aid, bid, abalance, filler
      from pgbench_accounts order by aid;
   select 'branches:'::text, bid, bbalance, filler
      from pgbench_branches order by bid;
   select 'tellers:'::text, tid, bid, tbalance, filler
      from pgbench_tellers order by tid;
   select 'history:'::text, tid, bid, aid, delta, mtime, filler, id
      from pgbench_history order by id;
_EOF_
psql -U $REPLICATIONUSER -h $SLAVEHOST $SLAVEDBNAME >dump.tmp.2.$$ <<_EOF_
   select 'accounts:'::text, aid, bid, abalance, filler
      from pgbench_accounts order by aid;
   select 'branches:'::text, bid, bbalance, filler
      from pgbench_branches order by bid;
   select 'tellers:'::text, tid, bid, tbalance, filler
      from pgbench_tellers order by tid;
   select 'history:'::text, tid, bid, aid, delta, mtime, filler, id
      from pgbench_history order by id;
_EOF_

if diff dump.tmp.1.$$ dump.tmp.2.$$ >$CLUSTERNAME.diff ; then
   echo "success - databases are equal."
   rm dump.tmp.?.$$
   rm $CLUSTERNAME.diff
else
   echo "FAILED - see $CLUSTERNAME.diff for database differences"
fi
```

Note that there is somewhat more sophisticated documentation of the process in the Slony-I source code tree in a file called `slony-I-basic-mstr-slv.txt`.

If this script returns **FAILED** please contact the developers at http://slony.info/. Be sure to be prepared with useful diagnostic information including the logs generated by slon(1) processes and the output of Section 5.1.1.

2.1.3.2 Using the altperl Scripts

Using the Section 6.1.1 scripts is an alternative way to get started; it allows you to avoid writing slonik scripts, at least for some of the simple ways of configuring Slony-I. The **slonik_build_env** script will generate output providing details you need to build a `slon_tools.conf`, which is required by these scripts. An example `slon_tools.conf` is provided in the distribution to get you started. The altperl scripts all reference this central configuration file centralize cluster configuration information. Once slon_tools.conf has been created, you can proceed as follows:

```
# Initialize cluster:
$ slonik_init_cluster  | slonik

# Start slon  (here 1 and 2 are node numbers)
$ slon_start 1
$ slon_start 2

# Create Sets (here 1 is a set number)
$ slonik_create_set 1 | slonik
```

```
# subscribe set to second node (1= set ID, 2= node ID)
$ slonik_subscribe_set 1 2 | slonik
```

You have now replicated your first database.

2.2 Starting & Stopping Replication

The slon(1) program is a daemon process that replicates data from one machine to another. The slon process is responsible for the following tasks

- Generating 'SYNC' events on the local database
- Processing events from remote nodes.
- Applying the updates pulled from a remote database to user tables to the local database.
- Performing cleanup tasks

2.2.1 Deploying Slon Processes

Each database in your cluster needs a slon process which it will act as the "node controller" for. The slon instance will consider itself "local" to that database and establish "remote" connections to any other databases for which a SLONIK STORE PATH(7) has been defined.

The slon process for a particular database does not need to run on the same server as the database. It is recommended (for performance reasons) that the network connection between slon process and "local" database fairly fast but this is not required. One common way of deploying Slony-I is to have the slon process running on the same node as the database it is servicing. Another common deployment is to centralize the slon processes for all of the databases in a particular data-center on a single administrative server.

It is important that the network connection between the slon processes and the database servers it talks to be reliable. If the network connection goes away at the wrong time it can leave the database connection in a "zombied". Restarting the slon process will repair this situation.

2.2.2 Starting Slon On Unix Systems

The slon process gets installed in your PostgreSQL bin directory, this is the same directory that psql and the postgres binary get installed into. On a Unix system (including Linux variants) slon can be started either:

- Manually through the command line by invoking "slon" directly.
- By using the rc.d style start_slon.sh script found in the tools directory of the Slony-I source distribution.

2.2.2.1 Invoking slon Directly

To invoke slon directly you would execute the command

```
slon slony_example 'host=localhost dbname=pgbench user=pgbench'
```

See slon(1) for information on command line options.

2.2.2.2 start_slon.sh

To start slon via the start_slon.sh script you will first need to create a slon.conf file with the configuration options for slon. This is an example of a simple slon.conf file

```
cluster_name=slony_example
conn_info=host=localhost dbname=pgbench user=pgbench
```

You would then set the SLON_CONF environment variable to point at this file and start the slon.

```
export SLON_BIN=/usr/local/pgsql8.3/bin/slon
export SLON_CONF=/etc/slon/slon.conf
export SLON_LOG=/var/log/slon.log
/usr/local/pgsql8.3/bin/start_slon.sh start
```

2.2.3 Stopping Slon On a Unix System

On a Unix system the slon process (called the watchdog) slon will fork creating a child slon process (called the worker) that does all the work. The watchdog monitors the worker and restarts the worker when required. To terminate slon you would send the watchdog slon (the slon process that you started) a SIGTERM. If you started slon through the start_slon.sh script then you can stop the slon via the "stop" command.

```
export SLON_BIN=/usr/local/pgsql8.3/bin/slon
export SLON_CONF=/etc/slon/slon.conf
export SLON_LOG=/var/log/slon.log
/usr/local/pgsql8.3/bin/start_slon.sh stop
```

2.2.4 Starting Slon On a MS-Windows System

On a MS-Windows system slon needs to be started as a service with a configuration file containing the settings for slon. An example of a configuration file is below.

```
cluster_name=slony_example
conn_info=host=localhost dbname=pgbench user=pgbench
```

You then need to add the slon service

```
pgsql\lib>regsvr32 slevent.dll

--
-- running slon
--
pgsql\bin>slon -regservice Slony-I
pgsql\bin>slon -addengine Slony-I slon.conf
pgsql\bin>slon -listengines Slony-I
```

2.2.5 Stopping slon On MS-Windows

On MS-Windows the service manager starts slon as a service. This slon processs acts as the slon worker. The service manager will start a new slon whenever the slon worker exists. To stop slon you need to disable the service. This can be done through the service manager GUI or with the following commands

```
pgsql\bin>slon -delengine Slony-I slon.conf
```

Chapter 3

Administration Tasks

3.1 Slony-I Building & Installation

Note
For Windows™ users: Unless you are planning on hacking the Slony-I code, it is highly recommended that you download and install a prebuilt binary distribution and jump straight to the configuration section below. Prebuilt binaries are available from the StackBuilder application included in the EnterpriseDB PostgreSQL installer
There are also RPM binaries available at that site for recent versions of Slony-I for recent versions of PostgreSQL.

This section discusses building Slony-I from source.

You should have obtained the Slony-I source from the previous step. Unpack it.

```
gunzip slony.tar.gz;
tar xf slony.tar
```

This will create a directory under the current directory with the Slony-I sources. Head into that that directory for the rest of the installation procedure.

3.1.1 Short Version

```
PGMAIN=/usr/local/pgsql839-freebsd-2008-09-03 \
./configure \
    --with-pgconfigdir=$PGMAIN/bin
gmake all; gmake install
```

3.1.2 Configuration

Slony-I normally needs to be built and installed by the PostgreSQL Unix user. The installation target must be identical to the existing PostgreSQL installation particularly in view of the fact that several Slony-I components represent libraries and SQL scripts that need to be in the Slony-I `lib` and `share` directories.

The first step of the installation procedure is to configure the source tree for your system. This is done by running the configure script.Slony-I is configured by pointing it to the various PostgreSQL library, binary, and include directories. For a full list of these options, use the command **./configure --help**.

It is sufficient, for the purposes of building a usable build, to run **configure `--with-pgconfigdir=/some/path/somewhere`**, where `/some/path/somewhere` is the directory where the PostgreSQL program pg_config is located. Based on the

output of pg_config, the `configure` script determines the various locations where PostgreSQL components are found, which indicate where the essential components of Slony-I must be installed.

For a full listing of configuration options, run the command **./configure --help**.

Warning
Beware: configure defaults to permit indicating values for various paths, including 'generic' values:

- `--bindir`=DIR
  user executables [EPREFIX/bin]
- `--sbindir`=DIR
  system admin executables [EPREFIX/sbin]
- `--libexecdir`=DIR
  program executables [EPREFIX/libexec]
- `--sysconfdir`=DIR
  read-only single-machine data [PREFIX/etc]
- `--sharedstatedir`=DIR
  modifiable architecture-independent data [PREFIX/com]
- `--localstatedir`=DIR
  modifiable single-machine data [PREFIX/var]
- `--libdir`=DIR
  object code libraries [EPREFIX/lib]
- `--includedir`=DIR
  C header files [PREFIX/include]
- `--oldincludedir`=DIR
  C header files for non-gcc [/usr/include]
- `--datarootdir`=DIR
  read-only arch.-independent data root [PREFIX/share]
- `--datadir`=DIR
  read-only architecture-independent data [DATAROOTDIR]
- `--infodir`=DIR
  info documentation [DATAROOTDIR/info]

- `--localedir`=DIR
  locale-dependent data [DATAROOTDIR/locale]
- `--mandir`=DIR
  man documentation [DATAROOTDIR/man]
- `--docdir`=DIR
  documentation root [DATAROOTDIR/doc/slony1]
- `--htmldir`=DIR
  html documentation [DOCDIR]
- `--dvidir`=DIR
  dvi documentation [DOCDIR]
- `--pdfdir`=DIR
  pdf documentation [DOCDIR]
- `--psdir`=DIR
  ps documentation [DOCDIR]

There are also PostgreSQL-specific options specified, which *should not be expressly set*, as pg_config should already provide correct values:

- `--with-pgbindir`=DIR

The compile of PostgreSQL must be expressly configured with the option **--enable-thread-safety** to provide correct client libraries.

Slony-I requires that the PostgreSQL server headers be installed. Some binary distributions of PostgreSQL include this is a a -dev package.

After running configure, you may wish to review the file `Makefile.global` to ensure it is looking in the right places for all of the components.

3.1.3 Example

After determining that the PostgreSQL instance to be used is installed in `/opt/dbs/pgsql746-aix-2005-04-01`:

```
PGMAIN=/opt/dbs/pgsql746-aix-2005-04-01 \
./configure \
    --with-pgconfigdir=$PGMAIN/bin
```

The configure script will run a number of tests to guess values for various dependent variables and try to detect some quirks of your system. Slony-I is known to need a modified version of libpq on specific platforms such as Solaris2.X on SPARC. A patch for libpq version 7.4.2 can be found at http://developer.postgresql.org/~wieck/slony1/download/threadsafe-libpq-742.diff.gz Similar patches may need to be constructed for other versions.

3.1.4 Build

To start the build process, type

```
gmake all
```

Be sure to use GNU make; on BSD systems, it is called gmake; on Linux, GNU make is typically the 'native' make, so the name of the command you type in may be either **make** or **gmake**. On other platforms, you may need additional packages or even install GNU make from scratch. The build may take anywhere from a few seconds to 2 minutes depending on how fast your hardware is at compiling things. The last line displayed should be

All of Slony-I is successfully made. Ready to install.

3.1.5 Installing Slony-I Once Built;

To install Slony-I, enter **gmake install**

This will install files into the postgresql install directory as specified by the **configure** `--prefix` option used in the PostgreSQL installation. Make sure you have appropriate permissions to write into that area. Commonly you need to do this either as root or as the postgres user.

The main list of files installed within the PostgreSQL instance is, for versions of Slony-I

- `$bindir/slon`
- `$bindir/slonik`
- `$libdir/slony1_funcs$(DLSUFFIX)`
- `$datadir/slony1_base.sql`
- `$datadir/slony1_funcs.sql`
- `$datadir/slony1_funcs.v83.sql`
- `$datadir/slony1_funcs.v84.sql`

3.1.6 Building on Win32

Building Slony-I on Win32 with the Microsoft SDK (Visual Studio) is different than building Slony-I on other platforms. Visual Studio builds can be done with out involving configure or gmake. To build Slony-I you need

- The Slony-I source from a source distribution tar (The distribution tar files contain pre-built copies of the parser and scanner generated files. The Win32 makefiles do not currently support building these).
- PostgreSQL binaries, headers and libraries.
- pthreads for win32
- The Microsoft SDK 6.1 or Visual Studio 2008 (other versions might work)
- gettext for win32

To compile the Slony-I binaries you will need to

- Set the environment variables PGSHARE,PG_INC,PG_LIB, PTHREADS_INC,PTHREADS_LIB, GETTEXT_LIB to point to the proper locations based on where these applications were installed. For example

```
set PG_INC=c:\Postgresql\9.0\include
set PG_LIB=c:\Postgresql\9.0\lib
set PGSHARE=c:\\Postgresql\\9.0\\share
set PTHREADS_INC=c:\pthreads-win32\include
set PTHREADS_LIB=c:\pthreads-win32\lib
set GETTEXT_LIB=c:\gettext\lib
```

  Note that the backslash characters must be escaped for PGSHARE as in the above example

From the Visual Studio or Microsoft Windows SDK command prompt run

```
cd src\backend
nmake /f win32.mak slony1_funcs.dll
cd ..\slon
nmake /f win32.mak slon.exe
cd ..\slonik
nmake /f win32.mak slonik.exe
```

src\backend\slony1_funcs.dll and any of the .sql files in src\backend need to be installed in your postgresql $share directory.

3.1.7 Building Documentation: Admin Guide

The document you are reading now is a fairly extensive 'Administrator's Guide' containing what wisdom has been discovered and written down about the care and feeding of Slony-I.

This is only built if you specify **--with-docs**

Note that you may have difficulty building the documentation on older Red Hat systems (RHEL4 and below) See Bug 159382 (For RHEL) See the `INSTALL` file for a workaround for Fedora...

A pre-built copy of the 'admin guide' should be readily available, either in the form of a separate tarball nearby, or in the directory `doc/adminguide/prebuilt`

3.1.8 Installing Slony-I from RPMs

Even though Slony-I can be compiled and run on most Linux distributions, it is also possible to install Slony-I using binary packages. Slony Global Development Team provides official RPMs and SRPMs for many versions or Red Hat and Fedora .

The RPMs are available at PostgreSQL RPM Repository. Please read the howto provided in the website for configuring yum to use that repository. Please note that the RPMs will look for RPM installation of PostgreSQL, so if you install PostgreSQL from source, you should manually ignore the RPM dependencies related to PostgreSQL.

Installing Slony-I using these RPMs is as easy as installing any RPM.

```
yum install slony1
```

yum will pick up dependencies. This repository provides Slony-I binaries built against every supported PostgreSQL version.

The RPM installs the files into their usual places. The configuration files are installed under `/etc`, the binary files are installed in `/usr/bin`, libraries are installed in `/usr/lib/pgsql`, and finally the docs are installed in `/usr/share/doc/slony1`.

3.1.9 Installing the Slony-I service on Windows™

On Windows™ systems, instead of running one slon(1) daemon per node, a single slon service is installed which can then be controlled through the **Services** control panel applet, or from a command prompt using the **net** command.

```
C:\Program Files\PostgreSQL\8.3\bin> slon -regservice my_slon
Service registered.
Before you can run Slony, you must also register an engine!

WARNING! Service is registered to run as Local System. You are
encouraged to change this to a low privilege account to increase
system security.
```

Once the service is installed, individual nodes can be setup by registering slon configuration files with the service.

```
C:\Program Files\PostgreSQL\8.3\bin> slon -addengine c:\node1.conf
Engine added.
```

Other, self explanatory commands include **slon -unregservice <service name>**, **slon -listengines <service name>** and **slon -delengine <service name> <config file>**.

For further information about the Windows™ port, you may want to see the following URLs:

- Slony-I Windows installer sample

3.2 Modifying Things in a Replication Cluster

3.2.1 Adding a Table To Replication

After your Slony-I cluster is setup and nodes are subscribed to your replication set you can still add more tables to replication. To do this you must first create the table on each node. You can do this using psql (on each node) or using the SLONIK EXECUTE SCRIPT(7) command. Next, you should create a new replication set and add the table (or sequence) to the new replication set. Then you subscribe your subscribers to the new replictation set. Once the subscription process is finished you can merge your new replication set into the old one.

```
slonik <<_EOF_
  #--
  # define the namespace the replication system uses in our example it is
  # slony_example
  #--
```

```
  cluster name = $CLUSTERNAME;

  #--
  # admin conninfo's are used by slonik to connect to the nodes one for each
  # node on each side of the cluster, the syntax is that of PQconnectdb in
  # the C-API
  # --
  node 1 admin conninfo = 'dbname=$MASTERDBNAME host=$MASTERHOST user=$REPLICATIONUSER';
  node 2 admin conninfo = 'dbname=$SLAVEDBNAME host=$SLAVEHOST user=$REPLICATIONUSER';
create set (id=2, origin=1, comment='a second replication set');
set add table (set id=2, origin=1, id=5, fully qualified name = 'public.newtable', comment ↩
    ='some new table');
subscribe set(id=1, provider=1,receiver=2);
merge set(id=1, add id=2,origin=1);
```

3.2.2 How To Add Columns To a Replicated Table

There are two approaches you can use for adding (or renaming) columns to an existing replicated table.

The first approach involves you using the SLONIK EXECUTE SCRIPT(7) command. With this approach you would

1. Create a SQL script with your ALTER table statements
2. Stop any application updates to the table you are changing (ie have an outage)
3. Use the slonik SLONIK EXECUTE SCRIPT(7) command to run your script

Your table should now be updated on all databases in the cluster.

Alternatively, if you have the altperl scripts installed, you may use **slonik_execute_script** for this purpose:

slonik_execute_script [options] set# full_path_to_sql_script_file

See **slonik_execute_script -h** for further options; note that this uses SLONIK EXECUTE SCRIPT(7) underneath.

There are a number of 'sharp edges' to note...

- You absolutely *must not* include transaction control commands, particularly **BEGIN** and **COMMIT**, inside these DDL scripts. Slony-I wraps DDL scripts with a **BEGIN/COMMIT** pair; adding extra transaction control will mean that parts of the DDL will commit outside the control of Slony-I
- Version 2.0 of Slony-I does not explicitly lock tables when performing an execute script. To avoid some race-conditions exposed by MVCC it is important that no other transactions are altering the tables being used by the ddl script while it is running

3.2.3 How to remove replication for a node

You will want to remove the various Slony-I components connected to the database(s).

We will just consider, for now, doing this to one node. If you have multiple nodes, you will have to repeat this as many times as necessary.

Components to be Removed:

- Log Triggers / Update Denial Triggers
- The 'cluster' schema containing Slony-I tables indicating the state of the node as well as various stored functions
- slon(1) process that manages the node

- Optionally, the SQL and pl/pgsql scripts and Slony-I binaries that are part of the PostgreSQL build. (Of course, this would make it challenging to restart replication; it is unlikely that you truly need to do this...)

The second approach involves using psql to alter the table directly on each database in the cluster.

1. Stop any application updates to the table you are changing(ie have on outage)
2. Connect to each database in the cluster (in turn) and make the required changes to the table

Warning
The psql approach is only safe with Slony-I 2.0 or greater

Things are not fundamentally different whether you are adding a brand new, fresh node, or if you had previously dropped a node and are recreating it. In either case, you are adding a node to replication.

3.2.4 Adding a Replication Node

To add a node to the replication cluster you should

1. Create a database for the node and install your application schema in it.

```
createdb -h $NEWSLAVE_HOST $SLAVEDB
pg_dump -h $MASTER_HOST -s $MASTERDB | psql -h $NEWSLAVE_HOST $SLAVEDB
```

2. Create the node with the SLONIK STORE NODE(7) command

```
node 5 admin conninfo='host=slavehost dbname=slavedb user=slony password=slony';
clustername=testcluster;

store node(id=5,comment='some slave node',event node=1);
```

3. Create paths between the new node and its provider node with the SLONIK STORE PATH(7) command.

```
node 5 admin conninfo='host=slavehost dbname=slavedb user=slony password=slony';
clustername=testcluster;
node 1 admin conninfo='host=masterhost dbname=masterdb user=slony password=slony';
# also include the admin conninfo lines for any other nodes in your cluster.
#
#
clustername=testcluster;
store path(server=1,client=5,conninfo='host=masterhost,dbname=masterdb,user=slony, ↩
    password=slony');
store path(server=5,client=1,conninfo='host=slavehost,dbname=masterdb,user=slony, ↩
    password=slony');
```

4. Subscribe the new node to the relevant replication sets

```
node 5 admin conninfo='host=slavehost dbname=slavedb user=slony password=slony';
clustername=testcluster;
node 1 admin conninfo='host=masterhost dbname=slavedb user=slony password=slony';
#
# also include the admin conninfo lines for any other nodes in the cluster
#
#
clustername=testcluster;
subscribe set(id=1,provider=1, receiver=5,forward=yes);
```

3.2.5 Adding a Cascaded Replica

In a standard Slony-I configuration all slaves(replicas) communicate directly with the master (origin). Sometimes it is more desirable to have some of your slaves(replicas) feed off of another replica. This is called a cascaded replica and is supported by Slony-I For example you might have a Slony-I cluster with 1 replication set (set id=1) and three nodes. The master (origin) for set 1 (node id=1), a node in a different data center that reads directly from the master (node id=2) and a third node in the same data center as the slave (node id=3). To the subscription sets in this configuration you need to make sure that paths exist between nodes 2 and nodes 3. Then to perform the subscription you could use the following slonik commands.

```
#Setup path between node 1==>2
store path(server=1,client=2,conninfo='host=masterhost,dbname=masterdb,user=slony,password= ↩
    slony');
store path(server=2,client=1,conninfo='host=slave2host,dbname=slave2db,user=slony,password= ↩
    slony');

#Setup path between node 2==>3
store path(server=3,client=2,conninfo='host=slave3host,dbname=slave3db,user=slony,password= ↩
    slony');
store path(server=2,client=3,conninfo='host=slave2host,dbname=slave2db,user=slony,password= ↩
    slony');

subscribe set(set id=1, provider=1, receiver=2,forward=yes);
subscribe set (set id=1,provider=2, receiver=3,forward=yes);
wait for event(origin=1, confirmed=all, wait on=1);
```

In the above example we define paths from 1==>2 and from 2==>3 but do not define a path between nodes 1===>3. If a path between node 1 and 3 was defined the data data for set 1 would still flow through node 2 because node 2 is the origin for set 1. However if node 2 were to fail nodes 1 and 3 would be unable to talk to each other unless a path between nodes 1 and nodes 3 had been defined.

3.2.6 How do I use Log Shipping?

Discussed in the Log Shipping section...

3.2.7 How To Remove Replication For a Node

You will want to remove the various Slony-I components connected to the database(s).

We will just discuss doing this to one node. If you have multiple nodes, you will have to repeat this as many times as necessary.

Removing slony from a node involves deleting the slony schema (tables, functions and triggers) from the node in question and telling the remaining nodes that the deleted node no longer exists. The slony SLONIK DROP NODE(7) command does both of these items while hte SLONIK UNINSTALL NODE(7) command only removes the slony schema from the node.

In the case of a failed node (where you used SLONIK FAILOVER(7) to switch to another node), you may need to use SLONIK UNINSTALL NODE(7) to drop out the triggers and schema and functions.

Warning
Removing Slony-I from a replica in versions before 2.0 is more complicated. If this applies to you then you should consult the Slony-I documentation for the version of Slony-I you are using.

3.2.8 Changing a Nodes Provider

For instance, you might want subscriber node 3 to draw data from node 1, when it is presently drawing data from node 2.

The SLONIK SUBSCRIBE SET(7) command can be used to do this. For existing subscriptions it can revise the subscription information.

```
subscribe set(id=1,origin=1, provider=2,forward=yes);
```

3.2.9 Moving The Master From One Node To Another

Sometimes you will want to promote one of your replicas (slaves) to become the master and at the same time turn the former master into a slave. Slony-I supports this with the SLONIK MOVE SET(7) command.

You must first pick a node that is connected to the former origin (otherwise it is not straightforward to reverse connections in the move to keep everything connected).

Second, you must run a slonik(1) script with the command SLONIK LOCK SET(7) to lock the set on the origin node. Note that at this point you have an application outage under way, as what this does is to put triggers on the origin that rejects updates.

Now, submit the slonik(1) SLONIK MOVE SET(7) request. It's perfectly reasonable to submit both requests in the same slonik(1) script. Now, the origin gets switched over to the new origin node. If the new node is a few events behind, it may take a little while for this to take place.

```
LOCK SET(id=1,ORIGIN=1);
MOVE SET(ID=1,OLD ORIGIN=1, NEW ORIGIN=3);
SYNC(ID=3);
WAIT FOR(ORIGIN=1, CONFIRMED=ALL,WAIT ON=1);
```

It is important to stop all non-Slony application activity against all tables in the replication set before locking the sets. The move set procedure involves obtaining a lock on every table in the replication set. Other activities on these tables can result in a deadlock.

3.3 Database Schema Changes (DDL)

When changes are made to the database schema, *e.g.* - adding fields to a table, it is necessary for this to be handled rather carefully, otherwise different nodes may get rather deranged because they disagree on how particular tables are built.

Slony-I can not automatically detect and replicate database schema changes however, Slony-I does provide facilities to assist in making database schema changes. Schema changes can be done on a replicated database either by using the Slony-I SLONIK EXECUTE SCRIPT(7) (slonik) command or by manually applying the changes to each node.

3.3.1 DDL Changes with Execute Script

The SLONIK EXECUTE SCRIPT(7) (slonik) command allows you to submit a SQL script (that can, but is not required to) contain DDL commands. This script will be executed on the event node and then (optionally) replicated to every other node in the cluster. You should keep the following in mind when using SLONIK EXECUTE SCRIPT(7)

- The script *must not* contain transaction **BEGIN** or **END** statements, as the script is already executed inside a transaction though nested transactions are allowed as long are processed within the scope of a single transaction whose **BEGIN** and **END** you do not control.
- If there is *anything* broken about the script, or about how it executes on a particular node (other than the event node), this will cause the slon(1) daemon for that node to panic and crash. You may see various expected messages (positive and negative) in Section 5.5.6.2. If you restart the slon, it will, most likely, try to *repeat* the DDL script, which will, almost certainly, fail the second time in the same way it did the first time.

  The implication of this is that it is *vital* that modifications not be made in a haphazard way on one node or another. The schemas must always stay in sync. If slon; fails due to a failed DDL change then you should manually (via psql) make the required changes so that the DDL change succeeds the next time slon attempts it.
- Slony-I 2.0.x and 2.1.x suffer from an issue where concurrent transactions involving the same tables as are referenced in the script might not be replayed in exactly the same order on the replica nodes. It is advisiable to not be concurrently inserting,deleting or updating rows to a table while a script changing that table (adding or deleting columns) is also running.

3.3.2 Applying DDL Changes Directly

DDL changes can be applied directly on a node through an application such as **psql**. The DDL changes will not be replicated by Slony-I and therefore must be manually applied to every relevant node. The following points should be kept in mind when applying DDL changes directly.

- While DDL changes are not automatically replicated, any **INSERT,UPDATE,DELETE** statements that you execute will be captured for replication, when run against the origin node. This means that you should not include DDL changes and DML inside the same script when apply DDL directly, because the script will not behave properly when you execute it on other nodes.

  If you, instead, apply DDL using **EXECUTE SCRIPT**, it is fine to intersperse DDL and DML within the script, as Slony-I handles that appropriately.

- You are responsible for ensuring that your scripts get applied on all other nodes at the correct point in the replication stream (*e.g.* - on or before the appropriate SYNC event). The best way of doing this with respect to adding and deleting columns is to make sure that new columns always get added on the replica nodes first and that columns being removed are dropped from the master before they are dropped from the replicas. That way, new columns are always available on the subscriber on or before the time they will be needed, and obsolete ones remain on the subscriber until after the last possible reference to them has been replicated.

Warning
If columns being added or dropped are mandatory (NOT NULL) or have default values, you will need to go through a longer process to ensure constraints are satisfied at each point in time on all nodes.
For instance, if dropping a column that has a NOT NULL constraint, it may take multiple ALTER TABLE statements on each node in order to successfully accomplish this, as the constraint needs to be relaxed first.

- DDL changes that rename a replicated table do not inform Slony-I of the new table name. If you change then name of a replicated table you must allow Slony-I to find the new table name by calling schemadocupdaterelname(p_only_on_node integer, p_set_id integer)

- DDL changes that alter either a primary key, a unique constraint that slony is using, or DDL changes that drop columns that come before the key or unique constraint that Slony-I is using will require Slony-I too reconfigure the arguments on the logtrigger. The function schemadocrepair_log_triggers(only_locked boolean) will reconfigure the trigger arguments of any Slony-I log triggers that are out of date. If `true` is passed to this function it will only adjust tables that are already locked by the current transaction (if you perform your **alter table** within a transaction and then call `repair_log_triggers()` as part of the same transaction then the altered tables will be locked). If you pass `false` to this function then the function will obtain an exclusive lock on any table that needs the trigger to be reconfigured.

3.4 Doing switchover and failover with Slony-I

3.4.1 Foreword

Slony-I is an asynchronous replication system. Because of that, it is almost certain that at the moment the current origin of a set fails, the final transactions committed at the origin will have not yet propagated to the subscribers. Systems are particularly likely to fail under heavy load; that is one of the corollaries of Murphy's Law. Therefore the principal goal is to *prevent* the main server from failing. The best way to do that is frequent maintenance.

Opening the case of a running server is not exactly what we should consider a 'professional' way to do system maintenance. And interestingly, those users who found it valuable to use replication for backup and failover purposes are the very ones that have the lowest tolerance for terms like 'system downtime.' To help support these requirements, Slony-I not only offers failover capabilities, but also the notion of controlled origin transfer.

It is assumed in this document that the reader is familiar with the slonik(1) utility and knows at least how to set up a simple 2 node replication system with Slony-I.

3.4.2 Controlled Switchover

We assume a current 'origin' as node1 with one 'subscriber' as node2 (*e.g.* - slave). A web application on a third server is accessing the database on node1. Both databases are up and running and replication is more or less in sync. We do controlled switchover using SLONIK MOVE SET(7).

- At the time of this writing switchover to another server requires the application to reconnect to the new database. So in order to avoid any complications, we simply shut down the web server. Users who use pg_pool for the applications database connections merely have to shut down the pool.

  What needs to be done, here, is highly dependent on the way that the application(s) that use the database are configured. The general point is thus: Applications that were connected to the old database must drop those connections and establish new connections to the database that has been promoted to the 'master' role. There are a number of ways that this may be configured, and therefore, a number of possible methods for accomplishing the change:

  – The application may store the name of the database in a file.

    In that case, the reconfiguration may require changing the value in the file, and stopping and restarting the application to get it to point to the new location.

  – A clever usage of DNS might involve creating a CNAME DNS record that establishes a name for the application to use to reference the node that is in the 'master' role.

    In that case, reconfiguration would require changing the CNAME to point to the new server, and possibly restarting the application to refresh database connections.

  – If you are using pg_pool or some similar 'connection pool manager,' then the reconfiguration involves reconfiguring this management tool, but is otherwise similar to the DNS/CNAME example above.

  Whether or not the application that accesses the database needs to be restarted depends on how it is coded to cope with failed database connections; if, after encountering an error it tries re-opening them, then there may be no need to restart it.

- A small slonik(1) script executes the following commands:

```
lock set (id = 1, origin = 1);
wait for event (origin = 1, confirmed = 2);
move set (id = 1, old origin = 1, new origin = 2);
wait for event (origin = 1, confirmed = 2, wait on=1);
```

  After these commands, the origin (master role) of data set 1 has been transferred to node2. And it is not simply transferred; it is done in a fashion such that node1 becomes a fully synchronized subscriber, actively replicating the set. So the two nodes have switched roles completely.

- After reconfiguring the web application (or pgpool) to connect to the database on node2, the web server is restarted and resumes normal operation.

  Done in one shell script, that does the application shutdown, slonik, move config files and startup all together, this entire procedure is likely to take less than 10 seconds.

You may now simply shutdown the server hosting node1 and do whatever is required to maintain the server. When slon(1) node1 is restarted later, it will start replicating again, and soon catch up. At this point the procedure to switch origins is executed again to restore the original configuration.

This is the preferred way to handle things; it runs quickly, under control of the administrators, and there is no need for there to be any loss of data.

After performing the configuration change, you should, run the Section 5.1.1 scripts in order to validate that the cluster state remains in good order after this change.

3.4.3 Failover

If some more serious problem occurs on the 'origin' server, it may be necessary to SLONIK FAILOVER(7) to a backup server. This is a highly undesirable circumstance, as transactions 'committed' on the origin, but not applied to the subscribers, will be

lost. You may have reported these transactions as 'successful' to outside users. As a result, failover should be considered a *last resort*. If the 'injured' origin server can be brought up to the point where it can limp along long enough to do a controlled switchover, that is *greatly* preferable.

Slony-I does not provide any automatic detection for failed systems. Abandoning committed transactions is a business decision that cannot be made by a database system. If someone wants to put the commands below into a script executed automatically from the network monitoring system, well ... it's *your* data, and it's *your* failover policy.

- The slonik(1) command

```
failover (id = 1, backup node = 2);
```

  causes node2 to assume the ownership (origin) of all sets that have node1 as their current origin. If there should happen to be additional nodes in the Slony-I cluster, all direct subscribers of node1 are instructed that this is happening. Slonik will also query all direct subscribers in order to determine out which node has the highest replication status (*e.g.* - the latest committed transaction) for each set, and the configuration will be changed in a way that node2 first applies those final before actually allowing write access to the tables.

  In addition, all nodes that subscribed directly to node1 will now use node2 as data provider for the set. This means that after the failover command succeeded, no node in the entire replication setup will receive anything from node1 any more.

  Note
  Note that in order for node 2 to be considered as a candidate for failover, it must have been set up with the SLONIK SUBSCRIBE SET(7) option **forwarding = yes**, which has the effect that replication log data is collected in sl_log_1/sl_log_2 on node 2. If replication log data is *not* being collected, then failover to that node is not possible.

- Reconfigure and restart the application (or pgpool) to cause it to reconnect to node2.

- Purge out the abandoned node

  You will find, after the failover, that there are still a full set of references to node 1 in sl_node, as well as in referring tables such as sl_confirm; since data in sl_log_1/sl_log_2 is still present, Slony-I cannot immediately purge out the node.

  After the failover is complete and all nodes have been reconfigured you can remove all remnants of node1's configuration information with the SLONIK DROP NODE(7) command:

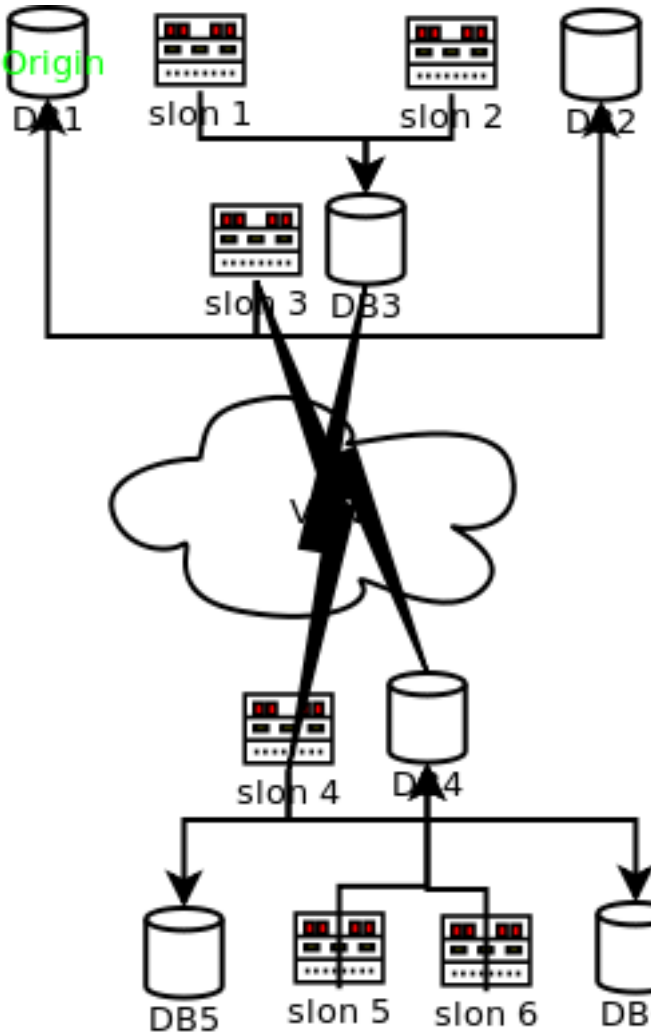
```
drop node (id = 1, event node = 2);
```

  Supposing the failure resulted from some catastrophic failure of the hardware supporting node 1, there might be no 'remains' left to look at. If the failure was not 'total', as might be the case if the node had to be abandoned due to a network communications failure, you will find that node 1 still 'imagines' itself to be as it was before the failure. See Section 3.4.6 for more details on the implications.

- After performing the configuration change, you should, as run the Section 5.1.1 scripts in order to validate that the cluster state remains in good order after this change.

3.4.4 Failover With Complex Node Set

Failover is relatively 'simple' if there are only two nodes; if a Slony-I cluster comprises many nodes, achieving a clean failover requires careful planning and execution.



Consider the following diagram describing a set of six nodes at two sites.

Let us assume that nodes 1, 2, and 3 reside at one data centre, and that we find ourselves needing to perform failover due to failure of that entire site. Causes could range from a persistent loss of communications to the physical destruction of the site; the cause is not actually important, as what we are concerned about is how to get Slony-I to properly fail over to the new site.

We will further assume that node 5 is to be the new origin, after failover.

The sequence of Slony-I reconfiguration required to properly failover this sort of node configuration is as follows:

- Resubscribe (using SLONIK SUBSCRIBE SET(7)) each node that is to be kept in the reformation of the cluster that is not already subscribed to the intended data provider.

  In the example cluster, this means we would likely wish to resubscribe nodes 4 and 6 to both point to node 5.

```
include </tmp/failover-preamble.slonik>;
subscribe set (id = 1, provider = 5, receiver = 4);
subscribe set (id = 1, provider = 5, receiver = 6);
wait for event(origin=1, confirmed=4,wait on=1);
wait for event(origin=1, confirmed=6, wait on=1);
```

- Drop all unimportant nodes, starting with leaf nodes.

  Since nodes 1, 2, and 3 are unaccessible, we must indicate the `EVENT NODE` so that the event reaches the still-live portions of the cluster.

```
include </tmp/failover-preamble.slonik>;
drop node (id=2, event node = 4);
drop node (id=3, event node = 4);
```

- Now, run **FAILOVER**.

```
include </tmp/failover-preamble.slonik>;
failover (id = 1, backup node = 5);
```

- Finally, drop the former origin from the cluster.

```
include </tmp/failover-preamble.slonik>;
drop node (id=1, event node = 4);
```

3.4.5 Automating FAIL OVER

If you do choose to automate **FAIL OVER** , it is important to do so *carefully.* You need to have good assurance that the failed node is well and truly failed, and you need to be able to assure that the failed node will not accidentally return into service, thereby allowing there to be two nodes out there able to respond in a 'master' role.

Note
The problem here requiring that you 'shoot the failed node in the head' is not fundamentally about replication or Slony-I; Slony-I handles this all reasonably gracefully, as once the node is marked as failed, the other nodes will 'shun' it, effectively ignoring it. The problem is instead with *your application.* Supposing the failed node can come back up sufficiently that it can respond to application requests, *that* is likely to be a problem, and one that hasn't anything to do with Slony-I. The trouble is if there are two databases that can respond as if they are 'master' systems.

When failover occurs, there therefore needs to be a mechanism to forcibly knock the failed node off the network in order to prevent applications from getting confused. This could take place via having an SNMP interface that does some combination of the following:

- Turns off power on the failed server.

  If care is not taken, the server may reappear when system administrators power it up.

- Modify firewall rules or other network configuration to drop the failed server's IP address from the network.

  If the server has multiple network interfaces, and therefore multiple IP addresses, this approach allows the 'application' addresses to be dropped/deactivated, but leave 'administrative' addresses open so that the server would remain accessible to system administrators.

3.4.6 After Failover, Reconfiguring Former Origin

What happens to the failed node will depend somewhat on the nature of the catastrophe that lead to needing to fail over to another node. If the node had to be abandoned because of physical destruction of its disk storage, there will likely not be anything of interest left. On the other hand, a node might be abandoned due to the failure of a network connection, in which case the former 'provider' can appear be functioning perfectly well. Nonetheless, once communications are restored, the fact of the **FAIL OVER** makes it mandatory that the failed node be abandoned.

After the above failover, the data stored on node 1 will rapidly become increasingly out of sync with the rest of the nodes, and must be treated as corrupt. Therefore, the only way to get node 1 back and transfer the origin role back to it is to rebuild it from scratch as a subscriber, let it catch up, and then follow the switchover procedure.

A good reason *not* to do this automatically is the fact that important updates (from a *business* perspective) may have been **commit**ted on the failing system. You probably want to analyze the last few transactions that made it into the failed node to see if some of them need to be reapplied on the 'live' cluster. For instance, if someone was entering bank deposits affecting customer accounts at the time of failure, you wouldn't want to lose that information.

Warning
It has been observed that there can be some very confusing results if a node is 'failed' due to a persistent network outage as opposed to failure of data storage. In such a scenario, the 'failed' node has a database in perfectly fine form; it is just that since it was cut off, it 'screams in silence.'
If the network connection is repaired, that node could reappear, and as far as *its* configuration is concerned, all is well, and it should communicate with the rest of its Slony-I cluster.
In *fact*, the only confusion taking place is on that node. The other nodes in the cluster are not confused at all; they know that this node is 'dead,' and that they should ignore it. But there is not a way to know this by looking at the 'failed' node. This points back to the design point that Slony-I is not a network monitoring tool. You need to have clear methods of communicating to applications and users what database hosts are to be used. If those methods are lacking, adding replication to the mix will worsen the potential for confusion, and failover will be a point at which there is enormous potential for confusion.

If the database is very large, it may take many hours to recover node1 as a functioning Slony-I node; that is another reason to consider failover as an undesirable 'final resort.'

3.4.7 Planning for Failover

Failover policies should be planned for ahead of time.

Most pointedly, any node that is expected to be a failover target must have its subscription(s) set up with the option **FORWARD = YES**. Otherwise, that node is not a candidate for being promoted to origin node.

This may simply involve thinking about what the priority lists should be of what should fail to what, as opposed to trying to automate it. But knowing what to do ahead of time cuts down on the number of mistakes made.

At Afilias, a variety of internal [**?**] guides have been created to provide checklists of what to do when certain 'unhappy' events take place. This sort of material is highly specific to the environment and the set of applications running there, so you would need to generate your own such documents. This is one of the vital components of any disaster recovery preparations.

Chapter 4

Advanced Concepts

4.1 Events & Confirmations

Slony-I transfers configuration changes and application data through events. Events in Slony-I have an origin, a type and some parameters. When an event is created it is inserted into the event queue (the sl_event table) on the node the event originates on. The remoteListener threads for each remote slon(1) process then picks up that event (by querying the table sl_event) and pass the event to the slon(1)'s remoteWorker thread for processing.

An event is uniquely identified via the combination of the node id of the node the event originates on and the event sequence number for that node. For example, (1,5000001) identifies event 5000001 originating from node 1. In contrast, (3,5000001) identifies a different event that originated on a different node.

4.1.1 SYNC Events

SYNC events are used to transfer application data for one node to the next. When data in a replicated table changes, a trigger fires that records information about the change in the sl_log_1 or sl_log_2 tables. The localListener thread in the slon processes will then periodically generate a SYNC event. When the SYNC event is created, Slony-I will determine the highest log_seqid assigned so far along with a list of log_seqid's that were assigned to transactions that have not yet been committed. This information is all stored as part of the SYNC event.

When the remoteWorker thread for a slon(1) processes a SYNC, it queries the rows from sl_log_1 and sl_log_2 that are covered by the SYNC (*e.g.* - log_seqid rows that had been committed at the time the SYNC was generated). The data modifications indicated by this logged data are then applied to the subscriber.

4.1.2 Event Confirmations

When an event is processed by the slon(1) process for a remote node, a CONFIRM message is generated by inserting a tuple into the sl_confirm table. This tuple indicates that a particular event has been confirmed by a particular receiver node. Confirmation messages are then transferred back to all other nodes in the cluster.

4.1.3 Event cleanup

The slon(1) cleanupThread periodically runs the schemadoccleanupevent(p_interval interval) database function that deletes all but the most recently confirmed event for each origin/receiver pair (this is safe to do because if an event has been confirmed by a receiver, then we know that all older events from that origin have also been confirmed by the receiver). Then the function deletes all SYNC events that are older than the oldest row left in sl_confirm (for each origin). The data for these deleted events will also be removed from the sl_log_1 and sl_log_2 tables.

When Slony-I is first enabled it will log the data to replicate to the sl_log_1 table. After a while it will stop logging to sl_log_1 and switch to logging in sl_log_2. When all the data in sl_log_1 is known to have been replicated to all the other nodes, Slony-I

will TRUNCATE the sl_log_1 table, clearing out this now-obsolete replication data. Then, it stops logging to sl_log_2, switching back to logging to the freshly truncated sl_log_1 table. This process is repeated periodically as Slony-I runs, keeping these tables from growing uncontrollably. By using TRUNCATE, we guarantee that the tables are properly emptied out.

4.1.4 Slonik and Event Confirmations

slonik(1) can submit configuration commands to different event nodes, as controlled by the parameters of each slonik command. If two commands are submitted to different nodes, it might be important to ensure they are processed by other nodes in a consistent order. The slonik(1) SLONIK WAIT FOR EVENT(7) command may be used to accomplish this, but as of Slony-I 2.1 this consistency is handled automatically by slonik(1) under a number of circumstances.

1. Before slonik submits an event to a node, it waits until that node has confirmed the last configuration event from the previous event node.
2. Before slonik submits a SLONIK SUBSCRIBE SET(7) command, it verifies that the provider node has confirmed all configuration events from all other nodes.
3. Before slonik(1) submits a SLONIK DROP NODE(7) event, it verifies that all nodes in the cluster (aside from the one being dropped, of course!) have already caught up with all other nodes
4. Before slonik submits a SLONIK CLONE PREPARE(7) it verifies that the node being cloned is caught up with all other nodes in the cluster.
5. Before slonik submits a SLONIK CREATE SET(7) command it verifies that any SLONIK DROP SET(7) commands have been confirmed by all nodes.

When slonik(1) starts up, it contacts all nodes for which it has SLONIK ADMIN CONNINFO(7) information, to find the last non-SYNC event from each node. Submitting commands from multiple slonik(1) instances at the same time will confuse slonik(1) and is not recommended. Whenever slonik(1) is waiting for an event confirmation, it displays a message every 10 seconds indicating which events are still outstanding. Any commands that might require slonik to wait for event confirmations may not be validly executed within a try block for the very same reasons that SLONIK WAIT FOR EVENT(7) command may not be used within a try block, namely that it is not reasonable to ask Slony-I to try to roll back events.

Automatic waiting for confirmations may be disabled in slonik(1) by running slonik(1) with the `-w` option.

4.2 Slony-I Listen Paths

Note
If you are running version Slony-I 1.1 or later it should be *completely unnecessary* to read this section as it introduces a way to automatically manage this part of its configuration. For earlier versions, however, it is needful.

If you have more than two or three nodes, and any degree of usage of cascaded subscribers (*e.g.* - subscribers that are subscribing through a subscriber node), you will have to be fairly careful about the configuration of 'listen paths' via the Slonik SLONIK STORE LISTEN(7) and SLONIK DROP LISTEN(7) statements that control the contents of the table sl_listen.

The 'listener' entries in this table control where each node expects to listen in order to get events propagated from other nodes. You might think that nodes only need to listen to the 'parent' from whom they are getting updates, but in reality, they need to be able to receive messages from *all* nodes in order to be able to conclude that **sync**s have been received everywhere, and that, therefore, entries in sl_log_1 and sl_log_2 have been applied everywhere, and can therefore be purged. this extra communication is needful so Slony-I is able to shift origins to other locations.

4.2.1 How Listening Can Break

On one occasion, I had a need to drop a subscriber node (#2) and recreate it. That node was the data provider for another subscriber (#3) that was, in effect, a 'cascaded slave.' Dropping the subscriber node initially didn't work, as slonik(1) informed me that there was a dependant node. I re-pointed the dependant node to the 'master' node for the subscription set, which, for a while, replicated without difficulties.

I then dropped the subscription on 'node 2', and started resubscribing it. that raised the Slony-I **set_subscription** event, which started copying tables. at that point in time, events stopped propagating to 'node 3', and while it was in perfectly ok shape, no events were making it to it.

The problem was that node #3 was expecting to receive events from node #2, which was busy processing the **set_subscription** event, and was not passing anything else on.

We dropped the listener rules that caused node #3 to listen to node 2, replacing them with rules where it expected its events to come from node #1 (the origin node for the replication set). At that moment, 'as if by magic', node #3 started replicating again, as it discovered a place to get **sync** events.

4.2.2 How the Listen Configuration Should Look

The simple cases tend to be simple to cope with. We need to instead look at a more complex node configuration.

Consider a set of nodes, 1 thru 6, where 1 is the origin, where 2-4 subscribe directly to the origin, and where 5 subscribes to 2, and 6 subscribes to 5.

Here is a 'listener network' that indicates where each node should listen for messages coming from each other node:

```
       1|   2|   3|   4|   5|   6|
----------------------------------------
   1   0    2    3    4    2    2
   2   1    0    1    1    5    5
   3   1    1    0    1    1    1
   4   1    1    1    0    1    1
   5   2    2    2    2    0    6
   6   5    5    5    5    5    0
```

Row 2 indicates all of the listen rules for node 2; it gets events for nodes 1, 3, and 4 through node 1, and gets events for nodes 5 and 6 from node 5.

The row of 5's at the bottom, for node 6, indicate that node 6 listens to node 5 to get events from nodes 1-5.

The set of slonik **set listen** statements to express this 'listener network' are as follows:

```
store listen (origin = 1, receiver = 2, provider = 1);
store listen (origin = 1, receiver = 3, provider = 1);
store listen (origin = 1, receiver = 4, provider = 1);
store listen (origin = 1, receiver = 5, provider = 2);
store listen (origin = 1, receiver = 6, provider = 5);
store listen (origin = 2, receiver = 1, provider = 2);
store listen (origin = 2, receiver = 3, provider = 1);
store listen (origin = 2, receiver = 4, provider = 1);
store listen (origin = 2, receiver = 5, provider = 2);
store listen (origin = 2, receiver = 6, provider = 5);
store listen (origin = 3, receiver = 1, provider = 3);
store listen (origin = 3, receiver = 2, provider = 1);
store listen (origin = 3, receiver = 4, provider = 1);
store listen (origin = 3, receiver = 5, provider = 2);
store listen (origin = 3, receiver = 6, provider = 5);
store listen (origin = 4, receiver = 1, provider = 4);
store listen (origin = 4, receiver = 2, provider = 1);
store listen (origin = 4, receiver = 3, provider = 1);
store listen (origin = 4, receiver = 5, provider = 2);
store listen (origin = 4, receiver = 6, provider = 5);
```

```
store listen (origin = 5, receiver = 1, provider = 2);
store listen (origin = 5, receiver = 2, provider = 5);
store listen (origin = 5, receiver = 3, provider = 1);
store listen (origin = 5, receiver = 4, provider = 1);
store listen (origin = 5, receiver = 6, provider = 5);
store listen (origin = 6, receiver = 1, provider = 2);
store listen (origin = 6, receiver = 2, provider = 5);
store listen (origin = 6, receiver = 3, provider = 1);
store listen (origin = 6, receiver = 4, provider = 1);
store listen (origin = 6, receiver = 5, provider = 6);
```

How we read these listen statements is thus...

When on the 'receiver' node, look to the 'provider' node to provide events coming from the 'origin' node.

The tool `init_cluster` in the `altperl` scripts produces optimized listener networks in both the tabular form shown above as well as in the form of slonik(1) statements.

There are three 'thorns' in this set of roses:

- If you change the shape of the node set, so that the nodes subscribe differently to things, you need to drop sl_listen entries and create new ones to indicate the new preferred paths between nodes. Until Slony-I 1.1;, there is no automated way at this point to do this 'reshaping'.
- If you *don't* change the sl_listen entries, events will likely continue to propagate so long as all of the nodes continue to run well. the problem will only be noticed when a node is taken down, 'orphaning' any nodes that are listening through it.
- you might have multiple replication sets that have *different* shapes for their respective trees of subscribers. there won't be a single 'best' listener configuration in that case.
- In order for there to be an sl_listen path, there *must* be a series of sl_path entries connecting the origin to the receiver. this means that if the contents of sl_path do not express a 'connected' network of nodes, then some nodes will not be reachable. this would typically happen, in practice, when you have two sets of nodes, one in one subnet, and another in another subnet, where there are only a couple of 'firewall' nodes that can talk between the subnets. cut out those nodes and the subnets stop communicating.

4.2.3 Automated Listen Path Generation

In Slony-I version 1.1, a heuristic scheme is introduced to automatically generate `sl_listen` entries. This happens, in order, based on three data sources:

- sl_subscribe entries are the first, most vital control as to what listens to what; we *know* there must be a direct path between each subscriber node and its provider.
- sl_path entries are the second indicator; if sl_subscribe has not already indicated 'how to listen,' then a node may listen directly to the event's origin if there is a suitable sl_path entry.
- Lastly, if there has been no guidance thus far based on the above data sources, then nodes can listen indirectly via every node that is either a provider for the receiver, or that is using the receiver as a provider.

Any time sl_subscribe or sl_path are modified, `RebuildListenEntries()` will be called to revise the listener paths.

4.3 Slony-I Trigger Handling

In PostgreSQL version 8.3, new functionality was added where triggers and rules may have their behaviour altered via **ALTER TABLE**, to specify the following alterations:

- **DISABLE TRIGGER trigger_name**

- **ENABLE TRIGGER trigger_name**
- **ENABLE REPLICA TRIGGER trigger_name**
- **ENABLE ALWAYS TRIGGER trigger_name**
- **DISABLE RULE rewrite_rule_name**
- **ENABLE RULE rewrite_rule_name**
- **ENABLE REPLICA RULE rewrite_rule_name**
- **ENABLE ALWAYS RULE rewrite_rule_name**

A new GUC variable, `session_replication_role` controls whether the session is in `origin`, `replica`, or `local` mode, which then, in combination with the above enabling/disabling options, controls whether or not the trigger function actually runs.

We may characterize when triggers fire, under Slony-I replication, based on the following table; the same rules apply to PostgreSQL rules.

Trigger Form	When Established	Log Trigger	denyaccess Trigger	Action - origin	Action - replica	Action - local
DISABLE TRIGGER	User request	disabled on subscriber	enabled on subscriber	does not fire	does not fire	does not fire
ENABLE TRIGGER	Default	enabled on subscriber	disabled on subscriber	fires	does not fire	fires
ENABLE REPLICA TRIGGER	User request	inappropriate	inappropriate	does not fire	fires	does not fire
ENABLE ALWAYS TRIGGER	User request	inappropriate	inappropriate	fires	fires	fires

Table 4.1: Trigger Behaviour

There are, correspondingly, now, several ways in which Slony-I interacts with this. Let us outline those times that are interesting:

- Before replication is set up, *every* database starts out in 'origin' status, and, by default, all triggers are of the **ENABLE TRIGGER** form, so they all run, as is normal in a system uninvolved in replication.
- When a Slony-I subscription is set up, on the origin node, both the `logtrigger` and `denyaccess` triggers are added, the former being enabled, and running, the latter being disabled, so it does not run.

  From a locking perspective, each SLONIK SET ADD TABLE(7) request will need to briefly take out an exclusive lock on each table as it attaches these triggers, which is much the same as has always been the case with Slony-I.
- On the subscriber, the subscription process will add the same triggers, but with the polarities 'reversed', to protect data from accidental corruption on subscribers.

  From a locking perspective, again, there is not much difference from earlier Slony-I behaviour, as the subscription process, due to running **TRUNCATE**, copying data, and altering table schemas, requires *extensive* exclusive table locks, and the changes in trigger behaviour do not change those requirements.
- If you restore a backup of a Slony-I node (taken by pg_dump or any other method), and drop the Slony-I namespace, this now cleanly removes all Slony-I components, leaving the database, including its schema, in a 'pristine', consistent fashion, ready for whatever use may be desired.
- Section 3.3 is now performed in quite a different way: rather than altering each replicated table to 'take it out of replicated mode', Slony-I instead simply shifts into the **local** status for the duration of this event.

  On the origin, this deactivates the `logtrigger` trigger.

  On each subscriber, this deactivates the `denyaccess` trigger.

- At the time of invoking SLONIK MOVE SET(7) against the former origin, Slony-I must transform that node into a subscriber, which requires dropping the `lockset` triggers, disabling the `logtrigger` triggers, and enabling the `denyaccess` triggers.

  At about the same time, when processing SLONIK MOVE SET(7) against the new origin, Slony-I must transform that node into an origin, which requires disabling the formerly active `denyaccess` triggers, and enabling the `logtrigger` triggers.

  From a locking perspective Slony-I will need to take out exclusive locks to disable and enable the respective triggers.

- Similarly to SLONIK MOVE SET(7), SLONIK FAILOVER(7) transforms a subscriber node into an origin, which requires disabling the formerly active `denyaccess` triggers, and enabling the `logtrigger` triggers. The locking implications are again, much the same, requiring an exclusive lock on each such table.

4.3.1 TRUNCATE in PostgreSQL 8.4+

In PostgreSQL 8.4, triggers were augmented to support the **TRUNCATE** event. Thus, one may create a trigger which runs when one requests **TRUNCATE** on a table, as follows:

```
create trigger "_@CLUSTERNAME@_truncatetrigger"
   before truncate on my_table
   for each statement
     execute procedure @NAMESPACE@.log_truncate(22);
```

Slony-I supports this on nodes running PostgreSQL 8.4 and above, as follows:

- Tables have an additional two triggers attached to them:
  - `log_truncate(tab_id)`
    Running on the origin, this captures **TRUNCATE** requests, and stores them in sl_log_1 and sl_log_2 so that they are applied at the appropriate point on subscriber nodes.
  - `truncate_deny()`
    Running on subscriber nodes, this forbids running **TRUNCATE** directly against replicated tables on these nodes, in much the same way `denyAccess()` forbids running **INSERT/UPDATE/DELETE** directly against replicated tables.
- For each table, the command **TRUNCATE TABLE ONLY my_schema.my_table CASCADE;** is submitted.

  Various options were considered (see Bugzilla Bug #134), after which **CASCADE** was concluded to be the appropriate answer.

  Warning
  If you have a subscriber node where additional tables have gotten attached via foreign keys to a replicated table, then running **TRUNCATE** against that parent table will also **TRUNCATE** *all the children.*
  Of course, it should be observed that this was a *terribly dangerous* thing to have done because deleting data from the parent table would already either:
  - Lead to deleting data from the child tables, this meaning the addition of **TRUNCATE** support is really no change at all;
  - Lead to foreign keys being broken on the subscriber, causing replication to keel over.

  (In effect, we're not really worsening things.)

- Note that if a request truncates several tables (*e.g.* - as where a table has a hierachy of children), then a request will be logged in sl_log_1/sl_log_2 for each table, and the **TRUNCATE CASCADE** will *effectively* mean that the child tables will be truncated, first indirectly, then directly. If there is a hierarchy of 3 tables, `t1`, `t2`, and `t3`, then `t3` will get truncated three times. It's empty after the first **TRUNCATE**, so additional iterations will be cheap.
- If mixing PostgreSQL 8.3 and higher versions within a cluster:
  - PostgreSQL 8.3 nodes will not capture **TRUNCATE** requests, neither to log the need to propagate the TRUNCATE, nor to prevent it, on either origin or replica.
  - PostgreSQL 8.4 nodes *do* capture **TRUNCATE** requests for both purposes.
  - If a PostgreSQL 8.4+ node captures a **TRUNCATE** request, it will apply fine against a subscriber running PostgreSQL 8.3.

4.4 Locking Issues

One of the usual merits of the use, by PostgreSQL, of Multi-Version Concurrency Control (MVCC) is that this eliminates a whole host of reasons to need to lock database objects. On some other database systems, you need to acquire a table lock in order to insert data into the table; that can *severely* hinder performance. On other systems, read locks can impede writes; with MVCC, PostgreSQL eliminates that whole class of locks in that 'old reads' can access 'old tuples.' Most of the time, this allows the gentle user of PostgreSQL to not need to worry very much about locks. Slony-I configuration events normally grab locks on an internal table, `sl_config_lock`, which should not be visible to applications unless they are performing actions on Slony-I components.

Unfortunately, there are several sorts of Slony-I events that do require exclusive locks on PostgreSQL tables, with the result that modifying Slony-I configuration can bring back some of those 'locking irritations.' In particular:

- set add table

  A momentary exclusive table lock must be acquired on the 'origin' node in order to add the trigger that collects updates for that table. It only needs to be acquired long enough to establish the new trigger.

- move set

  When a set origin is shifted from one node to another, exclusive locks must be acquired on each replicated table on both the old origin and the new origin in order to change the triggers on the tables.

- lock set

  This operation expressly requests locks on each of the tables in a given replication set on the origin node.

- During the **SUBSCRIBE_SET** event on a new subscriber.

  all tables in the replication set will be locked via an exclusive lock for the entire duration of the process of subscription. By locking the tables early, this means that the subscription cannot fail after copying some of the data due to some other process having held on to a table.

  In any case, note that this one began with the wording 'on a new subscriber.' The locks are applied *on the new subscriber.* They are *not* applied on the provider or on the origin.

- Each time an event is generated (including SYNC events) Slony-I obtains an exclusive lock on the sl_event table long enough to insert the event into sl_event. This is not normally an issue as Slony-I should be the only program using sl_event. However this means that any non-slony transactions that read from sl_event can cause replication to pause. If you pg_dump your database avoid dumping your Slony schemas or else pg_dump's locking will compete with Slony's own locking which could stop Slony replication for the duration of the pg_dump. Exclude the Slony schemas from pg_dump with --exclude-schema=schemaname to specifically exclude your Slony schema.

When Slony-I locks a table that a running application later tries to access the application will be blocked waiting for the lock. It is also possible for a running application to create a deadlock situation with Slony-I when they each have obtained locks that the other is waiting for.

Several possible solutions to this are:

- Announce an application outage to avoid deadlocks

  If you can temporarily block applications from using the database, that will provide a window of time during which there is nothing running against the database other than administrative processes under your control.

- Try the operation, hoping for things to work

  Since nothing prevents applications from leaving access locks in your way, you may find yourself deadlocked. But if the number of remaining locks are small, you may be able to negotiate with users to 'get in edgewise.'

- Use pgpool

  If you can use this or some similar 'connection broker', you may be able to tell the connection manager to stop using the database for a little while, thereby letting it 'block' the applications for you. What would be ideal would be for the connection manager to hold up user queries for a little while so that the brief database outage looks, to them, like a period where things were running slowly.

- Rapid Outage Management

  The following procedure may minimize the period of the outage:

  – Modify `pg_hba.conf` so that only the **slony** user will have access to the database.
  – Issue a **kill -SIGHUP** to the PostgreSQL postmaster.

    This will not kill off existing possibly-long-running queries, but will prevent new ones from coming in. There is an application impact in that incoming queries will be rejected until the end of the process.
  – If 'all looks good,' then it should be safe to proceed with the Slony-I operation.
  – If some old query is lingering around, you may need to **kill -SIGQUIT** one of the PostgreSQL processes. This will restart the backend and kill off any lingering queries. You probably need to restart the slon(1) processes that attach to the node.

    At that point, it will be safe to proceed with the Slony-I operation; there will be no competing processes.
  – Reset `pg_hba.conf` to allow other users in, and **kill -SIGHUP** the postmaster to make it reload the security configuration.

4.5 Log Shipping - Slony-I with Files

Slony-I has the ability to serialize the updates to go out into log files that can be kept in a spool directory.

The spool files could then be transferred via whatever means was desired to a 'slave system,' whether that be via FTP, rsync, or perhaps even by pushing them onto a 1GB 'USB key' to be sent to the destination by clipping it to the ankle of some sort of 'avian transport' system.

There are plenty of neat things you can do with a data stream in this form, including:

- Replicating to nodes that *aren't* securable
- Replicating to destinations where it is not possible to set up bidirection communications
- Supporting a different form of PITR (Point In Time Recovery) that filters out read-only transactions and updates to tables that are not of interest.
- If some disaster strikes, you can look at the logs of queries in detail

  This makes log shipping potentially useful even though you might not intend to actually create a log-shipped node.
- This is a really slick scheme for building load for doing tests
- We have a data 'escrow' system that would become incredibly cheaper given log shipping
- You may apply triggers on the 'disconnected node ' to do additional processing on the data

  For instance, you might take a fairly 'stateful' database and turn it into a 'temporal' one by use of triggers that implement the techniques described in [**?**] by Richard T. Snodgrass.

1. *Where are the 'spool files' for a subscription set generated?*

   Any slon subscriber node can generate them by adding the `-a` option.

   Note
   Notice that this implies that in order to use log shipping, you must have at least one subscriber node.

2. *What takes place when a SLONIK FAILOVER(7)/ SLONIK MOVE SET(7) takes place?*

   Nothing special. So long as the archiving node remains a subscriber, it will continue to generate logs.

   Warning
   If the archiving node becomes the origin, on the other hand, it will continue to generate logs.

3. *What if we run out of 'spool space'?*

   The node will stop accepting **SYNC**s until this problem is alleviated. The database being subscribed to will also fall behind.

4. *How do we set up a subscription?*

   The script in `tools` called slony1_dump.sh is a shell script that dumps the 'present' state of the subscriber node. You need to start the slon for the subscriber node with logging turned on. At any point after that, you can run slony1_dump.sh, which will pull the state of that subscriber as of some **SYNC** event. Once the dump completes, all the **SYNC** logs generated from the time that dump *started* may be added to the dump in order to get a 'log shipping subscriber.'

5. *What are the limitations of log shipping?*

   In the initial release, there are rather a lot of limitations. As releases progress, hopefully some of these limitations may be alleviated/eliminated.

   The log shipping functionality amounts to 'sniffing' the data applied at a particular subscriber node. As a result, you must have at least one 'regular' node; you cannot have a cluster that consists solely of an origin and a set of 'log shipping nodes.'.

   The 'log shipping node' tracks the entirety of the traffic going to a subscriber. You cannot separate things out if there are multiple replication sets.

   The 'log shipping node' presently only fully tracks **SYNC** events. This should be sufficient to cope with *some* changes in cluster configuration, but not others. A number of event types *are* handled in such a way that log shipping copes with them:

   - **SYNC** events are, of course, handled.
   - **DDL_SCRIPT** is handled.
   - **UNSUBSCRIBE_SET**

     This event, much like **SUBSCRIBE_SET** is not handled by the log shipping code. But its effect is, namely that **SYNC** events on the subscriber node will no longer contain updates to the set.

     Similarly, **SET_DROP_TABLE**, **SET_DROP_SEQUENCE**, **SET_MOVE_TABLE**, **SET_MOVE_SEQUENCE**, **DROP_S**
     MERGE_SET, **SUBSCRIBE_SET** will be handled 'apropriately'.
   - The various events involved in node configuration are irrelevant to log shipping: **STORE_NODE**, **ENABLE_NODE**, **DROP_NODE**, **STORE_PATH**, **DROP_PATH**, **STORE_LISTEN**, **DROP_LISTEN**
   - Events involved in describing how particular sets are to be initially configured are similarly irrelevant: **STORE_SET**, **SET_ADD_TABLE**, **SET_ADD_SEQUENCE**, **STORE_TRIGGER**, **DROP_TRIGGER**,

   It would be nice to be able to turn a 'log shipped' node into a fully communicating Slony-I node that you could failover to. This would be quite useful if you were trying to construct a cluster of (say) 6 nodes; you could start by creating one subscriber, and then use log shipping to populate the other 4 in parallel. This usage is not supported, but presumably one could take an application outage and promote the log-shipping node to a normal slony node with the OMIT COPY option of SUBSCRIBE SET.

4.5.1 Usage Hints

Note
Here are some more-or-less disorganized notes about how you might want to use log shipping...

- You *don't* want to blindly apply **SYNC** files because any given **SYNC** file may *not* be the right one. If it's wrong, then the result will be that the call to `setsyncTracking_offline()` will fail, and your psql session will **ABORT**, and then run through the remainder of that **SYNC** file looking for a **COMMIT** or **ROLLBACK** so that it can try to move on to the next transaction.

  But we *know* that the entire remainder of the file will fail! It is futile to go through the parsing effort of reading the remainder of the file.

  Better idea:

- The table, on the log shipped node, tracks which log it most recently applied in table `sl_archive_tracking`. Thus, you may predict the ID number of the next file by taking the latest counter from this table and adding 1.
- There is still variation as to the filename, depending on what the overall set of nodes in the cluster are. All nodes periodically generate **SYNC** events, even if they are not an origin node, and the log shipping system does generate logs for such events. As a result, when searching for the next file, it is necessary to search for files in a manner similar to the following:

```
ARCHIVEDIR=/var/spool/slony/archivelogs/node4
SLONYCLUSTER=mycluster
PGDATABASE=logshipdb
PGHOST=logshiphost
NEXTQUERY="select at_counter+1 from \"_${SLONYCLUSTER}\".sl_archive_tracking;"
nextseq=`psql -d ${PGDATABASE} -h ${PGHOST} -A -t -c "${NEXTQUERY}"`
filespec=`printf "slony1_log_*_%20d.sql" $nextseq`
for file in `find $ARCHIVEDIR -name "${filespec}"`; do
   psql -d ${PGDATABASE} -h ${PGHOST} -f ${file}
done
```

-

4.5.2 find-triggers-to-deactivate.sh

It was once pointed out (Bugzilla bug #19) that the dump of a schema may include triggers and rules that you may not wish to have running on the log shipped node.

The tool `tools/find-triggers-to-deactivate.sh` was created to assist with this task. It may be run against the node that is to be used as a schema source, and it will list the rules and triggers present on that node that may, in turn need to be deactivated.

It includes `logtrigger` and `denyaccess` triggers which will may be left out of the extracted schema, but it is still worth the Gentle Administrator verifying that such triggers are kept out of the log shipped replica.

4.5.3 slony_logshipper Tool

As of version 1.2.12, Slony-I has a tool designed to help apply logs, called slony_logshipper. It is run with three sorts of parameters:

- Options, chosen from the following:
  - `h`
    display this help text and exit
  - `v`
    display program version and exit
  - `q`
    quiet mode
  - `l`
    cause running daemon to reopen its logfile
  - `r`
    cause running daemon to resume after error
  - `t`
    cause running daemon to enter smart shutdown mode
  - `T`
    cause running daemon to enter immediate shutdown mode
  - `c`
    destroy existing semaphore set and message queue (use with caution)

  - `f`
    stay in foreground (don't daemonize)
  - `w`
    enter smart shutdown mode immediately

- A specified log shipper configuration file

  This configuration file consists of the following specifications:

  - **logfile = './offline_logs/logshipper.log';**
    Where the log shipper will leave messages.
  - **cluster name = 'T1';**
    Cluster name
  - **destination database = 'dbname=slony_test3';**
    Optional conninfo for the destination database. If given, the log shipper will connect to this database, and apply logs to it.
  - **archive dir = './offline_logs';**
    The archive directory is required when running in 'database-connected' mode to have a place to scan for missing (unapplied) archives.
  - **destination dir = './offline_result';**
    If specified, the log shipper will write the results of data massaging into result logfiles in this directory.
  - **max archives = 3600;**
    This fights eventual resource leakage; the daemon will enter 'smart shutdown' mode automatically after processing this many archives.
  - **ignore table "public"."history";**
    One may filter out single tables from log shipped replication
  - **ignore namespace "public";**
    One may filter out entire namespaces from log shipped replication
  - **rename namespace "public"."history" to "site_001"."history";**
    One may rename specific tables.
  - **rename namespace "public" to "site_001";**
    One may rename entire namespaces.
  - **post processing command = 'gzip -9 $inarchive';**
    Pre- and post-processing commands are executed via `system(3)`.

  An '@' as the first character causes the exit code to be ignored. Otherwise, a nonzero exit code is treated as an error and causes processing to abort.

  Pre- and post-processing commands have two further special variables defined:

  - `$inarchive` - indicating incoming archive filename
  - `$outnarchive` - indicating outgoing archive filename

- **error command = ' (echo "archive=$inarchive" echo "error messages:" echo "$errortext") | mail -s "Slony log shipping failed" postgres@localhost ';**

  The error command indicates a command to execute upon encountering an error. All logging since the last successful completion of an archive is available in the `$errortext` variable.

  In the example shown, this sends an email to the DBAs upon encountering an error.

- Archive File Names

  Each filename is added to the SystemV Message queue for processing by a slony_logshipper process.

Chapter 5

Deployment Considerations

5.1 Cluster Monitoring

As a prelude to the discussion, it is worth pointing out that since the bulk of Slony-I functionality is implemented via running database functions and SQL queries against tables within a Slony-I schema, most of the things that one might want to monitor about replication may be found by querying tables in the schema created for the cluster in each database in the cluster.

Here are some of the tables that contain information likely to be particularly interesting from a monitoring and diagnostic perspective.

sl_status

This view is the first, most obviously useful thing to look at from a monitoring perspective. It looks at the local node's events, and checks to see how quickly they are being confirmed on other nodes.

The view is primarily useful to run against an origin ('master') node, as it is only there where the events generated are generally expected to require interesting work to be done. The events generated on non-origin nodes tend to be **SYNC** events that require no replication work be done, and that are nearly no-ops, as a result.

sl_confirm

Contains confirmations of replication events; this may be used to infer which events have, and *have not* been processed.

sl_event

Contains information about the replication events processed on the local node.

sl_log_1 and sl_log_2

These tables contain replicable data. On an origin node, this is the 'queue' of data that has not necessarily been replicated everywhere. By examining the table, you may examine the details of what data is replicable.

sl_node

The list of nodes in the cluster.

sl_path

This table holds connection information indicating how slon(1) processes are to connect to remote nodes, whether to access events, or to request replication data.

sl_listen

This configuration table indicates how nodes listen for events coming from other nodes. Usually this is automatically populated; generally you can detect configuration problems by this table being 'underpopulated.'

sl_registry

A configuration table that may be used to store miscellaneous runtime data. Presently used only to manage switching between the two log tables.

sl_seqlog

Contains the 'last value' of replicated sequences.

sl_set

Contains definition information for replication sets, which is the mechanism used to group together related replicable tables and sequences.

sl_setsync

Contains information about the state of synchronization of each replication set, including transaction snapshot data.

sl_subscribe

Indicates what subscriptions are in effect for each replication set.

sl_table

Contains the list of tables being replicated.

5.1.1 test_slony_state

This invaluable script does various sorts of analysis of the state of a Slony-I cluster. Some administrators recommend running these scripts frequently (hourly seems suitable) to find problems as early as possible.

You specify arguments including `database`, `host`, `user`, `cluster`, `password`, and `port` to connect to any of the nodes on a cluster. You also specify a `mailprog` command (which should be a program equivalent to Unix mailx) and a recipient of email.

You may alternatively specify database connection parameters via the environment variables used by libpq, *e.g.* - using `PGPORT`, `PGDATABASE`, `PGUSER`, `PGSERVICE`, and such.

The script then rummages through sl_path to find all of the nodes in the cluster, and the DSNs to allow it to, in turn, connect to each of them.

For each node, the script examines the state of things, including such things as:

- Checking sl_listen for some 'analytically determinable' problems. It lists paths that are not covered.
- Providing a summary of events by origin node

  If a node hasn't submitted any events in a while, that likely suggests a problem.
- Summarizes the 'aging' of table sl_confirm

  If one or another of the nodes in the cluster hasn't reported back recently, that tends to lead to cleanups of tables like sl_log_1, sl_log_2 and sl_seqlog not taking place.
- Summarizes what transactions have been running for a long time

  This only works properly if the statistics collector is configured to collect command strings, as controlled by the option `stats_command_string = true` in `postgresql.conf` .

  If you have broken applications that hold connections open, this will find them.

  If you have broken applications that hold connections open, that has several unsalutory effects as described in the FAQ.

The script does some diagnosis work based on parameters in the script; if you don't like the values, pick your favorites!

Note
Note that there are two versions, one using the 'classic' `Pg.pm` Perl module for accessing PostgreSQL databases, and one, with `dbi` in its name, that uses the newer Perl `DBI` interface. It is likely going to be easier to find packaging for `DBI`.

5.1.2 Nagios Replication Checks

The script in the `tools` directory called **psql_replication_check.pl** represents some of the best answers arrived at in attempts to build replication tests to plug into the Nagios system monitoring tool.

A former script, `test_slony_replication.pl`, took a 'clever' approach where a 'test script' is periodically run, which rummages through the Slony-I configuration to find origin and subscribers, injects a change, and watches for its propagation through the system. It had two problems:

- Connectivity problems to the *single* host where the test ran would make it look as though replication was destroyed. Overall, this monitoring approach has been fragile to numerous error conditions.
- Nagios has no ability to benefit from the 'cleverness' of automatically exploring the set of nodes. You need to set up a Nagios monitoring rule for each and every node being monitored.

The new script, **psql_replication_check.pl**, takes the minimalist approach of assuming that the system is an online system that sees regular 'traffic,' so that you can define a view specifically for the replication test called `replication_status` which is expected to see regular updates. The view simply looks for the youngest 'transaction' on the node, and lists its timestamp, age, and some bit of application information that might seem useful to see.

- In an inventory system, that might be the order number for the most recently processed order.
- In a domain registry, that might be the name of the most recently created domain.

An instance of the script will need to be run for each node that is to be monitored; that is the way Nagios works.

5.1.3 Monitoring Slony-I using MRTG

One user reported on the Slony-I mailing list how to configure mrtg - Multi Router Traffic Grapher to monitor Slony-I replication.

... Since I use mrtg to graph data from multiple servers I use snmp (net-snmp to be exact). On database server, I added the following line to snmpd configuration:

```
exec replicationLagTime  /cvs/scripts/snmpReplicationLagTime.sh 2
where  /cvs/scripts/snmpReplicationLagTime.sh looks like this:
```

```
#!/bin/bash
/home/pgdba/work/bin/psql -U pgdba -h 127.0.0.1 -p 5800 -d _DBNAME_ -qAt -c
"select cast(extract(epoch from st_lag_time) as int8) FROM _irr.sl_status
WHERE st_received = $1"
```

Then, in mrtg configuration, add this target:

```
Target[db_replication_lagtime]:extOutput.3&extOutput.3:public at db::30:::
MaxBytes[db_replication_lagtime]: 400000000
Title[db_replication_lagtime]: db: replication lag time
PageTop[db_replication_lagtime]: <H1>db: replication lag time</H1>
Options[db_replication_lagtime]: gauge,nopercent,growright
```

Alternatively, Ismail Yenigul points out how he managed to monitor slony using MRTG without installing SNMPD.

Here is the mrtg configuration

```
Target[db_replication_lagtime]:`/bin/snmpReplicationLagTime.sh 2`
MaxBytes[db_replication_lagtime]: 400000000
Title[db_replication_lagtime]: db: replication lag time
PageTop[db_replication_lagtime]: <H1>db: replication lag time</H1>
Options[db_replication_lagtime]: gauge,nopercent,growright
```

and here is the modified version of the script

```
# cat /bin/snmpReplicationLagTime.sh
#!/bin/bash

output=`/usr/bin/psql -U slony -h 192.168.1.1 -d endersysecm -qAt -c
"select cast(extract(epoch from st_lag_time) as int8) FROM _mycluster.sl_status WHERE  ↩
    st_received = $1"`
echo $output
echo $output
echo
echo
# end of script#
```

Note
MRTG expects four lines from the script, and since there are only two lines provided, the output must be padded to four lines.

5.1.4 Bucardo-related Monitoring

The Bucardo replication system includes script, **check_postgres.pl**, which can monitor a variety of things about PostgreSQL status that includes monitoring Slony-I replication status; see check_postgres.pl - slony_status

5.1.5 search-logs.sh

This script is constructed to search for Slony-I log files at a given path (`LOGHOME`), based both on the naming conventions used by the Section 6.1.4 and Section 6.1.1.20 systems used for launching slon(1) processes.

Errors, if found, are listed, by log file, and emailed to the specified user (`LOGRECIPIENT`); if no email address is specified, output goes to standard output.

`LOGTIMESTAMP` allows overriding what hour to evaluate (rather than the last hour).

An administrator might run this script once an hour to monitor for replication problems.

5.1.6 Building MediaWiki Cluster Summary

The script `mkmediawiki.pl` , in `tools`, may be used to generate a cluster summary compatible with the popular MediaWiki software. Note that the `--categories` permits the user to specify a set of (comma-delimited) categories with which to associate the output. If you have a series of Slony-I clusters, passing in the option `--categories=slony1` leads to the MediaWiki instance generating a category page listing all Slony-I clusters so categorized on the wiki.

The gentle user might use the script as follows:

```
~/logtail.en>         mvs login -d mywiki.example.info -u "Chris Browne" -p `cat ~/. ↩
    wikipass` -w wiki/index.php
Doing login with host: logtail and lang: en
~/logtail.en> perl $SLONYHOME/tools/mkmediawiki.pl --host localhost --database  ↩
    slonyregress1 --cluster slony_regress1 --categories=Slony-I  > Slony_replication.wiki
~/logtail.en> mvs commit -m "More sophisticated generated Slony-I cluster docs"  ↩
    Slony_replication.wiki
Doing commit Slony_replication.wiki with host: logtail and lang: en
```

Note that **mvs** is a client written in Perl; on Debian GNU/Linux, the relevant package is called libwww-mediawiki-client-perl; other systems may have a packaged version of this under some similar name.

5.1.7 Analysis of a SYNC

The following is (as of 2.0) an extract from the slon(1) log for node #2 in a run of 'test1' from the regression tests.

```
DEBUG2 remoteWorkerThread_1: SYNC 19 processing
INFO   about to monitor_subscriber_query - pulling big actionid list 134885072
INFO   remoteWorkerThread_1: syncing set 1 with 4 table(s) from provider 1
DEBUG2  ssy_action_list length: 0
DEBUG2 remoteWorkerThread_1: current local log_status is 0
DEBUG2 remoteWorkerThread_1_1: current remote log_status = 0
DEBUG1 remoteHelperThread_1_1: 0.028 seconds delay for first row
DEBUG1 remoteHelperThread_1_1: 0.978 seconds until close cursor
INFO   remoteHelperThread_1_1: inserts=144 updates=1084 deletes=0
INFO   remoteWorkerThread_1: sync_helper timing:  pqexec (s/count)- provider 0.063/6 -  ↩
    subscriber 0.000/6
INFO   remoteWorkerThread_1: sync_helper timing:  large tuples 0.315/288
DEBUG2 remoteWorkerThread_1: cleanup
INFO   remoteWorkerThread_1: SYNC 19 done in 1.272 seconds
INFO   remoteWorkerThread_1: SYNC 19 sync_event timing:  pqexec (s/count)- provider 0.001/1 ↩
     - subscriber 0.004/1 - IUD 0.972/248
```

Here are some notes to interpret this output:

- Note the line that indicates

  ```
  inserts=144 updates=1084 deletes=0
  ```

  This indicates how many tuples were affected by this particular SYNC.

- Note the line indicating

  ```
  0.028 seconds delay for first row
  ```

  This indicates the time it took for the

  ```
  LOG
  cursor
  ```

  to get to the point of processing the first row of data. Normally, this takes a long time if the SYNC is a large one, and one requiring sorting of a sizable result set.

- Note the line indicating

  ```
  0.978 seconds until
  close cursor
  ```

  This indicates how long processing took against the provider.

- sync_helper timing: large tuples 0.315/288

  This breaks off, as a separate item, the number of large tuples (*e.g.* - where size exceeded the configuration parameter sync_max_rowsize) and where the tuples had to be processed individually.

- ```
  SYNC 19 done in 1.272 seconds
  ```

  This indicates that it took 1.272 seconds, in total, to process this set of SYNCs.

- ```
  SYNC 19 sync_event timing:  pqexec (s/count)- provider 0.001/1 - subscriber 0.004/0 - IUD  ↩
      0.972/248
  ```

  This records information about how many queries were issued against providers and subscribers in function `sync_event()`, and how long they took.

  Note that 248 does not match against the numbers of inserts, updates, and deletes, described earlier, as I/U/D requests are clustered into groups of queries that are submitted via a single `pqexec()` call on the subscriber.

- ```
  sync_helper timing:  pqexec (s/count)- provider 0.063/6 - subscriber 0.000/6
  ```

  This records information about how many queries were issued against providers and subscribers in function `sync_helper-()`, and how long they took.

5.2 Component Monitoring

There are several ways available to see what Slony-I processes are up to:

- Section 5.2.1
- Section 5.2.2

5.2.1 Looking at `pg_stat_activity` view

The standard PostgreSQL view `pg_stat_activity` indicates what the various database connections are up to.

On recent versions of PostgreSQL, this view includes an attribute, `application_name`, which Slony-I components populate based on the names of their respective threads.

5.2.2 Looking at sl_components view

Slony-I has a table, sl_components, introduced in Slony-I 2.1, which captures Slony-I activity for each node.

```
slonyregress1@localhost->  select * from _slony_regress1.sl_components  order by co_actor;
       co_actor       | co_pid | co_node | co_connection_pid |   co_activity    |       ↩
           co_starttime      | co_event | co_eventtype
----------------------+--------+---------+-------------------+------------------+-------------------

 local_cleanup        |  24586 |       0 |             24907 | thread main loop |  ↩
    2011-02-24 17:02:55+00 |          | n/a
 local_listen         |  24896 |       1 |             24900 | thread main loop |  ↩
    2011-02-24 17:03:07+00 |          | n/a
 local_monitor        |  24586 |       0 |             24909 | thread main loop |  ↩
    2011-02-24 17:02:55+00 |          | n/a
 local_sync           |  24517 |       0 |             24906 | thread main loop |  ↩
    2011-02-24 17:03:09+00 |          | n/a
 remote listener      |  24586 |       2 |             24910 | thread main loop |  ↩
    2011-02-24 17:03:03+00 |          | n/a
 remoteWorkerThread_2 |  24586 |       2 |             24908 | thread main loop |  ↩
    2011-02-24 17:02:55+00 |          | n/a
(6 rows)
```

This example indicates the various Slony-I threads that are typically running as part of a slon(1) process:

local_cleanup

This thread periodically wakes up to trim obsolete data and (optionally) vacuum Slony-I tables

local_listen

This thread listens for events taking place on the local node, and changes the slon(1)'s configuration as needed.

local_monitor

This thread is rather self-referential, here; it manages the queue of events to be published to the sl_components table.
```

local_sync

This thread generates SYNC events on the local database. If the local database is the origin for a replication set, those SYNC events are used to propagate changes to other nodes in the cluster.

remote_listener

This thread listens for events on a remote node database, and queues them into the remote worker thread for that node.

remoteWorkerThread_2

This thread waits for events (from the remote listener thread), and takes action. This is the thread that does most of the visible work of replication.

5.2.3 Notes On Interpreting Component Activity

- Many of these will typically report, as their activity, **thread main loop**, which indicates that the thread exists, and is simply executing its main loop, waiting to have work to do.

  Most threads will never indicate an event or event type, as they do not process Slony-I events.

- `local_monitor` thread never reports any activity.

  It would be a nice idea for this thread, which manages sl_components, to report in on its work. Unfortunately, the fact of adding in its own events would make it perpetually busy, as the action of processing the queue would add a monitoring entry, in effect becoming a repetitive recursive activity.

  It does report in when it starts up, which means you may expect that this entry indicates the time at which the slon(1) process began.

- Timestamps are based on the clock time of the slon(1) process.

  In order for the timestamps to be accurate, it is important to use NTP or similar technology to keep servers' clocks synchronized, as recommended in Section 1.2. If the host where a slon(1) runs has its time significantly out of sync with the database that it manages, queries against sl_components may provide results that will confuse the reader.

- process.

5.3 Partitioning Support

Slony-I does not directly provide support for the PostgreSQL methodology of partitioning via inheritance, but it does not, by the same token, prevent the Gentle User from using that sort of replication scheme, and then replicating the underlying tables.

One of the tests in the regression tests called `testinherit`, tests that Slony-I behaves as expected to replicate data across partitions. This test creates a master `sales_data` table, from which various children inherit:

- `us_east`
- `us_west`
- `canada`

The example is somewhat simplistic as it only provides rules to handle initial insertion into the respective partitions; it does not then support allowing tuples to migrate from partition to partition if they are altered via an **UPDATE** statement. On the other hand, unlike with many partitioning cases, this one permits the 'parent' table to contain tuples.

Things worth observing include:

- Each partition table must be added to replication individually.
- Slony-I is not aware of the relationship between partitions; it simply regards them as a series of individual tables.

5.3.1 Support for Dynamic Partition Addition

One common 'use case' of replication is to partition large data sets based on time period, whether weekly, monthly, quarterly, or annually, where there is therefore a need to periodically add a new partition.

The traditional approach taken to this in Slony-I would be the following:

- SLONIK EXECUTE SCRIPT(7) to add the new partition(s) on each node
- SLONIK CREATE SET(7) to create a temporary replication set
- SLONIK SET ADD TABLE(7) to add the table(s) to that set
- SLONIK SUBSCRIBE SET(7), once for each subscriber node, to set up replication of the table on each node
- SLONIK MERGE SET(7), once subscriptions are running, to eliminate the temporary set

In view of the fact that we can be certain that a thus-far-unused partition will be empty, we offer an alternative mechanism which evades the need to create extra replication sets and the need to submit multiple SLONIK SUBSCRIBE SET(7) requests. The alternative is as follows; we use SLONIK EXECUTE SCRIPT(7), extending the DDL script thus:

- Add the new partition(s) on each node
- Run a Slony-I stored function to mark the new partition as being a replicated table

  On the origin node, if the table is found to have tuples in it, the DDL script will be aborted, as the precondition that it be empty has been violated.

  On subscriber nodes, we may safely **TRUNCATE** the new table.

There are several stored functions provided to support this; the Gentle User may use whichever seems preferable. The 'base function' is `add_empty_table_to_replication()`; the others provide additional structure and validation of the arguments

- `add_empty_table_to_replication (set_id, tab_id, nspname, tabname, idxname, comment);`

  This is the 'base' function; you must specify the set ID, table ID, namespace name, table name, index name, and a comment, and this table will be added to replication.

  Note that the index name is optional; if NULL, the function will look up the primary key for the table, assuming one exists, and fail if it does not exist.

- `replicate_partition(tab_id, nspname, tabname, idxname, comment);`

  If it is known that the table to be replicated inherits from a replicated parent table, then this function can draw set and origin information from that parent table.

Note
As has been observed previously, Slony-I is unaware that tables are partitioned. Therefore, this approach may also be used with confidence to add any table to replication that is known to be empty.

5.4 Slony-I Upgrade

Minor Slony-I versions can be upgraded using the slonik SLONIK UPDATE FUNCTIONS(7) command. This includes upgrades from 2.1.x to a newer version 2.1.y version or from 2.0.x to 2.1.y.

When upgrading Slony-I, the installation on all nodes in a cluster must be upgraded at once, using the slonik(1) command SLONIK UPDATE FUNCTIONS(7).

While this requires temporarily stopping replication, it does not forcibly require an outage for applications that submit updates.

The proper upgrade procedure is thus:

- Stop the slon(1) processes on all nodes. (*e.g.* - old version of slon(1))
- Install the new version of slon(1) software on all nodes.
- Execute a slonik(1) script containing the command **update functions (id = [whatever]);** for each node in the cluster.

> **Note**
> Remember that your slonik upgrade script like all other slonik scripts must contain the proper preamble commands to function.

- Start all slons.

The overall operation is relatively safe: If there is any mismatch between component versions, the slon(1) will refuse to start up, which provides protection against corruption.

You need to be sure that the C library containing SPI trigger functions has been copied into place in the PostgreSQL build. There are multiple possible approaches to this:

The trickiest part of this is ensuring that the C library containing SPI functions is copied into place in the PostgreSQL build; the easiest and safest way to handle this is to have two separate PostgreSQL builds, one for each Slony-I version, where the postmaster is shut down and then restarted against the 'new' build; that approach requires a brief database outage on each node.

While that approach has been found to be easier and safer, nothing prevents one from carefully copying Slony-I components for the new version into place to overwrite the old version as the 'install' step. That might *not* work on Windows™ if it locks library files that are in use. It is also important to make sure that any connections to the database are restarted after the new binary is installed.

Run make install to install new Slony-I components on top of the old If you build Slony-I on the same system on which it is to be deployed, and build from sources, overwriting the old with the new is as easy as **make install**. There is no need to restart a database backend; just to stop slon(1) processes, run the **UPDATE FUNCTIONS** script, and start new slon(1) processes.

Unfortunately, this approach requires having a build environment on the same host as the deployment. That may not be consistent with efforts to use common PostgreSQL and Slony-I binaries across a set of nodes.

Create a new PostgreSQL and Slony-I build With this approach, the old PostgreSQL build with old Slony-I components persists after switching to a new PostgreSQL build with new Slony-I components. In order to switch to the new Slony-I build, you need to restart the PostgreSQL **postmaster**, therefore interrupting applications, in order to get it to be aware of the location of the new components.

5.4.1 Incompatibilties between 2.0 and 2.1

5.4.1.1 Automatic Wait For

Slonik will now sometimes wait for previously submitted events to be confirmed before submittng additional events. This is described in Section 4.1

5.4.1.2 SNMP Support

In version 2.0 Slony-I could be built with SNMP support. This allowed Slony-I to send SNMP messages. This has been removed in version 2.1

5.4.2 Incompatibilities between 1.2 and 2.0

5.4.2.1 TABLE ADD KEY issue in Slony-I 2.0

The TABLE ADD KEY slonik command has been removed in version 2.0. This means that all tables must have a set of columns that form a unique key for the table. If you are upgrading from a previous Slony-I version and are using a Slony-I created primary key then you will need to modify your table to have its own primary key before installing Slony-I version 2.0

5.4.2.2 New Trigger Handling in Slony-I Version 2

One of the major changes to Slony-I is that enabling/disabling of triggers and rules now takes place as plain SQL, supported by PostgreSQL 8.3+, rather than via 'hacking' on the system catalog.

As a result, Slony-I users should be aware of the PostgreSQL syntax for **ALTER TABLE**, as that is how they can accomplish what was formerly accomplished via SLONIK STORE TRIGGER(7) and SLONIK DROP TRIGGER(7).

5.4.2.3 SUBSCRIBE SET goes to the origin

New in 2.0.5 (but not older versions of 2.0.x) is that SLONIK SUBSCRIBE SET(7) commands are submitted by slonik to the set origin not the provider. This means that you only need to issue SLONIK WAIT FOR EVENT(7) on the set origin to wait for the subscription process to complete.

5.4.2.4 WAIT FOR EVENT requires WAIT ON

With version 2.0 the WAIT FOR EVENT slonik command requires that the WAIT ON parameter be specified. Any slonik scripts that were assuming a default value will need to be modified

5.4.3 Upgrading to Slony-I version 2.1 from version 2.0

Slony-I version 2.0 can be upgraded to version 2.1 using the slonik(1) command SLONIK UPDATE FUNCTIONS(7).

While this requires temporarily stopping replication, it does not forcibly require an outage for applications that submit updates.

The proper upgrade procedure is thus:

- Stop the slon(1) processes on all nodes. (*e.g.* - old version of slon(1))
- Install the new version of Slony-I; software on all nodes (including new versions of the shared functions and libraries) .
- Execute a slonik(1) script containing the command **update functions (id = [whatever]);** for each node in the cluster. This will alter the structure of some of the Slony-I configuration tables.

  Note
  Remember that your slonik upgrade script like all other slonik scripts must contain the proper preamble commands to function.

- Start all slons.

5.4.4 Upgrading to Slony-I version 2.1 from version 1.2 or earlier

The version 2 branch is *substantially* different from earlier releases, dropping support for versions of PostgreSQL prior to 8.3, as in version 8.3, support for a 'session replication role' was added, thereby eliminating the need for system catalog hacks as well as the not-entirely-well-supported `xxid` data type.

As a result of the replacement of the `xxid` type with a (native-to-8.3) PostgreSQL transaction XID type, the slonik(1) command SLONIK UPDATE FUNCTIONS(7) is quite inadequate to the process of upgrading earlier versions of Slony-I to version 2.

In version 2.0.2, we have added a new option to SLONIK SUBSCRIBE SET(7), **OMIT COPY**, which allows taking an alternative approach to upgrade which amounts to:

- Uninstall old version of Slony-I

  When Slony-I uninstalls itself, catalog corruptions are fixed back up.

- Install Slony-I version 2

- Resubscribe, with **OMIT COPY**

Warning
There is a large 'foot gun' here: during part of the process, Slony-I is not installed in any form, and if an application updates one or another of the databases, the resubscription, omitting copying data, will be left with data *out of sync.* The administrator *must take care*; Slony-I has no way to help ensure the integrity of the data during this process.

The following process is suggested to help make the upgrade process as safe as possible, given the above risks.

- Use Section 6.1.10 to generate a slonik(1) script to recreate the replication cluster.

  Be sure to verify the SLONIK ADMIN CONNINFO(7) statements, as the values are pulled are drawn from the PATH configuration, which may not necessarily be suitable for running slonik(1).

  This step may be done before the application outage.

- Determine what triggers have SLONIK STORE TRIGGER(7) configuration on subscriber nodes.

  Trigger handling has fundamentally changed between Slony-I 1.2 and 2.0.

  Generally speaking, what needs to happen is to query `sl_table` on each node, and, for any triggers found in `sl_table`, it is likely to be appropriate to set up a script indicating either **ENABLE REPLICA TRIGGER** or **ENABLE ALWAYS TRIGGER** for these triggers.

  This step may be done before the application outage.

- Begin an application outage during which updates should no longer be applied to the database.

- To ensure that applications cease to make changes, it would be appropriate to lock them out via modifications to `pg_hba.conf`

- Ensure replication is entirely caught up, via examination of the `sl_status` view, and any application data that may seem appropriate.

- Shut down slon(1) processes.

- Uninstall the old version of Slony-I from the database.

  This involves running a slonik(1) script that runs SLONIK UNINSTALL NODE(7) against each node in the cluster.

- Ensure new Slony-I binaries are in place.

  A convenient way to handle this is to have old and new in different directories alongside two PostgreSQL builds, stop the postmaster, repoint to the new directory, and restart the postmaster.

- Run the script that reconfigures replication as generated earlier.

  This script should probably be split into two portions to be run separately:

  - Firstly, set up nodes, paths, sets, and such
  - At this point, start up slon(1) processes
  - Then, run the portion which runs SLONIK SUBSCRIBE SET(7)

  Splitting the Section 6.1.10 script as described above is left as an exercise for the reader.

- If there were triggers that needed to be activated on subscriber nodes, this is the time to activate them.

- At this point, the cluster should be back up and running, ready to be reconfigured so that applications may access it again.

5.5 Log Analysis

Here are some of things that you may find in your Slony-I logs, and explanations of what they mean.

5.5.1 CONFIG notices

These entries are pretty straightforward. They are informative messages about your configuration.

Here are some typical entries that you will probably run into in your logs:

```
CONFIG main: local node id = 1
CONFIG main: loading current cluster configuration
CONFIG storeNode: no_id=3 no_comment='Node 3'
CONFIG storePath: pa_server=5 pa_client=1 pa_conninfo="host=127.0.0.1 dbname=foo user= ↩
    postgres port=6132" pa_connretry=10
CONFIG storeListen: li_origin=3 li_receiver=1 li_provider=3
CONFIG storeSet: set_id=1 set_origin=1 set_comment='Set 1'
CONFIG main: configuration complete - starting threads
```

5.5.2 INFO notices

Events that take place that seem like they will generally be of interest are recorded at the INFO level, and, just as with CONFIG notices, are always listed.

5.5.3 DEBUG Notices

Debug notices are of less interest, and will quite likely only need to be shown if you are running into some problem with Slony-I.

5.5.4 Thread name

Notices are always prefaced by the name of the thread from which the notice originates. You will see messages from the following threads:

localListenThread This is the local thread that listens for events on the local node.

remoteWorkerThread-X The thread processing remote events. You can expect to see one of these for each node that this node communicates with.

remoteListenThread-X Listens for events on a remote node database. You may expect to see one of these for each node in the cluster.

cleanupThread Takes care of things like vacuuming, cleaning out the confirm and event tables, and deleting old data.

syncThread Generates SYNC events.

How much information they display is controlled by the `log_level` slon(1) parameter; ERROR/WARN/CONFIG/INFO messages will always be displayed, while choosing increasing values from 1 to 4 will lead to additional DEBUG level messages being displayed.

5.5.5 How to read Slony-I logs

Note that as far as slon is concerned, there is no 'master' or 'slave.' They are just nodes.

What you can expect, initially, is to see, on both nodes, some events propagating back and forth. Firstly, there should be some events published to indicate creation of the nodes and paths. If you don't see those, then the nodes aren't properly communicating with one another, and nothing else will happen...

- Create the two nodes.

  No slons are running yet, so there are no logs to look at.

- Start the two slons

  The logs for each will start out very quiet, as neither node has much to say, and neither node knows how to talk to another node. Each node will periodically generate a **SYNC** event, but recognize *nothing* about what is going on on other nodes.

- Do the SLONIK STORE PATH(7) to set up communications paths. That will allow the nodes to start to become aware of one another.

  The slon logs should now start to receive events from 'foreign' nodes.

  If you look at the contents of the tables sl_node and sl_path and sl_listen, on each node, that should give a good idea as to where things stand. Until the slon(1) starts, each node may only be partly configured. If there are two nodes, there should be two entries in all three of these tables once the communications configuration is set up properly. If there are fewer entries than that, well, that should give you some idea of what is missing.

- You'll set up the set (SLONIK CREATE SET(7)), add tables (SLONIK SET ADD TABLE(7)), and sequences (SLONIK SET ADD SEQUENCE(7)), and will see relevant events Section 5.5.6.7 only in the logs for the origin node for the set.

- Then, when you submit the SLONIK SUBSCRIBE SET(7) request, the event should go to both nodes.

  The origin node has little more to do, after that... The subscriber will then have a **COPY_SET** event, which will lead to logging information about adding each table and copying its data. See Section 5.5.6.4 for more details.

After that, you'll mainly see two sorts of behaviour:

- On the origin, there won't be too terribly much logged, just indication that some **SYNC** events are being generated and confirmed by other nodes. See Section 5.5.6.6 to see the sorts of log entries to expect.

- On the subscriber, there will be reports of **SYNC** events, and that the subscriber pulls data from the provider for the relevant set(s). This will happen infrequently if there are no updates going to the origin node; it will happen frequently when the origin sees heavy updates.

5.5.6 Log Messages and Implications

This section lists numerous of the error messages found in Slony-I, along with a brief explanation of implications. It is a fairly comprehensive list, only leaving out some of the **DEBUG4** messages that are almost always uninteresting.

5.5.6.1 Log Messages Associated with Log Shipping

Most of these represent errors that come up if the Section 4.5 functionality breaks. You may expect things to break if the filesystem being used for log shipping fills, or if permissions on that directory are wrongly set.

- **ERROR: remoteWorkerThread_%d: log archive failed %s - %s\n**

  This indicates that an error was encountered trying to write a log shipping file. Normally the slon(1) will retry, and hopefully succeed.

- **DEBUG2: remoteWorkerThread_%d: writing archive log...**

  This indicates that a log shipping archive log is being written for a particular **SYNC** set.

- **INFO: remoteWorkerThread_%d: Run Archive Command %s**

  If slon(1) has been configured (`-x` aka `command_on_logarchive`) to run a command after generating each log shipping archive log, this reports when that process is spawned using `system()`.

- **ERROR: remoteWorkerThread_%d: Could not open COPY SET archive file %s - %s**

  Seems pretty self-explanatory...

- **ERROR: remoteWorkerThread_%d: Could not generate COPY SET archive header %s - %s**

  Probably means that we just filled up a filesystem...

- **ERROR remoteWorkerThread_%d: "update "_slony_regress1".sl_archive_counter set ac_num = ac_num + 1, ac_timestamp = CURRENT_TIMESTAMP; select ac_num, ac_timestamp from "_slony_regress1".sl_archive_counter; " PGRES_FATAL_E ERROR: could not serialize access due to concurrent update**

  This may occasionally occur when using logshipping; this will typically happen if there are 3 or more nodes, and there is an attempt to concurrently process events sourced from different nodes. This does not represent any serious problem; Slony-I will retry the event which failed without the need for administrative intervention.

5.5.6.2 Log Messages - DDL scripts

The handling of DDL is somewhat fragile, as described in Section 3.3; here are both informational and error messages that may occur in the progress of an SLONIK EXECUTE SCRIPT(7) request.

- **ERROR: remoteWorkerThread_%d: DDL preparation failed - set %d - only on node %**

  Something broke when applying a DDL script on one of the nodes. This is quite likely indicates that the node's schema differed from that on the origin; you may need to apply a change manually to the node to allow the event to proceed. The scary, scary alternative might be to delete the offending event, assuming it can't possibly work...

- **SLON_CONFIG: remoteWorkerThread_%d: DDL request with %d statements**

  This is informational, indicating how many SQL statements were processed.

- **SLON_ERROR: remoteWorkerThread_%d: DDL had invalid number of statements - %d**

  Occurs if there were < 0 statements (which should be impossible) or > MAXSTATEMENTS statements. Probably the script was bad...

- **ERROR: remoteWorkerThread_%d: malloc() failure in DDL_SCRIPT - could not allocate %d bytes of memory**

  This should only occur if you submit some extraordinarily large DDL script that makes a slon(1) run out of memory

- **CONFIG: remoteWorkerThread_%d: DDL Statement %d: [%s]**

  This lists each DDL statement as it is submitted.

- **ERROR: DDL Statement failed - %s**

  Oh, dear, one of the DDL statements that worked on the origin failed on this remote node...

- **CONFIG: DDL Statement success - %s**

  All's well...

- **ERROR: remoteWorkerThread_%d: Could not generate DDL archive tracker %s - %s**

  Apparently the DDL script couldn't be written to a log shipping file...

- **ERROR: remoteWorkerThread_%d: Could not submit DDL script %s - %s**

  Couldn't write the script to a log shipping file.

- **ERROR: remoteWorkerThread_%d: Could not close DDL script %s - %s**

  Couldn't close a log shipping file for a DDL script.

- **ERROR: Slony-I ddlScript_prepare(): set % not found**

  Set wasn't found on this node; you probably gave the wrong ID number...

- **ERROR: Slony-I ddlScript_prepare_int(): set % not found**

  Set wasn't found on this node; you probably gave the wrong ID number...

- **ERROR: Slony-I: alterTableForReplication(): Table with id % not found**

  Apparently the table wasn't found; could the schema be messed up?

- **ERROR: Slony-I: alterTableForReplication(): Table % is already in altered state**

  Curious... We're trying to set a table up for replication a *second* time?

- **ERROR: Slony-I: alterTableRestore(): Table with id % not found**

  This runs when a table is being restored to 'non-replicated' state; apparently the replicated table wasn't found.

- **ERROR: Slony-I: alterTableRestore(): Table % is not in altered state**

  Hmm. The table isn't in altered replicated state. That shouldn't be, if replication had been working properly...

- **NOTICE: Slony-I: alterTableForReplication(): multiple instances of trigger % on table %",**

  This normally happens if you have a table that had a trigger attached to it that replication hid due to this being a subscriber node, and then you added a trigger by the same name back to replication. Now, when trying to "fix up" triggers, those two triggers conflict.

  The DDL script will keep running and rerunning, or the UNINSTALL NODE will keep failing, until you drop the 'visible' trigger, by hand, much as you must have added it, by hand, earlier.

- **ERROR: Slony-I: Unable to disable triggers**

  This is the error that follows the 'multiple triggers' problem.

5.5.6.3 Threading Issues

There should not be any 'user-serviceable' aspects to the Slony-I threading model; each slon(1) creates a well-specified set of helper threads to manage the various database connections that it requires. The only way that anything should break on the threading side is if you have not compiled PostgreSQL libraries to 'play well' with threading, in which case you will be unable to compile Slony-I in the first place.

- **FATAL: remoteWorkerThread_%d: pthread_create() - %s**

  Couldn't create a new remote worker thread.

- **DEBUG1 remoteWorkerThread_%d: helper thread for provider %d created**

  This normally happens when the slon(1) starts: a thread is created for each node to which the local node should be listening for events.

- **DEBUG1: remoteWorkerThread_%d: helper thread for provider %d terminated**

  If subscriptions reshape such that a node no longer provides a subscription, then the thread that works on that node can be dropped.

- **DEBUG1: remoteWorkerThread_%d: disconnecting from data provider %d**

  A no-longer-used data provider may be dropped; if connection information is changed, the slon(1) needs to disconnect and reconnect.

- **DEBUG2: remoteWorkerThread_%d: ignore new events due to shutdown**

  If the slon(1) is shutting down, it is futile to process more events

- **DEBUG1: remoteWorkerThread_%d: node %d - no worker thread**

  Curious: we can't wake up the worker thread; there probably should already be one...

5.5.6.4 Log Entries At Subscription Time

Subscription time is quite a special time in Slony-I. If you have a large amount of data to be copied to subscribers, this may take a considerable period of time. Slony-I logs a fairly considerable amount of information about its progress, which is sure to be useful to the gentle reader. In particular, it generates log output every time it starts and finishes copying data for a given table as well as when it completes reindexing the table. That may not make a 28 hour subscription go any faster, but at least helps you have some idea of how it is progressing.

- **DEBUG1: copy_set %d**

  This indicates the beginning of copying data for a new subscription.

- **ERROR: remoteWorkerThread_%d: set %d not found in runtime configuration**

  slon(1) tried starting up a subscription; it couldn't find conninfo for the data source. Perhaps paths are not properly propagated?

- **ERROR: remoteWorkerThread_%d: node %d has no pa_conninfo**

  Apparently the conninfo configuration was *wrong*...

- **ERROR: copy set %d cannot connect to provider DB node %d**

  slon(1) couldn't connect to the provider. Is the conninfo wrong? Or perhaps authentication is misconfigured? Or perhaps the database is down?

- **DEBUG1: remoteWorkerThread_%d: connected to provider DB**

  Excellent: the copy set has a connection to its provider

- **ERROR: Slony-I: sequenceSetValue(): sequence % not found**

  Curious; the sequence object is missing. Could someone have dropped it from the schema by hand (*e.g.* - not using SLONIK EXECUTE SCRIPT(7))?

- **ERROR: Slony-I: subscribeSet() must be called on provider**

  This function should only get called on the provider node. slonik(1) normally handles this right, unless one had wrong DSNs in a slonik(1) script...

- **ERROR: Slony-I: subscribeSet(): set % not found**

  Hmm. The provider node isn't aware of this set. Wrong parms to a slonik(1) script?

- **ERROR: Slony-I: subscribeSet(): set origin and receiver cannot be identical**

  Duh, an origin node can't subscribe to itself.

- **ERROR: Slony-I: subscribeSet(): set provider and receiver cannot be identical**

  A receiver must subscribe to a *different* node...

- **Slony-I: subscribeSet(): provider % is not an active forwarding node for replication set %**

  You can only use a live, active, forwarding provider as a data source.

- Slony-I: subscribeSet_int(): set % is not active, cannot change provider

  You can't change the provider just yet...

- **Slony-I: subscribeSet_int(): set % not found**

  This node isn't aware of the set... Perhaps you submitted wrong parms?

- **Slony-I: unsubscribeSet() must be called on receiver**

  Seems obvious... This probably indicates a bad slonik(1) admin DSN...

- **Slony-I: Cannot unsubscribe set % while being provider**

  This should seem obvious; SLONIK UNSUBSCRIBE SET(7) will fail if a node has dependent subscribers for which it is the provider

- **Slony-I: cleanupEvent(): Single node - deleting events < %**

  If there's only one node, the cleanup event will delete old events so that you don't get 'build-up of crud.'

- **Slony-I: determineIdxnameUnique(): table % not found**

  Did you properly copy over the schema to a new node???

- **Slony-I: table % has no primary key**

  This likely signifies a bad loading of schema...

- **Slony-I: table % has no unique index %**

  This likely signifies a bad loading of schema...

- **WARN: remoteWorkerThread_%d: transactions earlier than XID %s are still in progress**

  This indicates that some old transaction is in progress from before the earliest available **SYNC** on the provider. Slony-I cannot start replicating until that transaction completes. This will repeat until thetransaction completes...

- **DEBUG2: remoteWorkerThread_%d: prepare to copy table %s**

  This indicates that slon(1) is beginning preparations to set up subscription for a table.

- **ERROR: remoteWorkerThread_%d: Could not lock table %s on subscriber**

  For whatever reason, the table could not be locked, so the subscription needs to be restarted. If the problem was something like a deadlock, retrying may help. If the problem was otherwise, you may need to intervene...

- **DEBUG2: remoteWorkerThread_%d: all tables for set %d found on subscriber**

  An informational message indicating that the first pass through the tables found no problems...

- **DEBUG2: remoteWorkerThread_%d: copy sequence %s**

  Processing some sequence...

- **DEBUG2: remoteWorkerThread_%d: copy table %s**

  slon(1) is starting to copy a table...

- **DEBUG3: remoteWorkerThread_%d: table %s Slony-I serial key added local**

  Just added new column to the table to provide surrogate primary key.

- **DEBUG3: remoteWorkerThread_%d: local table %s already has Slony-I serial key**

  Did not need to add serial key; apparently it was already there.

- **DEBUG3: remoteWorkerThread_%d: table %s does not require Slony-I serial key**

  Apparently this table didn't require a special serial key...

- **DEBUG3: remoteWorkerThread_%d: table %s Slony-I serial key added local**

- **DEBUG2: remoteWorkerThread_%d: Begin COPY of table %s**

  slon(1) is about to start the COPY on both sides to copy a table...

- **ERROR: remoteWorkerThread_%d: Could not generate copy_set request for %s - %s**

  This indicates that the **delete/copy** requests failed on the subscriber. The slon(1) will repeat the **COPY_SET** attempt; it will probably continue to fail..

- **ERROR: remoteWorkerThread_%d: copy to stdout on provider - %s %s**

  Evidently something about the COPY to `stdout` on the provider node broke... The event will be retried...

- **ERROR: remoteWorkerThread_%d: copy from stdin on local node - %s %s**

  Evidently something about the COPY into the table on the subscriber node broke... The event will be retried...

- **DEBUG2: remoteWorkerThread_%d: %d bytes copied for table %s**

  This message indicates that the COPY of the table has completed. This is followed by running **ANALYZE** and reindexing the table on the subscriber.

- **DEBUG2: remoteWorkerThread_%d: %.3f seconds to copy table %s**

  After this message, copying and reindexing and analyzing the table on the subscriber is complete.

- **DEBUG2: remoteWorkerThread_%d: set last_value of sequence %s (%s) to %s**

  As should be no surprise, this indicates that a sequence has been processed on the subscriber.

- **DEBUG2: remoteWorkerThread_%d: %.3 seconds to copy sequences**

  Summarizing the time spent processing sequences in the **COPY_SET** event.

- **ERROR: remoteWorkerThread_%d: query %s did not return a result**

  This indicates that the query, as part of final processing of **COPY_SET**, failed. The copy will restart...

- **DEBUG2: remoteWorkerThread_%d: copy_set no previous SYNC found, use enable event**

  This takes place if no past SYNC event was found; the current event gets set to the event point of the **ENABLE_SUBSCRIPTION** event.

- **DEBUG2: remoteWorkerThread_%d: copy_set SYNC found, use event seqno %s**

  This takes place if a SYNC event was found; the current event gets set as shown.

- **ERROR: remoteWorkerThread_%d: sl_setsync entry for set %d not found on provider**

  SYNC synchronization information was expected to be drawn from an existing subscriber, but wasn't found. Something replication-breakingly-bad has probably happened...

- **DEBUG1: remoteWorkerThread_%d: could not insert to sl_setsync_offline**

  Oh, dear. After setting up a subscriber, and getting pretty well everything ready, some writes to a log shipping file failed. Perhaps disk filled up...

- **DEBUG1: remoteWorkerThread_%d: %.3f seconds to build initial setsync status**

  Indicates the total time required to get the copy_set event finalized...

- **DEBUG1: remoteWorkerThread_%d: disconnected from provider DB**

  At the end of a subscribe set event, the subscriber's slon(1) will disconnect from the provider, clearing out connections...

- **DEBUG1: remoteWorkerThread_%d: copy_set %d done in %.3f seconds**

  Indicates the total time required to complete copy_set... This indicates a successful subscription!

5.5.6.5 Log Entries Associated with MERGE SET

These various exceptions cause SLONIK MERGE SET(7) to be rejected; something ought to be corrected before submitting the request again.

- **ERROR: Slony-I: merged set ids cannot be identical**

  It is illogical to try to merge a set with itself.

- **ERROR: Slony-I: set % not found**

  A missing set cannot be merged.

- **ERROR: Slony-I: set % does not originate on local node**

  The SLONIK MERGE SET(7) request must be submitted to the origin node for the sets that are to be merged.

- **ERROR: Slony-I: subscriber lists of set % and % are different**

  Sets can only be merged if they have identical subscriber lists.

- **ERROR: Slony-I: set % has subscriptions in progress - cannot merge**

  SLONIK MERGE SET(7) cannot proceed until all subscriptions have completed processing. If this message arises, that indicates that the subscriber lists *are* the same, but that one or more of the nodes has not yet completed setting up its subscription. It may be that waiting a short while will permit resubmitting the SLONIK MERGE SET(7) request.

5.5.6.6 Log Entries Associated With Normal SYNC activity

Some of these messages indicate exceptions, but the 'normal' stuff represents what you should expect to see most of the time when replication is just plain working.

- **DEBUG2: remoteWorkerThread_%d: forward confirm %d,%s received by %d**

  These events should occur frequently and routinely as nodes report confirmations of the events they receive.

- **DEBUG1: remoteWorkerThread_%d: SYNC %d processing**

  This indicates the start of processing of a **SYNC**

- **ERROR: remoteWorkerThread_%d: No pa_conninfo for data provider %d**

  Oh dear, we haven't connection information to connect to the data provider. That shouldn't be possible, normally...

- **ERROR: remoteListenThread_%d: timeout for event selection**

  This means that the listener thread (`src/slon/remote_listener.c`) timed out when trying to determine what events were outstanding for it.

  This could occur because network connections broke, in which case restarting the slon(1) might help.

  Alternatively, this might occur because the slon(1) for this node has been broken for a long time, and there are an enormous number of entries in `sl_event` on this or other nodes for the node to work through, and it is taking more than slon_conf_remote_listen_timeout seconds to run the query. In older versions of Slony-I, that configuration parameter did not exist; the timeout was fixed at 300 seconds. In newer versions, you might increase that timeout in the slon(1) config file to a larger value so that it can continue to completion. And then investigate why nobody was monitoring things such that replication broke for such a long time...

- **ERROR: remoteWorkerThread_%d: cannot connect to data provider %d on 'dsn'**

  Oh dear, we haven't got *correct* connection information to connect to the data provider.

- **DEBUG1: remoteWorkerThread_%d: connected to data provider %d on 'dsn'**

  Excellent; the slon(1) has connected to the provider.

- **WARN: remoteWorkerThread_%d: don't know what ev_seqno node %d confirmed for ev_origin %d**

  There's no confirmation information available for this node's provider; need to abort the **SYNC** and wait a bit in hopes that that information will emerge soon...

- **DEBUG1: remoteWorkerThread_%d: data provider %d only confirmed up to ev_seqno %d for ev_origin %d**

  The provider for this node is a subscriber, and apparently that subscriber is a bit behind. The slon(1) will need to wait for the provider to catch up until it has *new* data.

- **DEBUG2: remoteWorkerThread_%d: data provider %d confirmed up to ev_seqno %s for ev_origin %d - OK**

  All's well; the provider should have the data that the subscriber needs...

- **DEBUG2: remoteWorkerThread_%d: syncing set %d with %d table(s) from provider %d**

  This is declaring the plans for a **SYNC**: we have a set with some tables to process.

- **DEBUG2: remoteWorkerThread_%d: ssy_action_list value: %s length: %d**

  This portion of the query to collect log data to be applied has been known to 'bloat up'; this shows how it has gotten compressed...

- **DEBUG2: remoteWorkerThread_%d: Didn't add OR to provider**

  This indicates that there wasn't anything in a 'provider' clause in the query to collect log data to be applied, which shouldn't be. Things are quite likely to go bad at this point...

- **DEBUG2: remoteWorkerThread_%d: no sets need syncing for this event**

  This will be the case for all **SYNC** events generated on nodes that are not originating replication sets. You can expect to see these messages reasonably frequently.

- **DEBUG3: remoteWorkerThread_%d: activate helper %d**

  We're about to kick off a thread to help process **SYNC** data...

- **DEBUG4: remoteWorkerThread_%d: waiting for log data**

  The thread is waiting to get data to consume (e.g. - apply to the replica).

- **ERROR: remoteWorkerThread_%d: %s %s - qualification was %s**

  Apparently an application of replication data to the subscriber failed... This quite likely indicates some sort of serious corruption.

- **ERROR: remoteWorkerThread_%d: replication query did not affect one row (cmdTuples = %s) - query was: %s qualification was: %s**

  If `SLON_CHECK_CMDTUPLES` is set, slon(1) applies changes one tuple at a time, and verifies that each change affects exactly one tuple. Apparently that wasn't the case here, which suggests a corruption of replication. That's a rather bad thing...

- **ERROR: remoteWorkerThread_%d: SYNC aborted**

  The **SYNC** has been aborted. The slon(1) will likely retry this **SYNC** some time soon. If the **SYNC** continues to fail, there is some continuing problem, and replication will likely never catch up without intervention. It may be necessary to unsubscribe/resubscribe the affected slave set, or, if there is only one set on the slave node, it may be simpler to drop and recreate the slave node. If application connections may be shifted over to the master during this time, application downtime may not be necessary.

- **DEBUG2: remoteWorkerThread_%d: new sl_rowid_seq value: %s**

  This marks the progression of this internal Slony-I sequence.

- **DEBUG2: remoteWorkerThread_%d: SYNC %d done in %.3f seconds**

  This indicates the successful completion of a **SYNC**. Hurray!

- **DEBUG1: remoteWorkerThread_%d_d:%.3f seconds delay for first row**

  This indicates how long it took to get the first row from the LOG cursor that pulls in data from the sl_log tables.

- **ERROR: remoteWorkerThread_%d_d: large log_cmddata for actionseq %s not found**

  slon(1) could not find the data for one of the 'very large' sl_log table tuples that are pulled individually. This shouldn't happen.

- **DEBUG2: remoteWorkerThread_%d_d:%.3f seconds until close cursor**

  This indicates how long it took to complete reading data from the LOG cursor that pulls in data from the sl_log tables.

- **DEBUG2: remoteWorkerThread_%d_d: inserts=%d updates=%d deletes=%d**

  This reports how much activity was recorded in the current **SYNC** set.

- **DEBUG3: remoteWorkerThread_%d: compress_actionseq(list,subquery) Action list: %s**

  This indicates a portion of the LOG cursor query that is about to be compressed. (In some cases, this could grow to *enormous* size, blowing up the query parser...)

- **DEBUG3: remoteWorkerThread_%d: compressed actionseq subquery %s**

  This indicates what that subquery compressed into.

5.5.6.7 Log Entries - Adding Objects to Sets

These entries will be seen on an origin node's logs at the time you are configuring a replication set; some of them will be seen on subscribers at subscription time.

- **ERROR: Slony-I: setAddTable_int(): table % has no index %**

  Apparently a PK index was specified that is absent on this node...

- **ERROR: Slony-I setAddTable_int(): table % not found**

  Table wasn't found on this node; did you load the schema in properly?.

- **ERROR: Slony-I setAddTable_int(): table % is not a regular table**

  You tried to replicate something that isn't a table; you can't do that!

- **NOTICE: Slony-I setAddTable_int(): table % PK column % nullable**

  You tried to replicate a table where one of the columns in the would-be primary key is allowed to be null. All PK columns must be **NOT NULL.** This request is about to fail.

  A check for this condition was introduced in Slony-I version 1.2. If you have a 1.1 replica, it will continue to function after upgrading to 1.2, but you will experience this complaint when you try to add new subscribers.

  You can look for table/index combinations on an existing node that have NULLABLE columns in the primary key via the following query:

```
select c.relname as table_name, ic.relname as index_name, att.attname, att2.attnotnull
from _cluster.sl_table t, pg_catalog.pg_class c, pg_index i, pg_catalog.pg_class ic,  ↩
    pg_catalog.pg_attribute att, pg_catalog.pg_attribute att2
where t.tab_reloid = c.oid
    and t.tab_idxname = ic.relname
    and  ic.oid = i.indexrelid
    and att.attrelid = i.indexrelid
    and att2.attname = att.attname
    and att2.attrelid = c.oid
    and att2.attnotnull = 'f';
```

  These may be rectified via submitting, for each one, a query of the form: **alter table mytable alter column nullablecol set not null;** Running this against a subscriber where the table is empty will complete very quickly. It will take longer to apply this change to a table that already contains a great deal of data, as the alteration will scan the table to verify that there are no tuples where the column is NULL.

- **ERROR: Slony-I setAddTable_int(): table % not replicable!**

  This happens because of the NULLable PK column.

- **ERROR: Slony-I setAddTable_int(): table id % has already been assigned!**

  The table ID value needs to be assigned uniquely in SLONIK SET ADD TABLE(7); apparently you requested a value already in use.

- **ERROR: Slony-I setAddSequence(): set % not found**

  Apparently the set you requested is not available...

- **ERROR: Slony-I setAddSequence(): set % has remote origin**

  You may only add things at the origin node.

- **ERROR: Slony-I setAddSequence(): cannot add sequence to currently subscribed set %**

  Apparently the set you requested has already been subscribed. You cannot add tables/sequences to an already-subscribed set. You will need to create a new set, add the objects to that new set, and set up subscriptions to that.

- **ERROR: Slony-I setAddSequence_int(): set % not found**

  Apparently the set you requested is not available...

- **ERROR: Slony-I setAddSequence_int(): sequence % not found**

  Apparently the sequence you requested is not available on this node. How did you set up the schemas on the subscribers???

- **ERROR: Slony-I setAddSequence_int(): % is not a sequence**

  Seems pretty obvious :-).

- **ERROR: Slony-I setAddSequence_int(): sequence ID % has already been assigned**

  Each sequence ID added must be unique; apparently you have reused an ID.

5.5.6.8 Logging When Moving Objects Between Sets

- **ERROR: Slony-I setMoveTable_int(): table % not found**
  Table wasn't found on this node; you probably gave the wrong ID number...
- **ERROR: Slony-I setMoveTable_int(): set ids cannot be identical**
  Does it make sense to move a table from a set into the very same set?
- **ERROR: Slony-I setMoveTable_int(): set % not found**
  Set wasn't found on this node; you probably gave the wrong ID number...
- **ERROR: Slony-I setMoveTable_int(): set % does not originate on local node**
  Set wasn't found to have origin on this node; you probably gave the wrong EVENT NODE...
- **ERROR: Slony-I setMoveTable_int(): subscriber lists of set % and % are different**
  You can only move objects between sets that have identical subscriber lists.
- **ERROR: Slony-I setMoveSequence_int(): sequence % not found**
  Sequence wasn't found on this node; you probably gave the wrong ID number...
- **ERROR: Slony-I setMoveSequence_int(): set ids cannot be identical**
  Does it make sense to move a sequence from a set into the very same set?
- **ERROR: Slony-I setMoveSequence_int(): set % not found**
  Set wasn't found on this node; you probably gave the wrong ID number...
- **ERROR: Slony-I setMoveSequence_int(): set % does not originate on local node**
  Set wasn't found to have origin on this node; you probably gave the wrong EVENT NODE...
- **ERROR: Slony-I setMoveSequence_int(): subscriber lists of set % and % are different**
  You can only move objects between sets that have identical subscriber lists.

5.5.6.9 Issues with Dropping Objects

- **ERROR: Slony-I setDropTable(): table % not found**
  Table wasn't found on this node; are you sure you had the ID right?
- **ERROR: Slony-I setDropTable(): set % not found**
  The replication set wasn't found on this node; are you sure you had the ID right?
- **ERROR: Slony-I setDropTable(): set % has remote origin**
  The replication set doesn't originate on this node; you probably need to specify an **EVENT NODE** in the SLONIK SET DROP TABLE(7) command.
- **ERROR: Slony-I setDropSequence_int(): sequence % not found**
  Could this sequence be in another set?
- **ERROR: Slony-I setDropSequence_int(): set % not found**
  Could you have gotten the set ID wrong?
- **ERROR: Slony-I setDropSequence_int(): set % has origin at another node - submit this to that node**
  This message seems fairly self-explanatory...

5.5.6.10 Issues with MOVE SET, FAILOVER, DROP NODE

Many of these errors will occur if you submit a slonik(1) script that describes a reconfiguration incompatible with your cluster's current configuration. Those will lead to the feeling: 'Whew, I'm glad slonik(1) caught that for me!'

Some of the others lead to a slon(1) telling itself to fall over; all *should* be well when you restart it, as it will read in the revised, newly-correct configuration when it starts up.

Alas, a few indicate that 'something bad happened,' for which the resolution may not necessarily be easy. Nobody said that replication was easy, alas...

- **ERROR: Slony-I: DROP_NODE cannot initiate on the dropped node**

  You need to have an EVENT NODE other than the node that is to be dropped....

- **ERROR: Slony-I: Node % is still configured as a data provider**

  You cannot drop a node that is in use as a data provider; you need to reshape subscriptions so no nodes are dependent on it first.

- **ERROR: Slony-I: Node % is still origin of one or more sets**

  You can't drop a node if it is the origin for a set! Use SLONIK MOVE SET(7) or SLONIK FAILOVER(7) first.

- **ERROR: Slony-I: cannot failover - node % has no path to the backup node**

  You cannot failover to a node that isn't connected to all the subscribers, at least indirectly.

- **ERROR: Slony-I: cannot failover - node % is not subscribed to set %**

  You can't failover to a node that doesn't subscribe to all the relevant sets.

- **ERROR: Slony-I: cannot failover - subscription for set % is not active**

  If the subscription has been set up, but isn't yet active, that's still no good.

- **ERROR: Slony-I: cannot failover - node % is not a forwarder of set %**

  You can only failover or move a set to a node that has forwarding turned on.

- **NOTICE: failedNode: set % has no other direct receivers - move now**

  If the backup node is the only direct subscriber, then life is a bit simplified... No need to reshape any subscriptions!

- **NOTICE: failedNode set % has other direct receivers - change providers only**

  In this case, all direct subscribers are pointed to the backup node, and the backup node is pointed to receive from another node so it can get caught up.

- **NOTICE: Slony-I: Please drop schema _@CLUSTERNAME@**

  A node has been uninstalled; you may need to drop the schema...

5.5.6.11 Log Switching

These messages relate to the new-in-1.2 facility whereby Slony-I periodically switches back and forth between storing data in `sl_log_1` and `sl_log_2`.

- **Slony-I: Logswitch to sl_log_2 initiated'**

  Indicates that slon(1) is in the process of switching over to this log table.

- **Slony-I: Logswitch to sl_log_1 initiated'**

  Indicates that slon(1) is in the process of switching over to this log table.

- **Previous logswitch still in progress**

  An attempt was made to do a log switch while one was in progress...

- **ERROR: remoteWorkerThread_%d: cannot determine current log status**

  The attempt to read from sl_log_status, which determines whether we're working on `sl_log_1` or `sl_log_2` got no results; that can't be a good thing, as there certainly should be data here... Replication is likely about to halt...

- **DEBUG2: remoteWorkerThread_%d: current local log_status is %d**

  This indicates which of `sl_log_1` and `sl_log_2` are being used to store replication data.

5.5.6.12 Miscellanea

Perhaps these messages should be categorized further; that remains a task for the documentors.

- **ERROR: Slonik version: @MODULEVERSION@ != Slony-I version in PG build %**

  This is raised in `checkmoduleversion()` if there is a mismatch between the version of Slony-I as reported by slonik(1) versus what the PostgreSQL build has.

- **ERROR: Slony-I: registry key % is not an int4 value**

  Raised in `registry_get_int4()`, this complains if a requested value turns out to be NULL.

- **ERROR: registry key % is not a text value**

  Raised in `registry_get_text()`, this complains if a requested value turns out to be NULL.

- **ERROR: registry key % is not a timestamp value**

  Raised in `registry_get_timestamp()`, this complains if a requested value turns out to be NULL.

- **NOTICE: Slony-I: cleanup stale sl_nodelock entry for pid=%**

  This will occur when a slon(1) starts up after another has crashed; this is routine cleanup.

- **ERROR: Slony-I: This node is already initialized**

  This would typically happen if you submit SLONIK STORE NODE(7) against a node that has already been set up with the Slony-I schema.

- **ERROR: Slony-I: node % not found**

  An attempt to mark a node not listed locally as enabled should fail...

- **ERROR: Slony-I: node % is already active**

  An attempt to mark a node already marked as active as active should fail...

- **DEBUG4: remoteWorkerThread_%d: added active set %d to provider %d**

  Indicates that this set is being provided by this provider.

- **DEBUG2: remoteWorkerThread_%d: event %d ignored - unknown origin**

  Probably happens if events arrive before the **STORE_NODE** event that tells that the new node now exists...

- **WARN: remoteWorkerThread_%d: event %d ignored - origin inactive**

  This shouldn't occur now (2007) as we don't support the notion of deactivating a node...

- **DEBUG2: remoteWorkerThread_%d: event %d ignored - duplicate**

  This might be expected to happen if the event notification comes in concurrently from two sources...

- **DEBUG2: remoteWorkerThread_%d: unknown node %d**

  Happens if the slon(1) is unaware of this node; probably a sign of **STORE_NODE** requests not propagating...

- **insert or update on table "sl_path" violates foreign key constraint "pa_client-no_id-ref". DETAIL: Key (pa_client)=(2) is not present on table "s1_node**

  This happens if you try to do SLONIK SUBSCRIBE SET(7) when the node unaware of a would-be new node; probably a sign of **STORE_NODE** and **STORE_PATH** requests not propagating...

- **monitorThread: stack reallocation - size %d > warning threshold of 100. Stack perhaps isn't getting processed properly by monitoring thread**

  This is liable to occur if the monitoring thread isn't properly consuming updates from the stack, so the stack is growing in memory.

- **monitor_state - unable to allocate memory for actor (len %d)**

  This occurs when the slon(1) function `monitor_state()` is unable to allocate memory, which likely indicates that something has grown in memory in an unbounded way.

  Similar log entries may also occur for event types.

5.6 Performance Considerations

Slony-I is a trigger based replication engine. For each row of application data you insert, update, or delete in your database Slony-I will insert an additional row into the sl_log_1 or sl_log_2 tables. This means that Slony-I will likely have a negative impact on your databases performance. Predicting this impact is more difficult because the amount of impact is dependent on your database workload and hardware capabilities.

The following Slony-I operations are likely to impact performance:

- Changing data in a replicated table will result in rows being added to sl_log_1 or sl_log_2
- When a slon daemon generates a SYNC event on each node it will need to add to the sl_event table.
- Each remote slon daemon will need to query the sl_log_1, sl_log_2 and sl_event tables to pull the data to replicate

5.6.1 Vacuum Concerns

The sl_event and sl_confirm tables need to be regularly vacuumed because Slony-I will often add and delete rows to these tables. The autovacuum feature of PostgreSQL included in 8.3 and above is the recommended way of handling vacuums. If autovacuum does is not working well it can be configured to not vacuum the sl_event and sl_confirm tables (See the PostgreSQL documentation information on how to disable autovacuum on a per table basis). If Slony-I detects that autovacuum has been disabled for any of the Slony-I tables then it will try to vacuum the table itself as part of cleanupThread processing.

Note
Older versions of Slony-I and older versions of PostgreSQL had different vacuuming concerns. If your using an older version of Slony-I (prior to 2.0) then you should refer to the documentation for your Slony-I version to learn about the vacuuming concerns that apply to you.

5.6.2 Log Switching

Slony-I will frequently switch between sl_log_1 and sl_log_2 as the table that the Slony-I triggers log data into. Once all of the transactions in one of these tables have been replicated Slony-I will TRUNCATE the table. The usage of TRUNCATE in this manner eliminates the need to vacuum sl_log_1 and sl_log_2.

5.6.3 Long Running Transactions

Long running transactions can impact the performance of Slony-I because they prevent Log Switching from occurring. As long as your oldest transaction is open it will sl_log_1 or sl_log_2 from being truncated. This means that the other sl_log table will continue to grow in size. Long running transactions can also stop sl_event and sl_confirm from being vacuumed. The table bloat that occurs due to a long running transaction will mean that queries to these tables will take longer. This can lead to replication falling behind. If replication is behind then the data in these tables has remain until that data is replicated. The increased size of the Slony-I tables can cause replication to fall even further behind.

5.7 Security Considerations

The simplest and most common way of deploying Slony-I has been to create a Slony-I database user account on all nodes in the system and give that account database superuser privileges. This allows Slony-I to do 'anything it needs.'

In some environments, this is too much privilege to give out to an automated system, so this section describes how to minimize the privileges given out.

5.7.1 Minimum Privileges

The minimum privileges for running each component of Slony-I may be more specifically described.

slonik(1) The slonik admin connections need to connect to the database as a database superuser. As part of the installation of Slony-I, the slonik program will create C language functions in the database. This requires superuser access. Some slonik commands will enable and disable indicies which by manipulating pg_class. This also requires superuser access.

slon(1) Local Connection Each slon instance has a 'local' database connection. This is the database connection that is specified on the either the slon command line or the slon configuration file.

Slon needs to connect to this database with considerable 'write' privileges, and requires superuser access in a couple of places.

It must be able to

- Alter `pg_class` to deactivate indices when preparing to **COPY** a table
- Make alterations to any of the Slony-I created tables
- Make modifications (INSERT/UPDATE/DELETE/ALTER) to all replicated tables.
- set the session_replication_role to replica

slon(1) Remote Connections The Remote slon connection information is specified in the SLONIK STORE PATH(7) command when adding paths. The slon(1) daemon needs to connect to remote databases with sufficient permissions to:

- SELECT from sl_event
- SELECT the sl_log_1 and sl_log_2 tables
- SELECT any replicated tables that originate on the remote node. This is done as part of the initial **COPY** during the subscription process

Note that this role does not have any need to modify data; it purely involves **SELECT** access.

5.7.2 Lowering Authority Usage from Superuser

Traditionally, it has been stated that 'Slony-I needs to use superuser connections.' It turns out that this is not actually true, and and if there are particular concerns about excessive use of superuser roles, it is possible to reduce the 'security demands' of Slony-I considerably.

It is *simplest* to have the replication management user be a superuser, as, in that case, one need not think about what permissions to configure, but this is excessive.

There is only actually one place where Slony-I truly requires superuser access, and that is for installation (slonik) and on the local connetion slon uses.

5.7.3 Handling Database Authentication (Passwords)

The slon and slonik programs connect to PostgreSQL as a normal PostgreSQL client connection. How PostgreSQL authenticates the database connection is controlled through the normal libpq authentication options via the pg_hba.conf file. See the PostgreSQL manual for full details. If you choose to require password authentication for Slony-I connections then you have two options on where slon can obtain the passwords from.

- You can store the passwords as part of the conninfo string passed to the SLONIK STORE PATH(7) statement. This means that database passwords are stored inside of the database in cleartext.
- You can setup a .pgpass file on each node you are running slon on. slon will then retreive the passwords from the .pgpass file. You must make sure that each node running slon have passwords for all paths.

5.7.4 Other Good Security Practices

In order to be able to clearly identify which logical roles are being used, it seems advisable to set up users specifically for use by replication, one or more **slony** users.

As already discussed, these users may have specific permissions attached to indicate what capabilities they are intended to have.

It is also useful for these users to be present so that system monitoring and log monitoring processes are apprised of ‘who’ is doing things in the environment.

Chapter 6

Additional Utilities

6.1 Slony-I Administration Scripts

A number of tools have grown over the course of the history of Slony-I to help users manage their clusters. This section along with the ones on Section 5.1 discusses them.

6.1.1 altperl Scripts

There is a set of scripts to simplify administration of set of Slony-I instances. The scripts support having arbitrary numbers of nodes. They may be installed as part of the installation process:

./configure --with-perltools

This will produce a number of scripts with the prefix **slonik_**. They eliminate tedium by always referring to a central configuration file for the details of your site configuration. A documented sample of this file is provided in `altperl/slon_tools.conf-sample`. Most also include some command line help with the "--help" option, making them easier to learn and use.

Most generate Slonik scripts that are printed to STDOUT. At one time, the commands were passed directly to slonik(1) for execution. Unfortunately, this turned out to be a pretty large calibre 'foot gun', as minor typos on the command line led, on a couple of occasions, to pretty calamitous actions. The savvy administrator should review the script *before* piping it to slonik(1).

6.1.1.1 Support for Multiple Clusters

The UNIX environment variable `SLONYNODES` is used to determine what Perl configuration file will be used to control the shape of the nodes in a Slony-I cluster. If it is not provided, a default `slon_tools.conf` location will be referenced.

What variables are set up.

- `$CLUSTER_NAME`=orglogs; # What is the name of the replication cluster?
- `$LOGDIR`='/opt/OXRS/log/LOGDBS'; # What is the base directory for logs?
- `$APACHE_ROTATOR`="/opt/twcsds004/OXRS/apache/rotatelogs"; # If set, where to find Apache log rotator
- `foldCase` # If set to 1, object names (including schema names) will be folded to lower case. By default, your object names will be left alone. Note that PostgreSQL itself folds object names to lower case; if you create a table via the command **CREATE TABLE SOME_THING (Id INTEGER, STudlYName text);**, the result will be that all of those components are forced to lower case, thus equivalent to **create table some_thing (id integer, studlyname text);**, and the name of table and, in this case, the fields will all, in fact, be lower case.

You then define the set of nodes that are to be replicated using a set of calls to `add_node()`.

add_node (host => '10.20.30.40', dbname => 'orglogs', port => 5437, user => 'postgres', node => 4, parent => 1);

The set of parameters for `add_node()` are thus:

```
my %PARAMS =    (host=> undef,    # Host name
      dbname => 'template1',   # database name
    port => 5432,    # Port number
    user => 'postgres', # user to connect as
    node => undef,     # node number
    password => undef,   # password for user
    parent => 1,     # which node is parent to this node
    noforward => undef, # shall this node be set up to forward results?
    sslmode => undef, # SSL mode argument - determine
          # priority of SSL usage
          # = disable,allow,prefer,require
    options => undef  # extra command line options to pass to the
          # slon daemon
);
```

6.1.1.2 Set configuration - cluster.set1, cluster.set2

The UNIX environment variable `SLONYSET` is used to determine what Perl configuration file will be used to determine what objects will be contained in a particular replication set.

Unlike `SLONYNODES`, which is essential for *all* of the slonik(1)-generating scripts, this only needs to be set when running `create_set`, as that is the only script used to control what tables will be in a particular replication set.

6.1.1.3 slonik_build_env

Queries a database, generating output hopefully suitable for `slon_tools.conf` consisting of:

- a set of `add_node()` calls to configure the cluster
- The arrays `@KEYEDTABLES`, `@SERIALTABLES`, and `@SEQUENCES`

  Note that in Slony-I 2.0 and later, `@SERIALTABLES` is no longer meaningful as SLONIK SET ADD TABLE(7) no longer supports the `SERIAL` option.

6.1.1.4 slonik_print_preamble

This generates just the 'preamble' that is required by all slonik scripts. In effect, this provides a 'skeleton' slonik script that does not do anything.

6.1.1.5 slonik_create_set

This requires `SLONYSET` to be set as well as `SLONYNODES`; it is used to generate the **slonik** script to set up a replication set consisting of a set of tables and sequences that are to be replicated.

6.1.1.6 slonik_drop_node

Generates Slonik script to drop a node from a Slony-I cluster.

6.1.1.7 slonik_drop_set

Generates Slonik script to drop a replication set (*e.g.* - set of tables and sequences) from a Slony-I cluster.

This represents a pretty big potential 'foot gun' as this eliminates a replication set all at once. A typo that points it to the wrong set could be rather damaging. Compare to Section 6.1.1.25 and Section 6.1.1.6; with both of those, attempting to drop a subscription or a node that is vital to your operations will be blocked (via a foreign key constraint violation) if there exists a downstream subscriber that would be adversely affected. In contrast, there will be no warnings or errors if you drop a set; the set will simply disappear from replication.

6.1.1.8 slonik_drop_table

Generates Slonik script to drop a table from replication. Requires, as input, the ID number of the table (available from table `sl_table`) that is to be dropped.

6.1.1.9 slonik_execute_script

Generates Slonik script to push DDL changes to a replication set.

6.1.1.10 slonik_failover

Generates Slonik script to request failover from a dead node to some new origin

6.1.1.11 slonik_init_cluster

Generates Slonik script to initialize a whole Slony-I cluster, including setting up the nodes, communications paths, and the listener routing.

6.1.1.12 slonik_merge_sets

Generates Slonik script to merge two replication sets together.

6.1.1.13 slonik_move_set

Generates Slonik script to move the origin of a particular set to a different node.

6.1.1.14 replication_test

Script to test whether Slony-I is successfully replicating data.

6.1.1.15 slonik_restart_node

Generates Slonik script to request the restart of a node. This was particularly useful pre-1.0.5 when nodes could get snarled up when slon daemons died.

6.1.1.16 slonik_restart_nodes

Generates Slonik script to restart all nodes in the cluster. Not particularly useful.

6.1.1.17 slony_show_configuration

Displays an overview of how the environment (e.g. - `SLONYNODES`) is set to configure things.

6.1.1.18 slon_kill

Kills slony watchdog and all slon daemons for the specified set. It only works if those processes are running on the local host, of course!

6.1.1.19 slon_start

This starts a slon daemon for the specified cluster and node, and uses slon_watchdog to keep it running.

6.1.1.20 slon_watchdog

Used by **slon_start**.

6.1.1.21 slon_watchdog2

This is a somewhat smarter watchdog; it monitors a particular Slony-I node, and restarts the slon process if it hasn't seen updates go in in 20 minutes or more.

This is helpful if there is an unreliable network connection such that the slon sometimes stops working without becoming aware of it.

6.1.1.22 slonik_store_node

Adds a node to an existing cluster.

6.1.1.23 slonik_subscribe_set

Generates Slonik script to subscribe a particular node to a particular replication set.

6.1.1.24 slonik_uninstall_nodes

This goes through and drops the Slony-I schema from each node; use this if you want to destroy replication throughout a cluster. As its effects are necessarily rather destructive, this has the potential to be pretty unsafe.

6.1.1.25 slonik_unsubscribe_set

Generates Slonik script to unsubscribe a node from a replication set.

6.1.1.26 slonik_update_nodes

Generates Slonik script to tell all the nodes to update the Slony-I functions. This will typically be needed when you upgrade from one version of Slony-I to another.

6.1.2 mkslonconf.sh

This is a shell script designed to rummage through a Slony-I cluster and generate a set of `slon.conf` files that slon(1) accesses via the **slon -f slon.conf** option.

With all of the configuration residing in a configuration file for each slon(1), they can be invoked with minimal muss and fuss, with no risk of forgetting the **-a** option and thereby breaking a log shipping node.

Running it requires the following environment configuration:

- Firstly, the environment needs to be set up with suitable parameters for libpq to connect to one of the databases in the cluster. Thus, you need some suitable combination of the following environment variables set:
  - `PGPORT`
  - `PGDATABASE`
  - `PGHOST`
  - `PGUSER`
  - `PGSERVICE`

- SLONYCLUSTER - the name of the Slony-I cluster to be 'rummaged'.
- MKDESTINATION - a directory for configuration to reside in; the script will create MKDESTINATION/$SLONYCLUSTER/conf for the slon(1) configuration files, and MKDESTINATION/$SLONYCLUSTER/pid for slon(1) to store PID files in.
- LOGHOME - a directory for log files to reside in; a directory of the form **$LOGHOME/$SLONYCLUSTER/node[number]** will be created for each node.

For any 'new' nodes that it discovers, this script will create a new slon(1) conf file.

Warning
It is fair to say that there are several conditions to beware of; none of these should be greatly surprising...

- The DSN is pulled from the minimum value found for each node in sl_path. You may very well need to modify this.
- Various parameters are set to default values; you may wish to customize them by hand.
- If you are running slon(1) processes on multiple nodes (*e.g.* - as when running Slony-I across a WAN), this script will happily create fresh new config files for slon(1)s you wanted to have run on another host.

  Be sure to check out what nodes it set up before restarting slon(1)s.

  This would usually only cause some minor inconvenience due to, for instance, a slon(1) running at a non-preferred site, and either failing due to lack of network connectivity (in which no damage is done!) or running a bit less efficiently than it might have due to living at the wrong end of the network 'pipe.'

  On the other hand, if you are running a log shipping node at the remote site, accidentally introducing a slon(1) that *isn't* collecting logs could ruin your whole week.

The file layout set up by mkslonconf.sh was specifically set up to allow managing slon(1)s across a multiplicity of clusters using the script in the following section...

6.1.3 start_slon.sh

This rc.d-style script was introduced in Slony-I version 2.0; it provides automatable ways of:

- Starting the slon(1), via **start_slon.sh start**

Attempts to start the slon(1), checking first to verify that it is not already running, that configuration exists, and that the log file location is writable. Failure cases include:

- No slon runtime configuration file exists,
- A slon(1) is found with the PID indicated via the runtime configuration,
- The specified SLON_LOG location is not writable.
- Stopping the slon(1), via **start_slon.sh stop**

  This fails (doing nothing) if the PID (indicated via the runtime configuration file) does not exist;
- Monitoring the status of the slon(1), via **start_slon.sh status**

  This indicates whether or not the slon(1) is running, and, if so, prints out the process ID.

The following environment variables are used to control slon(1) configuration:

SLON_BIN_PATH

This indicates where the slon(1) binary program is found.

SLON_CONF

This indicates the location of the slon runtime configuration file that controls how the slon(1) behaves.

Note that this file is *required* to contain a value for log_pid_file; that is necessary to allow this script to detect whether the slon(1) is running or not.

SLON_LOG

This file is the location where slon(1) log files are to be stored, if need be. There is an option slon_conf_syslog for slon(1) to use syslog to manage logging; in that case, you may prefer to set `SLON_LOG` to `/dev/null`.

Note that these environment variables may either be set, in the script, or overridden by values passed in from the environment. The latter usage makes it easy to use this script in conjunction with the regression tests so that it is regularly tested.

6.1.4 launch_clusters.sh

This is another shell script which uses the configuration as set up by `mkslonconf.sh` and is intended to support an approach to running Slony-I involving regularly (*e.g.* via a cron process) checking to ensure that slon(1) processes are running.

It uses the following environment variables:

- `PATH` which needs to contain, preferably at the beginning, a path to the slon(1) binaries that should be run.
- `SLHOME` indicates the 'home' directory for slon(1) configuration files; they are expected to be arranged in subdirectories, one for each cluster, with filenames of the form `node1.conf`, `node2.conf`, and such

  The script uses the command **find $SLHOME/$cluster/conf -name "node[0-9]*.conf"** to find slon(1) configuration files.

  If you remove some of these files, or rename them so their names do not conform to the **find** command, they won't be found; that is an easy way to drop nodes out of this system.
- `LOGHOME` indicates the 'home' directory for log storage.

  This script does not assume the use of the Apache log rotator to manage logs; in that PostgreSQL version 8 does its own log rotation, it seems undesirable to retain a dependancy on specific log rotation 'technology.'
- `CLUSTERS` is a list of Slony-I clusters under management.

In effect, you could run this every five minutes, and it would launch any missing slon(1) processes.

6.1.5 slony1_extract_schema.sh

You may find that you wish to create a new node some time well after creating a cluster. The script `slony1_extract_schema.sh` will help you with this.

A command line might look like the following:

PGPORT=5881 PGHOST=master.int.example.info ./slony1_extract_schema.sh payroll payroll temppayroll

It performs the following:

- It dumps the origin node's schema, including the data in the Slony-I cluster schema.

  Note that the extra environment variables `PGPORT` and `PGHOST` to indicate additional information about where the database resides.
- This data is loaded into the freshly created temporary database, `temppayroll`
- The table and sequence OIDs in Slony-I tables are corrected to point to the temporary database's configuration.
- A slonik script is run to perform SLONIK UNINSTALL NODE(7) on the temporary database. This eliminates all the special Slony-I tables, schema, and removes Slony-I triggers from replicated tables.
- Finally, pg_dump is run against the temporary database, delivering a copy of the cleaned up schema to standard output.

6.1.6 slony-cluster-analysis

If you are running a lot of replicated databases, where there are numerous Slony-I clusters, it can get painful to track and document this. The following tools may be of some assistance in this.

slony-cluster-analysis.sh is a shell script intended to provide some over-time analysis of the configuration of a Slony-I cluster. You pass in the usual libpq environment variables (`PGHOST`, `PGPORT`, `PGDATABASE`, and such) to connect to a member of a Slony-I cluster, and pass the name of the cluster as an argument.

The script then does the following:

- Runs a series of queries against the Slony-I tables to get lists of nodes, paths, sets, and tables.
- This is stowed in a temporary file in `/tmp`
- A comparison is done between the present configuration and the configuration the last time the tool was run. If the configuration differs, an email of the difference (generated using diff) is sent to a configurable email address.
- If the configuration has changed, the old configuration file is renamed to indicate when the script noticed the change.
- Ultimately, the current configuration is stowed in `LOGDIR` in a filename like `cluster.last`

There is a sample 'wrapper' script, `slony-cluster-analysis-mass.sh`, which sets things up to point to a whole bunch of Slony-I clusters.

This should make it easier for a group of DBAs to keep track of two things:

- Documenting the current state of system configuration.
- Noticing when configuration changes.

6.1.7 Generating slonik scripts using `configure-replication.sh`

The `tools` script `configure-replication.sh` is intended to automate generating slonik scripts to configure replication.

This script uses a number (possibly large, if your configuration needs to be particularly complex) of environment variables to determine the shape of the configuration of a cluster. It uses default values extensively, and in many cases, relatively few environment values need to be set in order to get a viable configuration.

6.1.7.1 Global Values

There are some values that will be used universally across a cluster:

`CLUSTER` Name of Slony-I cluster

`NUMNODES` Number of nodes to set up

`PGUSER` name of PostgreSQL user controlling replication

Traditionally, people have used a database superuser for this, but that is not necessary as discussed Section 5.7.2

`PGPORT` default port number

`PGDATABASE` default database name

`TABLES` a list of fully qualified table names (*e.g.* - complete with namespace, such as **public.my_table**)

`SEQUENCES` a list of fully qualified sequence names (*e.g.* - complete with namespace, such as **public.my_sequence**)

Defaults are provided for *all* of these values, so that if you run `configure-replication.sh` without setting any environment variables, you will get a set of slonik scripts. They may not correspond, of course, to any database you actually want to use...

6.1.7.2 Node-Specific Values

For each node, there are also four environment variables; for node 1:

DB1 database to connect to

USER1 superuser to connect as

PORT1 port

HOST1 host

It is quite likely that DB*, USER*, and PORT* should be drawn from the global PGDATABASE, PGUSER, and PGPORT values above; having the discipline of that sort of uniformity is usually a good thing.

In contrast, HOST* values should be set explicitly for HOST1, HOST2, ..., as you don't get much benefit from the redundancy replication provides if all your databases are on the same server!

6.1.7.3 Resulting slonik scripts

slonik config files are generated in a temp directory under /tmp. The usage is thus:

- preamble.slonik is a 'preamble' containing connection info used by the other scripts.

  Verify the info in this one closely; you may want to keep this permanently to use with future maintenance you may want to do on the cluster.

- create_nodes.slonik

  This is the first script to run; it sets up the requested nodes as being Slony-I nodes, adding in some Slony-I-specific config tables and such.

  You can/should start slon processes any time after this step has run.

- store_paths.slonik

  This is the second script to run; it indicates how the slon(1)s should intercommunicate. It assumes that all slon(1)s can talk to all nodes, which may not be a valid assumption in a complexly-firewalled environment. If that assumption is untrue, you will need to modify the script to fix the paths.

- create_set.slonik

  This sets up the replication set consisting of the whole bunch of tables and sequences that make up your application's database schema.

  When you run this script, all that happens is that triggers are added on the origin node (node #1) that start collecting updates; replication won't start until #5...

  There are two assumptions in this script that could be invalidated by circumstances:

  – That all of the tables and sequences have been included.
    This becomes invalid if new tables get added to your schema and don't get added to the TABLES list.
  – That all tables have been defined with primary keys.
    Best practice is to always have and use true primary keys. If you have tables that require choosing a candidate primary key, you will have to modify this script by hand to accomodate that.

- subscribe_set_2.slonik

  And 3, and 4, and 5, if you set the number of nodes higher...

  This is the step that 'fires up' replication.

  The assumption that the script generator makes is that all the subscriber nodes will want to subscribe directly to the origin node. If you plan to have 'sub-clusters,' perhaps where there is something of a 'master' location at each data centre, you may need to revise that.

  The slon processes really ought to be running by the time you attempt running this step. To do otherwise would be rather foolish.

6.1.8 slon.in-profiles

In the tools area, slon.in-profiles is a script that might be used to start up slon(1) instances at the time of system startup. It is designed to interact with the FreeBSD Ports system.

6.1.9 duplicate-node.sh

In the tools area, duplicate-node.sh is a script that may be used to help create a new node that duplicates one of the ones in the cluster.

The script expects the following parameters:

- Cluster name
- New node number
- Origin node
- Node being duplicated
- New node

For each of the nodes specified, the script offers flags to specify libpq-style parameters for PGHOST, PGPORT, PGDATABASE, and PGUSER; it is expected that .pgpass will be used for storage of passwords, as is generally considered best practice. Those values may inherit from the libpq environment variables, if not set, which is useful when using this for testing. When 'used in anger,' however, it is likely that nearly all of the 14 available parameters should be used.

The script prepares files, normally in /tmp, and will report the name of the directory that it creates that contain SQL and slonik(1) scripts to set up the new node.

- schema.sql
  This is drawn from the origin node, and contains the 'pristine' database schema that must be applied first.
- slonik.preamble
  This 'preamble' is used by the subsequent set of slonik scripts.
- step1-storenode.slonik
  A slonik(1) script to set up the new node.
- step2-storepath.slonik
  A slonik(1) script to set up path communications between the provider node and the new node.
- step3-subscribe-sets.slonik
  A slonik(1) script to request subscriptions for all replications sets.

For testing purposes, this is sufficient to get a new node working. The configuration may not necessarily reflect what is desired as a final state:

- Additional communications paths may be desirable in order to have redundancy.
- It is assumed, in the generated scripts, that the new node should support forwarding; that may not be true.
- It may be desirable later, after the subscription process is complete, to revise subscriptions.

6.1.10 slonikconfdump.sh

The tool `tools/slonikconfdump.sh` was created to help dump out a slonik(1) script to duplicate the configuration of a functioning Slony-I cluster. It should be particularly useful when upgrading Slony-I to version 2.0; see Section 5.4.4 for more details.

It dumps out:

- Cluster name
- Node connection information

  Note that it uses the first value it finds (*e.g.* - for the lowest numbered client node).
- Nodes
- Sets
- Tables
- Sequences
- Subscriptions

  Note that the subscriptions are ordered by set, then by provider, then by receiver. This ordering does not necessarily indicate the order in which subscriptions need to be applied.

It may be run as follows:

```
chris@dba2:Slony-I/CMD/slony1-2.0/tools> SLONYCLUSTER=slony_regress1 PGDATABASE= ↩
    slonyregress1 bash slonikconfdump.sh
# building slonik config files for cluster slony_regress1
# generated by: slonikconfdump.sh
# Generated on:  Tue Jun 9 17:34:12 EDT 2009
cluster name=slony_regress1;
include <admin-conninfos.slonik>;  # Draw in ADMIN CONNINFO lines
node 1 admin conninfo='dbname=slonyregress1 host=localhost user=chris port=7083';
node 2 admin conninfo='dbname=slonyregress2 host=localhost user=chris port=7083';
init cluster (id=1, comment='Regress test node');
store node (id=2, comment='node 2');
store path (server=1, client=2, conninfo='dbname=slonyregress1 host=localhost user=chris  ↩
    port=7083', connretry=10);
store path (server=2, client=1, conninfo='dbname=slonyregress2 host=localhost user=chris  ↩
    port=7083', connretry=10);
create set (id=1, origin=1, comment='All test1 tables');
set add table (id=1, set id=1, origin=1, fully qualified name='"public"."table1"', comment ↩
    ='accounts table, key='table1_pkey');
set add table (id=2, set id=1, origin=1, fully qualified name='"public"."table2"', comment ↩
    ='public.table2, key='table2_id_key');
set add table (id=4, set id=1, origin=1, fully qualified name='"public"."table4"', comment ↩
    ='a table of many types, key='table4_pkey');
set add table (id=5, set id=1, origin=1, fully qualified name='"public"."table5"', comment ↩
    ='a table with composite PK strewn across the table, key='table5_pkey');
subscribe set (id=1, provider=1, receiver=2, forward=YES);
chris@dba2:Slony-I/CMD/slony1-2.0/tools>
```

The output should be reviewed before it is applied elsewhere. Particular attention should be paid to the **ADMIN CONNINFO**, as it picks the first value that it sees for each node; in a complex environment, where visibility of nodes may vary from subnet to subnet, it may not pick the right value. In addition, **SUBSCRIBE SET** statements do not necessarily indicate the order in which subscriptions need to be applied.

6.1.11 Parallel to Watchdog: generate_syncs.sh

A new script for Slony-I 1.1 is generate_syncs.sh, which addresses the following kind of situation.

Supposing you have some possibly-flakey server where the slon daemon that might not run all the time, you might return from a weekend away only to discover the following situation.

On Friday night, something went 'bump' and while the database came back up, none of the slon daemons survived. Your online application then saw nearly three days worth of reasonably heavy transaction load.

When you restart slon on Monday, it hasn't done a SYNC on the master since Friday, so that the next 'SYNC set' comprises all of the updates between Friday and Monday. Yuck.

If you run generate_syncs.sh as a cron job every 20 minutes, it will force in a periodic **SYNC** on the origin, which means that between Friday and Monday, the numerous updates are split into more than 100 syncs, which can be applied incrementally, making the cleanup a lot less unpleasant.

Note that if **SYNC**s *are* running regularly, this script won't bother doing anything.

6.2 Slony-I Watchdog

6.2.1 Watchdogs: Keeping Slons Running

There are a couple of 'watchdog' scripts available that monitor things, and restart the slon processes should they happen to die for some reason, such as a network 'glitch' that causes loss of connectivity.

You might want to run them...

The 'best new way' of managing slon(1) processes is via the combination of Section 6.1.2, which creates a configuration file for each node in a cluster, and Section 6.1.4, which uses those configuration files.

This approach is preferable to elder 'watchdog' systems in that you can very precisely 'nail down,' in each config file, the exact desired configuration for each node, and not need to be concerned with what options the watchdog script may or may not give you. This is particularly important if you are using log shipping, where forgetting the **-a** option could ruin your log shipped node, and thereby your whole day.

6.3 Testing Slony-I State

6.3.1 test_slony_state

In the `tools` directory, you will find Section 5.1.1 scripts called `test_slony_state.pl` and `test_slony_state-dbi.pl`. One uses the Perl/DBI interface; the other uses the Pg interface.

Both do essentially the same thing, namely to connect to a Slony-I node (you can pick any one), and from that, determine all the nodes in the cluster. They then run a series of queries (read only, so this should be quite safe to run) which examine various Slony-I tables, looking for a variety of sorts of conditions suggestive of problems, including:

- Bloating of tables like pg_listener, sl_log_1, sl_log_2, sl_seqlog
- Listen paths
- Analysis of Event propagation
- Analysis of Event confirmation propagation

  If communications is a *little* broken, replication may happen, but confirmations may not get back, which prevents nodes from clearing out old events and old replication data.

Running this once an hour or once a day can help you detect symptoms of problems early, before they lead to performance degradation.

6.3.2 Replication Test Scripts

In the directory `tools` may be found four scripts that may be used to do monitoring of Slony-I instances:

- **test_slony_replication** is a Perl script to which you can pass connection information to get to a Slony-I node. It then queries sl_path and other information on that node in order to determine the shape of the requested replication set.

  It then injects some test queries to a test table called `slony_test` which is defined as follows, and which needs to be added to the set of tables being replicated:

```
CREATE TABLE slony_test (
    description text,
    mod_date timestamp with time zone,
    "_Slony-I_testcluster_rowID" bigint DEFAULT nextval('"_testcluster".sl_rowid_seq':: ↩
        text) NOT NULL
);
```

  The last column in that table was defined by Slony-I as one lacking a primary key...

  This script generates a line of output for each Slony-I node that is active for the requested replication set in a file called `cluster.fact.log`.

  There is an additional `finalquery` option that allows you to pass in an application-specific SQL query that can determine something about the state of your application.

- **log.pm** is a Perl module that manages logging for the Perl scripts.

- **run_rep_tests.sh** is a 'wrapper' script that runs **test_slony_replication**.

  If you have several Slony-I clusters, you might set up configuration in this file to connect to all those clusters.

- **nagios_slony_test** is a script that was constructed to query the log files so that you might run the replication tests every so often (we run them every 6 minutes), and then a system monitoring tool such as Nagios can be set up to use this script to query the state indicated in those logs.

  It seemed rather more efficient to have a cron job run the tests and have Nagios check the results rather than having Nagios run the tests directly. The tests can exercise the whole Slony-I cluster at once rather than Nagios invoking updates over and over again.

6.3.3 Other Replication Tests

The methodology of the previous section is designed with a view to minimizing the cost of submitting replication test queries; on a busy cluster, supporting hundreds of users, the cost associated with running a few queries is likely to be pretty irrelevant, and the setup cost to configure the tables and data injectors is pretty high.

Three other methods for analyzing the state of replication have stood out:

- For an application-oriented test, it has been useful to set up a view on some frequently updated table that pulls application-specific information.

  For instance, one might look either at some statistics about a most recently created application object, or an application transaction. For instance:

  create view replication_test as select now() - txn_time as age, object_name from transaction_table order by txn_time desc limit 1;

  create view replication_test as select now() - created_on as age, object_name from object_table order by id desc limit 1;

  There is a downside: This approach requires that you have regular activity going through the system that will lead to there being new transactions on a regular basis. If something breaks down with your application, you may start getting spurious warnings about replication being behind, despite the fact that replication is working fine.

- The Slony-I-defined view, `sl_status` provides information as to how up to date different nodes are. Its contents are only really interesting on origin nodes, as the events generated on other nodes are generally ignorable.

- See also the Section 5.1.3 discussion.

6.4 Log Files

slon(1) daemons generate some more-or-less verbose log files, depending on what debugging level is turned on. You might assortedly wish to:

- Use a log rotator like Apache rotatelogs to have a sequence of log files so that no one of them gets too big;
- Purge out old log files, periodically.

6.5 mkservice

6.5.1 slon-mkservice.sh

Create a slon service directory for use with svscan from daemontools. This uses multilog in a pretty basic way, which seems to be standard for daemontools / multilog setups. If you want clever logging, see logrep below. Currently this script has very limited error handling capabilities.

For non-interactive use, set the following environment variables. `BASEDIR SYSUSR PASSFILE DBUSER HOST PORT DATABASE CLUSTER SLON_BINARY` If any of the above are not set, the script asks for configuration information interactively.

- `BASEDIR` where you want the service directory structure for the slon to be created. This should *not* be the `/var/service` directory.
- `SYSUSR` the unix user under which the slon (and multilog) process should run.
- `PASSFILE` location of the `.pgpass` file to be used. (default `~sysusr/.pgpass`)
- `DBUSER` the postgres user the slon should connect as (default slony)
- `HOST` what database server to connect to (default localhost)
- `PORT` what port to connect to (default 5432)
- `DATABASE` which database to connect to (default dbuser)
- `CLUSTER` the name of your Slony1 cluster? (default database)
- `SLON_BINARY` the full path name of the slon binary (default **which slon**)

6.5.2 logrep-mkservice.sh

This uses **tail -F** to pull data from log files allowing you to use multilog filters (by setting the CRITERIA) to create special purpose log files. The goal is to provide a way to monitor log files in near realtime for 'interesting' data without either hacking up the initial log file or wasting CPU/IO by re-scanning the same log repeatedly.

For non-interactive use, set the following environment variables. `BASEDIR SYSUSR SOURCE EXTENSION CRITERIA` If any of the above are not set, the script asks for configuration information interactively.

- `BASEDIR` where you want the service directory structure for the logrep to be created. This should *not* be the `/var/service` directory.
- `SYSUSR` unix user under which the service should run.
- `SOURCE` name of the service with the log you want to follow.
- `EXTENSION` a tag to differentiate this logrep from others using the same source.
- `CRITERIA` the multilog filter you want to use.

A trivial example of this would be to provide a log file of all slon ERROR messages which could be used to trigger a nagios alarm. **EXTENSION='ERRORS' CRITERIA="'-*' '+* * ERROR*'"** (Reset the monitor by rotating the log using **svc -a $svc_dir**)

A more interesting application is a subscription progress log. **EXTENSION='COPY' CRITERIA="'-*' '+* * ERROR*' '+* * WARN*' '+* * CONFIG enableSubscription*' '+* * DEBUG2 remoteWorkerThread_* prepare to copy table*' '+* * DEBUG2 remoteWorkerThread_* all tables for set * found on subscriber*' '+* * DEBUG2 remoteWorkerThread_* copy*' '+* * DEBUG2 remoteWorkerThread_* Begin COPY of table*' '+* * DEBUG2 remoteWorkerThread_* * bytes copied for table*' '+* * DEBUG2 remoteWorkerThread_* * seconds to*' '+* * DEBUG2 remoteWorkerThread_* set last_value of sequence*' '+* * DEBUG2 remoteWorkerThread_* copy_set*'"**

If you have a subscription log then it's easy to determine if a given slon is in the process of handling copies or other subscription activity. If the log isn't empty, and doesn't end with a **"CONFIG enableSubscription: sub_set:1"** (or whatever set number you've subscribed) then the slon is currently in the middle of initial copies.

If you happen to be monitoring the mtime of your primary slony logs to determine if your slon has gone brain-dead, checking this is a good way to avoid mistakenly clobbering it in the middle of a subscribe. As a bonus, recall that since the the slons are running under svscan, you only need to kill it (via the svc interface) and let svscan start it up again laster. I've also found the COPY logs handy for following subscribe activity interactively.

6.6 Slony-I Test Suites

Slony-I has had (thus far) three test suites:

- Ducttape tests

  These were introduced as part of the original Slony-I distribution, and induced load via running pgbench.

  Unfortunately, the tests required human intervention to control invokation and shutdown of tests, so running them could not be readily automated.

- Test bed framework

  Slony-I version 1.1.5, introduced a test framework intended to better support automation of the tests. It eliminated the use of xterm, and tests were self-contained and self-controlled, so that one could run a series of tests.

  Unfortunately, the framework did not include any way of inducing distributed load, so as to test scenarios involving sophisticated concurrent activity.

- clustertest framework

  Introduced during testing of Slony-I version 2.0 during 2010, and released in early 2011, this framework is intended to be a better replacement for all of the preceding test frameworks.

6.7 Clustertest Test Framework

6.7.1 Introduction and Overview

The clustertest framework is implemented in Java, where tests are implemented in the interpreted JavaScript language. The use of Java made it much easier to implement tests involving concurrent activities, both in terms of inducing test load, and in, concurrently changing configuration of the replication cluster.

It consists of two physical portions:

- A framework, implemented in Java

  This software is available at clustertest-framework @ GitHub .

  This framework makes use of libraries from several other open source projects:

- `js.jar`
  This is for `org.mozilla.javascript`, the Mozilla JavaScript interpreter
- `junit-4.8.1.jar`
  JUnit, a unit test framework.
- `log4j-1.2.15.jar`
  Log4J is a popular Java-based framework for generating event logs.
- `postgresql-8.4-701.jdbc3.jar`
  This is the PostgreSQL JDBC driver.

To build the framework, it is necessary to have a Java compiler and the build tool, Ant, installed. To build all the `.jar` files used by the framework, one will run the command, with output similar to the following:

```
% ant jar
Buildfile: /var/lib/postgresql/PostgreSQL/clustertest-framework/build.xml

compile-common:
    [mkdir] Created dir: /var/lib/postgresql/PostgreSQL/clustertest-framework/build/ ↩
        classes
    [javac] /var/lib/postgresql/PostgreSQL/clustertest-framework/build.xml:23: warning: ' ↩
        includeantruntime' was not set, defaulting to build.sysclasspath=last; set to  ↩
        false for repeatable builds

compile-testcoordinator:
    [javac] /var/lib/postgresql/PostgreSQL/clustertest-framework/build.xml:44: warning: ' ↩
        includeantruntime' was not set, defaulting to build.sysclasspath=last; set to  ↩
        false for repeatable builds
    [javac] Compiling 25 source files to /var/lib/postgresql/PostgreSQL/clustertest- ↩
        framework/build/classes
    [javac] Note: /var/lib/postgresql/PostgreSQL/clustertest-framework/src/info/slony/ ↩
        clustertest/testcoordinator/script/ClientScript.java uses or overrides a  ↩
        deprecated API.
    [javac] Note: Recompile with -Xlint:deprecation for details.
     © Copying 1 file to /var/lib/postgresql/PostgreSQL/clustertest-framework/build/ ↩
         classes/info/slony/clustertest/testcoordinator

jar-common:
    [mkdir] Created dir: /var/lib/postgresql/PostgreSQL/clustertest-framework/build/jar
      [jar] Building MANIFEST-only jar: /var/lib/postgresql/PostgreSQL/clustertest- ↩
          framework/build/jar/clustertest-common.jar

compile-client:
    [javac] /var/lib/postgresql/PostgreSQL/clustertest-framework/build.xml:30: warning: ' ↩
        includeantruntime' was not set, defaulting to build.sysclasspath=last; set to  ↩
        false for repeatable builds
    [javac] Compiling 1 source file to /var/lib/postgresql/PostgreSQL/clustertest- ↩
        framework/build/classes
     © Copying 2 files to /var/lib/postgresql/PostgreSQL/clustertest-framework/build/ ↩
         classes/info/slony/clustertest/client

jar-client:
      [jar] Building jar: /var/lib/postgresql/PostgreSQL/clustertest-framework/build/jar/ ↩
          clustertest-client.jar

jar-coordinator:
      [jar] Building jar: /var/lib/postgresql/PostgreSQL/clustertest-framework/build/jar/ ↩
          clustertest-coordinator.jar

jar:

BUILD SUCCESSFUL
```

```
Total time: 2 seconds
```

At this time, there is no 'regression test' for the test framework; to validate that it works requires running tests that use it.

It includes classes supporting PostgreSQL- and Slony-I-specific functionality such as:

- `CreateDbScript`
  Creates a database
- `DropDbScript`
  Drops a database
- `LogShippingDaemon`
  Starts up Slony-I logshipping daemon
- `LogShippingDumpScript`
  Dumps and loads logshipping-based schema
- `PgCommand`
  Run a PostgreSQL shell command (such as **psql**, **createdb**, and such)
- `PgDumpCommand`
  Dump a PostgreSQL database
- `PsqlCommandExec`
  Run SQL
- `ShellExecScript`
  Run a shell script/command
- `SlonLauncher`
  Start up a slon(1) process
- `SlonikScript`
  Run a slonik(1) script

- Tests integrated into the Slony-I software distribution, that consist of a combination of shell scripts, JavaScript, and SQL scripts.

  See the directory `clustertest` in the Slony-I software distribution, which has two sets of tests:

  - Section 6.7.2
  - Section 6.7.3

6.7.2 DISORDER - DIStributed ORDER test

The *DISORDER* or *DIStributed ORDER* test is intended to provide concurrency tests involving a reasonably sophisticated schema to validate various aspects of Slony-I behavior under concurrent load.

It consists of:

- A schema for an inventory management application.

  Major objects include customers, inventory items, orders, order lines, and shipments.

  There are foreign key relationships between the various items, as well as triggers that maintain inventory and customer balances. Some of these relationships involve **ON DELETE CASCADE**, and so some actions may induce large numbers of cascaded updates.

- Stored procedures to induce creation of the various sorts of objects, purchases, shipments, and additions and removals of customers and products.

- Some tests are intended to be run against replicas, validating that balances add up. We believe that PostgreSQL applies changes in a transactional fashion such that they will always **COMMIT** leaving the visible state consistent; certain of the tests look for inconsistencies.

- There are JavaScript test scripts that induce all sorts of manipulations of replication clusters to validate that replication configuration changes succeed and fail as expected.

6.7.2.1 Configuring DISORDER

DISORDER test configuration may be found in the following files:

- `conf/disorder.properties.sample`

  This file contains Java style properties indicating how to connect to the various databases used by the DISORDER tests, including paths to tools such as slon(1) and slonik(1)

  The sample file is to be copied to `conf/disorder.properties`, and customized to indicate your local configuration. By using a `.sample` file, a developer may run tests within a Git tree, and not need to worry about their customizations interfering with the 'canonical' sample configuration provided.

- `conf/java.conf.sample`

  This is a shell script containing a path indicating where the clustertest Java code (*e.g.* - the `clustertest-coordinator.jar` file) may be found. This is also used, indirectly to determine where additional Java .jar files such as the JDBC driver are located.

  As with the disorder properties, above, this needs to be copied to `conf/java.conf`, and customized to indicate one's own local configuration.

- `conf/log4j.properties`

  See documentation for Log4J for more details as to how this is configured; the defaults provided likely do not need to be altered.

6.7.3 Regression Tests

These tests represent a re-coding of the tests previously implemented as shell scripts using the clustertest framework.

These tests have gradually been enhanced to provide coverage of scenarios with which Slony-I has had problems; it is to be expected that new bugs may lead to the addition of further tests.

6.7.3.1 Configuring Regression Tests

Similar to the Section 6.7.2.1 for DISORDER tests, there are three configuration parameters:

- `conf/slonyregress.properties.sample`

  This file contains Java style properties indicating how to connect to the various databases used by the regression tests, including paths to tools such as slon(1) and slonik(1)

  The sample file is to be copied to `conf/slonyregress.properties`, and customized to indicate your local configuration. By using a `.sample` file, a developer may run tests within a Git tree, and not need to worry about their customizations interfering with the 'canonical' sample configuration provided.

- `conf/java.conf.sample`

  This is a shell script containing a path indicating where the clustertest Java code (*e.g.* - the `clustertest-coordinator.jar` file) may be found. This is also used, indirectly to determine where additional Java .jar files such as the JDBC driver are located.

- `conf/log4j.properties`

  Identical to configuration for DISORDER.

6.8 Slony-I Test Bed Framework

Version 1.1.5 of Slony-I introduced a common test bed framework intended to better support running a comprehensive set of tests at least somewhat automatically.

The new test framework is mostly written in Bourne shell, and is intended to be portable to both Bash (widely used on Linux) and Korn shell (widely found on commercial UNIX systems). The code lives in the source tree under the `tests` directory.

At present, nearly all of the tests make use of only two databases that, by default, are on a single PostgreSQL postmaster on one host. This is perfectly fine for those tests that involve verifying that Slony-I functions properly on various sorts of data. Those tests do things like varying date styles, and creating tables and sequences that involve unusual names to verify that quoting is being handled properly.

It is also possible to configure environment variables so that the replicated nodes will be placed on different database backends, optionally on remote hosts, running varying versions of PostgreSQL.

Here are some of the vital files...

- `run_test.sh`

This is the central script for running tests. Typical usage is thus:

./run_test.sh

```
usage ./run_test.sh testname
```

You need to specify the subdirectory name of the test set to be run; each such set is stored in a subdirectory of `tests`.

You may need to set one or more of the following environment variables to reflect your local configuration. For instance, the writer runs 'test1' against PostgreSQL 8.0.x using the following command line:

```
PGBINDIR=/opt/OXRS/dbs/pgsql8/bin PGPORT=5532 PGUSER=cbbrowne ./run_test.sh test1
```

PGBINDIR

This determines where the test scripts look for PostgreSQL and Slony-I binaries. The default is `/usr/local/pgsql/bin`.

There are also variables `PGBINDIR1` thru `PGBINDIR13` which allows you to specify a separate path for each database instance. That will be particularly useful when testing interoperability of Slony-I across different versions of PostgreSQL on different platforms. In order to create a database of each respective version, you need to point to an initdb of the appropriate version.

PGPORT

This indicates what port the backend is on. By default, 5432 is used.

There are also variables `PORT1` thru `PORT13` which allow you to specify a separate port number for each database instance. That will be particularly useful when testing interoperability of Slony-I across different versions of PostgreSQL.

PGUSER

By default, the user `postgres` is used; this is taken as the default user ID to use for all of the databases.

There are also variables `USER1` thru `USER13` which allow specifying a separate user name for each database instance. The tests assume this to be a PostgreSQL 'superuser.'

WEAKUSER

By default, the user `postgres` is used; this is taken as the default user ID to use for the SLONIK STORE PATH(7) connections to all of the databases.

There are also variables `WEAKUSER1` thru `WEAKUSER13` which allow specifying a separate user name for each database instance. This user *does not* need to be a PostgreSQL 'superuser.' This user can start out with no permissions; it winds up granted read permissions on the tables that the test uses, plus read access throughout the Slony-I schema, plus write access to one table and sequence used to manage node locks.

HOST

By default, `localhost` is used.

There are also variables `HOST1` thru `HOST13` which allow specifying a separate host for each database instance.

DB1 thru DB13

By default, `slonyregress1` thru `slonyregress13` are used.

You may override these from the environment if you have some reason to use different names.

ENCODING

By default, `UNICODE` is used, so that tests can create UTF8 tables and test the multibyte capabilities.

MY_MKTEMP_IS_DECREPIT

If your version of Linux uses a variation of mktemp that does not generate a full path to the location of the desired temporary file/directory, then set this value.

TMPDIR

By default, the tests will generate their output in `/tmp`, `/usr/tmp`, or `/var/tmp`, unless you set your own value for this environment variable.

SLTOOLDIR

Where to look for Slony-I tools such as slony1_dump.sh.

ARCHIVE[n]

If set to 'true', for a particular node, which will normally get configured out of human sight in the generic-to-a-particular-test file `settings.ik`, then this node will be used as a data source for Section 4.5, and this causes the test tools to set up a directory for the archive_dir option.

LOGSHIP[n]

If set to 'true', for a particular node, which will normally get configured out of human sight in `settings.ik` for a particular test, then this indicates that this node is being created via Section 4.5, and a slon(1) is not required for this node.

SLONCONF[n]

If set to 'true', for a particular node, typically handled in `settings.ik` for a given test, then configuration will be set up in a per-node slon.conf runtime config file.

SLONYTESTER

Email address of the person who might be contacted about the test results. This is stored in the `SLONYTESTFILE`, and may eventually be aggregated in some sort of buildfarm-like registry.

SLONYTESTFILE

File in which to store summary results from tests. Eventually, this may be used to construct a buildfarm-like repository of aggregated test results.

random_number and random_string

If you run **make** in the `test` directory, C programs random_number and random_string will be built which will then be used when generating random data in lieu of using shell/SQL capabilities that are much slower than the C programs.

Within each test, you will find the following files:

- `README`

  This file contains a description of the test, and is displayed to the reader when the test is invoked.

- `generate_dml.sh`

  This contains script code that generates SQL to perform updates.

- `init_add_tables.ik`

  This is a slonik(1) script for adding the tables for the test to repliation.

- `init_cluster.ik`

  slonik(1) to initialize the cluster for the test.

- `init_create_set.ik`

  slonik(1) to initialize additional nodes to be used in the test.

- `init_schema.sql`

  An SQL script to create the tables and sequences required at the start of the test.

- `init_data.sql`

  An SQL script to initialize the schema with whatever state is required for the 'master' node.

- `init_subscribe_set.ik`

  A slonik(1) script to set up subscriptions.

- `settings.ik`

  A shell script that is used to control the size of the cluster, how many nodes are to be created, and where the origin is.

- `schema.diff`

  A series of SQL queries, one per line, that are to be used to validate that the data matches across all the nodes. Note that in order to avoid spurious failures, the queries must use unambiguous **ORDER BY** clauses.

If there are additional test steps, such as running SLONIK EXECUTE SCRIPT(7), additional slonik(1) and SQL scripts may be necessary.

Part I

Reference

Chapter 7

slon

slon — Slony-I daemon

Synopsis

`slon` [*option*...] [*clustername*] [*conninfo*]

Description

slon is the daemon application that 'runs' Slony-I replication. A slon instance must be run for each node in a Slony-I cluster.

Options

`-d` *`log_level`* The `log_level` specifies which levels of debugging messages slon should display when logging its activity.

The nine levels of logging are:

- Fatal
- Error
- Warn
- Config
- Info
- Debug1
- Debug2
- Debug3
- Debug4

The first five non-debugging log levels (from Fatal to Info) are *always* displayed in the logs. In early versions of Slony-I, the 'suggested' `log_level` value was 2, which would list output at all levels down to debugging level 2. In Slony-I version 2, it is recommended to set `log_level` to 0; most of the consistently interesting log information is generated at levels higher than that.

-s *SYNC check interval* The `sync_interval`, measured in milliseconds, indicates how often slon should check to see if a **SYNC** should be introduced. Default is 2000 ms. The main loop in `sync_Thread_main()` sleeps for intervals of `sync_interval` milliseconds between iterations.

Short sync check intervals keep the origin on a 'short leash', updating its subscribers more frequently. If you have replicated sequences that are frequently updated *without* there being tables that are affected, this keeps there from being times when only sequences are updated, and therefore *no* syncs take place

If the node is not an origin for any replication set, so no updates are coming in, it is somewhat wasteful for this value to be much less the `sync_interval_timeout` value.

-t *SYNC interval timeout* At the end of each `sync_interval_timeout` timeout period, a **SYNC** will be generated on the 'local' node even if there has been no replicable data updated that would have caused a **SYNC** to be generated.

If application activity ceases, whether because the application is shut down, or because human users have gone home and stopped introducing updates, the slon(1) will iterate away, waking up every `sync_interval` milliseconds, and, as no updates are being made, no **SYNC** events would be generated. Without this timeout parameter, *no* **SYNC** events would be generated, and it would appear that replication was falling behind.

The `sync_interval_timeout` value will lead to eventually generating a **SYNC**, even though there was no real replication work to be done. The lower that this parameter is set, the more frequently slon(1) will generate **SYNC** events when the application is not generating replicable activity; this will have two effects:

- The system will do more replication work.
  (Of course, since there is no application load on the database, and no data to replicate, this load will be very easy to handle.
- Replication will appear to be kept more 'up to date.'
  (Of course, since there is no replicable activity going on, being 'more up to date' is something of a mirage.)

Default is 10000 ms and maximum is 120000 ms. By default, you can expect each node to 'report in' with a **SYNC** every 10 seconds.

Note that **SYNC** events are also generated on subscriber nodes. Since they are not actually generating any data to replicate to other nodes, these **SYNC** events are of not terribly much value.

-g *group size* This controls the maximum **SYNC** group size, `sync_group_maxsize`; defaults to 6. Thus, if a particular node is behind by 200 **SYNC**s, it will try to group them together into groups of a maximum size of `sync_group_maxsize`. This can be expected to reduce transaction overhead due to having fewer transactions to **COMMIT**.

The default of 6 is probably suitable for small systems that can devote only very limited bits of memory to slon. If you have plenty of memory, it would be reasonable to increase this, as it will increase the amount of work done in each transaction, and will allow a subscriber that is behind by a lot to catch up more quickly.

Slon processes usually stay pretty small; even with large value for this option, slon would be expected to only grow to a few MB in size.

The big advantage in increasing this parameter comes from cutting down on the number of transaction **COMMIT**s; moving from 1 to 2 will provide considerable benefit, but the benefits will progressively fall off once the transactions being processed get to be reasonably large. There isn't likely to be a material difference in performance between 80 and 90; at that point, whether 'bigger is better' will depend on whether the bigger set of **SYNC**s makes the `LOG` cursor behave badly due to consuming more memory and requiring more time to sortt.

In Slony-I version 1.0, slon will always attempt to group **SYNC**s together to this maximum, which *won't* be ideal if replication has been somewhat destabilized by there being very large updates (*e.g.* - a single transaction that updates hundreds of thousands of rows) or by **SYNC**s being disrupted on an origin node with the result that there are a few **SYNC**s that are very large. You might run into the problem that grouping together some very large **SYNC**s knocks over a slon process. When it picks up again, it will try to process the same large grouped set of **SYNC**s, and run into the same problem over and over until an administrator interrupts this and changes the `-g` value to break this 'deadlock.'

In Slony-I version 1.1 and later versions, the slon instead adaptively 'ramps up' from doing 1 **SYNC** at a time towards the maximum group size. As a result, if there are a couple of **SYNC**s that cause problems, the slon will (with any relevant watchdog assistance) always be able to get to the point where it processes the troublesome **SYNC**s one by one, hopefully making operator assistance unnecessary.

-o *desired sync time* A 'maximum' time planned for grouped **SYNC**s.

If replication is running behind, slon will gradually increase the numbers of **SYNC**s grouped together, targetting that (based on the time taken for the *last* group of **SYNC**s) they shouldn't take more than the specified `desired_sync_time` value.

The default value for `desired_sync_time` is 60000ms, equal to one minute.

That way, you can expect (or at least hope!) that you'll get a **COMMIT** roughly once per minute.

It isn't *totally* predictable, as it is entirely possible for someone to request a *very large update,* all as one transaction, that can 'blow up' the length of the resulting **SYNC** to be nearly arbitrarily long. In such a case, the heuristic will back off for the *next* group.

The overall effect is to improve Slony-I's ability to cope with variations in traffic. By starting with 1 **SYNC**, and gradually moving to more, even if there turn out to be variations large enough to cause PostgreSQL backends to crash, Slony-I will back off down to start with one sync at a time, if need be, so that if it is at all possible for replication to progress, it will.

-c *cleanup cycles* The value `vac_frequency` indicates how often to **VACUUM** in cleanup cycles.

Set this to zero to disable slon-initiated vacuuming. If you are using something like pg_autovacuum to initiate vacuums, you may not need for slon to initiate vacuums itself. If you are not, there are some tables Slony-I uses that collect a *lot* of dead tuples that should be vacuumed frequently, notably `pg_listener`.

In Slony-I version 1.1, this changes a little; the cleanup thread tracks, from iteration to iteration, the earliest transaction ID still active in the system. If this doesn't change, from one iteration to the next, then an old transaction is still active, and therefore a **VACUUM** will do no good. The cleanup thread instead merely does an **ANALYZE** on these tables to update the statistics in `pg_statistics`.

-p *PID filename* `pid_file` contains the filename in which the PID (process ID) of the slon is stored.

This may make it easier to construct scripts to monitor multiple slon processes running on a single host.

-f *config file* File from which to read slon configuration.

This configuration is discussed further in Slon Run-time Configuration. If there are to be a complex set of configuration parameters, or if there are parameters you do not wish to be visible in the process environment variables (such as passwords), it may be convenient to draw many or all parameters from a configuration file. You might either put common parameters for all slon processes in a commonly-used configuration file, allowing the command line to specify little other than the connection info. Alternatively, you might create a configuration file for each node.

-a *archive directory* `archive_dir` indicates a directory in which to place a sequence of **SYNC** archive files for use in log shipping mode.

-x *command to run on log archive* `command_on_logarchive` indicates a command to be run each time a SYNC file is successfully generated.

See more details on slon_conf_command_on_log_archive.

-q *quit based on SYNC provider* `quit_sync_provider` indicates which provider's worker thread should be watched in order to terminate after a certain event. This must be used in conjunction with the `-r` option below...

This allows you to have a slon stop replicating after a certain point.

-r *quit at event number* `quit_sync_finalsync` indicates the event number after which the remote worker thread for the provider above should terminate. This must be used in conjunction with the `-q` option above...

-l *lag interval* `lag_interval` indicates an interval value such as **3 minutes** or **4 hours** or **2 days** that indicates that this node is to lag its providers by the specified interval of time. This causes events to be ignored until they reach the age corresponding to the interval.

Warning
There is a concommittant downside to this lag; events that require all nodes to synchronize, as typically happens with SLONIK FAILOVER(7) and SLONIK MOVE SET(7), will have to wait for this lagging node.
That might not be ideal behaviour at failover time, or at the time when you want to run SLONIK EXECUTE SCRIPT(7).

Exit Status

slon returns 0 to the shell if it finished normally. It returns via `exit (-1)` (which will likely provide a return value of either 127 or 255, depending on your system) if it encounters any fatal error.

7.1 Run-time Configuration

There are several configuration parameters that affect the behavior of the replication system. In this section, we describe how to set the slon daemon's configuration parameters; the following subsections discuss each parameter in detail.

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.

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.

Some options may be set through the Command-line, these options override any conflicting settings in the configuration file.

7.2 Logging

`syslog` (integer) Sets up logging to syslog. If this parameter is 1, messages go both to syslog and the standard output. A value of 2 sends output only to syslog (some messages will still go to the standard output/error). The default is 0, which means syslog is off.

`syslog_facility` (string) Sets the syslog 'facility' to be used when syslog enabled. Valid values are LOCAL0, LOCAL1, LOCAL2, LOCAL3, LOCAL4, LOCAL5, LOCAL6, LOCAL7. The default is LOCAL0.

`syslog_ident` (string) Sets the program name used to identify slon messages in syslog. The default is slon.

`log_level` (integer) Debug log level (higher value ==> more output). Range: [0,4], default 0

There are nine log message types; using this option, some or all of the 'debugging' levels may be left out of the slon logs. In Slony-I version 2, a lot of log message levels have been revised in an attempt to ensure the 'interesting stuff' comes in at CONFIG/INFO levels, so that you could run at level 0, omitting all of the 'DEBUG' messages, and still have meaningful contents in the logs.

`log_pid` (boolean) Determins, if you would like the pid of the (parent) slon process to appear in each log line entry.

`log_timestamp` (boolean) Determines if you would like the timestamp of the event being logged to appear in each log line entry.

Note that if `syslog` usage is configured, then this is ignored; it is assumed that syslog will be supplying timestamps, and timestamps are therefore suppressed.

`log_timestamp_format` (string) An interval in seconds at which the remote worker thread will output the query, used to select log rows from the data provider, together with it's EXPLAIN query plan. The default value of 0 turns this feature off. The allowed range is 0 (off) to 86400 (once per day).

`explain_interval` (integer) A `strftime()`-conformant format string for use if `log_timestamp` is enabled. The default is '%Y-%m-%d %H:%M:%S %Z'

`pid_file` (string) Location and filename you would like for a file containing the Process ID of the slon process. The default is not defined in which case no file is written.

`monitor_interval` (integer) Indicates the number of milliseconds the monitoring thread waits to queue up status entries before dumping such updates into the components table.

`monitor_threads` (bool) Indicates whether or not the monitoring thread is to be run. The default is to do so.

7.3 Connection settings

cluster_name (string) Set the cluster name that this instance of slon is running against. The default is to read it off the command line.

conn_info (string) Set slon's connection info; default is to read it off the command line.

sql_on_connection (string) Execute this SQL on each node at slon connect time. Useful to set logging levels, or to tune the planner/memory settings. You can specify multiple statements by separating them with a ;

tcp_keepalive (bool) Enables sending of TCP KEEP alive requests between slon and the PostgreSQL backends. Defaults to true.

tcp_keepalive_idle (integer) The number of seconds of idle activity after which a TCP KEEPALIVE will be sent across the network. The tcp_keepalive parameter must be enabled for this to take effect. The default value is 0 which means use the operating systems default. Setting this parameter has no effect on Win32 systems.

tcp_keepalive_count (integer) The number of keep alive requets to the server that need to be lost before the connection is declared dead. tcp_keep_alive must be turned on for this parameter to take effect. The default value is 0 which means use the operating systems default. Setting this parameter has no effect on Win32 systems.

tcp_keepalive_interval (integer) The number of seconds between TCP keep alive requests. tcp_keepalive must be enabled for this parameter to take effect. The default value is 0 which means use the operating systems default. Setting this parameter has no effect on Win32 systems.

7.4 Archive Logging Options

archive_dir (text) This indicates in what directory sync archive files should be stored.

command_on_logarchive (text) This indicates a Unix command to be submitted each time an archive log is successfully generated.

The command will be passed one parameter, namely the full pathname of the archive file. Thus, supposing we have the settings...

command_on_logarchive = /usr/local/bin/logstuff

archive_dir = /var/log/slony1/archivelogs/payroll

A typical log file might be named something like `/var/log/slony1/archivelogs/payroll/slony1_log_1_00000000000000000036.sql`

The command run after that SYNC was generated would be:

/usr/local/bin/logstuff /var/log/slony1/archivelogs/payroll/slony1_log_1_00000000000000sql

Warning
Note that this is run via `system(const char *COMMAND)`; if the program that is run takes five minutes to run, that will defer the next **SYNC** by five minutes. You probably don't want the archiving command to do much 'in-line' work.

7.5 Event Tuning

sync_interval (integer) Check for updates at least this often in milliseconds. Range: [10-60000], default 100

This parameter is primarily of concern on nodes that originate replication sets. On a non-origin node, there will never be update activity that would induce a SYNC; instead, the timeout value described below will induce a SYNC every so often *despite absence of changes to replicate.*

sync_interval_timeout (integer) Maximum amount of time in milliseconds before issuing a **SYNC** event, This prevents a possible race condition in which the action sequence is bumped by the trigger while inserting the log row, which makes this bump is immediately visible to the sync thread, but the resulting log rows are not visible yet. If the **SYNC** is picked up by the subscriber, processed and finished before the transaction commits, this transaction's changes will not be replicated until the next **SYNC**. But if all application activity suddenly stops, there will be no more sequence bumps, so the high frequent `-s` check won't detect that. Thus, the need for `sync_interval_timeout`. Range: [0-120000], default 1000

This parameter is likely to be primarily of concern on nodes that originate replication sets, though it does affect how often events are generated on other nodes.

On a non-origin node, there never is activity to cause a SYNC to get generated; as a result, there will be a SYNC generated every `sync_interval_timeout` milliseconds. There are no subscribers looking for those SYNCs, so these events do not lead to any replication activity. They will, however, clutter sl_event up a little, so it would be undesirable for this timeout value to be set too terribly low. 120000ms represents 2 minutes, which is not a terrible value.

The two values function together in varying ways:

On an origin node, `sync_interval` is the *minimum* time period that will be covered by a SYNC, and during periods of heavy application activity, it may be that a SYNC is being generated every `sync_interval` milliseconds.

On that same origin node, there may be quiet intervals, when no replicatable changes are being submitted. A SYNC will be induced, anyways, every `sync_interval_timeout` milliseconds.

On a subscriber node that does not originate any sets, only the 'timeout-induced' SYNCs will occur.

sync_group_maxsize (integer) Maximum number of **SYNC** events that a subscriber node will group together when/if a subscriber falls behind. **SYNC**s are batched only if there are that many available and if they are contiguous. Every other event type in between leads to a smaller batch. And if there is only one **SYNC** available, even though you used `-g600`, the slon(1) will apply just the one that is available. As soon as a subscriber catches up, it will tend to apply each **SYNC** by itself, as a singleton, unless processing should fall behind for some reason. Range: [0,10000], default: 20

vac_frequency (integer) Sets how many cleanup cycles to run before a vacuum is done. 0 disables the builtin vacuum, intended to be used with the pg_autovacuum daemon. Range: [0,100], default: 3

cleanup_interval (interval) Controls how quickly old events are trimmed out. That subsequently controls when the data in the log tables, `sl_log_1` and `sl_log_2`, get trimmed out. Default: '10 minutes'.

cleanup_deletelogs (boolean) Controls whether or not we use DELETE to trim old data from the log tables, `sl_log_1` and `sl_log_2`. Default: false

desired_sync_time (integer) Maximum time planned for grouped **SYNC**s. If replication is behind, slon will try to increase numbers of syncs done targetting that they should take this quantity of time to process. This is in Range [10000,600000] ms, default 60000.

If the value is set to 0, this logic will be ignored.

quit_sync_provider (integer) This must be used in conjunction with quit_sync_finalsync, and indicates which provider node's worker thread should be watched to see if the slon should terminate due to reaching some desired 'final' event number.

If the value is set to 0, this logic will be ignored.

quit_sync_finalsync (integer) Final event number to process. This must be used in conjunction with quit_sync_finalsync, and allows the slon to terminate itself once it reaches a certain event for the specified provider.

If the value is set to 0, this logic will be ignored.

lag_interval (string/interval) Indicates an interval by which this node should lag its providers. If set, this is used in the event processing loop to modify what events are to be considered for queueing; those events newer than **now() - lag_interval::interval** are left out, to be processed later.

If the value is left empty, this logic will be ignored.

sync_max_rowsize (integer) Size above which an sl_log_? row's log_cmddata is considered large. Up to 500 rows of this size are allowed in memory at once. Rows larger than that count into the sync_max_largemem space allocated and free()'ed on demand.

The default value is 8192, meaning that your expected memory consumption (for the LOG cursor) should not exceed 8MB.

sync_max_largemem (integer) Maximum memory allocated for large rows, where log_cmddata are larger than sync_max_rowsize.

Note that the algorithm reads rows until *after* this value is exceeded. Otherwise, a tuple larger than this value would stall replication. As a result, don't assume that memory consumption will remain smaller than this value.

The default value is 5242880.

remote_listen_timeout (integer) How long, in milliseconds, should the remote listener wait before treating the event selection criteria as having timed out? Range: [30-30000], default 300ms

Chapter 8

slonik

slonik — Slony-I command processor

Synopsis

`slonik` [*options*] [*filename*]

Options

-w Suppress slonik's behaviour of automatically waiting for event confirmations before submitting events to a different node. If this option is specified, your slonik script may require explicit SLONIK WAIT FOR EVENT(7) commands in order to behave properly, as was the behaviour of slonik prior to version 2.1.

Description

slonik is the command processor application that is used to set up and modify configurations of Slony-I replication clusters.

Outline

The slonik command line utility is supposed to be used embedded into shell scripts; it reads commands from files or stdin.

It reads a set of Slonik statements, which are written in a scripting language with syntax similar to that of SQL, and performs the set of configuration changes on the slony nodes specified in the script.

Nearly all of the real configuration work is actually done by calling stored procedures after loading the Slony-I support base into a database. Slonik was created because these stored procedures have special requirements as to on which particular node in the replication system they are called. The absence of named parameters for stored procedures makes it rather hard to do this from the psql prompt, and psql lacks the ability to maintain multiple connections with open transactions to multiple databases.

The format of the Slonik 'language' is very similar to that of SQL, and the parser is based on a similar set of formatting rules for such things as numbers and strings. Note that slonik is declarative, using literal values throughout. It is anticipated that Slonik scripts will typically be *generated* by scripts, such as Bash or Perl, and these sorts of scripting languages already have perfectly good ways of managing variables, doing iteration, and such.

See also Slonik Command Language reference.

Exit Status

slonik returns 0 to the shell if it finished normally. Scripts may specify return codes.

8.1 Slonik Command Summary

Abstract

Slonik is a command line utility designed specifically to setup and modify configurations of the Slony-I replication system.

8.2 General outline

The slonik commandline utility is supposed to be used embedded into shell scripts and reads commands from files or stdin (via here documents for example). Nearly all of the *real* configuration work is done by calling stored procedures after loading the Slony-I support base into a database. You may find documentation for those procedures in the Slony-I Schema Documentation, as well as in comments associated with them in the database.

Slonik was created because:

- The stored procedures have special requirements as to on which particular node in the replication system they are called,
- The lack of named parameters for stored procedures makes it rather difficult to do this from the psql prompt, and
- psql lacks the ability to maintain multiple connections with open transactions.

8.2.1 Commands

The slonik command language is format free. Commands begin with keywords and are terminated with a semicolon. Most commands have a list of parameters, some of which have default values and that are therefore optional. The parameters of commands are enclosed in parentheses. Each option consists of one or more keywords, followed by an equal sign, followed by a value. Multiple options inside the parentheses are separated by commas. All keywords are case insensitive. The language should remind the reader of SQL.

Option values may be:

- integer values
- string literals enclosed in single quotes
- boolean values {TRUE|ON|YES} or {FALSE|OFF|NO}
- keywords for special cases

8.2.2 Comments

Comments begin at a hash sign (#) and extend to the end of the line.

8.2.3 Command groups

Commands can be combined into groups of commands with optional **on error** and **on success** conditionals. The syntax for this is:

```
try {
commands;
}
[on error { commands; }]
[on success { commands; }]
```

Those commands are grouped together into one transaction per participating node.

Note that this does not enforce grouping of the actions as a single transaction on all nodes. For instance, consider the following slonik code:

```
try {
    execute script (set id = 1, filename = '/tmp/script1.sql', event node=1);
    execute script (set id = 1, filename = '/tmp/script2.sql', event node=1);
}
```

This *would* be processed within a single BEGIN/COMMIT on node 1. However, the requests are separated into two **DDL_SCRIPT** events so that each will be run individually, in separate transactions, on other nodes in the cluster.

Chapter 9

Slonik Meta Commands

The following commands may be used to somewhat abstract the definitions of components of Slonik scripts; SLONIK INCLUDE(7) grouping configuration into central files that may be reused, and SLONIK DEFINE(7) allowing mnemonic identifiers to replace cryptic numeric object IDs.

9.1 SLONIK INCLUDE

INCLUDE — pulling in slonik code from another file

Synopsis

`include [ <pathname>]`

Description

This draws the specified slonik script inline into the present script. If the `pathname` specifies a relative path, slonik(1) will search relative to the current working directory.

Nested include files are supported. The scanner and parser report the proper file names and line numbers when they run into an error.

Example

```
    include </tmp/preamble.slonik>;
```

Version Information

This command was introduced in Slony-I 1.1

9.2 SLONIK DEFINE

DEFINE — Defining a named symbol

Synopsis

```
define [ name ][ value ]
```

Description

This defines a named symbol. Symbol names must follow the slonik rules for constructing identifiers, by starting with a letter, followed by letters, numbers, and underscores.

Symbol values may contain spaces and may recursively contain symbol references.

Symbols are referenced by using a '@' followed by the symbol name. Note that symbol referencing is suppressed inside string literals.

Example

```
define    cluster movies;
define    sakai   1;
define    chen    2;
define    fqn     fully qualified name;

cluster name = @cluster;
node @sakai admin conninfo = 'service=sakai-replication';
node @chen  admin conninfo = 'service=chen-replication';
define setMovies    id = 1;
define sakaiMovies  @setMovies, origin = @sakai;

create set ( @sakaiMovies, comment = 'movies' );

set add table( set @sakaiMovies, id = 1, @fqn = 'public.customers',
               comment = 'sakai customers' );
set add table( set @sakaiMovies, id = 2, @fqn = 'public.tapes',
               comment = 'sakai tapes' );
echo 'But @sakaiMovies will display as a string, and is not expanded';
```

Version Information

This command was introduced in Slony-I 1.1

Chapter 10

Slonik Preamble Commands

The following commands must appear as a 'preamble' at the beginning of each slonik command script. They do not cause any direct action on any of the nodes in the replication system, but affect the execution of the entire script.

10.1 SLONIK CLUSTER NAME

CLUSTER NAME — preamble - identifying Slony-I cluster

Synopsis

```
CLUSTER NAME = [clustername;]
```

Description

Must be the very first statement in every slonik script. It defines the namespace in which all Slony-I specific functions, procedures, tables and sequences are defined. The namespace name is built by prefixing the given string literal with an underscore. This namespace will be identical in all databases that participate in the same replication group.

No user objects are supposed to live in this namespace, and the namespace is not allowed to exist prior to adding a database to the replication system. Thus, if you add a new node using **pg_dump -s** on a database that is already in the cluster of replicated databases, you will need to drop the namespace via the SQL command **DROP SCHEMA _testcluster CASCADE;** .

Example

```
    CLUSTER NAME = testcluster;
```

Version Information

This command was introduced in Slony-I 1.0

10.2 SLONIK ADMIN CONNINFO

ADMIN CONNINFO — preamble - identifying PostgreSQL database

Synopsis

```
NODE ival ADMIN CONNINFO = 'DSN'; [ ival;][ 'conninfo']
```

Description

Describes how the slonik utility can reach a node's database in the cluster from where it is run (likely the DBA's workstation). The conninfo string is the string agrument given to the `PQconnectdb()` libpq function.

The slonik utility will not try to connect to a given database unless some subsequent command requires the connection.

Note
As mentioned in the original documents, Slony-I is designed as an enterprise replication system for data centers. It has been assumed throughout the entire development that the database servers and administrative workstations involved in replication and/or setup and configuration activities can use simple authentication schemes like 'trust'. Alternatively, libpq can read passwords from `.pgpass` .

Note
If you need to change the DSN information for a node, as would happen if the IP address for a host were to change, you must submit the new information using the SLONIK STORE PATH(7) command, and that configuration will be propagated. Existing slon processes may need to be restarted in order to become aware of the configuration change.

Example

```
    NODE 1 ADMIN CONNINFO = 'dbname=testdb host=server1 user=slony';
```

Version Information

This command was introduced in Slony-I 1.0

Chapter 11

Configuration and Action commmands

11.1 SLONIK ECHO

ECHO — Generic output tool

Synopsis

```
echo [ 'string']
```

Description

Prints the string literal on standard output.

Example

```
    ECHO 'Node 1 initialized successfully';
```

Version Information

This command was introduced in Slony-I 1.0

11.2 SLONIK DATE

DATE — Display current date

Synopsis

```
date[ (format)]
```

Description

Prints the current date. Accepts an optional strftime()-conformant format string.

Example

```
DATE;
DATE(format='%Y-%m-%d %H:%M:%S %Z');
```

Version Information

This command was introduced in Slony-I 2.1

11.3 SLONIK EXIT

EXIT — Terminate Slonik script with signal

Synopsis

```
exit [ [-]ival]
```

Description

Terminates script execution immediately, rolling back every open transaction on all database connections. The slonik utility will return the given value as its program termination code. Note that on Unix, exit statuses are restricted to the range 0-255.

Example

```
EXIT 0;
```

Version Information

This command was introduced in Slony-I 1.0

11.4 SLONIK INIT CLUSTER

INIT CLUSTER — Initialize Slony-I cluster

Synopsis

```
INIT CLUSTER [ID = integer] [COMMENT = 'string']
```

Description

Initialize the first node in a new Slony-I replication cluster. The initialization process consists of creating the cluster namespace, loading all the base tables, functions, procedures and initializing the node, using schemadocinitializelocalnode(p_comment integer, p_local_node_id text) and schemadocenablenode(p_no_id integer).

ID The unique, numeric ID number of the node.

COMMENT = 'comment text' A descriptive text added to the node entry in the table sl_node.

For this process to work, the SQL scripts of the Slony-I system must be installed on the DBA workstation (the computer currently executing the slonik utility), while on the system where the node database is running the shared objects of the Slony-I system must be installed in the PostgreSQL library directory. Also the procedural language PL/pgSQL is assumed to already be installed in the target database.

Example

```
INIT CLUSTER (
   ID = 1,
   COMMENT = 'Node 1'
);
```

Note
This command functions very similarly to SLONIK STORE NODE(7), the difference being that **INIT CLUSTER** does not need to draw configuration from other existing nodes.

Note
Be aware that some objects are created that contain the cluster name as part of their name. (Notably, partial indexes on `sl_log_1` and `sl_log_2`.) As a result, *really long* cluster names are a bad idea, as they can make object names 'blow up' past the typical maximum name length of 63 characters.

Locking Behaviour

This command creates a new namespace and configures tables therein; no public objects should be locked during the duration of this.

Slonik Event Confirmation Behaviour

Slonik does not wait for event confirmations before performing this command.

Version Information

This command was introduced in Slony-I 1.0

11.5 SLONIK STORE NODE

STORE NODE — Initialize Slony-I node

Synopsis

```
STORE NODE (options);
```

Description

Initialize a new node and add it to the configuration of an existing cluster.

The initialization process consists of creating the cluster namespace in the new node (the database itself must already exist), loading all the base tables, functions, procedures and initializing the node. The existing configuration of the rest of the cluster is copied from the 'event node'.

ID = ival The unique, immutable numeric ID number of the new node.

Note that the ID is *immutable* because it is used as the basis for inter-node event communications.

COMMENT = 'description' A descriptive text added to the node entry in the table sl_node

SPOOLNODE = boolean Specifies that the new node is a virtual spool node for file archiving of replication log. If true, slonik will not attempt to initialize a database with the replication schema.

Warning
Never use the SPOOLNODE value - no released version of Slony-I has ever behaved in the fashion described in the preceding fashion. Log shipping, as it finally emerged in 1.2.11, does not require initializing 'spool nodes'.

EVENT NODE = ival The ID of the node used to create the configuration event that tells all existing nodes about the new node. It must be the ID of a pre-existing node in the cluster, not the ID of the new node.

This uses schemadocinitializelocalnode(p_comment integer, p_local_node_id text) and schemadocenablenode(p_no_id integer).

Example

```
STORE NODE ( ID = 2, COMMENT = 'Node 2', EVENT NODE = 1 );
```

Locking Behaviour

This command creates a new namespace and configures tables therein; no public objects should be locked during the duration of this.

Slonik Event Confirmation Behaviour

Slonik waits for the command submitted to the previous event node to be confirmed on the specified event node before submitting this command.

Version Information

This command was introduced in Slony-I 1.0. The SPOOLNODE parameter was introduced in version 1.1, but was vestigial in that version. The described functionality for SPOOLNODE arrived in version 1.2, but SPOOLNODE was not used for this purpose. In later versions, hopefully SPOOLNODE will be unavailable.

In version 2.0, the default value for EVENT NODE was removed, so a node must be specified.

11.6 SLONIK DROP NODE

DROP NODE — Remove the node from participating in the replication

Synopsis

```
DROP NODE (options);
```

Description

Drop a node. This command removes the specified node entirely from the replication systems configuration. If the replication daemon is still running on that node (and processing events), it will attempt to uninstall the replication system and terminate itself.

ID = ival Node ID of the node to remove.

EVENT NODE = ival Node ID of the node to generate the event.

This uses schemadocdropnode(p_no_id integer).

When you invoke **DROP NODE**, one of the steps is to run **UNINSTALL NODE**.

Example

```
    DROP NODE ( ID = 2, EVENT NODE = 1 );
```

Locking Behaviour

When dropping triggers off of application tables, this will require exclusive access to each replicated table on the node being discarded.

Dangerous/Unintuitive Behaviour

If you are using connections that cache query plans (this is particularly common for Java application frameworks with connection pools), the connections may cache query plans that include the pre-**DROP NODE** state of things, and you will get error messages indicating missing OIDs.

After dropping a node, you may also need to recycle connections in your application.

You cannot submit this to an **EVENT NODE** that is the number of the node being dropped; the request must go to some node that will remain in the cluster.

Slonik Event Confirmation Behaviour

Slonik waits until nodes (other than the one being dropped) are caught up with non-SYNC events from all other nodes before submitting the DROP NODE command.

Version Information

This command was introduced in Slony-I 1.0

In version 2.0, the default value for `EVENT NODE` was removed, so a node must be specified.

11.7 SLONIK UNINSTALL NODE

UNINSTALL NODE — Decommission Slony-I node

Synopsis

```
UNINSTALL NODE (options);
```

Description

Restores all tables to the unlocked state, with all original user triggers, constraints and rules, eventually added Slony-I specific serial key columns dropped and the Slony-I schema dropped. The node becomes a standalone database. The data is left untouched.

ID = ival Node ID of the node to uninstall.

This uses schemadocuninstallnode().

The difference between **UNINSTALL NODE** and **DROP NODE** is that all **UNINSTALL NODE** does is to remove the Slony-I configuration; it doesn't drop the node's configuration from replication.

Example

```
UNINSTALL NODE ( ID = 2 );
```

Locking Behaviour

When dropping triggers off of application tables, this will require exclusive access to each replicated table on the node being discarded.

Dangerous/Unintuitive Behaviour

If you are using connections that cache query plans (this is particularly common for Java application frameworks with connection pools), the connections may cache query plans that include the pre-**UNINSTALL NODE** state of things, and you will get error messages indicating missing OIDs.

After dropping a node, you may also need to recycle connections in your application.

Slonik Event Confirmation Behaviour

Slonik does not wait for event confirmations before performing this command

Version Information

This command was introduced in Slony-I 1.0

11.8 SLONIK RESTART NODE

RESTART NODE — Restart Slony-I node

Synopsis

```
RESTART NODE options;
```

Description

Causes an eventually running replication daemon (slon process) on the specified node to shutdown and restart itself. Theoretically, this command should be obsolete. In practice, TCP timeouts can delay critical configuration changes to actually happen in the case where a former forwarding node failed and needs to be bypassed by subscribers.

ID = ival Node ID of the node to restart.

Example

```
RESTART NODE ( ID = 2 );
```

Locking Behaviour

No application-visible locking should take place.

Slonik Event Confirmation Behaviour

Slonik does not wait for event confirmations before performing this command

Version Information

This command was introduced in Slony-I 1.0; frequent use became unnecessary as of version 1.0.5. There are, however, occasional cases where it is necessary to interrupt a slon process, and this allows this to be scripted via slonik.

11.9 SLONIK STORE PATH

STORE PATH — Configure Slony-I node connection

Synopsis

```
STORE PATH (options);
```

Description

Configures how the replication daemon of one node connects to the database of another node. If the replication system is supposed to use a special backbone network segment, this is the place to user the special IP addresses or hostnames. An existing configuration can be overwritten.

The conninfo string must contain all information to connect to the database as the replication superuser. The names ‘server’ or ‘client’ have nothing to do with the particular role of a node within the cluster configuration. It should be simply viewed as ‘the server’ has the message or data that ‘the client is supposed to get.’ For a simple 2 node setup, paths into both directions must be configured.

It does not do any harm to configure path information from every node to every other node (full cross product). The connections are not established unless they are required to actually transfer events or confirmations because of *listen* entries or data because of *subscriptions*.

SERVER = ival Node ID of the database to connect to.

CLIENT = ival Node ID of the replication daemon connecting.

CONNINFO = string `PQconnectdb()` argument to establish the connection.

CONNRETRY = ival Number of seconds to wait before another attempt to connect is made in case the server is unavailable. Default is 10.

This uses schemadocstorepath(p_pa_connretry integer, p_pa_conninfo integer, p_pa_client text, p_pa_server integer).

Example

```
STORE PATH ( SERVER = 1, CLIENT = 2,
             CONNINFO = 'dbname=testdb host=server1 user=slony'
           );
```

Locking Behaviour

No application-visible locking should take place.

Slonik Event Confirmation Behaviour

Slonik does not wait for event confirmations before performing this command

Version Information

This command was introduced in Slony-I 1.0

11.10 SLONIK DROP PATH

DROP PATH — Delete Slony-I connection information

Synopsis

```
DROP PATH (options);
```

Description

Remove the connection information between 'server' and 'client'.

SERVER = ival Node ID of the database to connect to.

CLIENT = ival Node ID of the replication daemon connecting.

EVENT NODE = ival The ID of the node used to create the configuration event that tells all existing nodes about dropping the path. Defaults to the 'client', if omitted.

Example

```
DROP PATH ( SERVER = 1, CLIENT = 2 );
```

Locking Behaviour

No application-visible locking should take place.

Slonik Event Confirmation Behaviour

Slonik does not wait for event confirmations before performing this command

Version Information

This command was introduced in Slony-I 1.0

11.11 SLONIK STORE LISTEN

STORE LISTEN — Configure Slony-I node to indicate where to listen for events

Synopsis

```
STORE LISTEN (options);
```

Description

A 'listen' entry causes a node (receiver) to query an event provider for events that originate from a specific node, as well as confirmations from every existing node. It requires a 'path' to exist so that the receiver (as client) can connect to the provider (as server).

Every node in the system must listen for events from every other node in the system. As a general rule of thumb, a subscriber (see SLONIK SUBSCRIBE SET(7)) should listen for events of the set's origin on the same provider, where it receives the data from. In turn, the origin of the data set should listen for events from the origin in the opposite direction. A node can listen for events from one and the same origin on different providers at the same time. However, to process **SYNC** events from that origin, all data providers must have the same or higher sync status, so this will not result in any faster replication behaviour.

ORIGIN = ival Node ID of the event origin the receiver is listening for.

PROVIDER = ival Node ID of the node from which the receiver gets events that come from the origin. If not specified, default is the origin.

RECEIVER = ival The ID of the node receiving the events.

This uses schemadocstorelisten(p_receiver integer, p_provider integer, p_origin integer).

For more details, see Section 4.2.

Example

```
    STORE LISTEN ( ORIGIN = 1, RECEIVER = 2, PROVIDER = 3 );
```

Locking Behaviour

No application-visible locking should take place.

Slonik Event Confirmation Behaviour

Slonik waits for the command submitted to the previous event node to be confirmed on the specified event node before submitting this command.

Version Information

This command was introduced in Slony-I 1.0. As of version 1.1, you *should* no longer need to use this command, as listen paths are generated automatically.

11.12 SLONIK DROP LISTEN

DROP LISTEN — Eliminate configuration indicating how Slony-I node listens for events

Synopsis

```
DROP LISTEN (options);
```

Description

Remove a 'listen' configuration entry.

ORIGIN = ival Node ID of the event origin the receiver is listening for.

PROVIDER = ival Node ID of the node from which the receiver gets events that come from the origin. If not specified, default is the origin.

RECEIVER = ival The ID of the node receiving the events.

This uses schemadocdroplisten(p_li_receiver integer, p_li_provider integer, p_li_origin integer).

Example

```
    DROP LISTEN ( ORIGIN = 1, RECEIVER = 2, PROVIDER = 3 );
```

Locking Behaviour

No application-visible locking should take place.

Slonik Event Confirmation Behaviour

Slonik waits for the command submitted to the previous event node to be confirmed on the specified event node before submitting this command.

Version Information

This command was introduced in Slony-I 1.0. As of version 1.1, you should not need to use it anymore.

11.13 SLONIK TABLE ADD KEY

TABLE ADD KEY — Add primary key for use by Slony-I for a table with no suitable key

Version Information

This command was introduced in Slony-I 1.0

In Slony-I version 2.0, this command is removed as obsolete because triggers are no longer 'messed around with' in the system catalogue.

11.14 SLONIK TABLE DROP KEY

TABLE DROP KEY — Removes a primary key added by TABLE ADD KEY

Version Information

This command was introduced in Slony-I 1.0

In Slony-I version 2.0, this command is removed as obsolete because triggers are no longer 'messed around with' in the system catalogue.

11.15 SLONIK CREATE SET

CREATE SET — Create Slony-I replication set

Synopsis

```
CREATE SET (options);
```

Description

In the Slony-I replication system, replicated tables are organized in sets. As a general rule of thumb, a set should contain all the tables of one application, that have relationships. In a well designed application, this is equal to all the tables in one schema.

The smallest unit one node can subscribe for replication from another node is a set. A set always has an origin. In classical replication terms, that would be the 'master.' Since in Slony-I a node can be the 'master' over one set, while receiving replication data in the 'slave' role for another at the same time, this terminology may easily become misleading and should therefore be replaced with 'set origin' and 'subscriber'.

ID = ival ID of the set to be created.

ORIGIN = ival Initial origin node of the set.

COMMENT = 'string' A descriptive text added to the set entry.

If none is provided, a default value is set; **A replication set so boring no one thought to give it a name**.

This uses schemadocstoreset(p_set_comment integer, p_set_id text) .

Example

```
    CREATE SET ( ID = 1,
              ORIGIN = 1,
              COMMENT = 'Tables for ticketing system' );
```

Locking Behaviour

No application-visible locking should take place.

Slonik Event Confirmation Behaviour

Slonik waits for the command submitted to the previous event node to be confirmed on the specified event node before submitting this command. Slonik will also wait until any outstanding DROP SET commands are confirmed by all nodes before it submits the CREATE SET command.

Version Information

This command was introduced in Slony-I 1.0

Until version 1.2, it would crash if no comment was provided.

11.16 SLONIK DROP SET

DROP SET — Discard Slony-I replication set

Synopsis

```
DROP SET (options);
```

Description

Drop a set of tables from the Slony-I configuration. This automatically unsubscribes all nodes from the set and restores the original triggers and rules on all subscribers.

ID = ival ID of the set to be dropped.

ORIGIN = ival Current origin node of the set.

This uses schemadocdropset(p_set_id integer).

Example

```
DROP SET ( ID = 5,
           ORIGIN = 2 );
```

Locking Behaviour

On each node, this will require taking out exclusive locks on each replicated table in order to modify the table schema to clean up the triggers and rules.

Slonik Event Confirmation Behaviour

Slonik waits for the command submitted to the previous event node to be confirmed on the specified event node before submitting this command.

Version Information

This command was introduced in Slony-I 1.0

11.17 SLONIK MERGE SET

MERGE SET — Merge Slony-I replication sets together

Synopsis

```
MERGE SET (options);
```

Description

Merge a set of tables and sequences into another one. This function is a workaround for the problem that it is not possible to add tables/sequences to already-subscribed sets. One may create a temporary set, add the new objects to that, subscribe all nodes currently subscribed to the other set to this new one, and then merge the two together, eliminating the set ID that was being added.

This operation will refuse to be run if the two sets do not have *exactly* the same set of subscribers.

ID = ival Unique ID of the set to contain the union of the two formerly separate sets.

ADD ID = ival Unique ID of the set whose objects should be transferred into the above set.

ORIGIN = ival Current origin node for both sets.

This uses schemadocmergeset(p_add_id integer, p_set_id integer).

Example

```
# Assuming that node 1 is the origin of set 999 that has direct subscribers 2 and 3
SUBSCRIBE SET (ID = 999, PROVIDER = 1, RECEIVER = 2);
SUBSCRIBE SET (ID = 999, PROVIDER = 1, RECEIVER = 3);
MERGE SET ( ID = 1, ADD ID = 999, ORIGIN = 1 );
```

Locking Behaviour

No application-visible locking should take place.

Dangerous/Unintuitive Behaviour

Merging takes place based on the configuration on the origin node. If a merge is requested while subscriptions are still being processed, this can cause in-progress subscribers' replication to break, as they'll be looking for configuration for this set which the merge request deletes. Do not be too quick to merge sets.

Slonik Event Confirmation Behaviour

Slonik waits for the command submitted to the previous event node to be confirmed on the specified event node before submitting this command. Slonik will also wait for any in progress subscriptions involving the ADD ID to be subscribed before submitting the MERGE SET command.

Version Information

This command was introduced in Slony-I 1.0.5. In 1.2.1, a race condition was rectified where the merge request would be submitted while subscriptions were still in process on subscribers; it refuses to merge before subscriptions are complete.

11.18 SLONIK SET ADD TABLE

SET ADD TABLE — Add a table to a Slony-I replication set

Synopsis

```
SET ADD TABLE (options);
```

Description

Add an existing user table to a replication set. The set cannot currently be subscribed by any other node - that functionality is supported by the SLONIK MERGE SET(7) command.

SET ID = ival ID of the set to which the table is to be added.

ORIGIN = ival Origin node for the set. (Optional)

ID = ival Unique ID of the table. These ID's are not only used to uniquely identify the individual table within the replication system. The numeric value of this ID also determines the order in which the tables are locked in a SLONIK LOCK SET(7) command for example. So these numbers might represent any applicable table hierarchy to make sure the slonik command scripts do not deadlock at any critical moment. If this parameter is omitted then slonik will check every node that it can connect to and find the highest table id being used across all nodes.

This ID must be unique across all sets; you cannot have two tables in the same cluster with the same ID.

Note that Slony-I generates an in-memory array indicating all of the fully qualified table names; if you use large table ID numbers, the sparsely-utilized array can lead to substantial wastage of memory. Each potential table ID consumes a pointer to a char, commonly costing 4 bytes per table ID on 32 bit architectures, and 8 bytes per table ID on 64 bit architectures.

FULLY QUALIFIED NAME = 'string' The full table name including the name of the schema. This can be omitted if `TABLES` is specified instead

TABLES = 'string' A POSIX regular expression that specifies the list of tables that should be added. This regular expression is evaluated by PostgreSQL against the list of fully qualified table names on the set origin to find the tables that should be added. If `TABLES` is omitted then `FULLY QUALIFIED NAME` must be specified.

Warning
The TABLES option requires that all the tables are in 'good form' to be replicated en masse. The request will fail, not configuring any tables for replication, if it encounters any of the following problems:

- Each table must have a PRIMARY KEY defined, and a candidate primary key will not suffice.
- If a table is found that is already replicated, the request will fail.
- The TABLES option needs to automatically assign table ID values, and looks through the configuration on every node specified by SLONIK ADMIN CONNINFO(7), finding the largest ID in use, and starting after that for the table IDs that it assigns.

  It considers it a 'benign' failure to find a node that does not yet have a Slony-I schema assigned, as that may be expected to occur if tables are configured before all the nodes have been configured using SLONIK STORE NODE(7). If there is no Slony-I schema, then that node certainly hasn't contributed anything to an increase in the table IDs in use.

  On the other hand, if a node specified by SLONIK ADMIN CONNINFO(7) is not available to be queried, the request *will fail.*

COMMENT = 'string' A descriptive text added to the table entry.

ADD SEQUENCES= boolean A boolean value that indicates if any sequences attached to columns in this table should also be automatically added to the replication set. This defaults to false

This uses schemadocsetaddtable(p_tab_comment integer, p_tab_idxname integer, p_fqname text, p_tab_id name, p_set_id text).

Example

```
SET ADD TABLE (
    SET ID = 1,
    ORIGIN = 1,
    ID = 20,
    FULLY QUALIFIED NAME = 'public.tracker_ticket',
    COMMENT = 'Support ticket',
    ADD SEQUENCES=false
);

or

SET ADD TABLE (
    SET ID=1,
    TABLES='public\\.tracker*'
);
```

Error Messages

Here are some of the error messages you may encounter if adding tables incorrectly:

Slony-I: setAddTable_int: table public.my_table PK column id nullable Primary keys (or candidates thereof) are required to have all column defined as **NOT NULL**. If you have a PK candidate that has columns that are not thus restricted, Slony-I will reject the table with this message.

Slony-I: setAddTable_int: table id 14 has already been assigned! The table id, stored in s_l_table.tab_id, is required to be unique across all tables/nodes/sets. Apparently you have tried to reused a table ID.

Slony-I: setAddTable_int(): table public.my_table has no index mt_idx_14 This will normally occur with candidate primary keys; apparently the index specified is not available on this node.

Slony-I: setAddTable_int(): table public.my_table not found Worse than an index missing, the whole table is missing. Apparently whatever process you were using to get the schema into place everywhere didn't work properly.

Slony-I: setAddTable_int(): public.my_view is not a regular table You can only replicate (at least, using **SET ADD TABLE**) objects that are ordinary tables. That doesn't include views or indexes. (Indexes can come along for the ride, but you don't ask to replicate an index...)

Slony-I: setAddTable_int(): set 4 not found You need to define a replication set before assigning tables to it.

Slony-I: setAddTable(): set 4 has remote origin This will occur if set 4 is configured with, as origin, node 1, and then you submit a **SET ADD TABLE** request involving that set to some other node than node 1. This would be expected to occur if there was some confusion in the **admin conninfo** configuration in the slonik script preamble...

Slony-I: cannot add table to currently subscribed set 1 Slony-I does not support adding tables to sets that are already participating in subscriptions. Instead, you need to define a new replication set, and add any new tables to *that* set. You might then use SLONIK MERGE SET(7) to merge the new set into an existing one, if that seems appropriate.

Locking Behaviour

On the origin node, this operation requires a brief exclusive lock on the table in order to alter it to add replication triggers. On subscriber nodes, corresponding locking takes place at the time of the **SUBSCRIBE_SET** event.

Slonik Event Confirmation Behaviour

Slonik waits for the command submitted to the previous event node to be confirmed on the specified event node before submitting this command.

Version Information

This command was introduced in Slony-I 1.0

11.19 SLONIK SET ADD SEQUENCE

SET ADD SEQUENCE — Add a sequence to a Slony-I replication set

Synopsis

```
SET ADD SEQUENCE (options);
```

Description

Add an existing user sequence to a replication set. The set cannot currently be subscribed by any other node - that functionality is supported by the SLONIK MERGE SET(7) command.

SET ID = ival ID of the set to which the sequence is to be added.

ORIGIN = ival Origin node for the set. (optional)

ID = ival Unique ID of the sequence.

> **Note**
> Note that this ID needs to be unique *across sequences* throughout the cluster; the numbering of tables is separate, so you might have a table with ID 20 and a sequence with ID 20, and they would be recognized as separate.

This parameter is optional. If this parameter is omitted then slonik will check every node that it can connect to and find the highest table id being used across all nodes.

FULLY QUALIFIED NAME = 'string' The full sequence name including schema name. If SEQUENCES is specified then FULLY QUALIFIED NAME should be omitted.

SEQUENCES = 'string' A POSIX regular expression that matches to the sequences that should be added to the replication set. This regular expression is passed to postgresql for evaluation on the set origin against fully qualified sequence names. This parameter is optional. If FULLY QUALIFIED NAME is omitted then SEQUENCES must be specified.

COMMENT = 'string' A descriptive text added to the sequence entry.

This uses schemadocsetaddsequence(p_seq_comment integer, p_fqname integer, p_seq_id text, p_set_id text).

Example

```
    SET ADD SEQUENCE (
    SET ID = 1,
    ORIGIN = 1,
    ID = 20,
    FULLY QUALIFIED NAME = 'public.tracker_ticket_id_seq',
    COMMENT = 'Support ticket ID sequence'
    );

  or

  SET ADD SEQUENCE(
  SET ID=1,
  SEQUENCES='public.tracker_ticket_id_seq'
  );
```

Locking Behaviour

No application-visible locking should take place.

Slonik Event Confirmation Behaviour

Slonik waits for the command submitted to the previous event node to be confirmed on the specified event node before submitting this command.

Version Information

This command was introduced in Slony-I 1.0

11.20 SLONIK SET DROP TABLE

SET DROP TABLE — Remove a table from a Slony-I replication set

Synopsis

```
SET DROP TABLE (options);
```

Description

Drop a table from a replication set.

ORIGIN = ival Origin node for the set. A future version of slonik might figure out this information by itself.

ID = ival Unique ID of the table.

This uses schemadocsetdroptable(p_tab_id integer).

Example

```
SET DROP TABLE (
ORIGIN = 1,
ID = 20
);
```

Locking Behaviour

This operation must acquire an exclusive lock on the table being dropped from replication in order to alter it to drop the replication trigger. On subscriber nodes, this also involves adding back any rules/triggers that have been hidden.

Slonik Event Confirmation Behaviour

Slonik waits for the command submitted to the previous event node to be confirmed on the specified event node before submitting this command.

Version Information

This command was introduced in Slony-I 1.0.5

11.21 SLONIK SET DROP SEQUENCE

SET DROP SEQUENCE — Remove a sequence from a Slony-I replication set

Synopsis

```
SET DROP SEQUENCE (options);
```

Description

Drops an existing user sequence from a replication set.

ORIGIN = ival Origin node for the set. A future version of slonik might figure out this information by itself.

ID = ival Unique ID of the sequence.

This uses schemadocsetdropsequence(p_seq_id integer).

Example

```
SET DROP SEQUENCE (
ORIGIN = 1,
ID = 20
);
```

Locking Behaviour

No application-visible locking should take place.

Slonik Event Confirmation Behaviour

Slonik waits for the command submitted to the previous event node to be confirmed on the specified event node before submitting this command.

Version Information

This command was introduced in Slony-I 1.0.5

11.22 SLONIK SET MOVE TABLE

SET MOVE TABLE — Move a table from one Slony-I replication set to another

Synopsis

```
SET MOVE TABLE (options);
```

Description

Change the set a table belongs to. The current set and the new set must origin on the same node and subscribed by the same nodes.

Caution
Due to the way subscribing to new sets works make absolutely sure that the subscription of all nodes to the sets is completely processed before moving tables. Moving a table too early to a new set causes the subscriber to try and add the table already during the subscription process, which fails with a duplicate key error and breaks replication.

ORIGIN = ival Current origin of the set. A future version of slonik might figure out this information by itself.

ID = ival Unique ID of the table.

NEW SET = ival Unique ID of the set to which the table should be added.

This uses schemadocsetmovetable(p_new_set_id integer, p_tab_id integer).

Example

```
SET MOVE TABLE (
    ORIGIN = 1,
    ID = 20,
    NEW SET = 3
);
```

Locking Behaviour

No application-visible locking should take place.

Slonik Event Confirmation Behaviour

Slonik waits for the command submitted to the previous event node to be confirmed on the specified event node before submitting this command.

Version Information

This command was introduced in Slony-I 1.0.5

11.23 SLONIK SET MOVE SEQUENCE

SET MOVE SEQUENCE — Move a sequence from one Slony-I replication set to another

Synopsis

```
SET MOVE SEQUENCE (options);
```

Description

Change the set a sequence belongs to. The current set and the new set must originate on the same node and subscribed by the same nodes.

Caution
Due to the way subscribing to new sets works make absolutely sure that the subscription of all nodes to the sets is completely processed before moving sequences. Moving a sequence too early to a new set causes the subscriber to try and add the sequence already during the subscription process, which fails with a duplicate key error and breaks replication.

ORIGIN = ival Origin node for the set. A future version of slonik might figure out this information by itself.

ID = ival Unique ID of the sequence.

NEW SET = ival Unique ID of the set to which the sequence should be moved.

This uses schemadocsetmovesequence(p_new_set_id integer, p_seq_id integer).

Example

```
SET MOVE SEQUENCE (
    ORIGIN = 1,
    ID = 20,
    NEW SET = 3
);
```

Locking Behaviour

No application-visible locking should take place.

Slonik Event Confirmation Behaviour

Slonik waits for the command submitted to the previous event node to be confirmed on the specified event node before submitting this command.

Version Information

This command was introduced in Slony-I 1.0.5

11.24 SLONIK STORE TRIGGER

STORE TRIGGER — Indicate that a trigger should not be disabled by Slony-I on a subscriber node

Version Information

This command was introduced in Slony-I 1.0

In Slony-I version 2.0, this command is removed as obsolete because triggers are no longer ‘messed around with’ in the system catalogue.

11.25 SLONIK DROP TRIGGER

DROP TRIGGER — Return a trigger to default behavior, where it will not fire on subscriber nodes

Version Information

This command was introduced in Slony-I 1.0

In Slony-I version 2.0, this command is removed as obsolete because triggers are no longer ‘messed around with’ in the system catalogue.

11.26 SLONIK SUBSCRIBE SET

SUBSCRIBE SET — Start replication of Slony-I set

Synopsis

```
SUBSCRIBE SET (options);
```

Description

This performs one of two actions:

- Initiates replication for a replication set

  Causes a node (subscriber) to start replicating a set of tables either from the origin or from another provider node, which must itself already be be an active, forwarding subscriber.

  The application tables contained in the set must already exist and should ideally be empty. The current version of Slony-I will *not* attempt to copy the schema of the set. The replication daemon will start copying the current content of the set from the given provider and then try to catch up with any update activity that happened during that copy process. After successful subscription, the tables are guarded on the subscriber, using triggers, against accidental updates by the application.

  If the tables on the subscriber are *not* empty, then the **COPY SET** event (which is part of the subscription process) may wind up doing more work than should be strictly necessary:

  – It attempts to **TRUNCATE** the table, which will be efficient.
  – If that fails (a foreign key relationship might prevent TRUNCATE from working), it uses **DELETE** to delete all 'old' entries in the table
  – Those old entries clutter up the table until it is next **VACUUM**ed *after* the subscription process is complete
  – The indices for the table will contain entries for the old, deleted entries, which will slow the process of inserting new entries into the index.

  Warning
  This operation can take a (potentially distinctly) non-zero period of time. If you have a great deal of data in a particular set of tables, it may take hours or even (if 'a great deal' indicates 'tens or hundreds gigabytes of data') possibly multiple days for this event to complete.
  The **SUBSCRIBE SET** request will, nonetheless, return fairly much immediately, even though the work, being handled by the **COPY SET** event, is still in progress. If you need to set up subscriptions for a set of cascading nodes, you will need to wait for each subscriber to complete subscribing before submitting requests for subscriptions that use that node as a provider.

  ```
  Slony-I: provider 2 is not an active forwarding node for replication set 1
  ```

  In effect, such subscription requests will be ignored until the provider is ready.

- Revising subscription information for already-subscribed nodes.

  If you need to revise subscription information for a node, you *also* submit the new information using this command, and the new configuration will be propagated throughout the replication network. The normal reason to revise this information is that you want a node to subscribe to a *different* provider node, or for a node to become a 'forwarding' subscriber so it may later become the provider for a later subscriber.

ID = ival ID of the set to subscribe

PROVIDER = ival Node ID of the data provider from which this node draws data.

RECEIVER = ival Node ID of the new subscriber

FORWARD = boolean Flag whether or not the new subscriber should store the log information during replication to make it possible candidate for the provider role for future nodes. Any node that is intended to be a candidate for FAILOVER *must* have **FORWARD = yes**.

OMIT COPY = boolean Flag whether or not the subscription process should omit doing the **COPY** of the existing data in the set. In effect, use this option indicates 'Trust me, the data is already in sync!'

This is notably useful for the following sorts of cases:

- Major inter-version upgrades (*e.g.* - as from Slony-I 1.2 to 2.0) may be done quickly.
- Cloning a 'master node'. SLONIK CLONE PREPARE(7)/SLONIK CLONE FINISH(7)
-

Example

```
SUBSCRIBE SET (
   ID = 1,
   PROVIDER = 1,
   RECEIVER = 3,
   FORWARD = YES
);
  WAIT FOR EVENT(
    ORIGIN=1,
    CONFIRMED=ALL,
    WAIT ON=1
);
```

Forwarding Behaviour

The **FORWARD=boolean** flag indicates whether the subscriber will store log information in tables sl_log_1 and sl_log_2. Several implications fall from this...

By storing the data in these tables on the subscriber, there is some additional processing burden. If you are certain that you would never want to SLONIK MOVE SET(7) or SLONIK FAILOVER(7) to a particular subscriber, it is worth considering turning off forwarding on that node.

There is, however, a case where having forwarding turned off opens up a perhaps-unexpected failure condition; a rule of thumb should be that *all nodes that connect directly to the origin* should have forwarding turned on. Supposing one such 'direct subscriber' has forwarding turned off, it is possible for that node to be forcibly lost in a case of failover. The problem comes if that node gets ahead of other nodes.

Let's suppose that the origin, node 1 is at SYNC number 88901, a non-forwarding node, node 2 has processed up to SYNC 88897, and other forwarding nodes, 3, 4, and 5, have only processed data up to SYNC 88895. At that moment, the disk system on the origin node catches fire. Node 2 has the *data* up to SYNC 88897, but there is no remaining node that contains, in sl_log_1 or sl_log_2, the data for SYNCs 88896 and 88897, so there is no way to bring nodes 3-5 up to that point.

At that point, there are only two choices: To drop node 2, because there is no way to continue managing it, or to drop all nodes *but* 2, because there is no way to bring them up to SYNC 88897.

That dilemma may be avoided by making sure that all nodes directly subscribing to the origin have forwarding turned on.

Dangerous/Unintuitive Behaviour

- The fact that the request returns immediately even though the subscription may take considerable time to complete may be a bit surprising.

  Processing of the subscription involves *two* events; the **SUBSCRIBE_SET**, initiated on the set origin node, and an **ENABLE_SUBSCRIPTION**. This means that SLONIK WAIT FOR EVENT(7) must be used following a SUBSCRIBE SET to wait until the last event on the set origin completes.
- This command has *two* purposes; setting up subscriptions (which should be unsurprising) and *revising subscriptions*, which isn't so obvious to intuition.
- New subscriptions are set up by using **DELETE** or **TRUNCATE** to empty the table on a subscriber. If you created a new node by copying data from an existing node, it might 'seem intuitive' that that data should be kept; that is not the case - the former contents are discarded and the node is populated *from scratch*.
- The **OMIT COPY** option has the potential to be a large 'foot gun' in that it allows the administrator to push replication sets out of sync.

Locking Behaviour

This operation does *not* require acquiring any locks on the provider node.

On the subscriber node, it will have the effect of locking every table in the replication set. In version 1.2 and later, exclusive locks are acquired at the beginning of the process.

Slonik Event Confirmation Behaviour

Slonik waits until the provider has confirmed all outstanding configuration events from any other node before contacting the provider to determine the set origin. Slonik then waits for the command submitted to the previous event node to be confirmed on the origin before submitting this command to the origin.

Version Information

This command was introduced in Slony-I 1.0

The **OMIT COPY** option was introduced in Slony-I 2.0.3.

In Slony-I 2.0.5 the SUBSCRIBE SET command gets submitted directly against the set origin. Prior to this change the SUBSCRIBE SET was submitted against the provider

11.27 SLONIK UNSUBSCRIBE SET

UNSUBSCRIBE SET — End replication of Slony-I set

Synopsis

```
UNSUBSCRIBE SET (options);
```

Description

Stops the subscriber from replicating the set. The tables are opened up for full access by the client application on the former subscriber. The tables are not truncated or otherwise modified. All original triggers, rules and constraints are restored.

ID = ival ID of the set to unsubscribe

RECEIVER = ival Node ID of the (former) subscriber

This uses schemadocunsubscribeset(p_sub_receiver integer, p_sub_set integer).

Example

```
UNSUBSCRIBE SET (
   ID = 1,
   RECEIVER = 3
);
```

Locking Behaviour

Exclusive locks on each replicated table will be taken out on the subscriber in order to drop replication triggers from the tables and restore other triggers/rules.

Slonik Event Confirmation Behaviour

Slonik waits for the command submitted to the previous event node to be confirmed on the specified event node before submitting this command.

Dangerous/Unintuitive Behaviour

Resubscribing an unsubscribed set requires a *complete fresh copy* of data from the provider to be transferred since the tables have been subject to possible independent modifications.

Version Information

This command was introduced in Slony-I 1.0

11.28 SLONIK LOCK SET

LOCK SET — Guard Slony-I replication set to prepare for **MOVE SET**

Synopsis

```
LOCK SET (options);
```

Description

Guards a replication set against client application updates in preparation for a SLONIK MOVE SET(7) command.

This command must be the first in a possible statement group (**try**). The reason for this is that it needs to commit the changes made to the tables (adding a special trigger function) before it can wait for every concurrent transaction to finish. At the same time it cannot hold an open transaction to the same database itself since this would result in blocking itself forever.

Note that this is a locking operation, which means that it can get stuck behind other database activity.

The operation waits for transaction IDs to advance in order that data is not missed on the new origin. Thus, if you have long-running transactions running on the source node, this operation will wait for those transactions to complete. Unfortunately, if you have another database on the same postmaster as the origin node, long running transactions on that database will also be considered even though they are essentially independent.

ID = ival ID of the set to lock

ORIGIN = ival Node ID of the current set origin

This uses schemadoclockset(p_set_id integer).

Example

```
LOCK SET (
   ID = 1,
   ORIGIN = 3
);
```

Locking Behaviour

Exclusive locks on each replicated table will be taken out on the origin node, and triggers are added to each such table that reject table updates.

Slonik Event Confirmation Behaviour

Slonik does not wait for event confirmations before performing this command.

Version Information

This command was introduced in Slony-I 1.0

11.29 SLONIK UNLOCK SET

UNLOCK SET — Unlock a Slony-I set that was locked

Synopsis

```
UNLOCK SET (options);
```

Description

Unlocks a previously locked set.

ID = ival ID of the set to unlock

ORIGIN = ival Node ID of the current set origin

This uses schemadocunlockset(p_set_id integer).

Example

```
UNLOCK SET (
   ID = 1,
   ORIGIN = 3
);
```

Locking Behaviour

Exclusive locks on each replicated table will be taken out on the origin node, as the triggers are removed from each table that reject table updates.

Slonik Event Confirmation Behaviour

Slonik does not wait for event confirmations before performing this command.

Version Information

This command was introduced in Slony-I 1.0

11.30 SLONIK MOVE SET

MOVE SET — Change origin of a Slony-I replication set

Synopsis

```
MOVE SET (options);
```

Description

Changes the origin of a set from one node to another. The new origin must be a current subscriber of the set. The set must currently be locked on the old origin.

After this command, the set cannot be unlocked on the old origin any more. The old origin will continue as a forwarding subscriber of the set and the subscription chain from the old origin to the new origin will be reversed, hop by hop. As soon as the new origin has finished processing the event (that includes any outstanding sync events that happened before, *i.e.* fully catching up), the new origin will take over and open all tables in the set for client application update activity.

This is *not* failover, as it requires a functioning old origin node (you needed to lock the set on the old origin). You would probably prefer to **MOVE SET** instead of **FAILOVER**, if at all possible, as **FAILOVER** winds up discarding the old origin node as being corrupted. Before **MOVE SET** will function a **LOCK SET** is needed.

Note that this is a locking operation, which means that it can get stuck behind other database activity.

ID = ival ID of the set to transfer

OLD ORIGIN = ival Node ID of the current set origin

NEW ORIGIN = ival Node ID of the new set origin

This uses schemadocmoveset(p_new_origin integer, p_set_id integer).

Example

```
LOCK SET (
   ID = 1,
   ORIGIN = 1
);
MOVE SET (
   ID = 1,
   OLD ORIGIN = 1,
   NEW ORIGIN = 3
);
```

Locking Behaviour

Exclusive locks on each replicated table will be taken out on both the old origin node and the new origin node, as replication triggers are changed on both nodes: on the former origin, each table has two triggers (logtrigger and lockset) dropped and a denyaccess trigger added; on the new origin, the denyaccess trigger is dropped and a logtrigger trigger added.

Slonik Event Confirmation Behaviour

Slonik waits for the command submitted to the previous event node to be confirmed on the specified event node before submitting this command.

Version Information

This command was introduced in Slony-I 1.0

11.31 SLONIK FAILOVER

FAILOVER — Fail a broken replication set over to a backup node

Synopsis

```
FAILOVER (options);
```

Description

The **FAILOVER** command causes the backup node to take over all sets that currently originate on the failed node. slonik will contact all other direct subscribers of the failed node to determine which node has the highest sync status for each set. If another node has a higher sync status than the backup node, the replication will first be redirected so that the backup node replicates against that other node, before assuming the origin role and allowing update activity.

After successful failover, all former direct subscribers of the failed node become direct subscribers of the backup node. The failed node is abandoned, and can and should be removed from the configuration with SLONIK DROP NODE(7).

ID = ival ID of the failed node

BACKUP NODE = ival Node ID of the node that will take over all sets originating on the failed node

This uses schemadocfailednode(p_backup_node integer, p_failed_node integer).

Example

```
FAILOVER (
   ID = 1,
   BACKUP NODE = 2
);
```

Locking Behaviour

Exclusive locks on each replicated table will be taken out on both the new origin node as replication triggers are changed. If the new origin was not completely up to date, and replication data must be drawn from some other node that is more up to date, the new origin will not become usable until those updates are complete.

Dangerous/Unintuitive Behaviour

This command will abandon the status of the failed node. There is no possibility to let the failed node join the cluster again without rebuilding it from scratch as a slave. If at all possible, you would likely prefer to use SLONIK MOVE SET(7) instead, as that does *not* abandon the failed node.

If there are many nodes in a cluster, and failover includes dropping out additional nodes (*e.g.* when it is necessary to treat *all* nodes at a site including an origin as well as subscribers as failed), it is necessary to carefully sequence the actions.

Slonik Event Confirmation Behaviour

Slonik will submit the FAILOVER_EVENT without waiting but wait until the most ahead node has received confirmations of the FAILOVER_EVENT from all nodes before completing.

Version Information

This command was introduced in Slony-I 1.0

In version 2.0, the default `BACKUP NODE` value of 1 was removed, so it is mandatory to provide a value for this parameter.

11.32 SLONIK EXECUTE SCRIPT

EXECUTE SCRIPT — Execute SQL/DDL script

Synopsis

```
EXECUTE SCRIPT (options);
```

Description

Executes a script containing arbitrary SQL statements on all nodes that are subscribed to a set at a common controlled point within the replication transaction stream.

The specified event origin must be the origin of the set. The script file must not contain any **START** or **COMMIT TRANSACTION** calls. (This changes somewhat in PostgreSQL 8.0 once nested transactions, aka savepoints, are supported) In addition, non-deterministic DML statements (like updating a field with `CURRENT_TIMESTAMP`) must be avoided, since the data changes done by the script are explicitly not replicated.

`SET ID = ival` The unique numeric ID number of the set affected by the script

`FILENAME = '/path/to/file'` The name of the file containing the SQL script to execute. This might be a relative path, relative to the location of the slonik instance you are running, or, preferably, an absolute path on the system where slonik is to run.

The *contents* of the file are propagated as part of the event, so the file does not need to be accessible on any of the nodes.

`EVENT NODE = ival` (Mandatory unless EXECUTE ONLY ON is given) The ID of the current origin of the set. If EXECUTE ONLY ON is given, EVENT NODE must specify the same node or be omitted.

`EXECUTE ONLY ON = ival` (Optional) The ID of the only node to actually execute the script. This option causes the script to be propagated by all nodes but executed only by one. The default is to execute the script on all nodes that are subscribed to the set.

See also the warnings in Section 3.3.

Note that this is a potentially heavily-locking operation, which means that it can get stuck behind other database activity.

Note that if you need to make reference to the cluster name, you can use the token **@CLUSTERNAME@**; if you need to make reference to the Slony-I namespace, you can use the token **@NAMESPACE@**; both will be expanded into the appropriate replacement tokens.

This uses schemadocddlscript_complete(p_only_on_node integer, p_script text, p_set_id integer).

Example

```
EXECUTE SCRIPT (
   SET ID = 1,
   FILENAME = '/tmp/changes_2008-04-01.sql',
   EVENT NODE = 1
);
```

Locking Behaviour

Up until the 2.0 branch, each replicated table received an exclusive lock, on the origin node, in order to remove the replication triggers; after the DDL script completes, those locks will be cleared. In the 2.0 branch this is no longer the case. EXECUTE SCRIPT won't obtain any locks on your application tables though the script that you executing probably will. Due to bug #137 you should avoid concurrent writes to the tables being modified by the script while the script is running.

After the DDL script has run on the origin node, it will then run on subscriber nodes, where replicated tables will be similarly altered to remove replication triggers, therefore requiring that exclusive locks be taken out on each node, in turn.

Slonik Event Confirmation Behaviour

Slonik waits for the command submitted to the previous event node to be confirmed on the specified event node before submitting this command.

Version Information

This command was introduced in Slony-I 1.0.

Before Slony-I version 1.2, the entire DDL script was submitted as one `PQexec()` request, with the implication that the *entire* script was parsed based on the state of the database before invokation of the script. This means statements later in the script cannot depend on DDL changes made by earlier statements in the same script. Thus, you cannot add a column to a table and add constraints to that column later in the same request.

In Slony-I version 1.2, the DDL script is split into statements, and each statement is submitted separately. As a result, it is fine for later statements to refer to objects or attributes created or modified in earlier statements. Furthermore, in version 1.2, the **slonik** output includes a listing of each statement as it is processed, on the set origin node. Similarly, the statements processed are listed in slon logs on the other nodes.

In Slony-I version 1.0, this would only lock the tables in the specified replication set. As of 1.1 (until 2.0), *all replicated tables* are locked (*e.g.* - triggers are removed at the start, and restored at the end). This deals with the risk that one might request DDL changes on tables in multiple replication sets. With version 2.0 no locks on application tables are obtained by Slony-I

In version 2.0, the default value for `EVENT NODE` was removed, so a node must be specified.

As of version 2.0.7, the log triggers on all replicated tables are checked to ensure their parameters match the primary key on the table. If they *do not* match, those tables that are exclusively locked as a result of the DDL request will have the triggers recreated to match the primary key. Tables that do not have an exclusive lock will *not* be corrected, but a warning message will be generated. The function `repair_log_triggers(only_locked boolean)` may be used manually to correct the triggers on those tables.

11.33 SLONIK UPDATE FUNCTIONS

UPDATE FUNCTIONS — Reload stored functions

Synopsis

```
UPDATE FUNCTIONS (options);
```

Description

Reloads stored functions for a node.

Reloads all stored procedure and function definitions in the Slony-I schema for the specified node. This command is usually part of the Slony-I software upgrade procedure.

ID = ival The node to refresh.

Example

```
UPDATE FUNCTIONS (
    ID = 3        # Update functions on node 3
);
```

Locking Behaviour

No application-visible locking should take place.

Slonik Event Confirmation Behaviour

Slonik does not wait for event confirmations before performing this command.

Version Information

This command was introduced in Slony-I 1.0

Oddities

Any mismatch between slonik(1) and the C libraries 'living' in the PostgreSQL installation will result in this failing to do what is expected, and, more than likely, failing to run at all. You may *think* you are upgrading to version 1.1.5, but if you are running slonik(1) from version 1.1.2, or if you didn't restart the database with a version that has 1.1.5 libraries, and instead are referencing C stored functions from version 1.1.1, the attempt to upgrade will fail, because the sets of C functions have regularly changed between major versions.

Before Slony-I 1.2, the error messages that would result would be not terribly informative; what you'd find, in PostgreSQL logs, is some error message about being unable to load some stored function that happens to be implemented in C. As of 1.2, one of the first things done is to load a stored function to verify version numbers; it complains in a much more direct fashion if you have some versioning mismatch.

11.34 SLONIK WAIT FOR EVENT

WAIT FOR EVENT — Have Slonik script wait for previous event to complete

Synopsis

```
WAIT FOR EVENT (options);
```

Description

Waits for event Confirmation.

Slonik remembers the last event generated on every node during script execution (events generated by earlier calls are currently not checked). In certain situations it is necessary that events generated on one node (such as **CREATE SET**) are processed on another node before issuing more commands (for instance, SLONIK SUBSCRIBE SET(7)). **WAIT FOR EVENT** may be used to cause the slonik script to wait for confirmation of an event, which hopefully means that the subscriber node is ready for the next action.

WAIT FOR EVENT must be called outside of any **try** block in order to work, since new confirm messages don't become visible within a transaction.

ORIGIN = ival | ALL The origin of the event(s) to wait for.

CONFIRMED = ival | ALL The node ID of the receiver that must confirm the event(s).

WAIT ON = ival The ID of the node where the sl_confirm table is to be checked.

TIMEOUT = ival The number of seconds to wait. Default is 600 (10 minutes). **TIMEOUT = 0** causes the script to wait indefinitely.

Example

```
WAIT FOR EVENT (
  ORIGIN = ALL,
  CONFIRMED = ALL,
  WAIT ON = 1
);
```

Locking Behaviour

No application-visible locking should take place.

Version Information

This command was introduced in Slony-I 1.0

In version 2.0, the default value for `WAIT ON` was removed, so a node must be specified.

Oddities

Not all events return interesting results. For instance, many people have run afoul of problems with SLONIK SUBSCRIBE SET(7), when subscribing a new set. Be aware (and beware!) that a SLONIK SUBSCRIBE SET(7) request will return the event confirmation almost immediately, even though there might be several hours of work to do before the subscription is ready. The trouble with SLONIK SUBSCRIBE SET(7) is that it is processed as *two* events, one on the origin node, with a second event, to enable the subscription, on the subscriber.

In order to more reliably monitor from within a slonik(1) script that SLONIK SUBSCRIBE SET(7) is complete, you may submit a SLONIK SYNC(7) event after the subscription, and have the WAIT request wait on the **SYNC** event, as follows.

```
    # Assuming that node 1 is the origin for set 999 that has direct subscribers 2 and 3
    SUBSCRIBE SET (ID = 999, PROVIDER = 1, RECEIVER = 2);
    WAIT FOR EVENT (ORIGIN = 1, CONFIRMED = ALL, WAIT ON=1);
    SUBSCRIBE SET (ID = 999, PROVIDER = 1, RECEIVER = 3);
    WAIT FOR EVENT (ORIGIN = 1, CONFIRMED = ALL, WAIT ON=1);
    MERGE SET ( ID = 1, ADD ID = 999, ORIGIN = 1 );
```

11.35 SLONIK REPAIR CONFIG

REPAIR CONFIG — Resets the name-to-oid mapping of tables in a replication set, useful for restoring a node after a pg_dump.

Synopsis

```
REPAIR CONFIG (options);
```

Description

Resets name-to-oid mapping.

SET ID = ival Which set to clean up after.

EVENT NODE = ival The node ID where this should be submitted.

EXECUTE ONLY ON = ival The ID of the only node where the mappings are to be updated. If not specified, the default is to execute this on all nodes subscribed to the set.

Example

```
REPAIR CONFIG (
  SET ID = 1,
  EVENT NODE = 2
);
```

Locking Behaviour

No application-visible locking should take place.

Slonik Event Confirmation Behaviour

Slonik does not wait for event confirmations before performing this command.

Version Information

This command was introduced in Slony-I 1.1

11.36 SLONIK SYNC

SYNC — Generate an ordinary SYNC event

Synopsis

```
SYNC (options);
```

Description

Generates a SYNC event on a specified node.

ID = ival The node on which to generate the SYNC event.

Example

```
    SYNC (ID = 1);
    WAIT FOR EVENT (ORIGIN = 1, CONFIRMED = 2, WAIT ON=1);
```

Locking Behaviour

No application-visible locking should take place.

Slonik Event Confirmation Behaviour

Slonik does not wait for event confirmations before performing this command.

Version Information

This command was introduced in Slony-I 1.1.6 / 1.2.1

11.37 SLONIK SLEEP

SLEEP — Sleep using system `sleep()`

Synopsis

`sleep [ seconds]`

Description

Sleeps for the specified number of seconds.

Example

```
    sleep (seconds = 5);
```

Slonik Event Confirmation Behaviour

Slonik does not wait for event confirmations before performing this command.

Version Information

This command was introduced in Slony-I 1.1.6 / 1.2.1.

11.38 SLONIK CLONE PREPARE

CLONE PREPARE — Prepare for cloning a node.

Synopsis

```
clone prepare [ id][ provider][ comment]
```

Description

Prepares for cloning a specified subscriber node.

This duplicates the 'provider' node's configuration under a new node ID in preparation for the node to be copied via standard database tools.

Note that in order that we be certain that this new node be consistent with all nodes, it is important to issue a SYNC event against every node aside from the provider and wait to start copying the provider database at least until all those SYNC events have been confirmed by the provider. Otherwise, it is possible for the clone to miss some events.

Example

```
clone prepare (id = 33, provider = 22, comment='Clone 33');
sync (id=11);
sync (id=22);
```

Slonik Event Confirmation Behaviour

Slonik will wait until the node being cloned (the provider) is caught up with all other nodes before submitting the clone prepare command

Version Information

This command was introduced in Slony-I 2.0.

11.39 SLONIK CLONE FINISH

CLONE FINISH — Complete cloning a node.

Synopsis

```
clone prepare [ id][ provider]
```

Description

Finishes cloning a specified node.

This completes the work done by SLONIK CLONE PREPARE(7), establishing confirmation data for the new ‘clone’ based on the status found for the ‘provider’ node.

Example

```
clone finish (id = 33, provider = 22);
```

Slonik Event Confirmation Behaviour

Slonik does not wait for event confirmations before performing this command.

Version Information

This command was introduced in Slony-I 2.0.

Chapter 12

Appendix

12.1 Frequently Asked Questions

Slony-I FAQ: Building and Installing Slony-I

1. *I am using Frotznik Freenix 4.5, with its FFPM (Frotznik Freenix Package Manager) package management system. It comes with FFPM packages for PostgreSQL 7.4.7, which are what I am using for my databases, but they don't include Slony-I in the packaging. How do I add Slony-I to this?*

   Frotznik Freenix is new to me, so it's a bit dangerous to give really hard-and-fast definitive answers. The answers differ somewhat between the various combinations of PostgreSQL and Slony-I versions; the newer versions generally somewhat easier to cope with than are the older versions. In general, you almost certainly need to compile Slony-I from sources; depending on versioning of both Slony-I and PostgreSQL, you *may* need to compile PostgreSQL from scratch. (Whether you need to *use* the PostgreSQL compile is another matter; you probably don't...)

   - Slony-I version 1.0.5 and earlier require having a fully configured copy of PostgreSQL sources available when you compile Slony-I.

     Hopefully you can make the configuration this closely match against the configuration in use by the packaged version of PostgreSQL by checking the configuration using the command **pg_config --configure**.
   - Slony-I version 1.1 simplifies this considerably; it does not require the full copy of PostgreSQL sources, but can, instead, refer to the various locations where PostgreSQL libraries, binaries, configuration, and **#include** files are located.
   - PostgreSQL 8.0 and higher is generally easier to deal with in that a 'default' installation includes all of the **#include** files.

     If you are using an earlier version of PostgreSQL, you may find it necessary to resort to a source installation if the packaged version did not install the 'server **#include**' files, which are installed by the command **make install-all-headers** .

   In effect, the 'worst case' scenario takes place if you are using a version of Slony-I earlier than 1.1 with an 'elderly' version of PostgreSQL, in which case you can expect to need to compile PostgreSQL from scratch in order to have everything that the Slony-I compile needs even though you are using a 'packaged' version of PostgreSQL. If you are running a recent PostgreSQL and a recent Slony-I, then the codependencies can be fairly small, and you may not need extra PostgreSQL sources. These improvements should ease the production of Slony-I packages so that you might soon even be able to hope to avoid compiling Slony-I.

2. *I tried building Slony-I 1.1 and got the following error message:*

   ```
   configure: error: Headers for libpqserver are not found in the includeserverdir.
      This is the path to postgres.h. Please specify the includeserverdir with
      --with-pgincludeserverdir=<dir>
   ```

   You are almost certainly running version PostgreSQL 7.4 or earlier, where server headers are not installed by default if you just do a **make install** of PostgreSQL. You need to install server headers when you install PostgreSQL via the command **make install-all-headers**.

3. *I'm trying to upgrade to a newer version of Slony-I and am running into a problem with SLONIK UPDATE FUNCTIONS(7). When I run SLONIK UPDATE FUNCTIONS(7), my postmaster falls over with a Signal 11. There aren't any seeming errors in the log files, aside from the PostgreSQL logs indicating that, yes indeed, the postmaster fell over. I connected a debugger to the core file, and it indicates that it was trying to commit a transaction at the time of the failure. By the way I'm on PostgreSQL 8.1.[0-3].*

   Unfortunately, early releases of PostgreSQL 8.1 had a problem where if you redefined a function (such as, say, `upgradeSchema(text)`), and then, in the same transaction, ran that function, the postmaster would fall over, and the transaction would fail to commit. The slonik(1) command SLONIK UPDATE FUNCTIONS(7) functions like that; it, in one transaction, tries to:

   - Load the new functions (from `slony1_funcs.sql`), notably including `upgradeSchema(text)`.
   - Run `upgradeSchema(text)` to do any necessary upgrades to the database schema.
   - Notify slon(1) processes of a change of configuration.

   Unfortunately, on PostgreSQL 8.1.0, 8.1.1, 8.1.2, and 8.1.3, this conflicts with a bug where using and modifying a plpgsql function in the same transaction leads to a crash. Several workarounds are available.

   The preferred answer would be to upgrade PostgreSQL to 8.1.4 or some later version. Changes between minor versions do not require rebuilding databases; it should merely require copying a suitable 8.1.x build into place, and restarting the postmaster with the new version.

   If that is unsuitable, it would be possible to perform the upgrade via a series of transactions, performing the equivalent of what slonik(1) does 'by hand':

   - Take `slony1_funcs.sql` and do three replacements within it:
     - Replace '@CLUSTERNAME@' with the name of the cluster
     - Replace '@MODULEVERSION@' with the Slony-I version string, such as '1.2.10'
     - Replace '@NAMESPACE@' with the 'double-quoted' name of the cluster namespace, such as "_MyCluster"
   - Load that 'remapped' set of functions into the database.
   - Run the stored function via **select `upgradeSchema('1.2.7');`** , assuming that the previous version of Slony-I in use was version 1.2.7.
   - Restarting all slon(1) processes would probably be a wise move with this sort of 'surgery.'

4. *Problem building on Fedora/x86-64 When trying to configure Slony-I on a Fedora x86-64 system, where yum was used to install the package `postgresql-libs.x86_64`, the following complaint comes up:*

```
configure: error: Your version of libpq doesn't have PQunescapeBytea
 this means that your version of PostgreSQL is lower than 7.3
 and thus not supported by Slony-I.
```

   This happened with PostgreSQL 8.2.5, which is certainly rather newer than 7.3.

   configure is looking for that symbol by compiling a little program that calls for it, and checking if the compile succeeds. On the **gcc** command line it uses **-lpq** to search for the library. Unfortunately, that package is missing a symlink, from `/usr/lib64/libpq.so` to `libpq.so.5.0`; that is why it fails to link to libpq. The *true* problem is that the compiler failed to find a library to link to, not that libpq lacked the function call. Eventually, this should be addressed by those that manage the `postgresql-libs.x86_64` package.

   Note that this same symptom can be the indication of similar classes of system configuration problems. Bad symlinks, bad permissions, bad behaviour on the part of your C compiler, all may potentially lead to this same error message. Thus, if you see this error, you need to look in the log file that is generated, `config.log`. Search down to near the end, and see what the *actual* complaint was. That will be helpful in tracking down the true root cause of the problem.

5. *I found conflicting types for `yyleng` between `parser.c` and `scan.c`. In one case, it used type int, conflicting with yy_size_t. What shall I do?*

   This has been observed on MacOS, where flex (which generates `scan.c`) and bison (which generates `parser.c`) diverged in their handling of this variable.

   - You might might 'hack' `scan.c` by hand to use the matching type.

- You might select different versions of bison or flex so as to get versions whose data types match.
- Note that you may have multiple versions of bison or flex around, and might need to modify `PATH` in order to select the appropriate one.

Slony-I FAQ: How Do I?

1. *I need to dump a database* without *getting Slony-I configuration (*e.g. *- triggers, functions, and such).*

   Up to version 1.2, this is fairly nontrivial, requiring careful choice of nodes, and some moderately heavy 'procedure'. One methodology is as follows:

   - First, dump the schema from the node that has the 'master' role. That is the only place, pre-2.0, where you can readily dump the schema using pg_dump and have a consistent schema. You may use the Slony-I tool Section 6.1.5 to do this.
   - Take the resulting schema, which will *not* include the Slony-I-specific bits, and split it into two pieces:
     - Firstly, the portion comprising all of the creations of tables in the schema.
     - Secondly, the portion consisting of creations of indices, constraints, and triggers.
   - Pull a data dump, using **pg_dump --data-only**, of some node of your choice. It doesn't need to be for the 'master' node. This dump will include the contents of the Slony-I-specific tables; you can discard that, or ignore it. Since the schema dump didn't contain table definitions for the Slony-I tables, they won't be loaded.
   - Finally, load the three components in proper order:
     - Schema (tables)
     - Data dump
     - Remainder of the schema

   In Slony-I 2.0, the answer becomes simpler: Just take a **pg_dump --exclude-schema=_Cluster** against *any* node. In 2.0, the schemas are no longer 'clobbered' on subscribers, so a straight pg_dump will do what you want.

2. *I'd like to renumber the node numbers in my cluster. How can I renumber nodes?*

   The first answer is 'you can't do that' - Slony-I node numbers are quite 'immutable.' Node numbers are deeply woven into the fibres of the schema, by virtue of being written into virtually every table in the system, but much more importantly by virtue of being used as the basis for event propagation. The only time that it might be 'OK' to modify a node number is at some time where we know that it is not in use, and we would need to do updates against each node in the cluster in an organized fashion. To do this in an automated fashion seems like a *huge* challenge, as it changes the structure of the very event propagation system that already needs to be working in order for such a change to propagate.

   If it is *enormously necessary* to renumber nodes, this might be accomplished by dropping and re-adding nodes to get rid of the node formerly using the node ID that needs to be held by another node.

Slony-I FAQ: Impossible Things People Try

1. *Can I use Slony-I to replicate changes back and forth on my database between my two offices?*

   At one level, it is *theoretically possible* to do something like that, if you design your application so that each office has its own distinct set of tables, and you then have some system for consolidating the data to give them some common view. However, this requires a great deal of design work to create an application that performs this consolidation.

   In practice, the term for that is 'multimaster replication,' and Slony-I does not support 'multimaster replication.'

2. *I want to replicate all of the databases for a shared-database system I am managing. There are multiple databases, being used by my customers.*

   For this purpose, something like PostgreSQL PITR (Point In Time Recovery) is likely to be much more suitable. Slony-I requires a slon process (and multiple connections) for each identifiable database, and if you have a PostgreSQL cluster hosting 50 or 100 databases, this will require hundreds of database connections. Typically, in 'shared hosting' situations, DML is being managed by customers, who can change anything they like whenever *they* want. Slony-I does not work out well when not used in a disciplined manner.

3. *I want to be able to make DDL changes, and have them replicated automatically.*

   Slony-I requires that Section 3.3 be planned for explicitly and carefully. Slony-I captures changes using triggers, and PostgreSQL does not provide a way to use triggers to capture DDL changes.

   Note
   There has been quite a bit of discussion, off and on, about how PostgreSQL might capture DDL changes in a way that would make triggers useful; nothing concrete has emerged after several years of discussion.

4. *I want to split my cluster into disjoint partitions that are not aware of one another. Slony-I keeps generating Section 4.2 that link those partitions together.*

   The notion that all nodes are aware of one another is deeply imbedded in the design of Slony-I. For instance, its handling of cleanup of obsolete data depends on being aware of whether any of the nodes are behind, and thus might still depend on older data.

5. *I want to change some of my node numbers. How do I 'rename' a node to have a different node number?*

   You don't. The node number is used to coordinate inter-node communications, and changing the node ID number 'on the fly' would make it essentially impossible to keep node configuration coordinated.

6. *My application uses OID attributes; is it possible to replicate tables like this?*

   It is worth noting that oids, as a regular table attribute, have been deprecated since PostgreSQL version 8.1, back in 2005. Slony-I has *never* collected oids to replicate them, and, with that functionality being deprecated, the developers do not intend to add this functionality. PostgreSQL implemented oids as a way to link its internal system tables together; to use them with application tables is considered *poor practice*, and it is recommended that you use sequences to populate your own ID column on application tables.

   Of course, nothing prevents you from creating a table *without* oids, and then add in your own application column called `oid`, preferably with type information **SERIAL NOT NULL UNIQUE**, which *can* be replicated, and which is likely to be suitable as a candidate primary key for the table.

Slony-I FAQ: Connection Issues

1. *I looked for the `_clustername` namespace, and it wasn't there.*

   If the DSNs are wrong, then slon(1) instances can't connect to the nodes.This will generally lead to nodes remaining entirely untouched.Recheck the connection configuration. By the way, since slon(1) links to libpq, you could have password information stored in `$HOME/.pgpass`, partially filling in right/wrong authentication information there.

2. *I created a 'superuser' account, **slony**, to run replication activities. As suggested, I set it up as a superuser, via the following query: **update pg_shadow set usesuper = 't' where usename in ('slony', 'molly', 'dumpy');** (that command also deals with other users I set up to run vacuums and backups). Unfortunately, I ran into a problem the next time I subscribed to a new set.*

```
DEBUG1 copy_set 28661
DEBUG1 remoteWorkerThread_1: connected to provider DB
DEBUG2 remoteWorkerThread_78: forward confirm 1,594436 received by 78
DEBUG2 remoteWorkerThread_1: copy table public.billing_discount
ERROR  remoteWorkerThread_1: "select "_mycluster".setAddTable_int(28661, 51, 'public. ↩
    billing_discount', 'billing_discount_pkey', 'Table public.billing_discount with  ↩
    candidate primary key billing_discount_pkey'); " PGRES_FATAL_ERROR ERROR:   ↩
    permission denied for relation pg_class
CONTEXT:  PL/pgSQL function "altertableforreplication" line 23 at select into variables
PL/pgSQL function "setaddtable_int" line 76 at perform
WARN   remoteWorkerThread_1: data copy for set 28661 failed - sleep 60 seconds
```

   *This continues to fail, over and over, until I restarted the slon to connect as **postgres** instead.*

   The problem is fairly self-evident; permission is being denied on the system table, `pg_class`.

   The 'fix' is thus:

```
update pg_shadow set usesuper = 't', usecatupd='t' where usename = 'slony';
```

In version 8.1 and higher, you may also need the following:

```
update pg_authid set rolcatupdate = 't', rolsuper='t' where rolname = 'slony';
```

3. *I'm trying to get a slave subscribed, and get the following messages in the logs:*

```
DEBUG1 copy_set 1
DEBUG1 remoteWorkerThread_1: connected to provider DB
WARN  remoteWorkerThread_1: transactions earlier than XID 127314958 are still in ↩
    progress
WARN  remoteWorkerThread_1: data copy for set 1 failed - sleep 60 seconds
```

There is evidently some reasonably old outstanding transaction blocking Slony-I from processing the sync. You might want to take a look at pg_locks to see what's up:

```
sampledb=# select * from pg_locks where transaction is not null order by transaction;
 relation | database | transaction |  pid    |     mode      | granted
----------+----------+-------------+---------+---------------+---------
          |          |   127314921 | 2605100 | ExclusiveLock | t
          |          |   127326504 | 5660904 | ExclusiveLock | t
(2 rows)
```

See? 127314921 is indeed older than 127314958, and it's still running. A long running G/L report, a runaway RT3 query, a pg_dump, all will open up transactions that may run for substantial periods of time. Until they complete, or are interrupted, you will continue to see the message ' data copy for set 1 failed - sleep 60 seconds '.By the way, if there is more than one database on the PostgreSQL cluster, and activity is taking place on the OTHER database, that will lead to there being 'transactions earlier than XID whatever' being found to be still in progress. The fact that it's a separate database on the cluster is irrelevant; Slony-I will wait until those old transactions terminate.

4. *Same as the above. What I forgot to mention, as well, was that I was trying to add* TWO *subscribers, concurrently.*

That doesn't work out: Slony-I can't work on the **COPY** commands concurrently. See `src/slon/remote_worker.c`, function `copy_set()`

```
$ ps -aef | egrep '[2]605100'
postgres 2605100  205018  0 18:53:43  pts/3  3:13 postgres: postgres sampledb localhost ↩
    COPY
```

This happens to be a **COPY** transaction involved in setting up the subscription for one of the nodes. All is well; the system is busy setting up the first subscriber; it won't start on the second one until the first one has completed subscribing. That represents one possible cause.This has the (perhaps unfortunate) implication that you cannot populate two slaves concurrently from a single provider. You have to subscribe one to the set, and only once it has completed setting up the subscription (copying table contents and such) can the second subscriber start setting up the subscription.

5. *We got bitten by something we didn't foresee when completely uninstalling a slony replication cluster from the master and slave...*

Warning
MAKE SURE YOU STOP YOUR APPLICATION RUNNING AGAINST YOUR MASTER DATABASE WHEN REMOVING THE WHOLE SLONY CLUSTER*, or at least re-cycle all your open connections after the event!*

The connections 'remember' or refer to OIDs which are removed by the uninstall node script. And you will get lots of errors as a result...

There are two notable areas of PostgreSQL that cache query plans and OIDs:

- Prepared statements

- pl/pgSQL functions

The problem isn't particularly a Slony-I one; it would occur any time such significant changes are made to the database schema. It shouldn't be expected to lead to data loss, but you'll see a wide range of OID-related errors.

The problem occurs when you are using some sort of 'connection pool' that keeps recycling old connections. If you restart the application after this, the new connections will create *new* query plans, and the errors will go away. If your connection pool drops the connections, and creates new ones, the new ones will have *new* query plans, and the errors will go away.

In our code we drop the connection on any error we cannot map to an expected condition. This would eventually recycle all connections on such unexpected problems after just one error per connection. Of course if the error surfaces as a constraint violation which is a recognized condition, this won't help either, and if the problem is persistent, the connections will keep recycling which will drop the effect of the pooling, in the latter case the pooling code could also announce an admin to take a look...

6. *I upgraded my cluster to Slony-I version 1.2. I'm now getting the following notice in the logs:*

```
NOTICE:  Slony-I: log switch to sl_log_2 still in progress - sl_log_1 not truncated
```

Both `sl_log_1` and `sl_log_2` are continuing to grow, and `sl_log_1` is never getting truncated. What's wrong?

This is symptomatic of the same issue as above with dropping replication: if there are still old connections lingering that are using old query plans that reference the old stored functions, resulting in the inserts to `sl_log_1` Closing those connections and opening new ones will resolve the issue.

In the longer term, there is an item on the PostgreSQL TODO list to implement dependancy checking that would flush cached query plans when dependent objects change.

7. *I pointed a subscribing node to a different provider and it stopped replicating*

We noticed this happening when we wanted to re-initialize a node, where we had configuration thus:

- Node 1 - provider
- Node 2 - subscriber to node 1 - the node we're reinitializing
- Node 3 - subscriber to node 2 - node that should keep replicating

The subscription for node 3 was changed to have node 1 as provider, and we did SLONIK DROP SET(7) /SLONIK SUBSCRIBE SET(7) for node 2 to get it repopulating.Unfortunately, replication suddenly stopped to node 3.The problem was that there was not a suitable set of 'listener paths' in sl_listen to allow the events from node 1 to propagate to node 3. The events were going through node 2, and blocking behind the SLONIK SUBSCRIBE SET(7) event that node 2 was working on.The following slonik script dropped out the listen paths where node 3 had to go through node 2, and added in direct listens between nodes 1 and 3.

```
cluster name = oxrslive;
 node 1 admin conninfo='host=32.85.68.220 dbname=oxrslive user=postgres port=5432';
 node 2 admin conninfo='host=32.85.68.216 dbname=oxrslive user=postgres port=5432';
 node 3 admin conninfo='host=32.85.68.244 dbname=oxrslive user=postgres port=5432';
 node 4 admin conninfo='host=10.28.103.132 dbname=oxrslive user=postgres port=5432';
try {
  store listen (origin = 1, receiver = 3, provider = 1);
  store listen (origin = 3, receiver = 1, provider = 3);
  drop listen (origin = 1, receiver = 3, provider = 2);
  drop listen (origin = 3, receiver = 1, provider = 2);
}
```

Immediately after this script was run, **SYNC** events started propagating again to node 3. This points out two principles:

- If you have multiple nodes, and cascaded subscribers, you need to be quite careful in populating the SLONIK STORE LISTEN(7) entries, and in modifying them if the structure of the replication 'tree' changes.
- Version 1.1 provides better tools to help manage this.

The issues of 'listener paths' are discussed further at Section 4.2

8. *I was starting a slon(1), and got the following 'FATAL' messages in its logs. What's up???*

```
2006-03-29 16:01:34 UTC CONFIG main: slon version 1.2.0 starting up
2006-03-29 16:01:34 UTC DEBUG2 slon: watchdog process started
2006-03-29 16:01:34 UTC DEBUG2 slon: watchdog ready - pid = 28326
2006-03-29 16:01:34 UTC DEBUG2 slon: worker process created - pid = 28327
2006-03-29 16:01:34 UTC CONFIG main: local node id = 1
2006-03-29 16:01:34 UTC DEBUG2 main: main process started
2006-03-29 16:01:34 UTC CONFIG main: launching sched_start_mainloop
2006-03-29 16:01:34 UTC CONFIG main: loading current cluster configuration
2006-03-29 16:01:34 UTC CONFIG storeSet: set_id=1 set_origin=1 set_comment='test set'
2006-03-29 16:01:34 UTC DEBUG2 sched_wakeup_node(): no_id=1 (0 threads + worker
    signaled)
2006-03-29 16:01:34 UTC DEBUG2 main: last local event sequence = 7
2006-03-29 16:01:34 UTC CONFIG main: configuration complete - starting threads
2006-03-29 16:01:34 UTC DEBUG1 localListenThread: thread starts
2006-03-29 16:01:34 UTC FATAL  localListenThread: "select "_test1538".cleanupNodelock()
    ; insert into "_test1538".sl_nodelock values (    1, 0, "pg_catalog".pg_backend_pid
    ()); " - ERROR:  duplicate key violates unique constraint "sl_nodelock-pkey"

2006-03-29 16:01:34 UTC FATAL  Do you already have a slon running against this node?
2006-03-29 16:01:34 UTC FATAL  Or perhaps a residual idle backend connection from a
    dead slon?
```

The table `sl_nodelock` is used as an 'interlock' to prevent two slon(1) processes from trying to manage the same node at the same time. The slon(1) tries inserting a record into the table; it can only succeed if it is the only node manager.

This error message is typically a sign that you have started up a second slon(1) process for a given node. The slon(1) asks the obvious question: 'Do you already have a slon running against this node?'

Supposing you experience some sort of network outage, the connection between slon(1) and database may fail, and the slon(1) may figure this out long before the PostgreSQL instance it was connected to does. The result is that there will be some number of idle connections left on the database server, which won't be closed out until TCP/IP timeouts complete, which seems to normally take about two hours. For that two hour period, the slon(1) will try to connect, over and over, and will get the above fatal message, over and over. An administrator may clean this out by logging onto the server and issuing **kill -2** to any of the offending connections. Unfortunately, since the problem took place within the networking layer, neither PostgreSQL nor Slony-I have a direct way of detecting this. You can *mostly* avoid this by making sure that slon(1) processes always run somewhere nearby the server that each one manages. If the slon(1) runs on the same server as the database it manages, any 'networking failure' that could interrupt local connections would be likely to be serious enough to threaten the entire server.

9. *When can I shut down slon(1) processes?*

   Generally, it's no big deal to shut down a slon(1) process. Each one is 'merely' a PostgreSQL client, managing one node, which spawns threads to manage receiving events from other nodes. The 'event listening' threads are no big deal; they are doing nothing fancier than periodically checking remote nodes to see if they have work to be done on this node. If you kill off the slon(1) these threads will be closed, which should have little or no impact on much of anything. Events generated while the slon(1) is down will be picked up when it is restarted. The 'node managing' thread is a bit more interesting; most of the time, you can expect, on a subscriber, for this thread to be processing **SYNC** events. If you shut off the slon(1) during an event, the transaction will fail, and be rolled back, so that when the slon(1) restarts, it will have to go back and reprocess the event. The only situation where this will cause *particular* 'heartburn' is if the event being processed was one which takes a long time to process, such as **COPY_SET** for a large replication set. The other thing that *might* cause trouble is if the slon(1) runs fairly distant from nodes that it connects to; you could discover that database connections are left **idle in transaction**. This would normally only occur if the network connection is destroyed without either slon(1) or database being made aware of it. In that case, you may discover that 'zombied' connections are left around for as long as two hours if you don't go in by hand and kill off the PostgreSQL backends. There is one other case that could cause trouble; when the slon(1) managing the origin node is not running, no **SYNC** events run against that node. If the slon(1) stays down for an extended period of time, and something like Section 6.1.11 isn't running, you could be left with *one big **SYNC*** to process when it comes back up. But that is only a concern if that slon(1) is down for an extended period of time; shutting it down for a few seconds shouldn't cause any great problem.

10. *Are there risks to doing so? How about benefits?*

In short, if you don't have something like an 18 hour **COPY_SET** under way, it's normally not at all a big deal to take a slon(1) down for a little while, or perhaps even cycle *all* the slon(1) processes.

11. *I was trying to subscribe a set involving a multiple GB table, and it failed.*

```
Jul 31 22:52:53 dbms TICKER[70295]: [153-1] CONFIG remoteWorkerThread_3: copy table " ↩
    public"."images"
Jul 31 22:52:53 dbms TICKER[70295]: [154-1] CONFIG remoteWorkerThread_3: Begin COPY of  ↩
    table "public"."images"
Jul 31 22:54:24 dbms TICKER[70295]: [155-1] ERROR  remoteWorkerThread_3: PGgetCopyData ↩
    () server closed the connection unexpectedly
Jul 31 22:54:24 dbms TICKER[70295]: [155-2]     This probably means the server  ↩
    terminated abnormally
Jul 31 22:54:24 dbms TICKER[70295]: [155-3]     before or while processing the request.
Jul 31 22:54:24 dbms TICKER[70295]: [156-1] WARN   remoteWorkerThread_3: data copy for  ↩
    set 1 failed 1 times - sleep 15 seconds
```

Oh, by the way, I'm using SSL-based PostgreSQL conenctions.

A further examination of PostgreSQL logs indicated errors of the form:

```
Jul 31 23:00:00 tickerforum postgres[27093]: [9593-2] STATEMENT:  copy "public"."images ↩
    "
("post_ordinal","ordinal","caption","image","login","file_type","thumb","thumb_width"," ↩
    thumb_height","hidden") to stdout;
Jul 31 23:00:00 tickerforum postgres[27093]: [9594-1] LOG:  SSL error: internal error
Jul 31 23:00:00 tickerforum postgres[27093]: [9594-2] STATEMENT:  copy "public"."images ↩
    " ("post_ordinal","ordinal","caption","image","login","file_type","thumb"," ↩
    thumb_width","thumb_height","hidden") to stdout;
Jul 31 23:00:01 tickerforum postgres[27093]: [9595-1] LOG:  SSL error: internal error
```

This demonstrates a problem with PostgreSQL handling of SSL connections, which is 'out of scope' for Slony-I proper (*e.g.* - there's no 'there' inside Slony-I for us to try to fix). The resolution to the underlying problem will presumably be handled upstream in the PostgreSQL project; the workaround is to, at least for the initial **SUBSCRIBE SET** event, switch to a non-SSL PostgreSQL connection.

Slony-I FAQ: Configuration Issues

1. *Slonik fails - cannot load PostgreSQL library - **PGRES_FATAL_ERROR load '$libdir/xxid';** When I run the sample setup script I get an error message similar to: **stdin:64: PGRES_FATAL_ERROR load '$libdir/xxid'; - ERROR: LOAD: could not open file '$libdir/xxid': No such file or directory***

   Evidently, you haven't got the `xxid.so` library in the `$libdir` directory that the PostgreSQL instance is using. Note that the Slony-I components need to be installed in the PostgreSQL software installation for *each and every one* of the nodes, not just on the origin node.This may also point to there being some other mismatch between the PostgreSQL binary instance and the Slony-I instance. If you compiled Slony-I yourself, on a machine that may have multiple PostgreSQL builds 'lying around,' it's possible that the slon or slonik binaries are asking to load something that isn't actually in the library directory for the PostgreSQL database cluster that it's hitting.Long and short: This points to a need to 'audit' what installations of PostgreSQL and Slony-I you have in place on the machine(s). Unfortunately, just about any mismatch will cause things not to link up quite right. ... Life is simplest if you only have one set of PostgreSQL binaries on a given server; in that case, there isn't a 'wrong place' in which Slony-I components might get installed. If you have several software installs, you'll have to verify that the right versions of Slony-I components are associated with the right PostgreSQL binaries.

2. *I tried creating a CLUSTER NAME with a "-" in it. That didn't work.*

   Slony-I uses the same rules for unquoted identifiers as the PostgreSQL main parser, so no, you probably shouldn't put a "-" in your identifier name. You may be able to defeat this by putting 'quotes' around identifier names, but it's still liable to bite you some, so this is something that is probably not worth working around.

3. *ps finds passwords on command line If I run a **ps** command, I, and everyone else, can see passwords on the command line.*

   Take the passwords out of the Slony configuration, and put them into `$(HOME)/.pgpass`.

4. *Table indexes with FQ namespace names*

```
set add table (set id = 1, origin = 1, id = 27,
               full qualified name = 'nspace.some_table',
               key = 'key_on_whatever',
               comment = 'Table some_table in namespace nspace with a candidate primary ↩
                    key');
```

   If you have **key = 'nspace.key_on_whatever'** the request will *FAIL*.

5. *Replication has fallen behind, and it appears that the queries to draw data from sl_log_1/sl_log_2 are taking a long time to pull just a few **SYNC**s.*

   Until version 1.1.1, there was only one index on sl_log_1/sl_log_2, and if there were multiple replication sets, some of the columns on the index would not provide meaningful selectivity. If there is no index on column `log_xid`, consider adding it. See `slony1_base.sql` for an example of how to create the index.

6. *I need to rename a column that is in the primary key for one of my replicated tables. That seems pretty dangerous, doesn't it? I have to drop the table out of replication and recreate it, right?*

   Actually, this is a scenario which works out remarkably cleanly. Slony-I does indeed make intense use of the primary key columns, but actually does so in a manner that allows this sort of change to be made very nearly transparently. Suppose you revise a column name, as with the SQL DDL **alter table accounts alter column aid rename to cid;** This revises the names of the columns in the table; it *simultaneously* renames the names of the columns in the primary key index. The result is that the normal course of things is that altering a column name affects both aspects simultaneously on a given node. The *ideal* and proper handling of this change would involve using SLONIK EXECUTE SCRIPT(7) to deploy the alteration, which ensures it is applied at exactly the right point in the transaction stream on each node. Interestingly, that isn't forcibly necessary. As long as the alteration is applied on the replication set's origin before application on subscribers, things won't break irrepairably. Some **SYNC** events that do not include changes to the altered table can make it through without any difficulty... At the point that the first update to the table is drawn in by a subscriber, *that* is the point at which **SYNC** events will start to fail, as the provider will indicate the 'new' set of columns whilst the subscriber still has the 'old' ones. If you then apply the alteration to the subscriber, it can retry the **SYNC**, at which point it will, finding the 'new' column names, work just fine.

7. *I have a PostgreSQL 7.2-based system that I* really, really *want to use Slony-I to help me upgrade it to 8.0. What is involved in getting Slony-I to work for that?*

   Rod Taylor has reported the following... This is approximately what you need to do:

   - Take the 7.3 templates and copy them to 7.2 -- or otherwise hardcode the version your using to pick up the 7.3 templates
   - Remove all traces of schemas from the code and sql templates. I basically changed the "." to an "_".
   - Bunch of work related to the XID datatype and functions. For example, Slony creates CASTs for the xid to xxid and back -- but 7.2 cannot create new casts that way so you need to edit system tables by hand. I recall creating an Operator Class and editing several functions as well.
   - sl_log_1 will have severe performance problems with any kind of data volume. This required a number of index and query changes to optimize for 7.2. 7.3 and above are quite a bit smarter in terms of optimizations they can apply.
   - Don't bother trying to make sequences work. Do them by hand after the upgrade using pg_dump and grep.

   Of course, now that you have done all of the above, it's not compatible with standard Slony now. So you either need to implement 7.2 in a less hackish way, or you can also hack up slony to work without schemas on newer versions of PostgreSQL so they can talk to each other. Almost immediately after getting the DB upgraded from 7.2 to 7.4, we deinstalled the hacked up Slony (by hand for the most part), and started a migration from 7.4 to 7.4 on a different machine using the regular Slony. This was primarily to ensure we didn't keep our system catalogues which had been manually fiddled with. All that said, we upgraded a few hundred GB from 7.2 to 7.4 with about 30 minutes actual downtime (versus 48 hours for a dump / restore cycle) and no data loss.

   That represents a sufficiently ugly set of 'hackery' that the developers are exceedingly reluctant to let it anywhere near to the production code. If someone were interested in 'productionizing' this, it would probably make sense to do so based on the Slony-I 1.0 branch, with the express plan of *not* trying to keep much in the way of forwards compatibility or long term maintainability of replicas. You should only head down this road if you are sufficiently comfortable with PostgreSQL and Slony-I that you are prepared to hack pretty heavily with the code.

8. *I had a network 'glitch' that led to my using SLONIK FAILOVER(7) to fail over to an alternate node. The failure wasn't a disk problem that would corrupt databases; why do I need to rebuild the failed node from scratch?*

   The action of SLONIK FAILOVER(7) is to *abandon* the failed node so that no more Slony-I activity goes to or from that node. As soon as that takes place, the failed node will progressively fall further and further out of sync.

   The *big* problem with trying to recover the failed node is that it may contain updates that never made it out of the origin. If they get retried, on the new origin, you may find that you have conflicting updates. In any case, you do have a sort of 'logical' corruption of the data even if there never was a disk failure making it 'physical.'

   As discusssed in Section 3.4, using SLONIK FAILOVER(7) should be considered a *last resort* as it implies that you are abandoning the origin node as being corrupted.

9. *After notification of a subscription on* another *node, replication falls over on one of the subscribers, with the following error message:*

```
ERROR  remoteWorkerThread_1: "begin transaction; set transaction isolation level  ↩
    serializable; lock table "_livesystem".sl_config_lock; select "_livesystem". ↩
    enableSubscription(25506, 1, 501); notify "_livesystem_Event"; notify " ↩
    _livesystem_Confirm"; insert into "_livesystem".sl_event     (ev_origin, ev_seqno,  ↩
    ev_timestamp,      ev_minxid, ev_maxxid, ev_xip, ev_type , ev_data1, ev_data2,  ↩
    ev_data3, ev_data4    ) values ('1', '4896546', '2005-01-23 16:08:55.037395',  ↩
    '1745281261', '1745281262', '', 'ENABLE_SUBSCRIPTION', '25506', '1', '501', 't');  ↩
    insert into "_livesystem".sl_confirm      (con_origin, con_received, con_seqno,  ↩
    con_timestamp)    values (1, 4, '4896546', CURRENT_TIMESTAMP); commit transaction;" ↩
     PGRES_FATAL_ERROR ERROR:  insert or update on table "sl_subscribe" violates  ↩
    foreign key constraint "sl_subscribe-sl_path-ref"
DETAIL:  Key (sub_provider,sub_receiver)=(1,501) is not present in table "sl_path".
```

   This is then followed by a series of failed syncs as the slon(1) shuts down:

```
DEBUG2 remoteListenThread_1: queue event 1,4897517 SYNC
DEBUG2 remoteListenThread_1: queue event 1,4897518 SYNC
DEBUG2 remoteListenThread_1: queue event 1,4897519 SYNC
DEBUG2 remoteListenThread_1: queue event 1,4897520 SYNC
DEBUG2 remoteWorker_event: ignore new events due to shutdown
DEBUG2 remoteListenThread_1: queue event 1,4897521 SYNC
DEBUG2 remoteWorker_event: ignore new events due to shutdown
DEBUG2 remoteListenThread_1: queue event 1,4897522 SYNC
DEBUG2 remoteWorker_event: ignore new events due to shutdown
DEBUG2 remoteListenThread_1: queue event 1,4897523 SYNC
```

   If you see a slon(1) shutting down with *ignore new events due to shutdown* log entries, you typically need to step back in the log to *before* they started failing to see indication of the root cause of the problem.

   In this particular case, the problem was that some of the SLONIK STORE PATH(7) commands had not yet made it to node 4 before the SLONIK SUBSCRIBE SET(7) command propagated. This demonstrates yet another example of the need to not do things in a rush; you need to be sure things are working right *before* making further configuration changes.

10. *I just used SLONIK MOVE SET(7) to move the origin to a new node. Unfortunately, some subscribers are still pointing to the former origin node, so I can't take it out of service for maintenance without stopping them from getting updates. What do I do?*

    You need to use SLONIK SUBSCRIBE SET(7) to alter the subscriptions for those nodes to have them subscribe to a provider that *will* be sticking around during the maintenance.

    Warning
    What you *don't* do is to SLONIK UNSUBSCRIBE SET(7); that would require reloading all data for the nodes from scratch later.

11. *After notification of a subscription on* another *node, replication falls over, starting with the following error message:*

```
ERROR  remoteWorkerThread_1: "begin transaction; set transaction isolation level  ↩
    serializable; lock table "_livesystem".sl_config_lock; select "_livesystem". ↩
    enableSubscription(25506, 1, 501); notify "_livesystem_Event"; notify " ↩
    _livesystem_Confirm"; insert into "_livesystem".sl_event     (ev_origin, ev_seqno,  ↩
    ev_timestamp,      ev_minxid, ev_maxxid, ev_xip, ev_type , ev_data1, ev_data2,  ↩
    ev_data3, ev_data4    ) values ('1', '4896546', '2005-01-23 16:08:55.037395',  ↩
    '1745281261', '1745281262', '', 'ENABLE_SUBSCRIPTION', '25506', '1', '501', 't');  ↩
    insert into "_livesystem".sl_confirm      (con_origin, con_received, con_seqno,  ↩
    con_timestamp)    values (1, 4, '4896546', CURRENT_TIMESTAMP); commit transaction;" ↩
     PGRES_FATAL_ERROR ERROR:  insert or update on table "sl_subscribe" violates  ↩
    foreign key constraint "sl_subscribe-sl_path-ref"
DETAIL:  Key (sub_provider,sub_receiver)=(1,501) is not present in table "sl_path".
```

This is then followed by a series of failed syncs as the slon(1) shuts down:

```
DEBUG2 remoteListenThread_1: queue event 1,4897517 SYNC
DEBUG2 remoteListenThread_1: queue event 1,4897518 SYNC
DEBUG2 remoteListenThread_1: queue event 1,4897519 SYNC
DEBUG2 remoteListenThread_1: queue event 1,4897520 SYNC
DEBUG2 remoteWorker_event: ignore new events due to shutdown
DEBUG2 remoteListenThread_1: queue event 1,4897521 SYNC
DEBUG2 remoteWorker_event: ignore new events due to shutdown
DEBUG2 remoteListenThread_1: queue event 1,4897522 SYNC
DEBUG2 remoteWorker_event: ignore new events due to shutdown
DEBUG2 remoteListenThread_1: queue event 1,4897523 SYNC
```

If you see a slon(1) shutting down with *ignore new events due to shutdown* log entries, you'll typically have to step back to *before* they started failing to see indication of the root cause of the problem.

In this particular case, the problem was that some of the SLONIK STORE PATH(7) commands had not yet made it to node 4 before the SLONIK SUBSCRIBE SET(7) command propagated. This is yet another example of the need to not do things too terribly quickly; you need to be sure things are working right *before* making further configuration changes.

12. *Is the ordering of tables in a set significant?*

    Most of the time, it isn't. You might imagine it of some value to order the tables in some particular way in order that 'parent' entries would make it in before their 'children' in some foreign key relationship; that *isn't* the case since foreign key constraint triggers are turned off on subscriber nodes.

    (Jan Wieck comments:) The order of table ID's is only significant during a SLONIK LOCK SET(7) in preparation of switchover. If that order is different from the order in which an application is acquiring its locks, it can lead to deadlocks that abort either the application or slon.

    (David Parker) I ran into one other case where the ordering of tables in the set was significant: in the presence of inherited tables. If a child table appears before its parent in a set, then the initial subscription will end up deleting that child table after it has possibly already received data, because the **copy_set** logic does a **delete**, not a **delete only**, so the delete of the parent will delete the new rows in the child as well.

13. *If you have a slonik(1) script something like this, it will hang on you and never complete, because you can't have **wait for event** inside a **try** block. A **try** block is executed as one transaction, and the event that you are waiting for can never arrive inside the scope of the transaction.*

```
try {
      echo 'Moving set 1 to node 3';
      lock set (id=1, origin=1);
      echo 'Set locked';
      wait for event (origin = 1, confirmed = 3);
      echo 'Moving set';
      move set (id=1, old origin=1, new origin=3);
      echo 'Set moved - waiting for event to be confirmed by node 3';
      wait for event (origin = 1, confirmed = 3);
      echo 'Confirmed';
} on error {
```

```
        echo 'Could not move set for cluster foo';
        unlock set (id=1, origin=1);
        exit -1;
}
```

You must not invoke SLONIK WAIT FOR EVENT(7) inside a 'try' block.

14. *Slony-I: cannot add table to currently subscribed set 1 I tried to add a table to a set, and got the following message:*

```
  Slony-I: cannot add table to currently subscribed set 1
```

You cannot add tables to sets that already have subscribers.The workaround to this is to create *ANOTHER* set, add the new tables to that new set, subscribe the same nodes subscribing to "set 1" to the new set, and then merge the sets together.

15. *ERROR: duplicate key violates unique constraint "sl_table-pkey"I tried setting up a second replication set, and got the following error:*

```
stdin:9: Could not create subscription set 2 for oxrslive!
stdin:11: PGRES_FATAL_ERROR select "_oxrslive".setAddTable(2, 1, 'public.replic_test',  ↩
    'replic_test__Slony-I_oxrslive_rowID_key', 'Table public.replic_test without  ↩
    primary key');  - ERROR:  duplicate key violates unique constraint "sl_table-pkey"
CONTEXT:  PL/pgSQL function "setaddtable_int" line 71 at SQL statement
```

The table IDs used in SLONIK SET ADD TABLE(7) are required to be unique *ACROSS ALL SETS*. Thus, you can't restart numbering at 1 for a second set; if you are numbering them consecutively, a subsequent set has to start with IDs after where the previous set(s) left off.

16. *One of my nodes fell over (slon(1) / postmaster was down) and nobody noticed for several days. Now, when the slon(1) for that node starts up, it runs for about five minutes, then terminates, with the error message:* ***ERROR: remoteListenThread_%d: timeout for event selection*** *What's wrong, and what do I do?*

The problem is that the listener thread (in `src/slon/remote_listener.c`) timed out when trying to determine what events were outstanding for that node. By default, the query will run for five minutes; if there were many days worth of outstanding events, this might take too long.

On versions of Slony-I before 1.1.7, 1.2.7, and 1.3, one answer would be to increase the timeout in `src/slon/remote_listener.c`, recompile slon(1), and retry.

Another would be to treat the node as having failed, and use the slonik(1) command SLONIK DROP NODE(7) to drop the node, and recreate it. If the database is heavily updated, it may well be cheaper to do this than it is to find a way to let it catch up.

In newer versions of Slony-I, there is a new configuration parameter called slon_conf_remote_listen_timeout; you'd alter the config file to increase the timeout, and try again. Of course, as mentioned above, it could be faster to drop the node and recreate it than to let it catch up across a week's worth of updates...

Slony-I FAQ: Performance Issues

1. *Replication has been slowing down, I'm seeing* ***FETCH 100 FROM LOG*** *queries running for a long time, sl_log_1/sl_log_2 is growing, and performance is, well, generally getting steadily worse.*

There are actually a number of possible causes for this sort of thing. There is a question involving similar pathology where the problem is that pg_listener grows because it is not vacuumed. Another ' proximate cause ' for this growth is for there to be a connection connected to the node that sits **IDLE IN TRANSACTION** for a very long time. That open transaction will have multiple negative effects, all of which will adversely affect performance:

- Vacuums on all tables, including `pg_listener`, will not clear out dead tuples from before the start of the idle transaction.
- The cleanup thread will be unable to clean out entries in sl_log_1, sl_log_2, and sl_seqlog, with the result that these tables will grow, ceaselessly, until the transaction is closed.

You can monitor for this condition inside the database only if the PostgreSQL `postgresql.conf` parameter `stats_command_string` is set to true. If that is set, then you may submit the query **select * from pg_stat_activity where current_query like '%IDLE% in transaction';** which will find relevant activity.

You should also be able to search for ' idle in transaction ' in the process table to find processes that are thus holding on to an ancient transaction.

It is also possible (though rarer) for the problem to be a transaction that is, for some other reason, being held open for a very long time. The `query_start` time in `pg_stat_activity` may show you some query that has been running way too long.

There are plans for PostgreSQL to have a timeout parameter, `open_idle_transaction_timeout`, which would cause old transactions to time out after some period of disuse. Buggy connection pool logic is a common culprit for this sort of thing. There are plans for pgpool to provide a better alternative, eventually, where connections would be shared inside a connection pool implemented in C. You may have some more or less buggy connection pool in your Java or PHP application; if a small set of *real* connections are held in pgpool, that will hide from the database the fact that the application imagines that numerous of them are left idle in transaction for hours at a time.

2. *After dropping a node, sl_log_1/sl_log_2 aren't getting purged out anymore.*

   This is a common scenario in versions before 1.0.5, as the 'clean up' that takes place when purging the node does not include purging out old entries from the Slony-I table, sl_confirm, for the recently departed node. The node is no longer around to update confirmations of what syncs have been applied on it, and therefore the cleanup thread that purges log entries thinks that it can't safely delete entries newer than the final sl_confirm entry, which rather curtails the ability to purge out old logs.Diagnosis: Run the following query to see if there are any 'phantom/obsolete/blocking' sl_confirm entries:

```
oxrsbar=# select * from _oxrsbar.sl_confirm where con_origin not in (select no_id from ↩
    _oxrsbar.sl_node) or con_received not in (select no_id from _oxrsbar.sl_node);
 con_origin | con_received | con_seqno |        con_timestamp
------------+--------------+-----------+----------------------------
          4 |          501 |     83999 | 2004-11-09 19:57:08.195969
          1 |            2 |   3345790 | 2004-11-14 10:33:43.850265
          2 |          501 |    102718 | 2004-11-14 10:33:47.702086
        501 |            2 |      6577 | 2004-11-14 10:34:45.717003
          4 |            5 |     83999 | 2004-11-14 21:11:11.111686
          4 |            3 |     83999 | 2004-11-24 16:32:39.020194
(6 rows)
```

   In version 1.0.5, the SLONIK DROP NODE(7) function purges out entries in sl_confirm for the departing node. In earlier versions, this needs to be done manually. Supposing the node number is 3, then the query would be:

```
delete from _namespace.sl_confirm where con_origin = 3 or con_received = 3;
```

   Alternatively, to go after 'all phantoms,' you could use

```
oxrsbar=# delete from _oxrsbar.sl_confirm where con_origin not in (select no_id from ↩
    _oxrsbar.sl_node) or con_received not in (select no_id from _oxrsbar.sl_node);
DELETE 6
```

   General 'due diligence' dictates starting with a **BEGIN**, looking at the contents of sl_confirm before, ensuring that only the expected records are purged, and then, only after that, confirming the change with a **COMMIT**. If you delete confirm entries for the wrong node, that could ruin your whole day.You'll need to run this on each node that remains...Note that as of 1.0.5, this is no longer an issue at all, as it purges unneeded entries from sl_confirm in two places:

   - At the time a node is dropped
   - At the start of each `cleanupEvent` run, which is the event in which old data is purged from sl_log_1, sl_log_2, and sl_seqlog

3. *The slon spent the weekend out of commission [for some reason], and it's taking a long time to get a sync through.*

   You might want to take a look at the tables sl_log_1 and sl_log_2 and do a summary to see if there are any really enormous Slony-I transactions in there. Up until at least 1.0.2, there needs to be a slon(1) connected to the origin in order for **SYNC** events to be generated.

Note
As of 1.0.2, function `generate_sync_event()` provides an alternative as backup...

If none are being generated, then all of the updates until the next one is generated will collect into one rather enormous Slony-I transaction.Conclusion: Even if there is not going to be a subscriber around, you *really* want to have a slon running to service the origin node.Slony-I 1.1 provides a stored procedure that allows **SYNC** counts to be updated on the origin based on a cron job even if there is no slon(1) daemon running.

4. *Some nodes start consistently falling behindI have been running Slony-I on a node for a while, and am seeing system performance suffering.I'm seeing long running queries of the form:*

```
fetch 100 from LOG;
```

This can be characteristic of `pg_listener` (which is the table containing **NOTIFY** data) having plenty of dead tuples in it. That makes **NOTIFY** events take a long time, and causes the affected node to gradually fall further and further behind.You quite likely need to do a **VACUUM FULL** on `pg_listener`, to vigorously clean it out, and need to vacuum `pg_listener` really frequently. Once every five minutes would likely be AOK. Slon daemons already vacuum a bunch of tables, and `cleanup_thread.c` contains a list of tables that are frequently vacuumed automatically. In Slony-I 1.0.2, `pg_listener` is not included. In 1.0.5 and later, it is regularly vacuumed, so this should cease to be a direct issue. In version 1.2, `pg_listener` will only be used when a node is only receiving events periodically, which means that the issue should mostly go away even in the presence of evil long running transactions...There is, however, still a scenario where this will still 'bite.' Under MVCC, vacuums cannot delete tuples that were made 'obsolete' at any time after the start time of the eldest transaction that is still open. Long running transactions will cause trouble, and should be avoided, even on subscriber nodes.

5. *I have submitted a SLONIK MOVE SET(7) / SLONIK EXECUTE SCRIPT(7) request, and it seems to be stuck on one of my nodes. Slony-I logs aren't displaying any errors or warnings*

Is it possible that you are running pg_autovacuum, and it has taken out locks on some tables in the replication set? That would somewhat-invisibly block Slony-I from performing operations that require locking acquisition of exclusive locks. You might check for these sorts of locks using the following query: **select l.*, c.relname from pg_locks l, pg_class c where c.oid = l.relation ;** A `ShareUpdateExclusiveLock` lock will block the Slony-I operations that need their own exclusive locks, which are likely queued up, marked as not being granted.

6. *I'm noticing in the logs that a slon(1) is frequently switching in and out of 'polling' mode as it is frequently reporting 'LISTEN - switch from polling mode to use LISTEN' and 'UNLISTEN - switch into polling mode'.*

The thresholds for switching between these modes are controlled by the configuration parameters slon_conf_sync_interval and slon_conf_sync_interval_timeout; if the timeout value (which defaults to 10000, implying 10s) is kept low, that makes it easy for the slon(1) to decide to return to 'listening' mode. You may want to increase the value of the timeout parameter.

Slony-I FAQ: Slony-I Bugs in Elder Versions

1. *The slon(1) processes servicing my subscribers are growing to enormous size, challenging system resources both in terms of swap space as well as moving towards breaking past the 2GB maximum process size on my system. By the way, the data that I am replicating includes some rather large records. We have records that are tens of megabytes in size. Perhaps that is somehow relevant?*

Yes, those very large records are at the root of the problem. The problem is that slon(1) normally draws in about 100 records at a time when a subscriber is processing the query which loads data from the provider. Thus, if the average record size is 10MB, this will draw in 1000MB of data which is then transformed into **INSERT** or **UPDATE** statements, in the slon(1) process' memory. That obviously leads to slon(1) growing to a fairly tremendous size. The number of records that are fetched is controlled by the value `SLON_DATA_FETCH_SIZE` , which is defined in the file `src/slon/slon.h`. The relevant extract of this is shown below.

```
#ifdef  SLON_CHECK_CMDTUPLES
#define SLON_COMMANDS_PER_LINE     1
#define SLON_DATA_FETCH_SIZE     100
#define SLON_WORKLINES_PER_HELPER (SLON_DATA_FETCH_SIZE * 4)
```

```
#else
#define SLON_COMMANDS_PER_LINE     10
#define SLON_DATA_FETCH_SIZE    10
#define SLON_WORKLINES_PER_HELPER (SLON_DATA_FETCH_SIZE * 50)
#endif
```

If you are experiencing this problem, you might modify the definition of SLON_DATA_FETCH_SIZE , perhaps reducing by a factor of 10, and recompile slon(1). There are two definitions as SLON_CHECK_CMDTUPLES allows doing some extra monitoring to ensure that subscribers have not fallen out of SYNC with the provider. By default, this option is turned off, so the default modification to make is to change the second definition of SLON_DATA_FETCH_SIZE from 10 to 1.

In version 1.2, configuration values sync_max_rowsize and sync_max_largemem are associated with a new algorithm that changes the logic as follows. Rather than fetching 100 rows worth of data at a time:

- The **fetch from LOG** query will draw in 500 rows at a time where the size of the attributes does not exceed sync_max_rowsize. With default values, this restricts this aspect of memory consumption to about 8MB.
- Tuples with larger attributes are loaded until aggregate size exceeds the parameter sync_max_largemem. By default, this restricts consumption of this sort to about 5MB. This value is not a strict upper bound; if you have a tuple with attributes 50MB in size, it forcibly *must* be loaded into memory. There is no way around that. But slon(1) at least won't be trying to load in 100 such records at a time, chewing up 10GB of memory by the time it's done.

This should alleviate problems people have been experiencing when they sporadically have series' of very large tuples.

2. *I am trying to replicate UNICODE data from PostgreSQL 8.0 to PostgreSQL 8.1, and am experiencing problems.*

   PostgreSQL 8.1 is quite a lot more strict about what UTF-8 mappings of Unicode characters it accepts as compared to version 8.0. If you intend to use Slony-I to update an older database to 8.1, and might have invalid UTF-8 values, you may be for an unpleasant surprise. Let us suppose we have a database running 8.0, encoding in UTF-8. That database will accept the sequence **'\060\242'** as UTF-8 compliant, even though it is really not. If you replicate into a PostgreSQL 8.1 instance, it will complain about this, either at subscribe time, where Slony-I will complain about detecting an invalid Unicode sequence during the COPY of the data, which will prevent the subscription from proceeding, or, upon adding data, later, where this will hang up replication fairly much irretrievably. (You could hack on the contents of sl_log_1, but that quickly gets *really* unattractive...)There have been discussions as to what might be done about this. No compelling strategy has yet emerged, as all are unattractive. If you are using Unicode with PostgreSQL 8.0, you run a considerable risk of corrupting data. If you use replication for a one-time conversion, there is a risk of failure due to the issues mentioned earlier; if that happens, it appears likely that the best answer is to fix the data on the 8.0 system, and retry. In view of the risks, running replication between versions seems to be something you should not keep running any longer than is necessary to migrate to 8.1. For more details, see the discussion on postgresql-hackers mailing list. .

3. *I am running Slony-I 1.1 and have a 4+ node setup where there are two subscription sets, 1 and 2, that do not share any nodes. I am discovering that confirmations for set 1 never get to the nodes subscribing to set 2, and that confirmations for set 2 never get to nodes subscribing to set 1. As a result, sl_log_1/sl_log_2 grow and grow, and are never purged. This was reported as Slony-I bug 1485 .*

   Apparently the code for RebuildListenEntries() does not suffice for this case. RebuildListenEntries() will be replaced in Slony-I version 1.2 with an algorithm that covers this case. In the interim, you'll want to manually add some sl_listen entries using SLONIK STORE LISTEN(7) or storeListen(), based on the (apparently not as obsolete as we thought) principles described in Section 4.2.

4. *I am finding some multibyte columns (Unicode, Big5) are being truncated a bit, clipping off the last character. Why?*

   This was a bug present until a little after Slony-I version 1.1.0; the way in which columns were being captured by the logtrigger() function could clip off the last byte of a column represented in a multibyte format. Check to see that your version of src/backend/slony1_funcs.c is 1.34 or better; the patch was introduced in CVS version 1.34 of that file.

5. *Bug #1226 indicates an error condition that can come up if you have a replication set that consists solely of sequences.*

   The short answer is that having a replication set consisting only of sequences is not a best practice.

   The problem with a sequence-only set comes up only if you have a case where the only subscriptions that are active for a particular subscriber to a particular provider are for 'sequence-only' sets. If a node gets into that state, replication will fail,

as the query that looks for data from sl_log_1/sl_log_2 has no tables to find, and the query will be malformed, and fail. If a replication set *with* tables is added back to the mix, everything will work out fine; it just *seems* scary. This problem should be resolved some time after Slony-I 1.1.0.

6. *I need to drop a table from a replication set*

   This can be accomplished several ways, not all equally desirable ;-).

   - You could drop the whole replication set, and recreate it with just the tables that you need. Alas, that means recopying a whole lot of data, and kills the usability of the cluster on the rest of the set while that's happening.
   - If you are running 1.0.5 or later, there is the command SET DROP TABLE, which will "do the trick."
   - If you are still using 1.0.1 or 1.0.2, the *essential functionality of SLONIK SET DROP TABLE(7) involves the functionality in `droptable_int()`. You can fiddle this by hand by finding the table ID for the table you want to get rid of, which you can find in sl_table, and then run the following three queries, on each host:*

     ```
     select _slonyschema.alterTableRestore(40);
     select _slonyschema.tableDropKey(40);
     delete from _slonyschema.sl_table where tab_id = 40;
     ```

     The schema will obviously depend on how you defined the Slony-I cluster. The table ID, in this case, 40, will need to change to the ID of the table you want to have go away.

     You'll have to run these three queries on all of the nodes, preferably firstly on the origin node, so that the dropping of this propagates properly. Implementing this via a slonik(1) statement with a new Slony-I event would do that. Submitting the three queries using SLONIK EXECUTE SCRIPT(7) could do that. Also possible would be to connect to each database and submit the queries by hand.

7. *I need to drop a sequence from a replication set*

   If you are running 1.0.5 or later, there is a SLONIK SET DROP SEQUENCE(7) command in Slonik to allow you to do this, parallelling SLONIK SET DROP TABLE(7).If you are running 1.0.2 or earlier, the process is a bit more manual.Supposing I want to get rid of the two sequences listed below, `whois_cachemgmt_seq` and `epp_whoi_cach_seq_`, we start by needing the `seq_id` values.

   ```
   oxrsorg=# select * from _oxrsorg.sl_sequence  where seq_id in (93,59);
    seq_id | seq_reloid | seq_set |       seq_comment
   --------+------------+---------+-------------------------------------
        93 |  107451516 |       1 | Sequence public.whois_cachemgmt_seq
        59 |  107451860 |       1 | Sequence public.epp_whoi_cach_seq_
   (2 rows)
   ```

   The data that needs to be deleted to stop Slony from continuing to replicate these are thus:

   ```
   delete from _oxrsorg.sl_seqlog where seql_seqid in (93, 59);
   delete from _oxrsorg.sl_sequence where seq_id in (93,59);
   ```

   Those two queries could be submitted to all of the nodes via schemadocddlscript_complete(p_only_on_node integer, p_script text, p_set_id integer) / SLONIK EXECUTE SCRIPT(7), thus eliminating the sequence everywhere 'at once.' Or they may be applied by hand to each of the nodes.Similarly to SLONIK SET DROP TABLE(7), this is implemented Slony-I version 1.0.5 as SLONIK SET DROP SEQUENCE(7).

8. *I set up my cluster using pgAdminIII, with cluster name 'MY-CLUSTER'. Time has passed, and I tried using Slonik to make a configuration change, and this is failing with the following error message:*

   ```
   ERROR: syntax error at or near -
   ```

   The problem here is that Slony-I expects cluster names to be valid SQL Identifiers, and slonik(1) enforces this. Unfortunately, pgAdminIII did not do so, and allowed using a cluster name that now causes *a problem.*

   If you have gotten into this spot, it's a problem that we mayn't be help resolve, terribly much. It's *conceivably possible* that running the SQL command **alter namespace "_My-Bad-Clustername" rename to "_BetterClusterName";** against each database may work. That shouldn't particularly *damage* things! On the other hand, when the problem has been experienced, users have found they needed to drop replication and rebuild the cluster.

A change in version 2.0.2 is that a function runs as part of loading functions into the database which checks the validity of the cluster name. If you try to use an invalid cluster name, loading the functions will fail, with a suitable error message, which should prevent things from going wrong even if you're using tools other than slonik(1) to manage setting up the cluster.

Slony-I FAQ: Hopefully Obsolete Issues

1. *slon(1) does not restart after crash After an immediate stop of PostgreSQL (simulation of system crash) in* `pg_listener` *a tuple with* ***relname='_${cluster_name}_Restart'*** *exists. slon doesn't start because it thinks another process is serving the cluster on this node. What can I do? The tuples can't be dropped from this relation. The logs claim that*

   Another slon daemon is serving this node already

   The problem is that the system table `pg_listener`, used by PostgreSQL to manage event notifications, contains some entries that are pointing to backends that no longer exist. The new slon(1) instance connects to the database, and is convinced, by the presence of these entries, that an old slon is still servicing this Slony-I node. The 'trash' in that table needs to be thrown away.It's handy to keep a slonik script similar to the following to run in such cases:

```
twcsds004[/opt/twcsds004/OXRS/slony-scripts]$ cat restart_org.slonik
cluster name = oxrsorg ;
node 1 admin conninfo = 'host=32.85.68.220 dbname=oxrsorg user=postgres port=5532';
node 2 admin conninfo = 'host=32.85.68.216 dbname=oxrsorg user=postgres port=5532';
node 3 admin conninfo = 'host=32.85.68.244 dbname=oxrsorg user=postgres port=5532';
node 4 admin conninfo = 'host=10.28.103.132 dbname=oxrsorg user=postgres port=5532';
restart node 1;
restart node 2;
restart node 3;
restart node 4;
```

   SLONIK RESTART NODE(7) cleans up dead notifications so that you can restart the node.As of version 1.0.5, the startup process of slon looks for this condition, and automatically cleans it up. As of version 8.1 of PostgreSQL, the functions that manipulate `pg_listener` do not support this usage, so for Slony-I versions after 1.1.2 (*e.g.* - 1.1.5), this 'interlock' behaviour is handled via a new table, and the issue should be transparently 'gone.'

2. *I tried the following query which did not work:*

```
sdb=# explain select query_start, current_query from pg_locks join
pg_stat_activity on pid = procpid where granted = true and transaction
in (select transaction from pg_locks where granted = false);

ERROR: could not find hash function for hash operator 716373
```

   It appears the Slony-I `xxid` *functions are claiming to be capable of hashing, but cannot actually do so. What's up?*

   Slony-I defined an XXID data type and operators on that type in order to allow manipulation of transaction IDs that are used to group together updates that are associated with the same transaction. Operators were not available for PostgreSQL 7.3 and earlier versions; in order to support version 7.3, custom functions had to be added. The = operator was marked as supporting hashing, but for that to work properly, the join operator must appear in a hash index operator class. That was not defined, and as a result, queries (like the one above) that decide to use hash joins will fail.

   This has *not* been considered a 'release-critical' bug, as Slony-I does not internally generate queries likely to use hash joins. This problem shouldn't injure Slony-I's ability to continue replicating.

   Future releases of Slony-I (*e.g.* 1.0.6, 1.1) will omit the **HASHES** indicator, so that

   Supposing you wish to repair an existing instance, so that your own queries will not run afoul of this problem, you may do so as follows:

```
/* cbbrowne@[local]/dba2 slony_test1=*/ \x
Expanded display is on.
/* cbbrowne@[local]/dba2 slony_test1=*/ select * from pg_operator where oprname = '='
and oprnamespace = (select oid from pg_namespace where nspname = 'public');
-[ RECORD 1 ]+-------------
```

```
oprname      | =
oprnamespace | 2200
oprowner     | 1
oprkind      | b
oprcanhash   | t
oprleft      | 82122344
oprright     | 82122344
oprresult    | 16
oprcom       | 82122365
oprnegate    | 82122363
oprlsortop   | 82122362
oprrsortop   | 82122362
oprltcmpop   | 82122362
oprgtcmpop   | 82122360
oprcode      | "_T1".xxideq
oprrest      | eqsel
oprjoin      | eqjoinsel

/* cbbrowne@[local]/dba2 slony_test1=*/ update pg_operator set oprcanhash = 'f' where
oprname = '=' and oprnamespace = 2200 ;
UPDATE 1
```

3. *I can do a **pg_dump** and load the data back in much faster than the **SUBSCRIBE SET** runs. Why is that?*

   Slony-I depends on there being an already existant index on the primary key, and leaves all indexes alone whilst using the PostgreSQL **COPY** command to load the data. Further hurting performance, the **COPY SET** event (an event that the subscription process generates) starts by deleting the contents of tables, which leaves the table full of dead tuples. When you use **pg_dump** to dump the contents of a database, and then load that, creation of indexes is deferred until the very end. It is *much* more efficient to create indexes against the entire table, at the end, than it is to build up the index incrementally as each row is added to the table.

   If you can drop unnecessary indices while the **COPY** takes place, that will improve performance quite a bit. If you can **TRUNCATE** tables that contain data that is about to be eliminated, that will improve performance *a lot.*

   Slony-I version 1.1.5 and later versions should handle this automatically; it 'thumps' on the indexes in the PostgreSQL catalog to hide them, in much the same way triggers are hidden, and then 'fixes' the index pointers and reindexes the table.

4. *Replication Fails - Unique Constraint ViolationReplication has been running for a while, successfully, when a node encounters a 'glitch,' and replication logs are filled with repetitions of the following:*

```
DEBUG2 remoteWorkerThread_1: syncing set 2 with 5 table(s) from provider 1
DEBUG2 remoteWorkerThread_1: syncing set 1 with 41 table(s) from provider 1
DEBUG2 remoteWorkerThread_1: syncing set 5 with 1 table(s) from provider 1
DEBUG2 remoteWorkerThread_1: syncing set 3 with 1 table(s) from provider 1
DEBUG2 remoteHelperThread_1_1: 0.135 seconds delay for first row
DEBUG2 remoteHelperThread_1_1: 0.343 seconds until close cursor
ERROR  remoteWorkerThread_1: "insert into "_oxrsapp".sl_log_1          (log_origin,  ↩
    log_xid, log_tableid,                log_actionseq, log_cmdtype,    log_cmddata)  ↩
    values   ('1', '919151224', '34', '35090538', 'D', '_rserv_ts=''9275244''');
delete from only public.epp_domain_host where _rserv_ts='9275244';insert into "_oxrsapp ↩
    ".sl_log_1   (log_origin, log_xid, log_tableid,    log_actionseq, log_cmdtype,    ↩
    log_cmddata) values   ('1', '919151224', '34', '35090539', 'D', '_rserv_ts ↩
    =''9275245''');
delete from only public.epp_domain_host where _rserv_ts='9275245';insert into "_oxrsapp ↩
    ".sl_log_1   (log_origin, log_xid, log_tableid,    log_actionseq, log_cmdtype,    ↩
    log_cmddata) values   ('1', '919151224', '26', '35090540', 'D', '_rserv_ts ↩
    =''24240590''');
delete from only public.epp_domain_contact where _rserv_ts='24240590';insert into " ↩
    _oxrsapp".sl_log_1   (log_origin, log_xid, log_tableid,    log_actionseq,  ↩
    log_cmdtype,   log_cmddata) values   ('1', '919151224', '26', '35090541', 'D', ' ↩
    _rserv_ts=''24240591''');
delete from only public.epp_domain_contact where _rserv_ts='24240591';insert into " ↩
    _oxrsapp".sl_log_1   (log_origin, log_xid, log_tableid,    log_actionseq,  ↩
```

```
    log_cmdtype,   log_cmddata) values   ('1', '919151224', '26', '35090542', 'D', ' ↩
    _rserv_ts=''24240589''');
delete from only public.epp_domain_contact where _rserv_ts='24240589';insert into " ↩
    _oxrsapp".sl_log_1   (log_origin, log_xid, log_tableid,    log_actionseq,  ↩
    log_cmdtype,   log_cmddata) values   ('1', '919151224', '11', '35090543', 'D', ' ↩
    _rserv_ts=''36968002''');
delete from only public.epp_domain_status where _rserv_ts='36968002';insert into " ↩
    _oxrsapp".sl_log_1    (log_origin, log_xid, log_tableid,    log_actionseq,  ↩
    log_cmdtype,   log_cmddata) values   ('1', '919151224', '11', '35090544', 'D', ' ↩
    _rserv_ts=''36968003''');
delete from only public.epp_domain_status where _rserv_ts='36968003';insert into " ↩
    _oxrsapp".sl_log_1    (log_origin, log_xid, log_tableid,    log_actionseq,  ↩
    log_cmdtype,   log_cmddata) values   ('1', '919151224', '24', '35090549', 'I', '( ↩
    contact_id,status,reason,_rserv_ts) values (''6972897'',''64'','''',''31044208'')') ↩
    ;
insert into public.contact_status (contact_id,status,reason,_rserv_ts) values  ↩
    ('6972897','64','','31044208');insert into "_oxrsapp".sl_log_1    (log_origin,  ↩
    log_xid, log_tableid,    log_actionseq, log_cmdtype,   log_cmddata) values   ('1',  ↩
    '919151224', '24', '35090550', 'D', '_rserv_ts=''18139332''');
delete from only public.contact_status where _rserv_ts='18139332';insert into "_oxrsapp ↩
    ".sl_log_1   (log_origin, log_xid, log_tableid,    log_actionseq, log_cmdtype,    ↩
    log_cmddata) values   ('1', '919151224', '24', '35090551', 'D', '_rserv_ts ↩
    =''18139333''');
delete from only public.contact_status where _rserv_ts='18139333';" ERROR:  duplicate  ↩
    key violates unique constraint "contact_status_pkey"
 - qualification was:
ERROR  remoteWorkerThread_1: SYNC aborted
```

The transaction rolls back, and Slony-I tries again, and again, and again. The problem is with one of the last *SQL statements, the one with* ***log_cmdtype = 'I'****. That isn't quite obvious; what takes place is that Slony-I groups 10 update queries together to diminish the number of network round trips.*

A *certain* cause for this has been difficult to arrive at.By the time we notice that there is a problem, the seemingly missed delete transaction has been cleaned out of sl_log_1, so there appears to be no recovery possible. What has seemed necessary, at this point, is to drop the replication set (or even the node), and restart replication from scratch on that node.In Slony-I 1.0.5, the handling of purges of sl_log_1 became more conservative, refusing to purge entries that haven't been successfully synced for at least 10 minutes on all nodes. It was not certain that that would prevent the 'glitch' from taking place, but it seemed plausible that it might leave enough sl_log_1 data to be able to do something about recovering from the condition or at least diagnosing it more exactly. And perhaps the problem was that sl_log_1 was being purged too aggressively, and this would resolve the issue completely. It is a shame to have to reconstruct a large replication node for this; if you discover that this problem recurs, it may be an idea to break replication down into multiple sets in order to diminish the work involved in restarting replication. If only one set has broken, you may only need to unsubscribe/drop and resubscribe the one set. In one case we found two lines in the SQL error message in the log file that contained *identical* insertions into sl_log_1. This *ought* to be impossible as is a primary key on sl_log_1. The latest (somewhat) punctured theory that comes from *that* was that perhaps this PK index has been corrupted (representing a PostgreSQL bug), and that perhaps the problem might be alleviated by running the query:

```
# reindex table _slonyschema.sl_log_1;
```

On at least one occasion, this has resolved the problem, so it is worth trying this.

This problem has been found to represent a PostgreSQL bug as opposed to one in Slony-I. Version 7.4.8 was released with two resolutions to race conditions that should resolve the issue. Thus, if you are running a version of PostgreSQL earlier than 7.4.8, you should consider upgrading to resolve this.

5. *I started doing a backup using pg_dump, and suddenly Slony stops*

   Ouch. What happens here is a conflict between:

   - pg_dump, which has taken out an **AccessShareLock** on all of the tables in the database, including the Slony-I ones, and
   - A Slony-I sync event, which wants to grab a **AccessExclusiveLock** on the table sl_event.

The initial query that will be blocked is thus:

```
select "_slonyschema".createEvent('_slonyschema, 'SYNC', NULL);
```

(You can see this in pg_stat_activity, if you have query display turned on in postgresql.conf)The actual query combination that is causing the lock is from the function Slony_I_ClusterStatus(), found in slony1_funcs.c, and is localized in the code that does:

```
  LOCK TABLE %s.sl_event;
  INSERT INTO %s.sl_event (...stuff...)
  SELECT currval('%s.sl_event_seq');
```

The **LOCK** statement will sit there and wait until **pg_dump** (or whatever else has pretty much any kind of access lock on sl_event) completes.Every subsequent query submitted that touches sl_event will block behind the createEvent call.There are a number of possible answers to this:

- Have pg_dump specify the schema dumped using --schema=whatever, and don't try dumping the cluster's schema.
- It would be nice to add an --exclude-schema option to pg_dump to exclude the Slony-I cluster schema. Maybe in 8.2...
- Note that 1.0.5 uses a more precise lock that is less exclusive that alleviates this problem.

1. *I was trying to request SLONIK EXECUTE SCRIPT(7) or SLONIK MOVE SET(7), and found messages as follows on one of the subscribers:*

```
NOTICE: Slony-I: multiple instances of trigger defrazzle on table frobozz
NOTICE: Slony-I: multiple instances of trigger derez on table tron
ERROR: Slony-I: Unable to disable triggers
```

The trouble would seem to be that you have added triggers on tables whose names conflict with triggers that were hidden by Slony-I. Slony-I hides triggers (save for those 'unhidden' via SLONIK STORE TRIGGER(7)) by repointing them to the primary key of the table. In the case of foreign key triggers, or other triggers used to do data validation, it should be quite unnecessary to run them on a subscriber, as equivalent triggers should have been invoked on the origin node. In contrast, triggers that do some form of 'cache invalidation' are ones you might want to have run on a subscriber. The *Right Way* to handle such triggers is normally to use SLONIK STORE TRIGGER(7), which tells Slony-I that a trigger should not get deactivated.

But some intrepid DBA might take matters into their own hands and install a trigger by hand on a subscriber, and the above condition generally has that as the cause. What to do? What to do? The answer is normally fairly simple: Drop out the 'extra' trigger on the subscriber before the event that tries to restore them runs. Ideally, if the DBA is particularly intrepid, and aware of this issue, that should take place *before* there is ever a chance for the error message to appear. If the DBA is not that intrepid, the answer is to connect to the offending node and drop the 'visible' version of the trigger using the SQL **DROP TRIGGER** command. That should allow the event to proceed. If the event was SLONIK EXECUTE SCRIPT(7), then the 'not-so-intrepid' DBA may need to add the trigger back, by hand, or, if they are wise, they should consider activating it using SLONIK STORE TRIGGER(7).

2. *Behaviour - all the subscriber nodes start to fall behind the origin, and all the logs on the subscriber nodes have the following error message repeating in them (when I encountered it, there was a nice long SQL statement above each entry):*

```
ERROR remoteWorkerThread_1: helper 1 finished with error
ERROR remoteWorkerThread_1: SYNC aborted
```

Cause: you have likely issued **alter table** statements directly on the databases instead of using the slonik SLONIK EXECUTE SCRIPT(7) command.The solution is to rebuild the trigger on the affected table and fix the entries in sl_log_1/sl_log_2 by hand.

- You'll need to identify from either the slon logs, or the PostgreSQL database logs exactly which statement it is that is causing the error.
- You need to fix the Slony-defined triggers on the table in question. This is done with the following procedure.

```
BEGIN;
LOCK TABLE table_name;
SELECT _oxrsorg.altertablerestore(tab_id);--tab_id is _slony_schema.sl_table.tab_id
SELECT _oxrsorg.altertableforreplication(tab_id);--tab_id is _slony_schema.sl_table. ↩
    tab_id
COMMIT;
```

You then need to find the rows in sl_log_1/sl_log_2 that have bad entries and fix them. You may want to take down the slon daemons for all nodes except the master; that way, if you make a mistake, it won't immediately propagate through to the subscribers.

Here is an example:

```
BEGIN;

LOCK TABLE customer_account;

SELECT _app1.altertablerestore(31);
SELECT _app1.altertableforreplication(31);
COMMIT;

BEGIN;
LOCK TABLE txn_log;

SELECT _app1.altertablerestore(41);
SELECT _app1.altertableforreplication(41);

COMMIT;

--fixing customer_account, which had an attempt to insert a "" into a timestamp with  ↩
    timezone.
BEGIN;

update _app1.sl_log_1 SET log_cmddata = 'balance=''60684.00'' where pkey=''49'''  ↩
    where log_actionseq = '67796036';
update _app1.sl_log_1 SET log_cmddata = 'balance=''60690.00'' where pkey=''49'''  ↩
    where log_actionseq = '67796194';
update _app1.sl_log_1 SET log_cmddata = 'balance=''60684.00'' where pkey=''49'''  ↩
    where log_actionseq = '67795881';
update _app1.sl_log_1 SET log_cmddata = 'balance=''1852.00'' where pkey=''57''' where ↩
     log_actionseq = '67796403';
update _app1.sl_log_1 SET log_cmddata = 'balance=''87906.00'' where pkey=''8''' where ↩
     log_actionseq = '68352967';
update _app1.sl_log_1 SET log_cmddata = 'balance=''125180.00'' where pkey=''60'''  ↩
    where log_actionseq = '68386951';
update _app1.sl_log_1 SET log_cmddata = 'balance=''125198.00'' where pkey=''60'''  ↩
    where log_actionseq = '68387055';
update _app1.sl_log_1 SET log_cmddata = 'balance=''125174.00'' where pkey=''60'''  ↩
    where log_actionseq = '68386682';
update _app1.sl_log_1 SET log_cmddata = 'balance=''125186.00'' where pkey=''60'''  ↩
    where log_actionseq = '68386992';
update _app1.sl_log_1 SET log_cmddata = 'balance=''125192.00'' where pkey=''60'''  ↩
    where log_actionseq = '68387029';
```

3. *Node #1 was dropped via SLONIK DROP NODE(7), and the slon(1) one of the other nodes is repeatedly failing with the error message:*

```
ERROR  remoteWorkerThread_3: "begin transaction; set transaction isolation level
 serializable; lock table "_mailermailer".sl_config_lock; select "_mailermailer"
.storeListen_int(2, 1, 3); notify "_mailermailer_Event"; notify "_mailermailer_C
onfirm"; insert into "_mailermailer".sl_event     (ev_origin, ev_seqno, ev_times
tamp,      ev_minxid, ev_maxxid, ev_xip, ev_type , ev_data1, ev_data2, ev_data3
```

```
   ) values ('3', '2215', '2005-02-18 10:30:42.529048', '3286814', '3286815', ''
, 'STORE_LISTEN', '2', '1', '3'); insert into "_mailermailer".sl_confirm
(con_origin, con_received, con_seqno, con_timestamp)    values (3, 2, '2215', CU
RRENT_TIMESTAMP); commit transaction;" PGRES_FATAL_ERROR ERROR:  insert or updat
e on table "sl_listen" violates foreign key constraint "sl_listen-sl_path-ref"
DETAIL:  Key (li_provider,li_receiver)=(1,3) is not present in table "sl_path".
DEBUG1 syncThread: thread done
```

Evidently, a SLONIK STORE LISTEN(7) request hadn't propagated yet before node 1 was dropped.

This points to a case where you'll need to do 'event surgery' on one or more of the nodes. A **STORE_LISTEN** event remains outstanding that wants to add a listen path that *cannot* be created because node 1 and all paths pointing to node 1 have gone away. Let's assume, for exposition purposes, that the remaining nodes are #2 and #3, and that the above error is being reported on node #3. That implies that the event is stored on node #2, as it wouldn't be on node #3 if it had not already been processed successfully. The easiest way to cope with this situation is to delete the offending sl_event entry on node #2. You'll connect to node #2's database, and search for the **STORE_LISTEN** event: **select * from sl_event where ev_type = 'STORE_LISTEN';** There may be several entries, only some of which need to be purged.

```
-# begin;  -- Don't straight delete them; open a transaction so you can respond to OOPS
BEGIN;
-# delete from sl_event where ev_type = 'STORE_LISTEN' and
-#  (ev_data1 = '1' or ev_data2 = '1' or ev_data3 = '1');
DELETE 3
-# -- Seems OK...
-# commit;
COMMIT
```

The next time the slon for node 3 starts up, it will no longer find the 'offensive' **STORE_LISTEN** events, and replication can continue. (You may then run into some other problem where an old stored event is referring to no-longer-existant configuration...)

4. *I have a database where we have been encountering the following error message in our application:*

   ```
   permission denied for sequence sl_action_seq
   ```

   When we traced it back, it was due to the application calling `lastval()` *to capture the most recent sequence update, which happened to catch the last update to a Slony-I internal sequence.*

   Slony-I uses sequences to provide primary key values for log entries, and therefore this kind of behaviour may (perhaps regrettably!) be expected. Calling `lastval()`, to 'anonymously' get 'the most recently updated sequence value', rather than using `currval('sequence_name')` is an unsafe thing to do in general, as anything you might add in that uses DBMS features for logging, archiving, or replication can throw in an extra sequence update that you weren't expecting. In general, use of `lastval()` doesn't seem terribly safe; using it when Slony-I (or any similar trigger-based replication system such as Londiste or Bucardo) can lead to capturing unexpected sequence updates.

12.2 Release Checklist

Here are things that should be done whenever a release is prepared:

- Positive build reports for each supported platform - although it is arguably less necessary for a comprehensive list if we are releasing a minor upgrade
- Some kind of Standard Test Plan
- If the release modified the set of version-specific SQL files in `src/backend` (*e.g.* - it added a new `slony1_base.v83.sql` or `slony1_funcs.v83.sql`), or if we have other changes to the shape of PostgreSQL version support, the function `load_slony_functions()` in `src/slonik/slonik.c` needs to be revised to reflect the new shape of things.

- The new release needs to be added to function `upgradeSchema(text)` in `src/backend/slony1_funcs.sql`.

  This takes place in a 'cross-branch' fashion; if we add version 1.1.9, in the 1.1 branch, then version 1.1.9 needs to be added to the 1.2 branch as well as to later branches (*e.g.* - 1.3, 1.4, HEAD). Earlier branches will normally not need to be made aware of versions added to later branches.

  This was not true for version 2 - version 2.0 was different enough from earlier versions that we rejected having a direct upgrade from 1.x to 2.0, so there are *no* versions in 1.x branches in `upgradeSchema(text)` for Slony-I version 2.0.

- Binary RPM packages

- If the release is a '.0' one, we need to open a new STABLE branch

  git checkout HEAD

  git checkout -b REL_3_0_STABLE

- Tag with the release ID. For version 1.1.2, this would be `REL_1_1_2`

  git tag -a REL_1_1_2

- Check out an exported copy via: **git archive REL_1_1_2 -o /tmp/slony1-engine-1.0.2.tar**

- Run autoconf so as to regenerate `configure` from `configure.ac`

- Purge directory `autom4te.cache` so it is not included in the build

  Does not need to be done by hand - the later **make distclean** step does this for you.

- Run `tools/release_checklist.sh`

  This does a bunch of consistency checks to make sure that various files that are supposed to contain version numbers contain consistent values.

- For instance, configure should contain, for release 1.1.2:

- PACKAGE_VERSION=REL_1_1_2

- PACKAGE_STRING=slony1 REL_1_1_2

- `config.h.in` needs to contain the version number in two forms; the definitions for `SLONY_I_VERSION_STRING` and `SLONY_I_VERSION_STRING_DEC` both need to be updated.

- `src/backend/slony1_funcs.sql` has major/minor/patch versions in functions `slonyVersionMajor()`, `slonyVersionMinor()`, and `slonyVersionPatchlevel()`. These need to be assigned 'by hand' at this point.

- It sure would be nice if more of these could be assigned automatically, somehow.

  Don't commit the new `configure`; we shouldn't be tracking this in Git.

- make sure that the generated files from .l and .y are created, for example `slony/conf-file.[ch]`

  Currently this is best done by issuing **./configure && make all && make clean** but that is a somewhat ugly approach.

  Slightly better may be **./configure && make src/slon/conf-file.c src/slonik/parser.c src/slonik/scan.c**

- Make sure that generated Makefiles and such from the previous step(s) are removed.

  make distclean will do that...

  Note that **make distclean** also clears out `autom4te.cache`, thus obsoleting some former steps that suggested that it was needful to delete them.

- Generate HTML tarball, and RTF/PDF, if possible... Note that the HTML version should include `*.html` (duh!), `*.jpg`, `*.png`, and `*.css`

  Note that, if starting from a 'clean' copy of the documentation, in order to properly build the HTML tarball, it is necessary to run **make html** *twice*, in order for the document index to be properly constructed.

  – The first time **make html** is run, the file `HTML.index` does not yet exist.
    When jade is run, against the document, a side-effect is to generate `HTML.index`, extracting all index tags from the Slony-I documentation.

  - The second time **make html** is run, `HTML.index` is transformed into `bookindex.sgml`, which jade may then use to generate a nice index page indicating all the index entries included in the documentation tree.

- Run **make clean** in `doc/adminguide` before generating the tarball in order that `bookindex.sgml` is NOT part of the tarball
- Alternatively, delete `doc/adminguide/bookindex.sgml`
- Rename the directory (*e.g.* - `slony1-engine`) to a name based on the release, *e.g.* - `slony1-1.1.2`
- Generate a tarball - **tar tfvj slony1-1.1.2.tar.bz2 slony1-1.1.2**
- Build the administrative documentation, and build a tarball as `slony1-1.1.2-html.tar.bz2`

  Make sure that the docs are inside a subdir in the tarball.
- Put these tarballs in a temporary area, and notify everyone that they should test them out ASAP based on the Standard Test Plan.

12.3 Using Slonik

It's a bit of a pain writing Slonik scripts by hand, particularly as you start working with Slony-I clusters that may be comprised of increasing numbers of nodes and sets. Some problems that have been noticed include the following:

- If you are using Slony-I as a 'master/slave' replication system with one 'master' node and one 'slave' node, it may be sufficiently mnemonic to call the 'master' node 1 and the 'slave' node 2.

  Unfortunately, as the number of nodes increases, the mapping of IDs to nodes becomes way less obvious, particularly if you have a cluster where the origin might shift from node to node over time.
- Similarly, if there is only one replication set, it's fine for that to be 'set 1,' but if there are a multiplicity of sets, the numbering involved in using set numbers may grow decreasingly intuitive.
- People have observed that Slonik does not provide any notion of iteration. It is common to want to create a set of similar SLONIK STORE PATH(7) entries, since, in most cases, hosts will likely access a particular server via the same host name or IP address.
- Users seem interested in wrapping everything possible in **TRY** blocks, which is regrettably somewhat *less* useful than might be hoped...

These have assortedly pointed to requests for such enhancements as:

- Named nodes, named sets

  This is supported in Slony-I 1.1 by the SLONIK DEFINE(7) and SLONIK INCLUDE(7) statements.

  The use of SLONIK INCLUDE(7) to allow creating 'preamble files' has proven an invaluable tool to reduce errors. The preamble file is set up *once*, verified *once*, and then that verified script may be used with confidence for each slonik script.
- Looping and control constructs

  It seems to make little sense to create a fullscale parser as Yet Another Little Language grows into a rather larger one. There are plenty of scripting languages out there that can be used to construct Slonik scripts; it is unattractive to force yet another one on people.

There are several ways to work around these issues that have been seen 'in the wild':

- Embedding generation of slonik inside shell scripts

  The test bed found in the `src/ducttape` directory takes this approach.
- The altperl tools use Perl code to generate Slonik scripts.

  You define the cluster's configuration as a set of Perl objects; each script walks through the Perl objects as needed to generate a slonik script for that script's purpose.

12.4 Embedding Slonik in Shell Scripts

As mentioned earlier, there are numerous Slony-I test scripts in `src/ducttape` that embed the generation of Slonik inside the shell script.

They mostly *don't* do this in a terribly sophisticated way. Typically, they use the following sort of structure:

```
DB1=slony_test1
DB2=slony_test2
slonik <<_EOF_
  cluster name = T1;
  node 1 admin conninfo = 'dbname=$DB1';
  node 2 admin conninfo = 'dbname=$DB2';

  try {
    create set (id = 1, origin = 1, comment =
                            'Set 1 - pgbench tables');
    set add table (set id = 1, origin = 1,
      id = 1, fully qualified name = 'public.accounts',
      comment = 'Table accounts');
    set add table (set id = 1, origin = 1,
      id = 2, fully qualified name = 'public.branches',
      comment = 'Table branches');
    set add table (set id = 1, origin = 1,
      id = 3, fully qualified name = 'public.tellers',
      comment = 'Table tellers');
    set add table (set id = 1, origin = 1,
      id = 4, fully qualified name = 'public.history',
      comment = 'Table accounts');
  }
  on error {
    exit 1;
  }
_EOF_
```

A more sophisticated approach might involve defining some common components, notably the 'preamble' that consists of the SLONIK CLUSTER NAME(7) SLONIK ADMIN CONNINFO(7) commands that are common to every Slonik script, thus:

```
CLUSTER=T1
DB1=slony_test1
DB2=slony_test2
PREAMBLE="cluster name = $CLUSTER
node 1 admin conninfo = 'dbname=$DB1';
node 2 admin conninfo = 'dbname=$DB2';
"
```

The `PREAMBLE` value could then be reused over and over again if the shell script invokes **slonik** multiple times. You might also consider using SLONIK INCLUDE(7) and place the preamble in a file that is **include**d.

Shell variables provide a simple way to assign names to sets and nodes:

```
origin=1
subscriber=2
mainset=1
slonik <<_EOF_
$PREAMBLE
try {
  create set (id = $mainset, origin = $origin,
                    comment = 'Set $mainset - pgbench tables');
  set add table (set id = $mainset, origin = $origin,
    id = 1, fully qualified name = 'public.accounts',
    comment = 'Table accounts');
```

```
  set add table (set id = $mainset, origin = $origin,
    id = 2, fully qualified name = 'public.branches',
    comment = 'Table branches');
  set add table (set id = $mainset, origin = $origin,
    id = 3, fully qualified name = 'public.tellers',
    comment = 'Table tellers');
  set add table (set id = $mainset, origin = $origin,
    id = 4, fully qualified name = 'public.history',
    comment = 'Table accounts');
} on error {
  exit 1;
}
_EOF_
```

The script might be further enhanced to loop through the list of tables as follows:

```
# Basic configuration
origin=1
subscriber=2
mainset=1
# List of tables to replicate
TABLES="accounts branches tellers history"
ADDTABLES=""
tnum=1
for table in `echo $TABLES`; do
  ADDTABLES="$ADDTABLES
   set add table ($set id = $mainset, origin = $origin,
   id = $tnum, fully qualified name = 'public.$table',
   comment = 'Table $tname');
"
  let "tnum=tnum+1"
done
slonik <<_EOF_
$PREAMBLE
try {
  create set (id = $mainset, origin = $origin,
                    comment = 'Set $mainset - pgbench tables');
$ADDTABLES
} on error {
  exit 1;
}
_EOF_
```

That is of somewhat dubious value if you only have 4 tables, but eliminating errors resulting from enumerating large lists of configuration by hand will make this pretty valuable for the larger examples you'll find in 'real life.'

You can do even more sophisticated things than this if your scripting language supports things like:

- 'Record' data structures that allow assigning things in parallel
- Functions, procedures, or subroutines, allowing you to implement useful functionality once, and then refer to it multiple times within a script
- Some sort of 'module import' system so that common functionality can be shared across many scripts

If you can depend on having Bash, zsh, or Korn shell available, well, those are all shells with extensions supporting reasonably sophisticated data structures and module systems. On Linux, Bash is fairly ubiquitous; on commercial UNIX™, Korn shell is fairly ubiquitous; on BSD, 'sophisticated' shells are an option rather than a default.

At that point, it makes sense to start looking at other scripting languages, of which Perl is the most ubiquitous, being widely available on Linux, UNIX™, and BSD.

12.5 More Slony-I Help

If you are having problems with Slony-I, you have several options for help:

12.5.1 Slony-I Website

http://slony.info/ - the official 'home' of Slony-I contains links to the documentation, mailing lists and source code.

12.5.2 Mailing Lists

The answer to your problem may exist in the Slony1-general mailing list archives, or you may choose to ask your question on the Slony1-general mailing list. The mailing list archives, and instructions for joining the list may be found here. .

If you are a question to the mailing list then you should try to include the following information:

- The version of Slony-I you are using
- The version of PostgreSQL you are using
- A description of your replication cluster. This should include the number of replication sets, which node is the master for each set.
- The text of any error messages you are receiving from slony
- If you received the error while running a slonik script then try to include the script

It is a good idea to run the Section 5.1.1 tool before posting a question to the mailing list. It may give some clues as to what is wrong, and the results are likely to be of some assistance in analyzing the problem.

12.5.3 Other Sources

- There are several articles here Varlena GeneralBits that may be helpful but was written for an older version of Slony-I.
- IRC - There are usually some people on #slony on irc.freenode.net who may be able to answer some of your questions. Many people leave themselves logged into IRC and only periodically check the channel. It might take a while before someone answers your questions
- pgpool

  pgpool is a connection pool server for PostgreSQL; it allows an application to connect to it as if it were a standard PostgreSQL server. It caches connections, which reduces the overhead involved in establishing them. It supports a 'scheduled switchover' feature, which would allow dynamically switching over from one server to another. That would be very useful when doing a SLONIK MOVE SET(7), as it would allow applications to be switched to point to the new origin without needing to update their configuration.
- Slony1-ctl - Another set of administration scripts for slony

Chapter 13

Schema schemadoc

13.1 Table: sl_archive_counter

Table used to generate the log shipping archive number.

STRUCTURE OF SL_ARCHIVE_COUNTER

ac_num bigint
: Counter of SYNC ID used in log shipping as the archive number

ac_timestamp timestamp with time zone
: Time at which the archive log was generated on the subscriber

13.2 Table: sl_components

Table used to monitor what various slon/slonik components are doing

STRUCTURE OF SL_COMPONENTS

co_actor text `PRIMARY KEY`
: which component am I?

co_pid integer `NOT NULL`
: my process/thread PID on node where slon runs

co_node integer `NOT NULL`
: which node am I servicing?

co_connection_pid integer `NOT NULL`
: PID of database connection being used on database server

co_activity text
: activity that I am up to

co_starttime timestamp with time zone `NOT NULL`
: when did my activity begin? (timestamp reported as per slon process on server running slon)

co_event bigint
: which event have I started processing?

co_eventtype text
: what kind of event am I processing? (commonly n/a for event loop main threads)

13.3 Table: sl_config_lock

This table exists solely to prevent overlapping execution of configuration change procedures and the resulting possible deadlocks.

STRUCTURE OF SL_CONFIG_LOCK

dummy integer

No data ever goes in this table so the contents never matter. Indeed, this column does not really need to exist.

13.4 Table: sl_confirm

Holds confirmation of replication events. After a period of time, Slony removes old confirmed events from both this table and the sl_event table.

STRUCTURE OF SL_CONFIRM

con_origin integer

The ID # (from sl_node.no_id) of the source node for this event

con_received integer

con_seqno bigint

The ID # for the event

con_timestamp timestamp with time zone `DEFAULT (timeofday())::timestamp with time zone`

When this event was confirmed

INDEXES ON SL_CONFIRM

sl_confirm_idx1 con_origin, con_received, con_seqno

sl_confirm_idx2 con_received, con_seqno

13.5 Table: sl_event

Holds information about replication events. After a period of time, Slony removes old confirmed events from both this table and the sl_confirm table.

STRUCTURE OF SL_EVENT

ev_origin integer `PRIMARY KEY`

The ID # (from sl_node.no_id) of the source node for this event

ev_seqno bigint `PRIMARY KEY`

The ID # for the event

ev_timestamp timestamp with time zone

When this event record was created

ev_snapshot txid_snapshot

TXID snapshot on provider node for this event

ev_type text

The type of event this record is for. SYNC = Synchronise STORE_NODE = ENABLE_NODE = DROP_NODE = STORE_PATH = DROP_PATH = STORE_LISTEN = DROP_LISTEN = STORE_SET = DROP_SET = MERGE_SET = SET_ADD_TABLE = SET_ADD_SEQUENCE = STORE_TRIGGER = DROP_TRIGGER = MOVE_SET = ACCEPT_SET = SET_DROP_TABLE = SET_DROP_SEQUENCE = SET_MOVE_TABLE = SET_MOVE_SEQUENCE = FAILOVER_SET = SUBSCRIBE_SET = ENABLE_SUBSCRIPTION = UNSUBSCRIBE_SET = DDL_SCRIPT = ADJUST_SEQ = RESET_CONFIG =

ev_data1 text

Data field containing an argument needed to process the event

ev_data2 text

Data field containing an argument needed to process the event

ev_data3 text

Data field containing an argument needed to process the event

ev_data4 text

Data field containing an argument needed to process the event

ev_data5 text

Data field containing an argument needed to process the event

ev_data6 text

Data field containing an argument needed to process the event

ev_data7 text

Data field containing an argument needed to process the event

ev_data8 text

Data field containing an argument needed to process the event

13.6 Table: sl_event_lock

This table exists solely to prevent multiple connections from concurrently creating new events and perhaps getting them out of order.

STRUCTURE OF SL_EVENT_LOCK

dummy integer

No data ever goes in this table so the contents never matter. Indeed, this column does not really need to exist.

13.7 Table: sl_listen

Indicates how nodes listen to events from other nodes in the Slony-I network.

STRUCTURE OF SL_LISTEN

li_origin integer PRIMARY KEY REFERENCES sl_node

The ID # (from sl_node.no_id) of the node this listener is operating on

li_provider integer PRIMARY KEY REFERENCES sl_path

The ID # (from sl_node.no_id) of the source node for this listening event

li_receiver integer PRIMARY KEY REFERENCES sl_path

The ID # (from sl_node.no_id) of the target node for this listening event

13.8 Table: sl_log_1

Stores each change to be propagated to subscriber nodes

STRUCTURE OF SL_LOG_1

log_origin integer
: Origin node from which the change came

log_txid bigint
: Transaction ID on the origin node

log_tableid integer
: The table ID (from sl_table.tab_id) that this log entry is to affect

log_actionseq bigint

log_cmdtype character(1)
: Replication action to take. U = Update, I = Insert, D = DELETE

log_cmddata text
: The data needed to perform the log action

INDEXES ON SL_LOG_1

sl_log_1_idx1 log_origin, log_txid, log_actionseq

13.9 Table: sl_log_2

Stores each change to be propagated to subscriber nodes

STRUCTURE OF SL_LOG_2

log_origin integer
: Origin node from which the change came

log_txid bigint
: Transaction ID on the origin node

log_tableid integer
: The table ID (from sl_table.tab_id) that this log entry is to affect

log_actionseq bigint

log_cmdtype character(1)
: Replication action to take. U = Update, I = Insert, D = DELETE

log_cmddata text
: The data needed to perform the log action

INDEXES ON SL_LOG_2

sl_log_2_idx1 log_origin, log_txid, log_actionseq

13.10 Table: sl_node

Holds the list of nodes associated with this namespace.

STRUCTURE OF SL_NODE

no_id integer PRIMARY KEY
: The unique ID number for the node

no_active boolean
: Is the node active in replication yet?

no_comment text
: A human-oriented description of the node

TABLES REFERENCING SL_LISTEN VIA FOREIGN KEY CONSTRAINTS

- sl_listen
- sl_path
- sl_set
- sl_setsync

13.11 Table: sl_nodelock

Used to prevent multiple slon instances and to identify the backends to kill in terminateNodeConnections().

STRUCTURE OF SL_NODELOCK

nl_nodeid integer PRIMARY KEY
: Clients node_id

nl_conncnt serial PRIMARY KEY
: Clients connection number

nl_backendpid integer
: PID of database backend owning this lock

13.12 Table: sl_path

Holds connection information for the paths between nodes, and the synchronisation delay

STRUCTURE OF SL_PATH

pa_server integer PRIMARY KEY REFERENCES sl_node
: The Node ID # (from sl_node.no_id) of the data source

pa_client integer PRIMARY KEY REFERENCES sl_node
: The Node ID # (from sl_node.no_id) of the data target

pa_conninfo text NOT NULL
: The PostgreSQL connection string used to connect to the source node.

pa_connretry integer

The synchronisation delay, in seconds

TABLES REFERENCING SL_LISTEN VIA FOREIGN KEY CONSTRAINTS

- sl_listen
- sl_subscribe

13.13 Table: sl_registry

Stores miscellaneous runtime data

STRUCTURE OF SL_REGISTRY

reg_key text `PRIMARY KEY`

Unique key of the runtime option

reg_int4 integer

Option value if type int4

reg_text text

Option value if type text

reg_timestamp timestamp with time zone

Option value if type timestamp

13.14 View: sl_seqlastvalue

STRUCTURE OF SL_SEQLASTVALUE

seq_id integer

seq_set integer

seq_reloid oid

seq_origin integer

seq_last_value bigint

```
SELECT sq.seq_id
, sq.seq_set
, sq.seq_reloid
, s.set_origin AS seq_origin
, sequencelastvalue
(
     (
           (quote_ident
                 (
                       (pgn.nspname)::text
                 ) || [apos  ].[apos  ]::text
           ) || quote_ident
           (
                 (pgc.relname)::text
           )
     )
) AS seq_last_value
FROM sl_sequence sq
, sl_set s
, pg_class pgc
, pg_namespace pgn
WHERE (
     (
           (s.set_id = sq.seq_set)
         AND (pgc.oid = sq.seq_reloid)
     )
   AND (pgn.oid = pgc.relnamespace)
);
```

Figure 13.1: Definition of view sl_seqlastvalue

13.15 Table: sl_seqlog

Log of Sequence updates

STRUCTURE OF SL_SEQLOG

seql_seqid integer

Sequence ID

seql_origin integer

Publisher node at which the sequence originates

seql_ev_seqno bigint

Slony-I Event with which this sequence update is associated

seql_last_value bigint

Last value published for this sequence

INDEXES ON SL_SEQLOG

sl_seqlog_idx seql_origin, seql_ev_seqno, seql_seqid

13.16 Table: sl_sequence

Similar to sl_table, each entry identifies a sequence being replicated.

STRUCTURE OF SL_SEQUENCE

seq_id integer `PRIMARY KEY`

An internally-used ID for Slony-I to use in its sequencing of updates

seq_reloid oid `UNIQUE NOT NULL`

The OID of the sequence object

seq_relname name `NOT NULL`

The name of the sequence in pg_catalog.pg_class.relname used to recover from a dump/restore cycle

seq_nspname name `NOT NULL`

The name of the schema in pg_catalog.pg_namespace.nspname used to recover from a dump/restore cycle

seq_set integer `REFERENCES` sl_set

Indicates which replication set the object is in

seq_comment text

A human-oriented comment

13.17 Table: sl_set

Holds definitions of replication sets.

STRUCTURE OF SL_SET

set_id integer `PRIMARY KEY`

A unique ID number for the set.

set_origin integer `REFERENCES` sl_node

The ID number of the source node for the replication set.

set_locked bigint

Transaction ID where the set was locked.

set_comment text

A human-oriented description of the set.

TABLES REFERENCING SL_SEQUENCE VIA FOREIGN KEY CONSTRAINTS

- sl_sequence
- sl_setsync
- sl_subscribe
- sl_table

13.18 Table: sl_setsync

SYNC information

STRUCTURE OF SL_SETSYNC

ssy_setid integer PRIMARY KEY REFERENCES sl_set

ID number of the replication set

ssy_origin integer REFERENCES sl_node

ID number of the node

ssy_seqno bigint

Slony-I sequence number

ssy_snapshot txid_snapshot

TXID in provider system seen by the event

ssy_action_list text

action list used during the subscription process. At the time a subscriber copies over data from the origin, it sees all tables in a state somewhere between two SYNC events. Therefore this list must contains all log_actionseqs that are visible at that time, whose operations have therefore already been included in the data copied at the time the initial data copy is done. Those actions may therefore be filtered out of the first SYNC done after subscribing.

13.19 Table: sl_subscribe

Holds a list of subscriptions on sets

STRUCTURE OF SL_SUBSCRIBE

sub_set integer PRIMARY KEY REFERENCES sl_set

ID # (from sl_set) of the set being subscribed to

sub_provider integer REFERENCES sl_path

ID# (from sl_node) of the node providing data

sub_receiver integer PRIMARY KEY REFERENCES sl_path

ID# (from sl_node) of the node receiving data from the provider

sub_forward boolean

Does this provider keep data in sl_log_1/sl_log_2 to allow it to be a provider for other nodes?

sub_active boolean

Has this subscription been activated? This is not set on the subscriber until AFTER the subscriber has received COPY data from the provider

13.20 Table: sl_table

Holds information about the tables being replicated.

STRUCTURE OF SL_TABLE

tab_id integer PRIMARY KEY

Unique key for Slony-I to use to identify the table

tab_reloid oid UNIQUE NOT NULL

The OID of the table in pg_catalog.pg_class.oid

tab_relname name NOT NULL

The name of the table in pg_catalog.pg_class.relname used to recover from a dump/restore cycle

tab_nspname name NOT NULL

The name of the schema in pg_catalog.pg_namespace.nspname used to recover from a dump/restore cycle

tab_set integer REFERENCES sl_set

ID of the replication set the table is in

tab_idxname name NOT NULL

The name of the primary index of the table

tab_altered boolean NOT NULL

Has the table been modified for replication?

tab_comment text

Human-oriented description of the table

13.21 add_empty_table_to_replication(p_comment integer, p_idxname integer, p_tabna text, p_nspname text, p_tab_id text, p_set_id text)

Function Properties

Language: PLPGSQL, *Return Type:* bigint Verify that a table is empty, and add it to replication. tab_idxname is optional - if NULL, then we use the primary key. Note that this function is to be run within an EXECUTE SCRIPT script, so it runs at the right place in the transaction stream on all nodes.

```
declare

  prec record;
  v_origin int4;
  v_isorigin boolean;
  v_fqname text;
  v_query text;
  v_rows integer;
  v_idxname text;

begin
-- Need to validate that the set exists; the set will tell us if this is the origin
  select set_origin into v_origin from sl_set where set_id = p_set_id;
  if not found then
  raise exception 'add_empty_table_to_replication: set % not found!', p_set_id;
  end if;

-- Need to be aware of whether or not this node is origin for the set
   v_isorigin := ( v_origin = getLocalNodeId('_schemadoc') );

   v_fqname := '"' || p_nspname || '"."' || p_tabname || '"';
-- Take out a lock on the table
   v_query := 'lock ' || v_fqname || ';';
   execute v_query;

   if v_isorigin then
  -- On the origin, verify that the table is empty, failing if it has any tuples
       v_query := 'select 1 as tuple from ' || v_fqname || ' limit 1;';
```

```
  execute v_query into prec;
        GET DIAGNOSTICS v_rows = ROW_COUNT;
  if v_rows = 0 then
    raise notice 'add_empty_table_to_replication: table % empty on origin - OK', v_fqname;
  else
    raise exception 'add_empty_table_to_replication: table % contained tuples on origin  ↩
       node %', v_fqname, v_origin;
  end if;
   else
  -- On other nodes, TRUNCATE the table
        v_query := 'truncate ' || v_fqname || ';';
  execute v_query;
   end if;
-- If p_idxname is NULL, then look up the PK index, and RAISE EXCEPTION if one does not  ↩
    exist
   if p_idxname is NULL then
  select c2.relname into prec from pg_catalog.pg_index i, pg_catalog.pg_class c1,  ↩
      pg_catalog.pg_class c2, pg_catalog.pg_namespace n where i.indrelid = c1.oid and i. ↩
      indexrelid = c2.oid and c1.relname = p_tabname and i.indisprimary and n.nspname =  ↩
      p_nspname and n.oid = c1.relnamespace;
  if not found then
    raise exception 'add_empty_table_to_replication: table % has no primary key and no  ↩
       candidate specified!', v_fqname;
  else
    v_idxname := prec.relname;
  end if;
   else
  v_idxname := p_idxname;
   end if;
   return setAddTable_int(p_set_id, p_tab_id, v_fqname, v_idxname, p_comment);
end
```

13.22 add_missing_table_field(p_type text, p_field text, p_table text, p_namespace text)

Function Properties
Language: PLPGSQL, *Return Type:* boolean Add a column of a given type to a table if it is missing

```
DECLARE
  v_row       record;
  v_query     text;
BEGIN
  select 1 into v_row from pg_namespace n, pg_class c, pg_attribute a
     where slon_quote_brute(n.nspname) = p_namespace and
         c.relnamespace = n.oid and
         slon_quote_brute(c.relname) = p_table and
         a.attrelid = c.oid and
         slon_quote_brute(a.attname) = p_field;
  if not found then
    raise notice 'Upgrade table %.% - add field %', p_namespace, p_table, p_field;
    v_query := 'alter table ' || p_namespace || '.' || p_table || ' add column ';
    v_query := v_query || p_field || ' ' || p_type || ';';
    execute v_query;
    return 't';
  else
    return 'f';
  end if;
END;
```

13.23 addpartiallogindices()

Function Properties
Language: PLPGSQL, *Return Type:* integer Add partial indexes, if possible, to the unused sl_log_? table for all origin nodes, and drop any that are no longer needed. This function presently gets run any time set origins are manipulated (FAILOVER, STORE SET, MOVE SET, DROP SET), as well as each time the system switches between sl_log_1 and sl_log_2.

```
DECLARE
  v_current_status  int4;
  v_log     int4;
  v_dummy   record;
  v_dummy2  record;
  idef    text;
  v_count   int4;
        v_iname         text;
  v_ilen int4;
  v_maxlen int4;
BEGIN
  v_count := 0;
  select last_value into v_current_status from sl_log_status;

  -- If status is 2 or 3 --> in process of cleanup --> unsafe to create indices
  if v_current_status in (2, 3) then
    return 0;
  end if;

  if v_current_status = 0 then   -- Which log should get indices?
    v_log := 2;
  else
    v_log := 1;
  end if;
--                                        PartInd_test_db_sl_log_2-node-1
  -- Add missing indices...
  for v_dummy in select distinct set_origin from sl_set loop
            v_iname := 'PartInd_schemadoc_sl_log_' || v_log::text || '-node-'
       || v_dummy.set_origin::text;
      -- raise notice 'Consider adding partial index % on sl_log_%', v_iname, v_log;
      -- raise notice 'schema: [_schemadoc] tablename:[sl_log_%]', v_log;
            select * into v_dummy2 from pg_catalog.pg_indexes where tablename = 'sl_log_'  ↩
                || v_log::text and  indexname = v_iname;
            if not found then
    -- raise notice 'index was not found - add it!';
        v_iname := 'PartInd_schemadoc_sl_log_' || v_log::text || '-node-' || v_dummy. ↩
            set_origin::text;
    v_ilen := pg_catalog.length(v_iname);
    v_maxlen := pg_catalog.current_setting('max_identifier_length'::text)::int4;
                if v_ilen > v_maxlen then
       raise exception 'Length of proposed index name [%] > max_identifier_length [%] -  ↩
           cluster name probably too long', v_ilen, v_maxlen;
    end if;

    idef := 'create index "' || v_iname ||
                        '" on sl_log_' || v_log::text || ' USING btree(log_txid) where ( ↩
                            log_origin = ' || v_dummy.set_origin::text || ');';
    execute idef;
    v_count := v_count + 1;
            else
                -- raise notice 'Index % already present - skipping', v_iname;
            end if;
  end loop;
```

```
  -- Remove unneeded indices...
  for v_dummy in select indexname from pg_catalog.pg_indexes i where i.tablename = 'sl_log_ ↩
      ' || v_log::text and
                       i.indexname like ('PartInd_schemadoc_sl_log_' || v_log::text || '- ↩
                           node-%') and
                       not exists (select 1 from sl_set where
         i.indexname = 'PartInd_schemadoc_sl_log_' || v_log::text || '-node-' || set_origin ↩
             ::text)
  loop
    -- raise notice 'Dropping obsolete index %d', v_dummy.indexname;
    idef := 'drop index "' || v_dummy.indexname || '";';
    execute idef;
    v_count := v_count - 1;
  end loop;
  return v_count;
END
```

13.24 altertableaddtriggers(p_tab_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer alterTableAddTriggers(tab_id) Adds the log and deny access triggers to a replicated table.

```
declare
  v_no_id       int4;
  v_tab_row     record;
  v_tab_fqname    text;
  v_tab_attkind   text;
  v_n         int4;
  v_trec  record;
  v_tgbad boolean;
begin
  -- ----
  -- Grab the central configuration lock
  -- ----
  lock table sl_config_lock;

  -- ----
  -- Get our local node ID
  -- ----
  v_no_id := getLocalNodeId('_schemadoc');

  -- ----
  -- Get the sl_table row and the current origin of the table.
  -- ----
  select T.tab_reloid, T.tab_set, T.tab_idxname,
      S.set_origin, PGX.indexrelid,
      slon_quote_brute(PGN.nspname) || '.' ||
      slon_quote_brute(PGC.relname) as tab_fqname
      into v_tab_row
      from sl_table T, sl_set S,
        "pg_catalog".pg_class PGC, "pg_catalog".pg_namespace PGN,
        "pg_catalog".pg_index PGX, "pg_catalog".pg_class PGXC
      where T.tab_id = p_tab_id
        and T.tab_set = S.set_id
        and T.tab_reloid = PGC.oid
        and PGC.relnamespace = PGN.oid
        and PGX.indrelid = T.tab_reloid
        and PGX.indexrelid = PGXC.oid
```

```
        and PGXC.relname = T.tab_idxname
        for update;
  if not found then
    raise exception 'Slony-I: alterTableAddTriggers(): Table with id % not found', p_tab_id ↩
        ;
  end if;
  v_tab_fqname = v_tab_row.tab_fqname;

  v_tab_attkind := determineAttKindUnique(v_tab_row.tab_fqname,
            v_tab_row.tab_idxname);

  execute 'lock table ' || v_tab_fqname || ' in access exclusive mode';

  -- ----
  -- Create the log and the deny access triggers
  -- ----
  execute 'create trigger "_schemadoc_logtrigger"' ||
      ' after insert or update or delete on ' ||
      v_tab_fqname || ' for each row execute procedure logTrigger (' ||
                              pg_catalog.quote_literal('_schemadoc') || ',' ||
        pg_catalog.quote_literal(p_tab_id::text) || ',' ||
        pg_catalog.quote_literal(v_tab_attkind) || ');';

  execute 'create trigger "_schemadoc_denyaccess" ' ||
      'before insert or update or delete on ' ||
      v_tab_fqname || ' for each row execute procedure ' ||
      'denyAccess (' || pg_catalog.quote_literal('_schemadoc') || ');';

  perform alterTableAddTruncateTrigger(v_tab_fqname, p_tab_id);

  perform alterTableConfigureTriggers (p_tab_id);
  return p_tab_id;
end;
```

13.25 altertableconfiguretriggers(p_tab_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer alterTableConfigureTriggers (tab_id) Set the enable/disable configuration for the replication triggers according to the origin of the set.

```
declare
  v_no_id       int4;
  v_tab_row     record;
  v_tab_fqname    text;
  v_n         int4;
begin
  -- ----
  -- Grab the central configuration lock
  -- ----
  lock table sl_config_lock;

  -- ----
  -- Get our local node ID
  -- ----
  v_no_id := getLocalNodeId('_schemadoc');

  -- ----
  -- Get the sl_table row and the current tables origin.
  -- ----
```

```
  select T.tab_reloid, T.tab_set,
      S.set_origin, PGX.indexrelid,
      slon_quote_brute(PGN.nspname) || '.' ||
      slon_quote_brute(PGC.relname) as tab_fqname
      into v_tab_row
      from sl_table T, sl_set S,
        "pg_catalog".pg_class PGC, "pg_catalog".pg_namespace PGN,
        "pg_catalog".pg_index PGX, "pg_catalog".pg_class PGXC
      where T.tab_id = p_tab_id
        and T.tab_set = S.set_id
        and T.tab_reloid = PGC.oid
        and PGC.relnamespace = PGN.oid
        and PGX.indrelid = T.tab_reloid
        and PGX.indexrelid = PGXC.oid
        and PGXC.relname = T.tab_idxname
        for update;
  if not found then
    raise exception 'Slony-I: alterTableConfigureTriggers(): Table with id % not found',  ↩
        p_tab_id;
  end if;
  v_tab_fqname = v_tab_row.tab_fqname;

  -- ----
  -- Configuration depends on the origin of the table
  -- ----
  if v_tab_row.set_origin = v_no_id then
    -- ----
    -- On the origin the log trigger is configured like a default
    -- user trigger and the deny access trigger is disabled.
    -- ----
    execute 'alter table ' || v_tab_fqname ||
        ' enable trigger "_schemadoc_logtrigger"';
    execute 'alter table ' || v_tab_fqname ||
        ' disable trigger "_schemadoc_denyaccess"';
        perform alterTableConfigureTruncateTrigger(v_tab_fqname,
        'enable', 'disable');
  else
    -- ----
    -- On a replica the log trigger is disabled and the
    -- deny access trigger fires in origin session role.
    -- ----
    execute 'alter table ' || v_tab_fqname ||
        ' disable trigger "_schemadoc_logtrigger"';
    execute 'alter table ' || v_tab_fqname ||
        ' enable trigger "_schemadoc_denyaccess"';
        perform alterTableConfigureTruncateTrigger(v_tab_fqname,
        'disable', 'enable');

  end if;

  return p_tab_id;
end;
```

13.26 altertabledroptriggers(p_tab_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer alterTableDropTriggers (tab_id) Remove the log and deny access triggers from a table.

```
declare
  v_no_id       int4;
  v_tab_row     record;
  v_tab_fqname    text;
  v_n         int4;
begin
  -- ----
  -- Grab the central configuration lock
  -- ----
  lock table sl_config_lock;

  -- ----
  -- Get our local node ID
  -- ----
  v_no_id := getLocalNodeId('_schemadoc');

  -- ----
  -- Get the sl_table row and the current tables origin.
  -- ----
  select T.tab_reloid, T.tab_set,
      S.set_origin, PGX.indexrelid,
      slon_quote_brute(PGN.nspname) || '.' ||
      slon_quote_brute(PGC.relname) as tab_fqname
      into v_tab_row
      from sl_table T, sl_set S,
        "pg_catalog".pg_class PGC, "pg_catalog".pg_namespace PGN,
        "pg_catalog".pg_index PGX, "pg_catalog".pg_class PGXC
      where T.tab_id = p_tab_id
        and T.tab_set = S.set_id
        and T.tab_reloid = PGC.oid
        and PGC.relnamespace = PGN.oid
        and PGX.indrelid = T.tab_reloid
        and PGX.indexrelid = PGXC.oid
        and PGXC.relname = T.tab_idxname
        for update;
  if not found then
    raise exception 'Slony-I: alterTableDropTriggers(): Table with id % not found',  ↩
        p_tab_id;
  end if;
  v_tab_fqname = v_tab_row.tab_fqname;

  execute 'lock table ' || v_tab_fqname || ' in access exclusive mode';

  -- ----
  -- Drop both triggers
  -- ----
  execute 'drop trigger "_schemadoc_logtrigger" on ' ||
      v_tab_fqname;

  execute 'drop trigger "_schemadoc_denyaccess" on ' ||
      v_tab_fqname;

  perform alterTableDropTruncateTrigger(v_tab_fqname, p_tab_id);

  return p_tab_id;
end;
```

13.27 checkmoduleversion()

Function Properties
Language: PLPGSQL, *Return Type:* text Inline test function that verifies that slonik request for STORE NODE/INIT CLUSTER is being run against a conformant set of schema/functions.

```
declare
  moduleversion text;
begin
  select into moduleversion getModuleVersion();
  if moduleversion <> '@MODULEVERSION@' then
      raise exception 'Slonik version: @MODULEVERSION@ != Slony-I version in PG build %',
            moduleversion;
  end if;
  return null;
end;
```

13.28 cleanupevent(p_interval interval)

Function Properties
Language: PLPGSQL, *Return Type:* integer cleaning old data out of sl_confirm, sl_event. Removes all but the last sl_confirm row per (origin,receiver), and then removes all events that are confirmed by all nodes in the whole cluster up to the last SYNC.

```
declare
  v_max_row record;
  v_min_row record;
  v_max_sync  int8;
  v_origin  int8;
  v_seqno   int8;
  v_xmin    bigint;
  v_rc            int8;
begin
  -- ----
  -- First remove all confirmations where origin/receiver no longer exist
  -- ----
  delete from sl_confirm
        where con_origin not in (select no_id from sl_node);
  delete from sl_confirm
        where con_received not in (select no_id from sl_node);
  -- ----
  -- Next remove all but the oldest confirm row per origin,receiver pair.
  -- Ignore confirmations that are younger than 10 minutes. We currently
  -- have an not confirmed suspicion that a possibly lost transaction due
  -- to a server crash might have been visible to another session, and
  -- that this led to log data that is needed again got removed.
  -- ----
  for v_max_row in select con_origin, con_received, max(con_seqno) as con_seqno
        from sl_confirm
        where con_timestamp < (CURRENT_TIMESTAMP - p_interval)
        group by con_origin, con_received
  loop
    delete from sl_confirm
        where con_origin = v_max_row.con_origin
        and con_received = v_max_row.con_received
        and con_seqno < v_max_row.con_seqno;
  end loop;

  -- ----
  -- Then remove all events that are confirmed by all nodes in the
```

```
-- whole cluster up to the last SYNC
-- ----
for v_min_row in select con_origin, min(con_seqno) as con_seqno
      from sl_confirm
      group by con_origin
loop
  select coalesce(max(ev_seqno), 0) into v_max_sync
      from sl_event
      where ev_origin = v_min_row.con_origin
      and ev_seqno <= v_min_row.con_seqno
      and ev_type = 'SYNC';
  if v_max_sync > 0 then
    delete from sl_event
        where ev_origin = v_min_row.con_origin
        and ev_seqno < v_max_sync;
  end if;
end loop;

-- ----
-- If cluster has only one node, then remove all events up to
-- the last SYNC - Bug #1538
      -- http://gborg.postgresql.org/project/slony1/bugs/bugupdate.php?1538
-- ----

select * into v_min_row from sl_node where
    no_id <> getLocalNodeId('_schemadoc') limit 1;
if not found then
  select ev_origin, ev_seqno into v_min_row from sl_event
  where ev_origin = getLocalNodeId('_schemadoc')
  order by ev_origin desc, ev_seqno desc limit 1;
  raise notice 'Slony-I: cleanupEvent(): Single node - deleting events < %', v_min_row. ↩
      ev_seqno;
    delete from sl_event
    where
      ev_origin = v_min_row.ev_origin and
      ev_seqno < v_min_row.ev_seqno;

      end if;

if exists (select * from "pg_catalog".pg_class c, "pg_catalog".pg_namespace n, " ↩
    pg_catalog".pg_attribute a where c.relname = 'sl_seqlog' and n.oid = c.relnamespace  ↩
    and a.attrelid = c.oid and a.attname = 'oid') then
              execute 'alter table sl_seqlog set without oids;';
end if;
-- ----
-- Also remove stale entries from the nodelock table.
-- ----
perform cleanupNodelock();

-- ----
-- Find the eldest event left, for each origin
-- ----
  for v_origin, v_seqno, v_xmin in
select ev_origin, ev_seqno, "pg_catalog".txid_snapshot_xmin(ev_snapshot) from sl_event
        where (ev_origin, ev_seqno) in (select ev_origin, min(ev_seqno) from sl_event  ↩
            where ev_type = 'SYNC' group by ev_origin)
loop
  delete from sl_seqlog where seql_origin = v_origin and seql_ev_seqno < v_seqno;
  end loop;

v_rc := logswitch_finish();
if v_rc = 0 then   -- no switch in progress
```

```
    perform logswitch_start();
  end if;

  return 0;
end;
```

13.29 cleanupnodelock()

Function Properties
Language: PLPGSQL, *Return Type:* integer Clean up stale entries when restarting slon

```
declare
  v_row   record;
begin
  for v_row in select nl_nodeid, nl_conncnt, nl_backendpid
      from sl_nodelock
      for update
  loop
    if killBackend(v_row.nl_backendpid, 'NULL') < 0 then
      raise notice 'Slony-I: cleanup stale sl_nodelock entry for pid=%',
          v_row.nl_backendpid;
      delete from sl_nodelock where
          nl_nodeid = v_row.nl_nodeid and
          nl_conncnt = v_row.nl_conncnt;
    end if;
  end loop;

  return 0;
end;
```

13.30 clonenodefinish(p_no_provider integer, p_no_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer Internal part of cloneNodePrepare().

```
declare
  v_row     record;
begin
  perform "pg_catalog".setval('sl_local_node_id', p_no_id);
  perform resetSession();
  for v_row in select sub_set from sl_subscribe
      where sub_receiver = p_no_id
  loop
    perform updateReloid(v_row.sub_set, p_no_id);
  end loop;

  perform RebuildListenEntries();

  delete from sl_confirm
    where con_received = p_no_id;
  insert into sl_confirm
    (con_origin, con_received, con_seqno, con_timestamp)
    select con_origin, p_no_id, con_seqno, con_timestamp
    from sl_confirm
    where con_received = p_no_provider;
  insert into sl_confirm
    (con_origin, con_received, con_seqno, con_timestamp)
```

```
    select p_no_provider, p_no_id,
        (select max(ev_seqno) from sl_event
          where ev_origin = p_no_provider), current_timestamp;

  return 0;
end;
```

13.31 clonenodeprepare(p_no_comment integer, p_no_provider integer, p_no_id text)

Function Properties
Language: PLPGSQL, *Return Type:* bigint Prepare for cloning a node.

```
begin
  perform cloneNodePrepare_int (p_no_id, p_no_provider, p_no_comment);
  return  createEvent('_schemadoc', 'CLONE_NODE',
                  p_no_id::text, p_no_provider::text,
                  p_no_comment::text);
end;
```

13.32 clonenodeprepare_int(p_no_comment integer, p_no_provider integer, p_no_id text)

Function Properties
Language: PLPGSQL, *Return Type:* integer Internal part of cloneNodePrepare().

```
begin
  insert into sl_node
    (no_id, no_active, no_comment)
    select p_no_id, no_active, p_no_comment
    from sl_node
    where no_id = p_no_provider;

  insert into sl_path
    (pa_server, pa_client, pa_conninfo, pa_connretry)
    select pa_server, p_no_id, 'Event pending', pa_connretry
    from sl_path
    where pa_client = p_no_provider;
  insert into sl_path
    (pa_server, pa_client, pa_conninfo, pa_connretry)
    select p_no_id, pa_client, 'Event pending', pa_connretry
    from sl_path
    where pa_server = p_no_provider;

  insert into sl_subscribe
    (sub_set, sub_provider, sub_receiver, sub_forward, sub_active)
    select sub_set, sub_provider, p_no_id, sub_forward, sub_active
    from sl_subscribe
    where sub_receiver = p_no_provider;

  insert into sl_confirm
    (con_origin, con_received, con_seqno, con_timestamp)
    select con_origin, p_no_id, con_seqno, con_timestamp
    from sl_confirm
    where con_received = p_no_provider;
```

```
  perform RebuildListenEntries();

  return 0;
end;
```

13.33 component_state(i_eventtype text, i_event integer, i_starttime integer, i_activity integer, i_conn_pid text, i_node timestamp with time zone, i_pid bigint, i_actor text)

Function Properties
Language: PLPGSQL, *Return Type:* integer Store state of a Slony component. Useful for monitoring

```
begin
  -- Trim out old state for this component
  if not exists (select 1 from sl_components where co_actor = i_actor) then
     insert into sl_components
             (co_actor, co_pid, co_node, co_connection_pid, co_activity, co_starttime,  ↩
                co_event, co_eventtype)
     values
              (i_actor, i_pid, i_node, i_conn_pid, i_activity, i_starttime, i_event,  ↩
                 i_eventtype);
  else
     update sl_components
              set
                 co_connection_pid = i_conn_pid, co_activity = i_activity, co_starttime =  ↩
                     i_starttime, co_event = i_event,
                 co_eventtype = i_eventtype
              where co_actor = i_actor
              and co_starttime < i_starttime;
  end if;
  return 1;
end
```

13.34 copyfields(p_tab_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* text Return a string consisting of what should be appended to a COPY statement to specify fields for the passed-in tab_id. In PG versions > 7.3, this looks like (field1,field2,...fieldn)

```
declare
  result text;
  prefix text;
  prec record;
begin
  result := '';
  prefix := '(';   -- Initially, prefix is the opening paren

  for prec in select slon_quote_input(a.attname) as column from sl_table t, pg_catalog. ↩
      pg_attribute a where t.tab_id = p_tab_id and t.tab_reloid = a.attrelid and a.attnum >  ↩
      0 and a.attisdropped = false order by attnum
  loop
    result := result || prefix || prec.column;
    prefix := ',';   -- Subsequently, prepend columns with commas
  end loop;
```

```
  result := result || ')';
  return result;
end;
```

13.35 createevent(ev_data1 name, p_event_type text, p_cluster_name text)

Function Properties
Language: C, *Return Type:* bigint FUNCTION createEvent (cluster_name, ev_type [, ev_data [...]]) Create an sl_event entry

```
_Slony_I_createEvent
```

13.36 createevent(ev_data2 name, ev_data1 text, p_event_type text, p_cluster_name text)

Function Properties
Language: C, *Return Type:* bigint FUNCTION createEvent (cluster_name, ev_type [, ev_data [...]]) Create an sl_event entry

```
_Slony_I_createEvent
```

13.37 createevent(ev_data3 name, ev_data2 text, ev_data1 text, p_event_type text, p_cluster_name text)

Function Properties
Language: C, *Return Type:* bigint FUNCTION createEvent (cluster_name, ev_type [, ev_data [...]]) Create an sl_event entry

```
_Slony_I_createEvent
```

13.38 createevent(ev_data4 name, ev_data3 text, ev_data2 text, ev_data1 text, p_event_type text, p_cluster_name text)

Function Properties
Language: C, *Return Type:* bigint FUNCTION createEvent (cluster_name, ev_type [, ev_data [...]]) Create an sl_event entry

```
_Slony_I_createEvent
```

13.39 createevent(ev_data5 name, ev_data4 text, ev_data3 text, ev_data2 text, ev_data1 text, p_event_type text, p_cluster_name text)

Function Properties
Language: C, *Return Type:* bigint FUNCTION createEvent (cluster_name, ev_type [, ev_data [...]]) Create an sl_event entry

```
_Slony_I_createEvent
```

13.40 createevent(ev_data6 name, ev_data5 text, ev_data4 text, ev_data3 text, ev_data2 text, ev_data1 text, p_event_type text, p_cluster_name text)

Function Properties
Language: C, *Return Type:* bigint FUNCTION createEvent (cluster_name, ev_type [, ev_data [...]]) Create an sl_event entry

```
_Slony_I_createEvent
```

13.41 createevent(ev_data7 name, ev_data6 text, ev_data5 text, ev_data4 text, ev_data3 text, ev_data2 text, ev_data1 text, p_event_type text, p_cluster_name text)

Function Properties
Language: C, *Return Type:* bigint FUNCTION createEvent (cluster_name, ev_type [, ev_data [...]]) Create an sl_event entry

```
_Slony_I_createEvent
```

13.42 createevent(ev_data8 name, ev_data7 text, ev_data6 text, ev_data5 text, ev_data4 text, ev_data3 text, ev_data2 text, ev_data1 text, p_event_type text, p_cluster_name text)

Function Properties
Language: C, *Return Type:* bigint FUNCTION createEvent (cluster_name, ev_type [, ev_data [...]]) Create an sl_event entry

```
_Slony_I_createEvent
```

13.43 createevent(p_event_type name, p_cluster_name text)

Function Properties
Language: C, *Return Type:* bigint FUNCTION createEvent (cluster_name, ev_type [, ev_data [...]]) Create an sl_event entry

```
_Slony_I_createEvent
```

13.44 ddlscript_complete(p_only_on_node integer, p_script text, p_set_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* bigint ddlScript_complete(set_id, script, only_on_node) After script has run on origin, this fixes up relnames, restores triggers, and generates a DDL_SCRIPT event to request it to be run on replicated slaves.

```
declare
  v_set_origin    int4;
  v_query       text;
  v_row       record;
begin
  if p_only_on_node = -1 then
          perform ddlScript_complete_int(p_set_id,p_only_on_node);
```

```
    return  createEvent('_schemadoc', 'DDL_SCRIPT',
      p_set_id::text, p_script::text, p_only_on_node::text);
  end if;
  if p_only_on_node <> -1 then
    for v_row in execute
      'select setting from _slony1_saved_session_replication_role' loop
      v_query := 'set session_replication_role to ' || v_row.setting;
    end loop;
    execute v_query;
    execute 'drop table _slony1_saved_session_replication_role';
    perform ddlScript_complete_int(p_set_id,p_only_on_node);
  end if;
  return NULL;
end;
```

13.45 ddlscript_complete_int(p_only_on_node integer, p_set_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer ddlScript_complete_int(set_id, script, only_on_node) Complete processing the DDL_SCRIPT event. This puts tables back into replicated mode.

```
declare
  v_row       record;
begin
  perform updateRelname(p_set_id, p_only_on_node);
  perform repair_log_triggers(true);
  return p_set_id;
end;
```

13.46 ddlscript_prepare(p_only_on_node integer, p_set_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer Prepare for DDL script execution on origin

```
declare
  v_set_origin    int4;
begin
  -- ----
  -- Check that the set exists and originates here
  -- ----
  select set_origin into v_set_origin
      from sl_set
      where set_id = p_set_id
      for update;
  if not found then
    raise exception 'Slony-I: set % not found', p_set_id;
  end if;
  if p_only_on_node = -1 then
    if v_set_origin <> getLocalNodeId('_schemadoc') then
      raise exception 'Slony-I: set % does not originate on local node',
        p_set_id;
    end if;
    -- ----
    -- Create a SYNC event
    -- ----
    perform createEvent('_schemadoc', 'SYNC', NULL);
```

```
  else
    -- If running "ONLY ON NODE", there are two possibilities:
    -- 1.  Running on origin, where denyaccess() triggers are already shut off
    -- 2.  Running on replica, where we need the LOCAL role to suppress denyaccess()  ↩
        triggers
    execute 'create temp table _slony1_saved_session_replication_role (
          setting text);';
    execute 'insert into _slony1_saved_session_replication_role
          select setting from pg_catalog.pg_settings
          where name = ''session_replication_role'';';
    if (v_set_origin <> getLocalNodeId('_schemadoc')) then
      execute 'set session_replication_role to local;';
    end if;
  end if;
  return 1;
end;
```

13.47 ddlscript_prepare_int(p_only_on_node integer, p_set_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer ddlScript_prepare_int (set_id, only_on_node) Do preparatory work for a DDL script, restoring triggers/rules to original state.

```
declare
  v_set_origin    int4;
  v_no_id       int4;
  v_row       record;
begin
  -- ----
  -- Check that we either are the set origin or a current
  -- subscriber of the set.
  -- ----
  v_no_id := getLocalNodeId('_schemadoc');
  select set_origin into v_set_origin
      from sl_set
      where set_id = p_set_id
      for update;
  if not found then
    raise exception 'Slony-I: set % not found', p_set_id;
  end if;
  if v_set_origin <> v_no_id
      and not exists (select 1 from sl_subscribe
            where sub_set = p_set_id
            and sub_receiver = v_no_id)
  then
    return 0;
  end if;

  -- ----
  -- If execution on only one node is requested, check that
  -- we are that node.
  -- ----
  if p_only_on_node > 0 and p_only_on_node <> v_no_id then
    return 0;
  end if;

  return p_set_id;
end;
```

13.48 decode_tgargs(bytea)

Function Properties
Language: C, *Return Type:* text[] Translates the contents of pg_trigger.tgargs to an array of text arguments

```
_slon_decode_tgargs
```

13.49 deny_truncate()

Function Properties
Language: PLPGSQL, *Return Type:* trigger trigger function run when a replicated table receives a TRUNCATE request

```
  begin
    raise exception 'truncation of replicated table forbidden on subscriber node';
    end
```

13.50 denyaccess()

Function Properties
Language: C, *Return Type:* trigger Trigger function to prevent modifications to a table on a subscriber

```
_Slony_I_denyAccess
```

13.51 determineattkindunique(p_idx_name text, p_tab_fqname name)

Function Properties
Language: PLPGSQL, *Return Type:* text determineAttKindUnique (tab_fqname, indexname) Given a tablename, return the Slony-I specific attkind (used for the log trigger) of the table. Use the specified unique index or the primary key (if indexname is NULL).

```
declare
  v_tab_fqname_quoted text default '';
  v_idx_name_quoted text;
  v_idxrow    record;
  v_attrow    record;
  v_i       integer;
  v_attno     int2;
  v_attkind   text default '';
  v_attfound    bool;
begin
  v_tab_fqname_quoted := slon_quote_input(p_tab_fqname);
  v_idx_name_quoted := slon_quote_brute(p_idx_name);
  --
  -- Ensure that the table exists
  --
  if (select PGC.relname
        from "pg_catalog".pg_class PGC,
          "pg_catalog".pg_namespace PGN
        where slon_quote_brute(PGN.nspname) || '.' ||
          slon_quote_brute(PGC.relname) = v_tab_fqname_quoted
          and PGN.oid = PGC.relnamespace) is null then
    raise exception 'Slony-I: table % not found', v_tab_fqname_quoted;
  end if;
```

```
--
-- Lookup the tables primary key or the specified unique index
--
if p_idx_name isnull then
  raise exception 'Slony-I: index name must be specified';
else
  select PGXC.relname, PGX.indexrelid, PGX.indkey
      into v_idxrow
      from "pg_catalog".pg_class PGC,
        "pg_catalog".pg_namespace PGN,
        "pg_catalog".pg_index PGX,
        "pg_catalog".pg_class PGXC
      where slon_quote_brute(PGN.nspname) || '.' ||
        slon_quote_brute(PGC.relname) = v_tab_fqname_quoted
        and PGN.oid = PGC.relnamespace
        and PGX.indrelid = PGC.oid
        and PGX.indexrelid = PGXC.oid
        and PGX.indisunique
        and slon_quote_brute(PGXC.relname) = v_idx_name_quoted;
  if not found then
    raise exception 'Slony-I: table % has no unique index %',
        v_tab_fqname_quoted, v_idx_name_quoted;
  end if;
end if;

--
-- Loop over the tables attributes and check if they are
-- index attributes. If so, add a "k" to the return value,
-- otherwise add a "v".
--
for v_attrow in select PGA.attnum, PGA.attname
    from "pg_catalog".pg_class PGC,
        "pg_catalog".pg_namespace PGN,
      "pg_catalog".pg_attribute PGA
    where slon_quote_brute(PGN.nspname) || '.' ||
        slon_quote_brute(PGC.relname) = v_tab_fqname_quoted
      and PGN.oid = PGC.relnamespace
      and PGA.attrelid = PGC.oid
      and not PGA.attisdropped
      and PGA.attnum > 0
    order by attnum
loop
  v_attfound = 'f';

  v_i := 0;
  loop
    select indkey[v_i] into v_attno from "pg_catalog".pg_index
        where indexrelid = v_idxrow.indexrelid;
    if v_attno isnull or v_attno = 0 then
      exit;
    end if;
    if v_attrow.attnum = v_attno then
      v_attfound = 't';
      exit;
    end if;
    v_i := v_i + 1;
  end loop;

  if v_attfound then
    v_attkind := v_attkind || 'k';
  else
```

```
      v_attkind := v_attkind || 'v';
    end if;
  end loop;

  -- Strip off trailing v characters as they are not needed by the logtrigger
  v_attkind := pg_catalog.rtrim(v_attkind, 'v');

  --
  -- Return the resulting attkind
  --
  return v_attkind;
end;
```

13.52 determineidxnameunique(p_idx_name text, p_tab_fqname name)

Function Properties
Language: PLPGSQL, *Return Type:* name FUNCTION determineIdxnameUnique (tab_fqname, indexname) Given a tablename, tab_fqname, check that the unique index, indexname, exists or return the primary key index name for the table. If there is no unique index, it raises an exception.

```
declare
  v_tab_fqname_quoted text default '';
  v_idxrow    record;
begin
  v_tab_fqname_quoted := slon_quote_input(p_tab_fqname);
  --
  -- Ensure that the table exists
  --
  if (select PGC.relname
        from "pg_catalog".pg_class PGC,
          "pg_catalog".pg_namespace PGN
        where slon_quote_brute(PGN.nspname) || '.' ||
          slon_quote_brute(PGC.relname) = v_tab_fqname_quoted
          and PGN.oid = PGC.relnamespace) is null then
    raise exception 'Slony-I: determineIdxnameUnique(): table % not found',  ↩
       v_tab_fqname_quoted;
  end if;

  --
  -- Lookup the tables primary key or the specified unique index
  --
  if p_idx_name isnull then
    select PGXC.relname
        into v_idxrow
        from "pg_catalog".pg_class PGC,
          "pg_catalog".pg_namespace PGN,
          "pg_catalog".pg_index PGX,
          "pg_catalog".pg_class PGXC
        where slon_quote_brute(PGN.nspname) || '.' ||
          slon_quote_brute(PGC.relname) = v_tab_fqname_quoted
          and PGN.oid = PGC.relnamespace
          and PGX.indrelid = PGC.oid
          and PGX.indexrelid = PGXC.oid
          and PGX.indisprimary;
    if not found then
      raise exception 'Slony-I: table % has no primary key',
          v_tab_fqname_quoted;
    end if;
  else
```

```
    select PGXC.relname
        into v_idxrow
        from "pg_catalog".pg_class PGC,
          "pg_catalog".pg_namespace PGN,
          "pg_catalog".pg_index PGX,
          "pg_catalog".pg_class PGXC
        where slon_quote_brute(PGN.nspname) || '.' ||
          slon_quote_brute(PGC.relname) = v_tab_fqname_quoted
          and PGN.oid = PGC.relnamespace
          and PGX.indrelid = PGC.oid
          and PGX.indexrelid = PGXC.oid
          and PGX.indisunique
          and slon_quote_brute(PGXC.relname) = slon_quote_input(p_idx_name);
    if not found then
      raise exception 'Slony-I: table % has no unique index %',
          v_tab_fqname_quoted, p_idx_name;
    end if;
  end if;

  --
  -- Return the found index name
  --
  return v_idxrow.relname;
end;
```

13.53 disable_indexes_on_table(i_oid oid)

Function Properties
Language: PLPGSQL, *Return Type:* integer disable indexes on the specified table. Used during subscription process to suppress indexes, which allows COPY to go much faster. This may be set as a SECURITY DEFINER in order to eliminate the need for superuser access by Slony-I.

```
begin
  -- Setting pg_class.relhasindex to false will cause copy not to
  -- maintain any indexes. At the end of the copy we will reenable
  -- them and reindex the table. This bulk creating of indexes is
  -- faster.

  update pg_catalog.pg_class set relhasindex ='f' where oid = i_oid;
  return 1;
end
```

13.54 disablenode(p_no_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* bigint process DISABLE_NODE event for node no_id NOTE: This is not yet implemented!

```
begin
  -- **** TODO ****
  raise exception 'Slony-I: disableNode() not implemented';
end;
```

13.55 disablenode_int(p_no_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer

```
begin
  -- **** TODO ****
  raise exception 'Slony-I: disableNode_int() not implemented';
end;
```

13.56 droplisten(p_li_receiver integer, p_li_provider integer, p_li_origin integer)

Function Properties
Language: PLPGSQL, *Return Type:* bigint dropListen (li_origin, li_provider, li_receiver) Generate the DROP_LISTEN event.

```
begin
  perform dropListen_int(p_li_origin,
      p_li_provider, p_li_receiver);

  return  createEvent ('_schemadoc', 'DROP_LISTEN',
      p_li_origin::text, p_li_provider::text, p_li_receiver::text);
end;
```

13.57 droplisten_int(p_li_receiver integer, p_li_provider integer, p_li_origin integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer dropListen (li_origin, li_provider, li_receiver) Process the DROP_LISTEN event, deleting the sl_listen entry for the indicated (origin,provider,receiver) combination.

```
begin
  delete from sl_listen
      where li_origin = p_li_origin
      and li_provider = p_li_provider
      and li_receiver = p_li_receiver;
  if found then
    return 1;
  else
    return 0;
  end if;
end;
```

13.58 dropnode(p_no_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* bigint generate DROP_NODE event to drop node node_id from replication

```
declare
  v_node_row    record;
begin
  -- ----
  -- Check that this got called on a different node
  -- ----
```

```
  if p_no_id = getLocalNodeId('_schemadoc') then
    raise exception 'Slony-I: DROP_NODE cannot initiate on the dropped node';
  end if;

  select * into v_node_row from sl_node
      where no_id = p_no_id
      for update;
  if not found then
    raise exception 'Slony-I: unknown node ID %', p_no_id;
  end if;

  -- ----
  -- Make sure we do not break other nodes subscriptions with this
  -- ----
  if exists (select true from sl_subscribe
      where sub_provider = p_no_id)
  then
    raise exception 'Slony-I: Node % is still configured as a data provider',
        p_no_id;
  end if;

  -- ----
  -- Make sure no set originates there any more
  -- ----
  if exists (select true from sl_set
      where set_origin = p_no_id)
  then
    raise exception 'Slony-I: Node % is still origin of one or more sets',
        p_no_id;
  end if;

  -- ----
  -- Call the internal drop functionality and generate the event
  -- ----
  perform dropNode_int(p_no_id);
  return  createEvent('_schemadoc', 'DROP_NODE',
                  p_no_id::text);
end;
```

13.59 dropnode_int(p_no_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer internal function to process DROP_NODE event to drop node node_id from replication

```
declare
  v_tab_row   record;
begin
  -- ----
  -- If the dropped node is a remote node, clean the configuration
  -- from all traces for it.
  -- ----
  if p_no_id <> getLocalNodeId('_schemadoc') then
    delete from sl_subscribe
        where sub_receiver = p_no_id;
    delete from sl_listen
        where li_origin = p_no_id
          or li_provider = p_no_id
          or li_receiver = p_no_id;
```

```
    delete from sl_path
        where pa_server = p_no_id
          or pa_client = p_no_id;
    delete from sl_confirm
        where con_origin = p_no_id
          or con_received = p_no_id;
    delete from sl_event
        where ev_origin = p_no_id;
    delete from sl_node
        where no_id = p_no_id;

    return p_no_id;
  end if;

  -- ----
  -- This is us ... deactivate the node for now, the daemon
  -- will call uninstallNode() in a separate transaction.
  -- ----
  update sl_node
      set no_active = false
      where no_id = p_no_id;

  -- Rewrite sl_listen table
  perform RebuildListenEntries();

  return p_no_id;
end;
```

13.60 droppath(p_pa_client integer, p_pa_server integer)

Function Properties
Language: PLPGSQL, *Return Type:* bigint Generate DROP_PATH event to drop path from pa_server to pa_client

```
declare
  v_row     record;
begin
  -- ----
  -- There should be no existing subscriptions. Auto unsubscribing
  -- is considered too dangerous.
  -- ----
  for v_row in select sub_set, sub_provider, sub_receiver
      from sl_subscribe
      where sub_provider = p_pa_server
      and sub_receiver = p_pa_client
  loop
    raise exception
      'Slony-I: Path cannot be dropped, subscription of set % needs it',
      v_row.sub_set;
  end loop;

  -- ----
  -- Drop all sl_listen entries that depend on this path
  -- ----
  for v_row in select li_origin, li_provider, li_receiver
      from sl_listen
      where li_provider = p_pa_server
      and li_receiver = p_pa_client
  loop
    perform dropListen(
```

```
        v_row.li_origin, v_row.li_provider, v_row.li_receiver);
  end loop;

  -- ----
  -- Now drop the path and create the event
  -- ----
  perform dropPath_int(p_pa_server, p_pa_client);

  -- Rewrite sl_listen table
  perform RebuildListenEntries();

  return  createEvent ('_schemadoc', 'DROP_PATH',
      p_pa_server::text, p_pa_client::text);
end;
```

13.61 droppath_int(p_pa_client integer, p_pa_server integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer Process DROP_PATH event to drop path from pa_server to pa_client

```
begin
  -- ----
  -- Remove any dangling sl_listen entries with the server
  -- as provider and the client as receiver. This must have
  -- been cleared out before, but obviously was not.
  -- ----
  delete from sl_listen
      where li_provider = p_pa_server
      and li_receiver = p_pa_client;

  delete from sl_path
      where pa_server = p_pa_server
      and pa_client = p_pa_client;

  if found then
    -- Rewrite sl_listen table
    perform RebuildListenEntries();

    return 1;
  else
    -- Rewrite sl_listen table
    perform RebuildListenEntries();

    return 0;
  end if;
end;
```

13.62 dropset(p_set_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* bigint Process DROP_SET event to drop replication of set set_id. This involves: - Removing log and deny access triggers - Removing all traces of the set configuration, including sequences, tables, subscribers, syncs, and the set itself

```
declare
  v_origin      int4;
```

```
begin
  -- ----
  -- Check that the set exists and originates here
  -- ----
  select set_origin into v_origin from sl_set
      where set_id = p_set_id;
  if not found then
    raise exception 'Slony-I: set % not found', p_set_id;
  end if;
  if v_origin != getLocalNodeId('_schemadoc') then
    raise exception 'Slony-I: set % does not originate on local node',
        p_set_id;
  end if;

  -- ----
  -- Call the internal drop set functionality and generate the event
  -- ----
  perform dropSet_int(p_set_id);
  return  createEvent('_schemadoc', 'DROP_SET',
      p_set_id::text);
end;
```

13.63 dropset_int(p_set_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer

```
declare
  v_tab_row     record;
begin
  -- ----
  -- Restore all tables original triggers and rules and remove
  -- our replication stuff.
  -- ----
  for v_tab_row in select tab_id from sl_table
      where tab_set = p_set_id
      order by tab_id
  loop
    perform alterTableDropTriggers(v_tab_row.tab_id);
  end loop;

  -- ----
  -- Remove all traces of the set configuration
  -- ----
  delete from sl_sequence
      where seq_set = p_set_id;
  delete from sl_table
      where tab_set = p_set_id;
  delete from sl_subscribe
      where sub_set = p_set_id;
  delete from sl_setsync
      where ssy_setid = p_set_id;
  delete from sl_set
      where set_id = p_set_id;

  -- Regenerate sl_listen since we revised the subscriptions
  perform RebuildListenEntries();

  -- Run addPartialLogIndices() to try to add indices to unused sl_log_? table
```

```
  perform addPartialLogIndices();

  return p_set_id;
end;
```

13.64 enable_indexes_on_table(i_oid oid)

Function Properties
Language: PLPGSQL, *Return Type:* integer re-enable indexes on the specified table. This may be set as a SECURITY DEFINER in order to eliminate the need for superuser access by Slony-I.

```
begin
  update pg_catalog.pg_class set relhasindex ='t' where oid = i_oid;
  return 1;
end
```

13.65 enablenode(p_no_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* bigint no_id - Node ID # Generate the ENABLE_NODE event for node no_id

```
declare
  v_local_node_id int4;
  v_node_row    record;
begin
  -- ----
  -- Check that we are the node to activate and that we are
  -- currently disabled.
  -- ----
  v_local_node_id := getLocalNodeId('_schemadoc');
  select * into v_node_row
      from sl_node
      where no_id = p_no_id
      for update;
  if not found then
    raise exception 'Slony-I: node % not found', p_no_id;
  end if;
  if v_node_row.no_active then
    raise exception 'Slony-I: node % is already active', p_no_id;
  end if;

  -- ----
  -- Activate this node and generate the ENABLE_NODE event
  -- ----
  perform enableNode_int (p_no_id);
  return  createEvent('_schemadoc', 'ENABLE_NODE',
                 p_no_id::text);
end;
```

13.66 enablenode_int(p_no_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer no_id - Node ID # Internal function to process the ENABLE_NODE event for node no_id

```
declare
  v_local_node_id int4;
  v_node_row    record;
  v_sub_row   record;
begin
  -- ----
  -- Check that the node is inactive
  -- ----
  select * into v_node_row
      from sl_node
      where no_id = p_no_id
      for update;
  if not found then
    raise exception 'Slony-I: node % not found', p_no_id;
  end if;
  if v_node_row.no_active then
    return p_no_id;
  end if;

  -- ----
  -- Activate the node and generate sl_confirm status rows for it.
  -- ----
  update sl_node
      set no_active = 't'
      where no_id = p_no_id;
  insert into sl_confirm
      (con_origin, con_received, con_seqno)
      select no_id, p_no_id, 0 from sl_node
        where no_id != p_no_id
        and no_active;
  insert into sl_confirm
      (con_origin, con_received, con_seqno)
      select p_no_id, no_id, 0 from sl_node
        where no_id != p_no_id
        and no_active;

  -- ----
  -- Generate ENABLE_SUBSCRIPTION events for all sets that
  -- origin here and are subscribed by the just enabled node.
  -- ----
  v_local_node_id := getLocalNodeId('_schemadoc');
  for v_sub_row in select SUB.sub_set, SUB.sub_provider from
      sl_set S,
      sl_subscribe SUB
      where S.set_origin = v_local_node_id
      and S.set_id = SUB.sub_set
      and SUB.sub_receiver = p_no_id
      for update of S
  loop
    perform enableSubscription (v_sub_row.sub_set,
        v_sub_row.sub_provider, p_no_id);
  end loop;

  return p_no_id;
end;
```

13.67 enablesubscription(p_sub_receiver integer, p_sub_provider integer, p_sub_set integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer enableSubscription (sub_set, sub_provider, sub_receiver) Indicates that sub_receiver intends subscribing to set sub_set from sub_provider. Work is all done by the internal function enableSubscription_int (sub_set, sub_provider, sub_receiver).

```
begin
  return  enableSubscription_int (p_sub_set,
      p_sub_provider, p_sub_receiver);
end;
```

13.68 enablesubscription_int(p_sub_receiver integer, p_sub_provider integer, p_sub_s integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer enableSubscription_int (sub_set, sub_provider, sub_receiver) Internal function to enable subscription of node sub_receiver to set sub_set via node sub_provider. slon does most of the work; all we need do here is to remember that it happened. The function updates sl_subscribe, indicating that the subscription has become active.

```
declare
  v_n         int4;
begin
  -- ----
  -- The real work is done in the replication engine. All
  -- we have to do here is remembering that it happened.
  -- ----

  -- ----
  -- Well, not only ... we might be missing an important event here
  -- ----
  if not exists (select true from sl_path
      where pa_server = p_sub_provider
      and pa_client = p_sub_receiver)
  then
    insert into sl_path
        (pa_server, pa_client, pa_conninfo, pa_connretry)
        values
        (p_sub_provider, p_sub_receiver,
        '<event pending>', 10);
  end if;

  update sl_subscribe
      set sub_active = 't'
      where sub_set = p_sub_set
      and sub_receiver = p_sub_receiver;
  get diagnostics v_n = row_count;
  if v_n = 0 then
    insert into sl_subscribe
        (sub_set, sub_provider, sub_receiver,
        sub_forward, sub_active)
        values
        (p_sub_set, p_sub_provider, p_sub_receiver,
        false, true);
  end if;
```

```
  -- Rewrite sl_listen table
  perform RebuildListenEntries();

  return p_sub_set;
end;
```

13.69 failednode(p_backup_node integer, p_failed_node integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer Initiate failover from failed_node to backup_node. This function must be called on all nodes, and then waited for the restart of all node daemons.

```
declare
  v_row       record;
  v_row2        record;
  v_n         int4;
begin
  -- ----
  -- All consistency checks first
  -- Check that every node that has a path to the failed node
  -- also has a path to the backup node.
  -- ----
  for v_row in select P.pa_client
      from sl_path P
      where P.pa_server = p_failed_node
        and P.pa_client <> p_backup_node
        and not exists (select true from sl_path PP
              where PP.pa_server = p_backup_node
                and PP.pa_client = P.pa_client)
  loop
    raise exception 'Slony-I: cannot failover - node % has no path to the backup node',
        v_row.pa_client;
  end loop;

  -- ----
  -- Check all sets originating on the failed node
  -- ----
  for v_row in select set_id
      from sl_set
      where set_origin = p_failed_node
  loop
    -- ----
    -- Check that the backup node is subscribed to all sets
    -- that originate on the failed node
    -- ----
    select into v_row2 sub_forward, sub_active
        from sl_subscribe
        where sub_set = v_row.set_id
          and sub_receiver = p_backup_node;
    if not found then
      raise exception 'Slony-I: cannot failover - node % is not subscribed to set %',
          p_backup_node, v_row.set_id;
    end if;

    -- ----
    -- Check that the subscription is active
    -- ----
    if not v_row2.sub_active then
      raise exception 'Slony-I: cannot failover - subscription for set % is not active',
```

```
        v_row.set_id;
  end if;

  -- ----
  -- If there are other subscribers, the backup node needs to
  -- be a forwarder too.
  -- ----
  select into v_n count(*)
      from sl_subscribe
      where sub_set = v_row.set_id
        and sub_receiver <> p_backup_node;
  if v_n > 0 and not v_row2.sub_forward then
    raise exception 'Slony-I: cannot failover - node % is not a forwarder of set %',
        p_backup_node, v_row.set_id;
  end if;
end loop;

-- ----
-- Terminate all connections of the failed node the hard way
-- ----
perform terminateNodeConnections(p_failed_node);

-- ----
-- Move the sets
-- ----
for v_row in select S.set_id, (select count(*)
        from sl_subscribe SUB
        where S.set_id = SUB.sub_set
          and SUB.sub_receiver <> p_backup_node
          and SUB.sub_provider = p_failed_node)
        as num_direct_receivers
    from sl_set S
    where S.set_origin = p_failed_node
    for update
loop
  -- ----
  -- If the backup node is the only direct subscriber ...
  -- ----
  if v_row.num_direct_receivers = 0 then
          raise notice 'failedNode: set % has no other direct receivers - move now',  ↩
              v_row.set_id;
    -- ----
    -- backup_node is the only direct subscriber, move the set
    -- right now. On the backup node itself that includes restoring
    -- all user mode triggers, removing the protection trigger,
    -- adding the log trigger, removing the subscription and the
    -- obsolete setsync status.
    -- ----
    if p_backup_node = getLocalNodeId('_schemadoc') then
      update sl_set set set_origin = p_backup_node
          where set_id = v_row.set_id;

      delete from sl_setsync
          where ssy_setid = v_row.set_id;

      for v_row2 in select * from sl_table
          where tab_set = v_row.set_id
          order by tab_id
      loop
        perform alterTableConfigureTriggers(v_row2.tab_id);
      end loop;
    end if;
```

```
    delete from sl_subscribe
        where sub_set = v_row.set_id
          and sub_receiver = p_backup_node;
  else
    raise notice 'failedNode: set % has other direct receivers - change providers only',  ↩
        v_row.set_id;
    -- ----
    -- Backup node is not the only direct subscriber or not
    -- a direct subscriber at all.
    -- This means that at this moment, we redirect all possible
    -- direct subscribers to receive from the backup node, and the
    -- backup node itself to receive from another one.
    -- The admin utility will wait for the slon engine to
    -- restart and then call failedNode2() on the node with
    -- the highest SYNC and redirect this to it on
    -- backup node later.
    -- ----
    update sl_subscribe
        set sub_provider = (select min(SS.sub_receiver)
            from sl_subscribe SS
            where SS.sub_set = v_row.set_id
              and SS.sub_receiver <> p_backup_node
              and SS.sub_forward
              and exists (
                select 1 from sl_path
                  where pa_server = SS.sub_receiver
                    and pa_client = p_backup_node
              ))
        where sub_set = v_row.set_id
          and sub_receiver = p_backup_node;
      update sl_subscribe
                  set sub_provider = (select min(SS.sub_receiver)
                          from sl_subscribe SS
                          where SS.sub_set = v_row.set_id
                              and SS.sub_receiver <> p_failed_node
                              and SS.sub_forward
                              and exists (
                                  select 1 from sl_path
                                      where pa_server = SS.sub_receiver
                                        and pa_client = sl_subscribe.sub_receiver
                              ))
                  where sub_set = v_row.set_id
                      and sub_receiver <> p_backup_node;

    update sl_subscribe
        set sub_provider = p_backup_node
        where sub_set = v_row.set_id
          and sub_receiver <> p_backup_node
          and exists (
            select 1 from sl_path
              where pa_server = p_backup_node
                and pa_client = sl_subscribe.sub_receiver
          );
    delete from sl_subscribe
                 where sub_set = v_row.set_id
                     and sub_receiver = p_backup_node;

  end if;
end loop;
```

```
  -- Rewrite sl_listen table
  perform RebuildListenEntries();

  -- Run addPartialLogIndices() to try to add indices to unused sl_log_? table
  perform addPartialLogIndices();

  -- ----
  -- Make sure the node daemon will restart
  -- ----
  notify "_schemadoc_Restart";

  -- ----
  -- That is it - so far.
  -- ----
  return p_failed_node;
end;
```

13.70 failednode2(p_ev_seqfake integer, p_ev_seqno integer, p_set_id integer, p_backu bigint, p_failed_node bigint)

Function Properties
Language: PLPGSQL, *Return Type:* bigint FUNCTION failedNode2 (failed_node, backup_node, set_id, ev_seqno, ev_seqfake)
On the node that has the highest sequence number of the failed node, fake the FAILOVER_SET event.

```
declare
  v_row       record;
begin
  select * into v_row
      from sl_event
      where ev_origin = p_failed_node
      and ev_seqno = p_ev_seqno;
  if not found then
    raise exception 'Slony-I: event %,% not found',
        p_failed_node, p_ev_seqno;
  end if;

  insert into sl_event
      (ev_origin, ev_seqno, ev_timestamp,
      ev_snapshot,
      ev_type, ev_data1, ev_data2, ev_data3)
      values
      (p_failed_node, p_ev_seqfake, CURRENT_TIMESTAMP,
      v_row.ev_snapshot,
      'FAILOVER_SET', p_failed_node::text, p_backup_node::text,
      p_set_id::text);
  insert into sl_confirm
      (con_origin, con_received, con_seqno, con_timestamp)
      values
      (p_failed_node, getLocalNodeId('_schemadoc'),
      p_ev_seqfake, CURRENT_TIMESTAMP);
  notify "_schemadoc_Restart";

  perform failoverSet_int(p_failed_node,
      p_backup_node, p_set_id, p_ev_seqfake);

  return p_ev_seqfake;
end;
```

13.71 failoverset_int(p_wait_seqno integer, p_set_id integer, p_backup_node integer, p_failed_node bigint)

Function Properties
Language: PLPGSQL, *Return Type:* integer FUNCTION failoverSet_int (failed_node, backup_node, set_id, wait_seqno) Finish failover for one set.

```
declare
  v_row       record;
  v_last_sync     int8;
begin
  -- ----
  -- Change the origin of the set now to the backup node.
  -- On the backup node this includes changing all the
  -- trigger and protection stuff
  -- ----
  if p_backup_node = getLocalNodeId('_schemadoc') then
    delete from sl_setsync
        where ssy_setid = p_set_id;
    delete from sl_subscribe
        where sub_set = p_set_id
          and sub_receiver = p_backup_node;
    update sl_set
        set set_origin = p_backup_node
        where set_id = p_set_id;

    for v_row in select * from sl_table
        where tab_set = p_set_id
        order by tab_id
    loop
      perform alterTableConfigureTriggers(v_row.tab_id);
    end loop;
    insert into sl_event
        (ev_origin, ev_seqno, ev_timestamp,
        ev_snapshot,
        ev_type, ev_data1, ev_data2, ev_data3, ev_data4)
        values
        (p_backup_node, "pg_catalog".nextval('sl_event_seq'), CURRENT_TIMESTAMP,
        pg_catalog.txid_current_snapshot(),
        'ACCEPT_SET', p_set_id::text,
        p_failed_node::text, p_backup_node::text,
        p_wait_seqno::text);
  else
    delete from sl_subscribe
        where sub_set = p_set_id
          and sub_receiver = p_backup_node;
    update sl_set
        set set_origin = p_backup_node
        where set_id = p_set_id;
  end if;

  -- update sl_node
  --     set no_active=false WHERE
  --     no_id=p_failed_node;

  -- Rewrite sl_listen table
  perform RebuildListenEntries();

  -- ----
  -- If we are a subscriber of the set ourself, change our
  -- setsync status to reflect the new set origin.
```

```
  -- ----
  if exists (select true from sl_subscribe
      where sub_set = p_set_id
        and sub_receiver = getLocalNodeId(
            '_schemadoc'))
  then
    delete from sl_setsync
        where ssy_setid = p_set_id;

    select coalesce(max(ev_seqno), 0) into v_last_sync
        from sl_event
        where ev_origin = p_backup_node
          and ev_type = 'SYNC';
    if v_last_sync > 0 then
      insert into sl_setsync
          (ssy_setid, ssy_origin, ssy_seqno,
          ssy_snapshot, ssy_action_list)
          select p_set_id, p_backup_node, v_last_sync,
          ev_snapshot, NULL
          from sl_event
          where ev_origin = p_backup_node
            and ev_seqno = v_last_sync;
    else
      insert into sl_setsync
          (ssy_setid, ssy_origin, ssy_seqno,
          ssy_snapshot, ssy_action_list)
          values (p_set_id, p_backup_node, '0',
          '1:1:', NULL);
    end if;

  end if;

  return p_failed_node;
end;
```

13.72 finishtableaftercopy(p_tab_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer Reenable index maintenance and reindex the table

```
declare
  v_tab_oid   oid;
  v_tab_fqname  text;
begin
  -- ----
  -- Get the tables OID and fully qualified name
  -- ---
  select  PGC.oid,
      slon_quote_brute(PGN.nspname) || '.' ||
      slon_quote_brute(PGC.relname) as tab_fqname
    into v_tab_oid, v_tab_fqname
      from sl_table T,
        "pg_catalog".pg_class PGC, "pg_catalog".pg_namespace PGN
        where T.tab_id = p_tab_id
        and T.tab_reloid = PGC.oid
        and PGC.relnamespace = PGN.oid;
  if not found then
    raise exception 'Table with ID % not found in sl_table', p_tab_id;
  end if;
```

```
  -- ----
  -- Reenable indexes and reindex the table.
  -- ----
  perform enable_indexes_on_table(v_tab_oid);
  execute 'reindex table ' || slon_quote_input(v_tab_fqname);

  return 1;
end;
```

13.73 forwardconfirm(p_con_timestamp integer, p_con_seqno integer, p_con_received bigint, p_con_origin timestamp without time zone)

Function Properties
Language: PLPGSQL, *Return Type:* bigint forwardConfirm (p_con_origin, p_con_received, p_con_seqno, p_con_timestamp) Confirms (recorded in sl_confirm) that items from p_con_origin up to p_con_seqno have been received by node p_con_received as of p_con_timestamp, and raises an event to forward this confirmation.

```
declare
  v_max_seqno   bigint;
begin
  select into v_max_seqno coalesce(max(con_seqno), 0)
      from sl_confirm
      where con_origin = p_con_origin
      and con_received = p_con_received;
  if v_max_seqno < p_con_seqno then
    insert into sl_confirm
        (con_origin, con_received, con_seqno, con_timestamp)
        values (p_con_origin, p_con_received, p_con_seqno,
          p_con_timestamp);
    v_max_seqno = p_con_seqno;
  end if;

  return v_max_seqno;
end;
```

13.74 generate_sync_event(p_interval interval)

Function Properties
Language: PLPGSQL, *Return Type:* integer Generate a sync event if there has not been one in the requested interval, and this is a provider node.

```
declare
  v_node_row     record;

BEGIN
  select 1 into v_node_row from sl_event
          where ev_type = 'SYNC' and ev_origin = getLocalNodeId('_schemadoc')
          and ev_timestamp > now() - p_interval limit 1;
  if not found then
    -- If there has been no SYNC in the last interval, then push one
    perform createEvent('_schemadoc', 'SYNC', NULL);
    return 1;
  else
    return 0;
  end if;
```

```
end;
```

13.75 getlocalnodeid(p_cluster name)

Function Properties
Language: C, *Return Type:* integer Returns the node ID of the node being serviced on the local database

```
_Slony_I_getLocalNodeId
```

13.76 getmoduleversion()

Function Properties
Language: C, *Return Type:* text Returns the compiled-in version number of the Slony-I shared object

```
_Slony_I_getModuleVersion
```

13.77 initializelocalnode(p_comment integer, p_local_node_id text)

Function Properties
Language: PLPGSQL, *Return Type:* integer no_id - Node ID # no_comment - Human-oriented comment Initializes the new node, no_id

```
declare
  v_old_node_id   int4;
  v_first_log_no    int4;
  v_event_seq     int8;
begin
  -- ----
  -- Make sure this node is uninitialized or got reset
  -- ----
  select last_value::int4 into v_old_node_id from sl_local_node_id;
  if v_old_node_id != -1 then
    raise exception 'Slony-I: This node is already initialized';
  end if;

  -- ----
  -- Set sl_local_node_id to the requested value and add our
  -- own system to sl_node.
  -- ----
  perform setval('sl_local_node_id', p_local_node_id);
  perform storeNode_int (p_local_node_id, p_comment);

  if (pg_catalog.current_setting('max_identifier_length')::integer - pg_catalog.length(' ↩
      schemadoc')) < 5 then
    raise notice 'Slony-I: Cluster name length [%] versus system max_identifier_length [%]  ↩
        ', pg_catalog.length('schemadoc'), pg_catalog.current_setting('max_identifier_length ↩
        ');
    raise notice 'leaves narrow/no room for some Slony-I-generated objects (such as indexes ↩
        ).';
    raise notice 'You may run into problems later!';
  end if;

  return p_local_node_id;
end;
```

13.78 is_node_reachable(receiver_node_id integer, origin_node_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* boolean Is the receiver node reachable from the origin, via any of the listen paths?

```
declare
    listen_row record;
    reachable boolean;
begin
  reachable:=false;
  select * into listen_row from sl_listen where
       li_origin=origin_node_id and li_receiver=receiver_node_id;
  if found then
     reachable:=true;
  end if;
  return reachable;
end
```

13.79 issubscriptioninprogress(p_add_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* boolean Checks to see if a subscription for the indicated set is in progress. Returns true if a subscription is in progress. Otherwise false

```
begin
  if exists (select true from sl_event
      where ev_type = 'ENABLE_SUBSCRIPTION'
      and ev_data1 = p_add_id::text
      and ev_seqno > (select max(con_seqno) from sl_confirm
          where con_origin = ev_origin
          and con_received::text = ev_data3))
  then
    return true;
  else
    return false;
  end if;
end;
```

13.80 killbackend(p_signame integer, p_pid text)

Function Properties
Language: C, *Return Type:* integer Send a signal to a postgres process. Requires superuser rights

```
_Slony_I_killBackend
```

13.81 lockedset()

Function Properties
Language: C, *Return Type:* trigger Trigger function to prevent modifications to a table before and after a moveSet()

```
_Slony_I_lockedSet
```

13.82 lockset(p_set_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer lockSet(set_id) Add a special trigger to all tables of a set that disables access to it.

```
declare
  v_local_node_id   int4;
  v_set_row     record;
  v_tab_row     record;
begin
  -- ----
  -- Check that the set exists and that we are the origin
  -- and that it is not already locked.
  -- ----
  v_local_node_id := getLocalNodeId('_schemadoc');
  select * into v_set_row from sl_set
      where set_id = p_set_id
      for update;
  if not found then
    raise exception 'Slony-I: set % not found', p_set_id;
  end if;
  if v_set_row.set_origin <> v_local_node_id then
    raise exception 'Slony-I: set % does not originate on local node',
        p_set_id;
  end if;
  if v_set_row.set_locked notnull then
    raise exception 'Slony-I: set % is already locked', p_set_id;
  end if;

  -- ----
  -- Place the lockedSet trigger on all tables in the set.
  -- ----
  for v_tab_row in select T.tab_id,
      slon_quote_brute(PGN.nspname) || '.' ||
      slon_quote_brute(PGC.relname) as tab_fqname
      from sl_table T,
        "pg_catalog".pg_class PGC, "pg_catalog".pg_namespace PGN
      where T.tab_set = p_set_id
        and T.tab_reloid = PGC.oid
        and PGC.relnamespace = PGN.oid
      order by tab_id
  loop
    execute 'create trigger "_schemadoc_lockedset" ' ||
        'before insert or update or delete on ' ||
        v_tab_row.tab_fqname || ' for each row execute procedure
        lockedSet (''_schemadoc'');';
  end loop;

  -- ----
  -- Remember our snapshots xmax as for the set locking
  -- ----
  update sl_set
      set set_locked = "pg_catalog".txid_snapshot_xmax("pg_catalog".txid_current_snapshot() ↩
          )
      where set_id = p_set_id;

  return p_set_id;
end;
```

13.83 log_truncate()

Function Properties
Language: PLPGSQL, *Return Type:* trigger trigger function run when a replicated table receives a TRUNCATE request

```
  declare
    c_command text;
    c_log integer;
    c_node integer;
    c_tabid integer;
  begin
        c_tabid := tg_argv[0];
      c_node := getLocalNodeId('_schemadoc');
    c_command := 'TRUNCATE TABLE ONLY "' || tab_nspname || '"."' ||
          tab_relname || '" CASCADE'
          from sl_table where tab_id = c_tabid;
    select last_value into c_log from sl_log_status;
    if c_log in (0, 2) then
       insert into sl_log_1 (log_origin, log_txid, log_tableid, log_actionseq, log_cmdtype, ↩
            log_cmddata)
          values (c_node, pg_catalog.txid_current(), c_tabid, nextval('"_schemadoc". ↩
              sl_action_seq'), 'T', c_command);
    else   -- (1, 3)
       insert into sl_log_2 (log_origin, log_txid, log_tableid, log_actionseq, log_cmdtype, ↩
            log_cmddata)
          values (c_node, pg_catalog.txid_current(), c_tabid, nextval('"_schemadoc". ↩
              sl_action_seq'), 'T', c_command);
    end if;
    return NULL;
    end
```

13.84 logswitch_finish()

Function Properties
Language: PLPGSQL, *Return Type:* integer logswitch_finish() Attempt to finalize a log table switch in progress return values: -1 if switch in progress, but not complete 0 if no switch in progress 1 if performed truncate on sl_log_2 2 if performed truncate on sl_log_1

```
DECLARE
  v_current_status  int4;
  v_dummy       record;
  v_origin  int8;
  v_seqno   int8;
  v_xmin    bigint;
  v_purgeable boolean;
BEGIN
  -- ----
  -- Get the current log status.
  -- ----
  select last_value into v_current_status from sl_log_status;

  -- ----
  -- status value 0 or 1 means that there is no log switch in progress
  -- ----
  if v_current_status = 0 or v_current_status = 1 then
    return 0;
  end if;

  -- ----
```

```
-- status = 2: sl_log_1 active, cleanup sl_log_2
-- ----
if v_current_status = 2 then
  v_purgeable := 'true';

  -- ----
  -- Attempt to lock sl_log_2 in order to make sure there are no other transactions
  -- currently writing to it. Exit if it is still in use. This prevents TRUNCATE from
  -- blocking writers to sl_log_2 while it is waiting for a lock. It also prevents it
  -- immediately truncating log data generated inside the transaction which was active
  -- when logswitch_finish() was called (and was blocking TRUNCATE) as soon as that
  -- transaction is committed.
  -- ----
  begin
    lock table sl_log_2 in exclusive mode nowait;
  exception when lock_not_available then
    raise notice 'Slony-I: could not lock sl_log_2 - sl_log_2 not truncated';
    return -1;
  end;

  -- ----
  -- The cleanup thread calls us after it did the delete and
  -- vacuum of both log tables. If sl_log_2 is empty now, we
  -- can truncate it and the log switch is done.
  -- ----
        for v_origin, v_seqno, v_xmin in
    select ev_origin, ev_seqno, "pg_catalog".txid_snapshot_xmin(ev_snapshot) from  ↩
        sl_event
          where (ev_origin, ev_seqno) in (select ev_origin, min(ev_seqno) from sl_event  ↩
              where ev_type = 'SYNC' group by ev_origin)
  loop
    if exists (select 1 from sl_log_2 where log_origin = v_origin and log_txid >= v_xmin  ↩
        limit 1) then
      v_purgeable := 'false';
    end if;
        end loop;
  if not v_purgeable then
    -- ----
    -- Found a row ... log switch is still in progress.
    -- ----
    raise notice 'Slony-I: log switch to sl_log_1 still in progress - sl_log_2 not  ↩
        truncated';
    return -1;
  end if;

  raise notice 'Slony-I: log switch to sl_log_1 complete - truncate sl_log_2';
  truncate sl_log_2;
  if exists (select * from "pg_catalog".pg_class c, "pg_catalog".pg_namespace n, " ↩
      pg_catalog".pg_attribute a where c.relname = 'sl_log_2' and n.oid = c.relnamespace  ↩
      and a.attrelid = c.oid and a.attname = 'oid') then
                execute 'alter table sl_log_2 set without oids;';
  end if;
  perform "pg_catalog".setval('sl_log_status', 0);
  -- Run addPartialLogIndices() to try to add indices to unused sl_log_? table
  perform addPartialLogIndices();

  return 1;
end if;

-- ----
-- status = 3: sl_log_2 active, cleanup sl_log_1
-- ----
```

```
  if v_current_status = 3 then
    v_purgeable := 'true';

    -- ----
    -- Attempt to lock sl_log_1 in order to make sure there are no other transactions
    -- currently writing to it. Exit if it is still in use. This prevents TRUNCATE from
    -- blocking writes to sl_log_1 while it is waiting for a lock. It also prevents it
    -- immediately truncating log data generated inside the transaction which was active
    -- when logswitch_finish() was called (and was blocking TRUNCATE) as soon as that
    -- transaction is committed.
    -- ----
    begin
      lock table sl_log_1 in exclusive mode nowait;
    exception when lock_not_available then
      raise notice 'Slony-I: could not lock sl_log_1 - sl_log_1 not truncated';
      return -1;
    end;

    -- ----
    -- The cleanup thread calls us after it did the delete and
    -- vacuum of both log tables. If sl_log_2 is empty now, we
    -- can truncate it and the log switch is done.
    -- ----
          for v_origin, v_seqno, v_xmin in
      select ev_origin, ev_seqno, "pg_catalog".txid_snapshot_xmin(ev_snapshot) from  ↩
          sl_event
            where (ev_origin, ev_seqno) in (select ev_origin, min(ev_seqno) from sl_event  ↩
                where ev_type = 'SYNC' group by ev_origin)
    loop
      if (exists (select 1 from sl_log_1 where log_origin = v_origin and log_txid >= v_xmin ↩
           limit 1)) then
        v_purgeable := 'false';
      end if;
          end loop;
    if not v_purgeable then
      -- ----
      -- Found a row ... log switch is still in progress.
      -- ----
      raise notice 'Slony-I: log switch to sl_log_2 still in progress - sl_log_1 not  ↩
          truncated';
      return -1;
    end if;

    raise notice 'Slony-I: log switch to sl_log_2 complete - truncate sl_log_1';
    truncate sl_log_1;
    if exists (select * from "pg_catalog".pg_class c, "pg_catalog".pg_namespace n, " ↩
        pg_catalog".pg_attribute a where c.relname = 'sl_log_1' and n.oid = c.relnamespace  ↩
        and a.attrelid = c.oid and a.attname = 'oid') then
                  execute 'alter table sl_log_1 set without oids;';
    end if;
    perform "pg_catalog".setval('sl_log_status', 1);
    -- Run addPartialLogIndices() to try to add indices to unused sl_log_? table
    perform addPartialLogIndices();
    return 2;
  end if;
END;
```

13.85 logswitch_start()

Function Properties
Language: PLPGSQL, *Return Type:* integer logswitch_start() Initiate a log table switch if none is in progress

```
DECLARE
  v_current_status  int4;
BEGIN
  -- ----
  -- Get the current log status.
  -- ----
  select last_value into v_current_status from sl_log_status;

  -- ----
  -- status = 0: sl_log_1 active, sl_log_2 clean
  -- Initiate a switch to sl_log_2.
  -- ----
  if v_current_status = 0 then
    perform "pg_catalog".setval('sl_log_status', 3);
    perform registry_set_timestamp(
        'logswitch.laststart', now());
    raise notice 'Slony-I: Logswitch to sl_log_2 initiated';
    return 2;
  end if;

  -- ----
  -- status = 1: sl_log_2 active, sl_log_1 clean
  -- Initiate a switch to sl_log_1.
  -- ----
  if v_current_status = 1 then
    perform "pg_catalog".setval('sl_log_status', 2);
    perform registry_set_timestamp(
        'logswitch.laststart', now());
    raise notice 'Slony-I: Logswitch to sl_log_1 initiated';
    return 1;
  end if;

  raise exception 'Previous logswitch still in progress';
END;
```

13.86 logtrigger()

Function Properties
Language: C, *Return Type:* trigger This is the trigger that is executed on the origin node that causes updates to be recorded in sl_log_1/sl_log_2.

```
_Slony_I_logTrigger
```

13.87 mergeset(p_add_id integer, p_set_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* bigint Generate MERGE_SET event to request that sets be merged together. Both sets must exist, and originate on the same node. They must be subscribed by the same set of nodes.

```
declare
  v_origin      int4;
```

```
  in_progress     boolean;
begin
  -- ----
  -- Check that both sets exist and originate here
  -- ----
  if p_set_id = p_add_id then
    raise exception 'Slony-I: merged set ids cannot be identical';
  end if;
  select set_origin into v_origin from sl_set
      where set_id = p_set_id;
  if not found then
    raise exception 'Slony-I: set % not found', p_set_id;
  end if;
  if v_origin != getLocalNodeId('_schemadoc') then
    raise exception 'Slony-I: set % does not originate on local node',
        p_set_id;
  end if;

  select set_origin into v_origin from sl_set
      where set_id = p_add_id;
  if not found then
    raise exception 'Slony-I: set % not found', p_add_id;
  end if;
  if v_origin != getLocalNodeId('_schemadoc') then
    raise exception 'Slony-I: set % does not originate on local node',
        p_add_id;
  end if;

  -- ----
  -- Check that both sets are subscribed by the same set of nodes
  -- ----
  if exists (select true from sl_subscribe SUB1
        where SUB1.sub_set = p_set_id
        and SUB1.sub_receiver not in (select SUB2.sub_receiver
            from sl_subscribe SUB2
            where SUB2.sub_set = p_add_id))
  then
    raise exception 'Slony-I: subscriber lists of set % and % are different',
        p_set_id, p_add_id;
  end if;

  if exists (select true from sl_subscribe SUB1
        where SUB1.sub_set = p_add_id
        and SUB1.sub_receiver not in (select SUB2.sub_receiver
            from sl_subscribe SUB2
            where SUB2.sub_set = p_set_id))
  then
    raise exception 'Slony-I: subscriber lists of set % and % are different',
        p_add_id, p_set_id;
  end if;

  -- ----
  -- Check that all ENABLE_SUBSCRIPTION events for the set are confirmed
  -- ----
  select isSubscriptionInProgress(p_add_id) into in_progress ;

  if in_progress then
    raise exception 'Slony-I: set % has subscriptions in progress - cannot merge',
        p_add_id;
  end if;

  -- ----
```

```
  -- Create a SYNC event, merge the sets, create a MERGE_SET event
  -- ----
  perform createEvent('_schemadoc', 'SYNC', NULL);
  perform mergeSet_int(p_set_id, p_add_id);
  return  createEvent('_schemadoc', 'MERGE_SET',
      p_set_id::text, p_add_id::text);
end;
```

13.88 mergeset_int(p_add_id integer, p_set_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer mergeSet_int(set_id, add_id) - Perform MERGE_SET event, merging all objects from set add_id into set set_id.

```
begin
  update sl_sequence
      set seq_set = p_set_id
      where seq_set = p_add_id;
  update sl_table
      set tab_set = p_set_id
      where tab_set = p_add_id;
  delete from sl_subscribe
      where sub_set = p_add_id;
  delete from sl_setsync
      where ssy_setid = p_add_id;
  delete from sl_set
      where set_id = p_add_id;

  return p_set_id;
end;
```

13.89 moveset(p_new_origin integer, p_set_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* bigint moveSet(set_id, new_origin) Generate MOVE_SET event to request that the origin for set set_id be moved to node new_origin

```
declare
  v_local_node_id   int4;
  v_set_row     record;
  v_sub_row     record;
  v_sync_seqno    int8;
  v_lv_row      record;
begin
  -- ----
  -- Check that the set is locked and that this locking
  -- happened long enough ago.
  -- ----
  v_local_node_id := getLocalNodeId('_schemadoc');
  select * into v_set_row from sl_set
      where set_id = p_set_id
      for update;
  if not found then
    raise exception 'Slony-I: set % not found', p_set_id;
  end if;
  if v_set_row.set_origin <> v_local_node_id then
```

```
    raise exception 'Slony-I: set % does not originate on local node',
        p_set_id;
  end if;
  if v_set_row.set_locked isnull then
    raise exception 'Slony-I: set % is not locked', p_set_id;
  end if;
  if v_set_row.set_locked > "pg_catalog".txid_snapshot_xmin("pg_catalog". ↩
      txid_current_snapshot()) then
    raise exception 'Slony-I: cannot move set % yet, transactions < % are still in progress ↩
        ',
        p_set_id, v_set_row.set_locked;
  end if;

  -- ----
  -- Unlock the set
  -- ----
  perform unlockSet(p_set_id);

  -- ----
  -- Check that the new_origin is an active subscriber of the set
  -- ----
  select * into v_sub_row from sl_subscribe
      where sub_set = p_set_id
      and sub_receiver = p_new_origin;
  if not found then
    raise exception 'Slony-I: set % is not subscribed by node %',
        p_set_id, p_new_origin;
  end if;
  if not v_sub_row.sub_active then
    raise exception 'Slony-I: subsctiption of node % for set % is inactive',
        p_new_origin, p_set_id;
  end if;

  -- ----
  -- Reconfigure everything
  -- ----
  perform moveSet_int(p_set_id, v_local_node_id,
      p_new_origin, 0);

  perform RebuildListenEntries();

  -- ----
  -- At this time we hold access exclusive locks for every table
  -- in the set. But we did move the set to the new origin, so the
  -- createEvent() we are doing now will not record the sequences.
  -- ----
  v_sync_seqno := createEvent('_schemadoc', 'SYNC');
  insert into sl_seqlog
      (seql_seqid, seql_origin, seql_ev_seqno, seql_last_value)
      select seq_id, v_local_node_id, v_sync_seqno, seq_last_value
      from sl_seqlastvalue
      where seq_set = p_set_id;

  -- ----
  -- Finally we generate the real event
  -- ----
  return createEvent('_schemadoc', 'MOVE_SET',
      p_set_id::text, v_local_node_id::text, p_new_origin::text);
end;
```

13.90 moveset_int(p_wait_seqno integer, p_new_origin integer, p_old_origin integer, p_set_id bigint)

Function Properties
Language: PLPGSQL, *Return Type:* integer moveSet(set_id, old_origin, new_origin, wait_seqno) Process MOVE_SET event to request that the origin for set set_id be moved from old_origin to node new_origin

```
declare
  v_local_node_id   int4;
  v_tab_row     record;
  v_sub_row     record;
  v_sub_node      int4;
  v_sub_last      int4;
  v_sub_next      int4;
  v_last_sync     int8;
begin
  -- ----
  -- Get our local node ID
  -- ----
  v_local_node_id := getLocalNodeId('_schemadoc');

  -- On the new origin, raise an event - ACCEPT_SET
  if v_local_node_id = p_new_origin then
    -- Create a SYNC event as well so that the ACCEPT_SET has
    -- the same snapshot as the last SYNC generated by the new
    -- origin. This snapshot will be used by other nodes to
    -- finalize the setsync status.
    perform createEvent('_schemadoc', 'SYNC', NULL);
    perform createEvent('_schemadoc', 'ACCEPT_SET',
      p_set_id::text, p_old_origin::text,
      p_new_origin::text, p_wait_seqno::text);
  end if;

  -- ----
  -- Next we have to reverse the subscription path
  -- ----
  v_sub_last = p_new_origin;
  select sub_provider into v_sub_node
      from sl_subscribe
      where sub_set = p_set_id
      and sub_receiver = p_new_origin;
  if not found then
    raise exception 'Slony-I: subscription path broken in moveSet_int';
  end if;
  while v_sub_node <> p_old_origin loop
    -- ----
    -- Tracing node by node, the old receiver is now in
    -- v_sub_last and the old provider is in v_sub_node.
    -- ----

    -- ----
    -- Get the current provider of this node as next
    -- and change the provider to the previous one in
    -- the reverse chain.
    -- ----
    select sub_provider into v_sub_next
        from sl_subscribe
        where sub_set = p_set_id
          and sub_receiver = v_sub_node
        for update;
    if not found then
```

```
    raise exception 'Slony-I: subscription path broken in moveSet_int';
  end if;
  update sl_subscribe
      set sub_provider = v_sub_last
      where sub_set = p_set_id
        and sub_receiver = v_sub_node;

  v_sub_last = v_sub_node;
  v_sub_node = v_sub_next;
end loop;

-- ----
-- This includes creating a subscription for the old origin
-- ----
insert into sl_subscribe
    (sub_set, sub_provider, sub_receiver,
    sub_forward, sub_active)
    values (p_set_id, v_sub_last, p_old_origin, true, true);
if v_local_node_id = p_old_origin then
  select coalesce(max(ev_seqno), 0) into v_last_sync
      from sl_event
      where ev_origin = p_new_origin
        and ev_type = 'SYNC';
  if v_last_sync > 0 then
    insert into sl_setsync
        (ssy_setid, ssy_origin, ssy_seqno,
        ssy_snapshot, ssy_action_list)
        select p_set_id, p_new_origin, v_last_sync,
        ev_snapshot, NULL
        from sl_event
        where ev_origin = p_new_origin
          and ev_seqno = v_last_sync;
  else
    insert into sl_setsync
        (ssy_setid, ssy_origin, ssy_seqno,
        ssy_snapshot, ssy_action_list)
        values (p_set_id, p_new_origin, '0',
        '1:1:', NULL);
  end if;
end if;

-- ----
-- Now change the ownership of the set.
-- ----
update sl_set
    set set_origin = p_new_origin
    where set_id = p_set_id;

-- ----
-- On the new origin, delete the obsolete setsync information
-- and the subscription.
-- ----
if v_local_node_id = p_new_origin then
  delete from sl_setsync
      where ssy_setid = p_set_id;
else
  if v_local_node_id <> p_old_origin then
    --
    -- On every other node, change the setsync so that it will
    -- pick up from the new origins last known sync.
    --
    delete from sl_setsync
```

```
          where ssy_setid = p_set_id;
      select coalesce(max(ev_seqno), 0) into v_last_sync
          from sl_event
          where ev_origin = p_new_origin
            and ev_type = 'SYNC';
      if v_last_sync > 0 then
        insert into sl_setsync
            (ssy_setid, ssy_origin, ssy_seqno,
            ssy_snapshot, ssy_action_list)
            select p_set_id, p_new_origin, v_last_sync,
            ev_snapshot, NULL
            from sl_event
            where ev_origin = p_new_origin
              and ev_seqno = v_last_sync;
      else
        insert into sl_setsync
            (ssy_setid, ssy_origin, ssy_seqno,
            ssy_snapshot, ssy_action_list)
            values (p_set_id, p_new_origin,
            '0', '1:1:', NULL);
      end if;
    end if;
  end if;
  delete from sl_subscribe
      where sub_set = p_set_id
      and sub_receiver = p_new_origin;

  -- Regenerate sl_listen since we revised the subscriptions
  perform RebuildListenEntries();

  -- Run addPartialLogIndices() to try to add indices to unused sl_log_? table
  perform addPartialLogIndices();

  -- ----
  -- If we are the new or old origin, we have to
  -- adjust the log and deny access trigger configuration.
  -- ----
  if v_local_node_id = p_old_origin or v_local_node_id = p_new_origin then
    for v_tab_row in select tab_id from sl_table
        where tab_set = p_set_id
        order by tab_id
    loop
      perform alterTableConfigureTriggers(v_tab_row.tab_id);
    end loop;
  end if;

  return p_set_id;
end;
```

13.91 preparetableforcopy(p_tab_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer Delete all data and suppress index maintenance

```
declare
  v_tab_oid   oid;
  v_tab_fqname  text;
begin
  -- ----
```

```
  -- Get the OID and fully qualified name for the table
  -- ---
  select  PGC.oid,
      slon_quote_brute(PGN.nspname) || '.' ||
      slon_quote_brute(PGC.relname) as tab_fqname
    into v_tab_oid, v_tab_fqname
      from sl_table T,
        "pg_catalog".pg_class PGC, "pg_catalog".pg_namespace PGN
        where T.tab_id = p_tab_id
        and T.tab_reloid = PGC.oid
        and PGC.relnamespace = PGN.oid;
  if not found then
    raise exception 'Table with ID % not found in sl_table', p_tab_id;
  end if;

  -- ----
  -- Try using truncate to empty the table and fallback to
  -- delete on error.
  -- ----
  perform TruncateOnlyTable(v_tab_fqname);
  raise notice 'truncate of % succeeded', v_tab_fqname;

  -- suppress index activity
        perform disable_indexes_on_table(v_tab_oid);

  return 1;
  exception when others then
    raise notice 'truncate of % failed - doing delete', v_tab_fqname;
    perform disable_indexes_on_table(v_tab_oid);
    execute 'delete from only ' || slon_quote_input(v_tab_fqname);
    return 0;
end;
```

13.92 rebuildlistenentries()

Function Properties
Language: PLPGSQL, *Return Type:* integer RebuildListenEntries() Invoked by various subscription and path modifying functions, this rewrites the sl_listen entries, adding in all the ones required to allow communications between nodes in the Slony-I cluster.

```
declare
  v_row record;
begin
  -- First remove the entire configuration
  delete from sl_listen;

  -- Second populate the sl_listen configuration with a full
  -- network of all possible paths.
  insert into sl_listen
        (li_origin, li_provider, li_receiver)
      select pa_server, pa_server, pa_client from sl_path;
  while true loop
    insert into sl_listen
          (li_origin, li_provider, li_receiver)
      select distinct li_origin, pa_server, pa_client
        from sl_listen, sl_path
        where li_receiver = pa_server
          and li_origin <> pa_client
      except
```

```
    select li_origin, li_provider, li_receiver
      from sl_listen;

  if not found then
    exit;
  end if;
end loop;

-- We now replace specific event-origin,receiver combinations
-- with a configuration that tries to avoid events arriving at
-- a node before the data provider actually has the data ready.

-- Loop over every possible pair of receiver and event origin
for v_row in select N1.no_id as receiver, N2.no_id as origin
    from sl_node as N1, sl_node as N2
    where N1.no_id <> N2.no_id
loop
  -- 1st choice:
  -- If we use the event origin as a data provider for any
  -- set that originates on that very node, we are a direct
  -- subscriber to that origin and listen there only.
  if exists (select true from sl_set, sl_subscribe        , sl_node p
      where set_origin = v_row.origin
        and sub_set = set_id
        and sub_provider = v_row.origin
        and sub_receiver = v_row.receiver
        and sub_active
        and p.no_active
        and p.no_id=sub_provider
        )
  then
    delete from sl_listen
      where li_origin = v_row.origin
        and li_receiver = v_row.receiver;
    insert into sl_listen (li_origin, li_provider, li_receiver)
      values (v_row.origin, v_row.origin, v_row.receiver);
    continue;
  end if;

  -- 2nd choice:
  -- If we are subscribed to any set originating on this
  -- event origin, we want to listen on all data providers
  -- we use for this origin. We are a cascaded subscriber
  -- for sets from this node.
  if exists (select true from sl_set, sl_subscribe
          where set_origin = v_row.origin
            and sub_set = set_id
            and sub_receiver = v_row.receiver
            and sub_active)
  then
    delete from sl_listen
        where li_origin = v_row.origin
          and li_receiver = v_row.receiver;
    insert into sl_listen (li_origin, li_provider, li_receiver)
        select distinct set_origin, sub_provider, v_row.receiver
          from sl_set, sl_subscribe
          where set_origin = v_row.origin
            and sub_set = set_id
            and sub_receiver = v_row.receiver
            and sub_active;
    continue;
  end if;
```

```
  end loop ;

  return null ;
end ;
```

13.93 recreate_log_trigger(p_tab_attkind text, p_tab_id oid, p_fq_table_name text)

Function Properties
Language: PLPGSQL, *Return Type:* integer A function that drops and recreates the log trigger on the specified table. It is intended to be used after the primary_key/unique index has changed.

```
begin
  execute 'drop trigger "_schemadoc_logtrigger" on ' ||
    p_fq_table_name ;
    -- ----
  execute 'create trigger "_schemadoc_logtrigger"' ||
      ' after insert or update or delete on ' ||
      p_fq_table_name
      || ' for each row execute procedure logTrigger (' ||
                              pg_catalog.quote_literal('_schemadoc') || ',' ||
        pg_catalog.quote_literal(p_tab_id::text) || ',' ||
        pg_catalog.quote_literal(p_tab_attkind) || ');';
  return 0;
end
```

13.94 registernodeconnection(p_nodeid integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer Register (uniquely) the node connection so that only one slon can service the node

```
begin
  insert into sl_nodelock
    (nl_nodeid, nl_backendpid)
    values
    (p_nodeid, pg_backend_pid());

  return 0;
end;
```

13.95 registry_get_int4(p_default text, p_key integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer registry_get_int4(key, value) Get a registry value. If not present, set and return the default.

```
DECLARE
  v_value   int4;
BEGIN
  select reg_int4 into v_value from sl_registry
      where reg_key = p_key;
  if not found then
```

```
    v_value = p_default;
    if p_default notnull then
      perform registry_set_int4(p_key, p_default);
    end if;
  else
    if v_value is null then
      raise exception 'Slony-I: registry key % is not an int4 value',
          p_key;
    end if;
  end if;
  return v_value;
END;
```

13.96 registry_get_text(p_default text, p_key text)

Function Properties
Language: PLPGSQL, *Return Type:* text registry_get_text(key, value) Get a registry value. If not present, set and return the default.

```
DECLARE
  v_value   text;
BEGIN
  select reg_text into v_value from sl_registry
      where reg_key = p_key;
  if not found then
    v_value = p_default;
    if p_default notnull then
      perform registry_set_text(p_key, p_default);
    end if;
  else
    if v_value is null then
      raise exception 'Slony-I: registry key % is not a text value',
          p_key;
    end if;
  end if;
  return v_value;
END;
```

13.97 registry_get_timestamp(p_default text, p_key timestamp with time zone)

Function Properties
Language: PLPGSQL, *Return Type:* timestamp without time zone registry_get_timestamp(key, value) Get a registry value. If not present, set and return the default.

```
DECLARE
  v_value   timestamp;
BEGIN
  select reg_timestamp into v_value from sl_registry
      where reg_key = p_key;
  if not found then
    v_value = p_default;
    if p_default notnull then
      perform registry_set_timestamp(p_key, p_default);
    end if;
  else
    if v_value is null then
```

```
      raise exception 'Slony-I: registry key % is not an timestamp value',
          p_key;
    end if;
  end if;
  return v_value;
END;
```

13.98 registry_set_int4(p_value text, p_key integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer registry_set_int4(key, value) Set or delete a registry value

```
BEGIN
  if p_value is null then
    delete from sl_registry
        where reg_key = p_key;
  else
    lock table sl_registry;
    update sl_registry
        set reg_int4 = p_value
        where reg_key = p_key;
    if not found then
      insert into sl_registry (reg_key, reg_int4)
          values (p_key, p_value);
    end if;
  end if;
  return p_value;
END;
```

13.99 registry_set_text(p_value text, p_key text)

Function Properties
Language: PLPGSQL, *Return Type:* text registry_set_text(key, value) Set or delete a registry value

```
BEGIN
  if p_value is null then
    delete from sl_registry
        where reg_key = p_key;
  else
    lock table sl_registry;
    update sl_registry
        set reg_text = p_value
        where reg_key = p_key;
    if not found then
      insert into sl_registry (reg_key, reg_text)
          values (p_key, p_value);
    end if;
  end if;
  return p_value;
END;
```

13.100 registry_set_timestamp(p_value text, p_key timestamp with time zone)

Function Properties

Language: PLPGSQL, *Return Type:* timestamp without time zone registry_set_timestamp(key, value) Set or delete a registry value

```
BEGIN
  if p_value is null then
    delete from sl_registry
        where reg_key = p_key;
  else
    lock table sl_registry;
    update sl_registry
        set reg_timestamp = p_value
        where reg_key = p_key;
    if not found then
      insert into sl_registry (reg_key, reg_timestamp)
          values (p_key, p_value);
    end if;
  end if;
  return p_value;
END;
```

13.101 repair_log_triggers(only_locked boolean)

Function Properties
Language: PLPGSQL, *Return Type:* integer repair the log triggers as required. If only_locked is true then only tables that are already exclusivly locked by the current transaction are repaired. Otherwise all replicated tables with outdated trigger arguments are recreated.

```
declare
  retval integer;
  table_row record;
begin
  retval=0;
  for table_row in
    select  tab_nspname,tab_relname,
        tab_idxname, tab_id, mode,
        determineAttKindUnique(tab_nspname||
          '.'||tab_relname,tab_idxname) as attkind
    from
        sl_table
        left join
        pg_locks on (relation=tab_reloid and pid=pg_backend_pid()
        and mode='AccessExclusiveLock')
        ,pg_trigger
    where tab_reloid=tgrelid and
    determineAttKindUnique(tab_nspname||'.'
            ||tab_relname,tab_idxname)
      !=(decode_tgargs(tgargs))[2]
      and tgname =  '_schemadoc'
      || '_logtrigger'
    LOOP
        if (only_locked=false) or table_row.mode='AccessExclusiveLock' then
           perform recreate_log_trigger
               (table_row.tab_nspname||'.'||table_row.tab_relname,
               table_row.tab_id,table_row.attkind);
          retval=retval+1;
        else
           raise notice '%.% has an invalid configuration on the log trigger. This was not ↩
               corrected because only_lock is true and the table is not locked.',
           table_row.tab_nspname,table_row.tab_relname;
```

```
        end if;
    end loop;
  return retval;
end
```

13.102 replicate_partition(p_comment integer, p_idxname text, p_tabname text, p_nspname text, p_tab_id text)

Function Properties
Language: PLPGSQL, *Return Type:* bigint Add a partition table to replication. tab_idxname is optional - if NULL, then we use the primary key. This function looks up replication configuration via the parent table. Note that this function is to be run within an EXECUTE SCRIPT script, so it runs at the right place in the transaction stream on all nodes.

```
declare
  prec record;
  prec2 record;
  v_set_id int4;

begin
-- Look up the parent table; fail if it does not exist
   select c1.oid into prec from pg_catalog.pg_class c1, pg_catalog.pg_class c2, pg_catalog. ↩
       pg_inherits i, pg_catalog.pg_namespace n where c1.oid = i.inhparent  and c2.oid = i. ↩
       inhrelid and n.oid = c2.relnamespace and n.nspname = p_nspname and c2.relname =  ↩
       p_tabname;
   if not found then
  raise exception 'replicate_partition: No parent table found for %.%!', p_nspname,  ↩
      p_tabname;
   end if;

-- The parent table tells us what replication set to use
   select tab_set into prec2 from sl_table where tab_reloid = prec.oid;
   if not found then
  raise exception 'replicate_partition: Parent table % for new partition %.% is not  ↩
      replicated!', prec.oid, p_nspname, p_tabname;
   end if;

   v_set_id := prec2.tab_set;

-- Now, we have all the parameters necessary to run add_empty_table_to_replication...
   return add_empty_table_to_replication(v_set_id, p_tab_id, p_nspname, p_tabname,  ↩
       p_idxname, p_comment);
end
```

13.103 resetsession()

Function Properties
Language: C, *Return Type:* text

```
_Slony_I_resetSession
```

13.104 reshapesubscription(p_sub_receiver integer, p_sub_provider integer, p_sub_se integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer Run on a receiver/subscriber node when the provider for that subscription is being changed. Slonik will invoke this method before the SUBSCRIBE_SET event propogates to the receiver so listen paths can be updated.

```
begin
  update sl_subscribe set sub_provider=p_sub_provider
       WHERE sub_set=p_sub_set AND sub_receiver=p_sub_receiver;
  if found then
     perform RebuildListenEntries();
     notify "_schemadoc_Restart";
  end if;
  return 0;
end
```

13.105 seqtrack(p_seqval integer, p_seqid bigint)

Function Properties
Language: C, *Return Type:* bigint Returns NULL if seqval has not changed since the last call for seqid

```
_Slony_I_seqtrack
```

13.106 sequencelastvalue(p_seqname text)

Function Properties
Language: PLPGSQL, *Return Type:* bigint sequenceLastValue(p_seqname) Utility function used in sl_seqlastvalue view to compactly get the last value from the requested sequence.

```
declare
  v_seq_row record;
begin
  for v_seq_row in execute 'select last_value from ' || slon_quote_input(p_seqname)
  loop
    return v_seq_row.last_value;
  end loop;

  -- not reached
end;
```

13.107 sequencesetvalue(p_last_value integer, p_ev_seqno integer, p_seq_origin bigint, p_seq_id bigint)

Function Properties
Language: PLPGSQL, *Return Type:* integer sequenceSetValue (seq_id, seq_origin, ev_seqno, last_value) Set sequence seq_id to have new value last_value.

```
declare
  v_fqname      text;
begin
```

```
  -- ----
  -- Get the sequences fully qualified name
  -- ----
  select slon_quote_brute(PGN.nspname) || '.' ||
      slon_quote_brute(PGC.relname) into v_fqname
    from sl_sequence SQ,
      "pg_catalog".pg_class PGC, "pg_catalog".pg_namespace PGN
    where SQ.seq_id = p_seq_id
      and SQ.seq_reloid = PGC.oid
      and PGC.relnamespace = PGN.oid;
  if not found then
    raise exception 'Slony-I: sequenceSetValue(): sequence % not found', p_seq_id;
  end if;

  -- ----
  -- Update it to the new value
  -- ----
  execute 'select setval(''' || v_fqname ||
      ''', ' || p_last_value::text || ')';

  insert into sl_seqlog
      (seql_seqid, seql_origin, seql_ev_seqno, seql_last_value)
      values (p_seq_id, p_seq_origin, p_ev_seqno, p_last_value);

  return p_seq_id;
end;
```

13.108 setaddsequence(p_seq_comment integer, p_fqname integer, p_seq_id text, p_set_id text)

Function Properties
Language: PLPGSQL, *Return Type:* bigint setAddSequence (set_id, seq_id, seq_fqname, seq_comment) On the origin node for set set_id, add sequence seq_fqname to the replication set, and raise SET_ADD_SEQUENCE to cause this to replicate to subscriber nodes.

```
declare
  v_set_origin    int4;
begin
  -- ----
  -- Check that we are the origin of the set
  -- ----
  select set_origin into v_set_origin
      from sl_set
      where set_id = p_set_id;
  if not found then
    raise exception 'Slony-I: setAddSequence(): set % not found', p_set_id;
  end if;
  if v_set_origin != getLocalNodeId('_schemadoc') then
    raise exception 'Slony-I: setAddSequence(): set % has remote origin - submit to origin ↩
       node', p_set_id;
  end if;

  if exists (select true from sl_subscribe
      where sub_set = p_set_id)
  then
    raise exception 'Slony-I: cannot add sequence to currently subscribed set %',
        p_set_id;
  end if;
```

```
  -- ----
  -- Add the sequence to the set and generate the SET_ADD_SEQUENCE event
  -- ----
  perform setAddSequence_int(p_set_id, p_seq_id, p_fqname,
      p_seq_comment);
  return  createEvent('_schemadoc', 'SET_ADD_SEQUENCE',
            p_set_id::text, p_seq_id::text,
            p_fqname::text, p_seq_comment::text);
end;
```

13.109 setaddsequence_int(p_seq_comment integer, p_fqname integer, p_seq_id text, p_set_id text)

Function Properties
Language: PLPGSQL, *Return Type:* integer setAddSequence_int (set_id, seq_id, seq_fqname, seq_comment) This processes the SET_ADD_SEQUENCE event. On remote nodes that subscribe to set_id, add the sequence to the replication set.

```
declare
  v_local_node_id   int4;
  v_set_origin    int4;
  v_sub_provider    int4;
  v_relkind     char;
  v_seq_reloid    oid;
  v_seq_relname   name;
  v_seq_nspname   name;
  v_sync_row      record;
begin
  -- ----
  -- For sets with a remote origin, check that we are subscribed
  -- to that set. Otherwise we ignore the sequence because it might
  -- not even exist in our database.
  -- ----
  v_local_node_id := getLocalNodeId('_schemadoc');
  select set_origin into v_set_origin
      from sl_set
      where set_id = p_set_id;
  if not found then
    raise exception 'Slony-I: setAddSequence_int(): set % not found',
        p_set_id;
  end if;
  if v_set_origin != v_local_node_id then
    select sub_provider into v_sub_provider
        from sl_subscribe
        where sub_set = p_set_id
        and sub_receiver = getLocalNodeId('_schemadoc');
    if not found then
      return 0;
    end if;
  end if;

  -- ----
  -- Get the sequences OID and check that it is a sequence
  -- ----
  select PGC.oid, PGC.relkind, PGC.relname, PGN.nspname
    into v_seq_reloid, v_relkind, v_seq_relname, v_seq_nspname
      from "pg_catalog".pg_class PGC, "pg_catalog".pg_namespace PGN
      where PGC.relnamespace = PGN.oid
      and slon_quote_input(p_fqname) = slon_quote_brute(PGN.nspname) ||
          '.' || slon_quote_brute(PGC.relname);
```

```
  if not found then
    raise exception 'Slony-I: setAddSequence_int(): sequence % not found',
        p_fqname;
  end if;
  if v_relkind != 'S' then
    raise exception 'Slony-I: setAddSequence_int(): % is not a sequence',
        p_fqname;
  end if;

        select 1 into v_sync_row from sl_sequence where seq_id = p_seq_id;
  if not found then
              v_relkind := 'o';   -- all is OK
        else
                raise exception 'Slony-I: setAddSequence_int(): sequence ID % has already  ↩
                    been assigned', p_seq_id;
        end if;

  -- ----
  -- Add the sequence to sl_sequence
  -- ----
  insert into sl_sequence
    (seq_id, seq_reloid, seq_relname, seq_nspname, seq_set, seq_comment)
    values
    (p_seq_id, v_seq_reloid, v_seq_relname, v_seq_nspname,  p_set_id, p_seq_comment);

  -- ----
  -- On the set origin, fake a sl_seqlog row for the last sync event
  -- ----
  if v_set_origin = v_local_node_id then
    for v_sync_row in select coalesce (max(ev_seqno), 0) as ev_seqno
        from sl_event
        where ev_origin = v_local_node_id
          and ev_type = 'SYNC'
    loop
      insert into sl_seqlog
          (seql_seqid, seql_origin, seql_ev_seqno,
          seql_last_value) values
          (p_seq_id, v_local_node_id, v_sync_row.ev_seqno,
          sequenceLastValue(p_fqname));
    end loop;
  end if;

  return p_seq_id;
end;
```

13.110 setaddtable(p_tab_comment integer, p_tab_idxname integer, p_fqname text, p_tab_id name, p_set_id text)

Function Properties
Language: PLPGSQL, *Return Type:* bigint setAddTable (set_id, tab_id, tab_fqname, tab_idxname, tab_comment) Add table tab_fqname to replication set on origin node, and generate SET_ADD_TABLE event to allow this to propagate to other nodes. Note that the table id, tab_id, must be unique ACROSS ALL SETS.

```
declare
  v_set_origin    int4;
begin
  -- ----
  -- Check that we are the origin of the set
  -- ----
```

```
  select set_origin into v_set_origin
      from sl_set
      where set_id = p_set_id;
  if not found then
    raise exception 'Slony-I: setAddTable(): set % not found', p_set_id;
  end if;
  if v_set_origin != getLocalNodeId('_schemadoc') then
    raise exception 'Slony-I: setAddTable(): set % has remote origin', p_set_id;
  end if;

  if exists (select true from sl_subscribe
      where sub_set = p_set_id)
  then
    raise exception 'Slony-I: cannot add table to currently subscribed set % - must attach
        to an unsubscribed set',
        p_set_id;
  end if;

  -- ----
  -- Add the table to the set and generate the SET_ADD_TABLE event
  -- ----
  perform setAddTable_int(p_set_id, p_tab_id, p_fqname,
      p_tab_idxname, p_tab_comment);
  return  createEvent('_schemadoc', 'SET_ADD_TABLE',
      p_set_id::text, p_tab_id::text, p_fqname::text,
      p_tab_idxname::text, p_tab_comment::text);
end;
```

13.111 setaddtable_int(p_tab_comment integer, p_tab_idxname integer, p_fqname text, p_tab_id name, p_set_id text)

Function Properties
Language: PLPGSQL, *Return Type:* integer setAddTable_int (set_id, tab_id, tab_fqname, tab_idxname, tab_comment) This function processes the SET_ADD_TABLE event on remote nodes, adding a table to replication if the remote node is subscribing to its replication set.

```
declare
  v_tab_relname   name;
  v_tab_nspname   name;
  v_local_node_id   int4;
  v_set_origin    int4;
  v_sub_provider    int4;
  v_relkind   char;
  v_tab_reloid    oid;
  v_pkcand_nn   boolean;
  v_prec      record;
begin
  -- ----
  -- For sets with a remote origin, check that we are subscribed
  -- to that set. Otherwise we ignore the table because it might
  -- not even exist in our database.
  -- ----
  v_local_node_id := getLocalNodeId('_schemadoc');
  select set_origin into v_set_origin
      from sl_set
      where set_id = p_set_id;
  if not found then
    raise exception 'Slony-I: setAddTable_int(): set % not found',
        p_set_id;
```

```
end if;
if v_set_origin != v_local_node_id then
  select sub_provider into v_sub_provider
      from sl_subscribe
      where sub_set = p_set_id
      and sub_receiver = getLocalNodeId('_schemadoc');
  if not found then
    return 0;
  end if;
end if;

-- ----
-- Get the tables OID and check that it is a real table
-- ----
select PGC.oid, PGC.relkind, PGC.relname, PGN.nspname into v_tab_reloid, v_relkind,  ↩
    v_tab_relname, v_tab_nspname
    from "pg_catalog".pg_class PGC, "pg_catalog".pg_namespace PGN
    where PGC.relnamespace = PGN.oid
    and slon_quote_input(p_fqname) = slon_quote_brute(PGN.nspname) ||
        '.' || slon_quote_brute(PGC.relname);
if not found then
  raise exception 'Slony-I: setAddTable_int(): table % not found',
      p_fqname;
end if;
if v_relkind != 'r' then
  raise exception 'Slony-I: setAddTable_int(): % is not a regular table',
      p_fqname;
end if;

if not exists (select indexrelid
    from "pg_catalog".pg_index PGX, "pg_catalog".pg_class PGC
    where PGX.indrelid = v_tab_reloid
      and PGX.indexrelid = PGC.oid
      and PGC.relname = p_tab_idxname)
then
  raise exception 'Slony-I: setAddTable_int(): table % has no index %',
      p_fqname, p_tab_idxname;
end if;

-- ----
-- Verify that the columns in the PK (or candidate) are not NULLABLE
-- ----

v_pkcand_nn := 'f';
for v_prec in select attname from "pg_catalog".pg_attribute where attrelid =
                      (select oid from "pg_catalog".pg_class where oid = v_tab_reloid)
                  and attname in (select attname from "pg_catalog".pg_attribute where
                                  attrelid = (select oid from "pg_catalog".pg_class PGC,
                                  "pg_catalog".pg_index PGX where
                                  PGC.relname = p_tab_idxname and PGX.indexrelid=PGC.oid  ↩
                                      and
                                  PGX.indrelid = v_tab_reloid)) and attnotnull <> 't'
loop
  raise notice 'Slony-I: setAddTable_int: table % PK column % nullable', p_fqname, v_prec ↩
      .attname;
  v_pkcand_nn := 't';
end loop;
if v_pkcand_nn then
  raise exception 'Slony-I: setAddTable_int: table % not replicable!', p_fqname;
end if;

select * into v_prec from sl_table where tab_id = p_tab_id;
```

```
  if not found then
    v_pkcand_nn := 't';  -- No-op -- All is well
  else
    raise exception 'Slony-I: setAddTable_int: table id % has already been assigned!',  ↩
        p_tab_id;
  end if;

  -- ----
  -- Add the table to sl_table and create the trigger on it.
  -- ----
  insert into sl_table
      (tab_id, tab_reloid, tab_relname, tab_nspname,
      tab_set, tab_idxname, tab_altered, tab_comment)
      values
      (p_tab_id, v_tab_reloid, v_tab_relname, v_tab_nspname,
      p_set_id, p_tab_idxname, false, p_tab_comment);
  perform alterTableAddTriggers(p_tab_id);

  return p_tab_id;
end;
```

13.112 setdropsequence(p_seq_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* bigint setDropSequence (seq_id) On the origin node for the set, drop sequence seq_id from replication set, and raise SET_DROP_SEQUENCE to cause this to replicate to subscriber nodes.

```
declare
  v_set_id    int4;
  v_set_origin    int4;
begin
  -- ----
  -- Determine set id for this sequence
  -- ----
  select seq_set into v_set_id from sl_sequence where seq_id = p_seq_id;

  -- ----
  -- Ensure sequence exists
  -- ----
  if not found then
    raise exception 'Slony-I: setDropSequence_int(): sequence % not found',
      p_seq_id;
  end if;

  -- ----
  -- Check that we are the origin of the set
  -- ----
  select set_origin into v_set_origin
      from sl_set
      where set_id = v_set_id;
  if not found then
    raise exception 'Slony-I: setDropSequence(): set % not found', v_set_id;
  end if;
  if v_set_origin != getLocalNodeId('_schemadoc') then
    raise exception 'Slony-I: setDropSequence(): set % has origin at another node - submit  ↩
        this to that node', v_set_id;
  end if;

  -- ----
```

```
  -- Add the sequence to the set and generate the SET_ADD_SEQUENCE event
  -- ----
  perform setDropSequence_int(p_seq_id);
  return  createEvent('_schemadoc', 'SET_DROP_SEQUENCE',
          p_seq_id::text);
end;
```

13.113 setdropsequence_int(p_seq_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer setDropSequence_int (seq_id) This processes the SET_DROP_SEQUENCE event. On remote nodes that subscribe to the set containing sequence seq_id, drop the sequence from the replication set.

```
declare
  v_set_id    int4;
  v_local_node_id   int4;
  v_set_origin    int4;
  v_sub_provider    int4;
  v_relkind     char;
  v_sync_row      record;
begin
  -- ----
  -- Determine set id for this sequence
  -- ----
  select seq_set into v_set_id from sl_sequence where seq_id = p_seq_id;

  -- ----
  -- Ensure sequence exists
  -- ----
  if not found then
    return 0;
  end if;

  -- ----
  -- For sets with a remote origin, check that we are subscribed
  -- to that set. Otherwise we ignore the sequence because it might
  -- not even exist in our database.
  -- ----
  v_local_node_id := getLocalNodeId('_schemadoc');
  select set_origin into v_set_origin
      from sl_set
      where set_id = v_set_id;
  if not found then
    raise exception 'Slony-I: setDropSequence_int(): set % not found',
        v_set_id;
  end if;
  if v_set_origin != v_local_node_id then
    select sub_provider into v_sub_provider
        from sl_subscribe
        where sub_set = v_set_id
        and sub_receiver = getLocalNodeId('_schemadoc');
    if not found then
      return 0;
    end if;
  end if;

  -- ----
  -- drop the sequence from sl_sequence, sl_seqlog
  -- ----
```

```
  delete from sl_seqlog where seql_seqid = p_seq_id;
  delete from sl_sequence where seq_id = p_seq_id;

  return p_seq_id;
end;
```

13.114 setdroptable(p_tab_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* bigint setDropTable (tab_id) Drop table tab_id from set on origin node, and generate SET_DROP_TABLE event to allow this to propagate to other nodes.

```
declare
  v_set_id    int4;
  v_set_origin    int4;
begin
    -- ----
  -- Determine the set_id
    -- ----
  select tab_set into v_set_id from sl_table where tab_id = p_tab_id;

  -- ----
  -- Ensure table exists
  -- ----
  if not found then
    raise exception 'Slony-I: setDropTable_int(): table % not found',
      p_tab_id;
  end if;

  -- ----
  -- Check that we are the origin of the set
  -- ----
  select set_origin into v_set_origin
      from sl_set
      where set_id = v_set_id;
  if not found then
    raise exception 'Slony-I: setDropTable(): set % not found', v_set_id;
  end if;
  if v_set_origin != getLocalNodeId('_schemadoc') then
    raise exception 'Slony-I: setDropTable(): set % has remote origin', v_set_id;
  end if;

  -- ----
  -- Drop the table from the set and generate the SET_ADD_TABLE event
  -- ----
  perform setDropTable_int(p_tab_id);
  return  createEvent('_schemadoc', 'SET_DROP_TABLE',
        p_tab_id::text);
end;
```

13.115 setdroptable_int(p_tab_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer setDropTable_int (tab_id) This function processes the SET_DROP_TABLE event on remote nodes, dropping a table from replication if the remote node is subscribing to its replication set.

```
declare
  v_set_id    int4;
  v_local_node_id   int4;
  v_set_origin    int4;
  v_sub_provider    int4;
  v_tab_reloid    oid;
begin
    -- ----
  -- Determine the set_id
    -- ----
  select tab_set into v_set_id from sl_table where tab_id = p_tab_id;

  -- ----
  -- Ensure table exists
  -- ----
  if not found then
    return 0;
  end if;

  -- ----
  -- For sets with a remote origin, check that we are subscribed
  -- to that set. Otherwise we ignore the table because it might
  -- not even exist in our database.
  -- ----
  v_local_node_id := getLocalNodeId('_schemadoc');
  select set_origin into v_set_origin
      from sl_set
      where set_id = v_set_id;
  if not found then
    raise exception 'Slony-I: setDropTable_int(): set % not found',
        v_set_id;
  end if;
  if v_set_origin != v_local_node_id then
    select sub_provider into v_sub_provider
        from sl_subscribe
        where sub_set = v_set_id
        and sub_receiver = getLocalNodeId('_schemadoc');
    if not found then
      return 0;
    end if;
  end if;

  -- ----
  -- Drop the table from sl_table and drop trigger from it.
  -- ----
  perform alterTableDropTriggers(p_tab_id);
  delete from sl_table where tab_id = p_tab_id;
  return p_tab_id;
end;
```

13.116 setmovesequence(p_new_set_id integer, p_seq_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* bigint setMoveSequence(p_seq_id, p_new_set_id) - This generates the SET_MOVE_SEQUENCE event, after validation, notably that both sets exist, are distinct, and have exactly the same subscription lists

```
declare
  v_old_set_id    int4;
  v_origin      int4;
```

```
begin
  -- ----
  -- Get the sequences current set
  -- ----
  select seq_set into v_old_set_id from sl_sequence
      where seq_id = p_seq_id;
  if not found then
    raise exception 'Slony-I: setMoveSequence(): sequence %d not found', p_seq_id;
  end if;

  -- ----
  -- Check that both sets exist and originate here
  -- ----
  if p_new_set_id = v_old_set_id then
    raise exception 'Slony-I: setMoveSequence(): set ids cannot be identical';
  end if;
  select set_origin into v_origin from sl_set
      where set_id = p_new_set_id;
  if not found then
    raise exception 'Slony-I: setMoveSequence(): set % not found', p_new_set_id;
  end if;
  if v_origin != getLocalNodeId('_schemadoc') then
    raise exception 'Slony-I: setMoveSequence(): set % does not originate on local node',
        p_new_set_id;
  end if;

  select set_origin into v_origin from sl_set
      where set_id = v_old_set_id;
  if not found then
    raise exception 'Slony-I: set % not found', v_old_set_id;
  end if;
  if v_origin != getLocalNodeId('_schemadoc') then
    raise exception 'Slony-I: set % does not originate on local node',
        v_old_set_id;
  end if;

  -- ----
  -- Check that both sets are subscribed by the same set of nodes
  -- ----
  if exists (select true from sl_subscribe SUB1
        where SUB1.sub_set = p_new_set_id
        and SUB1.sub_receiver not in (select SUB2.sub_receiver
            from sl_subscribe SUB2
            where SUB2.sub_set = v_old_set_id))
  then
    raise exception 'Slony-I: subscriber lists of set % and % are different',
        p_new_set_id, v_old_set_id;
  end if;

  if exists (select true from sl_subscribe SUB1
        where SUB1.sub_set = v_old_set_id
        and SUB1.sub_receiver not in (select SUB2.sub_receiver
            from sl_subscribe SUB2
            where SUB2.sub_set = p_new_set_id))
  then
    raise exception 'Slony-I: subscriber lists of set % and % are different',
        v_old_set_id, p_new_set_id;
  end if;

  -- ----
  -- Change the set the sequence belongs to
  -- ----
```

```
  perform setMoveSequence_int(p_seq_id, p_new_set_id);
  return  createEvent('_schemadoc', 'SET_MOVE_SEQUENCE',
      p_seq_id::text, p_new_set_id::text);
end;
```

13.117 setmovesequence_int(p_new_set_id integer, p_seq_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer setMoveSequence_int(p_seq_id, p_new_set_id) - processes the SET_MOVE_SEQUENCE event, moving a sequence to another replication set.

```
begin
  -- ----
  -- Move the sequence to the new set
  -- ----
  update sl_sequence
      set seq_set = p_new_set_id
      where seq_id = p_seq_id;

  return p_seq_id;
end;
```

13.118 setmovetable(p_new_set_id integer, p_tab_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* bigint This processes the SET_MOVE_TABLE event. The table is moved to the destination set.

```
declare
  v_old_set_id    int4;
  v_origin      int4;
begin
  -- ----
  -- Get the tables current set
  -- ----
  select tab_set into v_old_set_id from sl_table
      where tab_id = p_tab_id;
  if not found then
    raise exception 'Slony-I: table %d not found', p_tab_id;
  end if;

  -- ----
  -- Check that both sets exist and originate here
  -- ----
  if p_new_set_id = v_old_set_id then
    raise exception 'Slony-I: set ids cannot be identical';
  end if;
  select set_origin into v_origin from sl_set
      where set_id = p_new_set_id;
  if not found then
    raise exception 'Slony-I: set % not found', p_new_set_id;
  end if;
  if v_origin != getLocalNodeId('_schemadoc') then
    raise exception 'Slony-I: set % does not originate on local node',
        p_new_set_id;
  end if;
```

```
  select set_origin into v_origin from sl_set
      where set_id = v_old_set_id;
  if not found then
    raise exception 'Slony-I: set % not found', v_old_set_id;
  end if;
  if v_origin != getLocalNodeId('_schemadoc') then
    raise exception 'Slony-I: set % does not originate on local node',
        v_old_set_id;
  end if;

  -- ----
  -- Check that both sets are subscribed by the same set of nodes
  -- ----
  if exists (select true from sl_subscribe SUB1
        where SUB1.sub_set = p_new_set_id
        and SUB1.sub_receiver not in (select SUB2.sub_receiver
            from sl_subscribe SUB2
            where SUB2.sub_set = v_old_set_id))
  then
    raise exception 'Slony-I: subscriber lists of set % and % are different',
        p_new_set_id, v_old_set_id;
  end if;

  if exists (select true from sl_subscribe SUB1
        where SUB1.sub_set = v_old_set_id
        and SUB1.sub_receiver not in (select SUB2.sub_receiver
            from sl_subscribe SUB2
            where SUB2.sub_set = p_new_set_id))
  then
    raise exception 'Slony-I: subscriber lists of set % and % are different',
        v_old_set_id, p_new_set_id;
  end if;

  -- ----
  -- Change the set the table belongs to
  -- ----
  perform createEvent('_schemadoc', 'SYNC', NULL);
  perform setMoveTable_int(p_tab_id, p_new_set_id);
  return  createEvent('_schemadoc', 'SET_MOVE_TABLE',
      p_tab_id::text, p_new_set_id::text);
end;
```

13.119 setmovetable_int(p_new_set_id integer, p_tab_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer

```
begin
  -- ----
  -- Move the table to the new set
  -- ----
  update sl_table
      set tab_set = p_new_set_id
      where tab_id = p_tab_id;

  return p_tab_id;
end;
```

13.120 slon_node_health_check()

Function Properties
Language: PLPGSQL, *Return Type:* boolean called when slon starts up to validate that there are not problems with node configuration. Returns t if all is OK, f if there is a problem.

```
declare
    prec record;
    all_ok boolean;
begin
    all_ok := 't'::boolean;
    -- validate that all tables in sl_table have:
    --      sl_table agreeing with pg_class
    for prec in select tab_id, tab_relname, tab_nspname from
    sl_table t where not exists (select 1 from pg_catalog.pg_class c, pg_catalog. ↩
        pg_namespace n
        where c.oid = t.tab_reloid and c.relname = t.tab_relname and c.relnamespace = n.oid ↩
             and n.nspname = t.tab_nspname) loop
        all_ok := 'f'::boolean;
        raise warning 'table [id,nsp,name]=[%,%,%] - sl_table does not match pg_class/ ↩
           pg_namespace', prec.tab_id, prec.tab_relname, prec.tab_nspname;
    end loop;
    if not all_ok then
       raise warning 'Mismatch found between sl_table and pg_class.  Slonik command REPAIR  ↩
          CONFIG may be useful to rectify this.';
    end if;
    return all_ok;
end
```

13.121 slon_quote_brute(p_tab_fqname text)

Function Properties
Language: PLPGSQL, *Return Type:* text Brutally quote the given text

```
declare
    v_fqname text default '';
begin
    v_fqname := '"' || replace(p_tab_fqname,'"','""') || '"';
    return v_fqname;
end;
```

13.122 slon_quote_input(p_tab_fqname text)

Function Properties
Language: PLPGSQL, *Return Type:* text quote all words that aren[apos]t quoted yet

```
  declare
     v_nsp_name text;
     v_tab_name text;
   v_i integer;
   v_l integer;
     v_pq2 integer;
begin
  v_l := length(p_tab_fqname);

  -- Let us search for the dot
```

```
  if p_tab_fqname like '"%' then
    -- if the first part of the ident starts with a double quote, search
    -- for the closing double quote, skipping over double double quotes.
    v_i := 2;
    while v_i <= v_l loop
      if substr(p_tab_fqname, v_i, 1) != '"' then
        v_i := v_i + 1;
      else
        v_i := v_i + 1;
        if substr(p_tab_fqname, v_i, 1) != '"' then
          exit;
        end if;
        v_i := v_i + 1;
      end if;
    end loop;
  else
    -- first part of ident is not quoted, search for the dot directly
    v_i := 1;
    while v_i <= v_l loop
      if substr(p_tab_fqname, v_i, 1) = '.' then
        exit;
      end if;
      v_i := v_i + 1;
    end loop;
  end if;

  -- v_i now points at the dot or behind the string.

  if substr(p_tab_fqname, v_i, 1) = '.' then
    -- There is a dot now, so split the ident into its namespace
    -- and objname parts and make sure each is quoted
    v_nsp_name := substr(p_tab_fqname, 1, v_i - 1);
    v_tab_name := substr(p_tab_fqname, v_i + 1);
    if v_nsp_name not like '"%' then
      v_nsp_name := '"' || replace(v_nsp_name, '"', '""') ||
                '"';
    end if;
    if v_tab_name not like '"%' then
      v_tab_name := '"' || replace(v_tab_name, '"', '""') ||
                '"';
    end if;

    return v_nsp_name || '.' || v_tab_name;
  else
    -- No dot ... must be just an ident without schema
    if p_tab_fqname like '"%' then
      return p_tab_fqname;
    else
      return '"' || replace(p_tab_fqname, '"', '""') || '"';
    end if;
  end if;

end;
```

13.123 slonyversion()

Function Properties
Language: PLPGSQL, *Return Type:* text Returns the version number of the slony schema

```
begin
  return slonyVersionMajor()::text || '.' ||
         slonyVersionMinor()::text || '.' ||
         slonyVersionPatchlevel()::text  ;
end;
```

13.124 slonyversionmajor()

Function Properties
Language: PLPGSQL, *Return Type:* integer Returns the major version number of the slony schema

```
begin
  return 2;
end;
```

13.125 slonyversionminor()

Function Properties
Language: PLPGSQL, *Return Type:* integer Returns the minor version number of the slony schema

```
begin
  return 1;
end;
```

13.126 slonyversionpatchlevel()

Function Properties
Language: PLPGSQL, *Return Type:* integer Returns the version patch level of the slony schema

```
begin
  return 1;
end;
```

13.127 store_application_name(i_name text)

Function Properties
Language: PLPGSQL, *Return Type:* text Set application_name GUC, if possible. Returns NULL if it fails to work.

```
declare
    p_command text;
begin
    if exists (select 1 from pg_catalog.pg_settings where name = 'application_name') then
       p_command := 'set application_name to '''|| i_name || ''';';
       execute p_command;
       return i_name;
    end if;
    return NULL::text;
end
```

13.128 storelisten(p_receiver integer, p_provider integer, p_origin integer)

Function Properties
Language: PLPGSQL, *Return Type:* bigint FUNCTION storeListen (li_origin, li_provider, li_receiver) generate STORE_LISTEN event, indicating that receiver node li_receiver listens to node li_provider in order to get messages coming from node li_origin.

```
begin
  perform storeListen_int (p_origin, p_provider, p_receiver);
  return  createEvent ('_schemadoc', 'STORE_LISTEN',
      p_origin::text, p_provider::text, p_receiver::text);
end;
```

13.129 storelisten_int(p_li_receiver integer, p_li_provider integer, p_li_origin integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer FUNCTION storeListen_int (li_origin, li_provider, li_receiver) Process STORE_LISTEN event, indicating that receiver node li_receiver listens to node li_provider in order to get messages coming from node li_origin.

```
declare
  v_exists    int4;
begin
  select 1 into v_exists
      from sl_listen
      where li_origin = p_li_origin
      and li_provider = p_li_provider
      and li_receiver = p_li_receiver;
  if not found then
    -- ----
    -- In case we receive STORE_LISTEN events before we know
    -- about the nodes involved in this, we generate those nodes
    -- as pending.
    -- ----
    if not exists (select 1 from sl_node
            where no_id = p_li_origin) then
      perform storeNode_int (p_li_origin, '<event pending>');
    end if;
    if not exists (select 1 from sl_node
            where no_id = p_li_provider) then
      perform storeNode_int (p_li_provider, '<event pending>');
    end if;
    if not exists (select 1 from sl_node
            where no_id = p_li_receiver) then
      perform storeNode_int (p_li_receiver, '<event pending>');
    end if;

    insert into sl_listen
        (li_origin, li_provider, li_receiver) values
        (p_li_origin, p_li_provider, p_li_receiver);
  end if;

  return 0;
end;
```

13.130 storenode(p_no_comment integer, p_no_id text)

Function Properties
Language: PLPGSQL, *Return Type:* bigint no_id - Node ID # no_comment - Human-oriented comment Generate the STORE_NODE event for node no_id

```
begin
  perform storeNode_int (p_no_id, p_no_comment);
  return  createEvent('_schemadoc', 'STORE_NODE',
                  p_no_id::text, p_no_comment::text);
end;
```

13.131 storenode_int(p_no_comment integer, p_no_id text)

Function Properties
Language: PLPGSQL, *Return Type:* integer no_id - Node ID # no_comment - Human-oriented comment Internal function to process the STORE_NODE event for node no_id

```
declare
  v_old_row   record;
begin
  -- ----
  -- Check if the node exists
  -- ----
  select * into v_old_row
      from sl_node
      where no_id = p_no_id
      for update;
  if found then
    -- ----
    -- Node exists, update the existing row.
    -- ----
    update sl_node
        set no_comment = p_no_comment
        where no_id = p_no_id;
  else
    -- ----
    -- New node, insert the sl_node row
    -- ----
    insert into sl_node
        (no_id, no_active, no_comment) values
        (p_no_id, 'f', p_no_comment);
  end if;

  return p_no_id;
end;
```

13.132 storepath(p_pa_connretry integer, p_pa_conninfo integer, p_pa_client text, p_pa_server integer)

Function Properties
Language: PLPGSQL, *Return Type:* bigint FUNCTION storePath (pa_server, pa_client, pa_conninfo, pa_connretry) Generate the STORE_PATH event indicating that node pa_client can access node pa_server using DSN pa_conninfo

```
begin
  perform storePath_int(p_pa_server, p_pa_client,
      p_pa_conninfo, p_pa_connretry);
  return  createEvent('_schemadoc', 'STORE_PATH',
      p_pa_server::text, p_pa_client::text,
      p_pa_conninfo::text, p_pa_connretry::text);
end;
```

13.133 storepath_int(p_pa_connretry integer, p_pa_conninfo integer, p_pa_client text, p_pa_server integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer FUNCTION storePath (pa_server, pa_client, pa_conninfo, pa_connretry) Process the STORE_PATH event indicating that node pa_client can access node pa_server using DSN pa_conninfo

```
declare
  v_dummy     int4;
begin
  -- ----
  -- Check if the path already exists
  -- ----
  select 1 into v_dummy
      from sl_path
      where pa_server = p_pa_server
      and pa_client = p_pa_client
      for update;
  if found then
    -- ----
    -- Path exists, update pa_conninfo
    -- ----
    update sl_path
        set pa_conninfo = p_pa_conninfo,
          pa_connretry = p_pa_connretry
        where pa_server = p_pa_server
        and pa_client = p_pa_client;
  else
    -- ----
    -- New path
    --
    -- In case we receive STORE_PATH events before we know
    -- about the nodes involved in this, we generate those nodes
    -- as pending.
    -- ----
    if not exists (select 1 from sl_node
            where no_id = p_pa_server) then
      perform storeNode_int (p_pa_server, '<event pending>');
    end if;
    if not exists (select 1 from sl_node
            where no_id = p_pa_client) then
      perform storeNode_int (p_pa_client, '<event pending>');
    end if;
    insert into sl_path
        (pa_server, pa_client, pa_conninfo, pa_connretry) values
        (p_pa_server, p_pa_client, p_pa_conninfo, p_pa_connretry);
  end if;

  -- Rewrite sl_listen table
  perform RebuildListenEntries();
```

```
  return 0;
end;
```

13.134 storeset(p_set_comment integer, p_set_id text)

Function Properties
Language: PLPGSQL, *Return Type:* bigint Generate STORE_SET event for set set_id with human readable comment set_comment

```
declare
  v_local_node_id   int4;
begin
  v_local_node_id := getLocalNodeId('_schemadoc');

  insert into sl_set
      (set_id, set_origin, set_comment) values
      (p_set_id, v_local_node_id, p_set_comment);

  return createEvent('_schemadoc', 'STORE_SET',
      p_set_id::text, v_local_node_id::text, p_set_comment::text);
end;
```

13.135 storeset_int(p_set_comment integer, p_set_origin integer, p_set_id text)

Function Properties
Language: PLPGSQL, *Return Type:* integer storeSet_int (set_id, set_origin, set_comment) Process the STORE_SET event, indicating the new set with given ID, origin node, and human readable comment.

```
declare
  v_dummy       int4;
begin
  select 1 into v_dummy
      from sl_set
      where set_id = p_set_id
      for update;
  if found then
    update sl_set
        set set_comment = p_set_comment
        where set_id = p_set_id;
  else
    if not exists (select 1 from sl_node
             where no_id = p_set_origin) then
      perform storeNode_int (p_set_origin, '<event pending>');
    end if;
    insert into sl_set
        (set_id, set_origin, set_comment) values
        (p_set_id, p_set_origin, p_set_comment);
  end if;

  -- Run addPartialLogIndices() to try to add indices to unused sl_log_? table
  perform addPartialLogIndices();

  return p_set_id;
end;
```

13.136 subscribeset(p_omit_copy integer, p_sub_forward integer, p_sub_receiver integer, p_sub_provider boolean, p_sub_set boolean)

Function Properties
Language: PLPGSQL, *Return Type:* bigint subscribeSet (sub_set, sub_provider, sub_receiver, sub_forward, omit_copy) Makes sure that the receiver is not the provider, then stores the subscription, and publishes the SUBSCRIBE_SET event to other nodes. If omit_copy is true, then no data copy will be done.

```
declare
  v_set_origin    int4;
  v_ev_seqno      int8;
  v_rec     record;
begin
  --
  -- Check that the receiver exists
  --
  if not exists (select no_id from sl_node where no_id=
                p_sub_receiver) then
          raise exception 'Slony-I: subscribeSet() receiver % does not exist' ,  ↩
              p_sub_receiver;
  end if;

  --
  -- Check that the provider exists
  --
  if not exists (select no_id from sl_node where no_id=
                p_sub_provider) then
          raise exception 'Slony-I: subscribeSet() provider % does not exist' ,  ↩
              p_sub_provider;
  end if;

  -- ----
  -- Check that the origin and provider of the set are remote
  -- ----
  select set_origin into v_set_origin
      from sl_set
      where set_id = p_sub_set;
  if not found then
    raise exception 'Slony-I: subscribeSet(): set % not found', p_sub_set;
  end if;
  if v_set_origin = p_sub_receiver then
    raise exception
        'Slony-I: subscribeSet(): set origin and receiver cannot be identical';
  end if;
  if p_sub_receiver = p_sub_provider then
    raise exception
        'Slony-I: subscribeSet(): set provider and receiver cannot be identical';
  end if;
  -- ----
  -- Check that this is called on the origin node
  -- ----
  if v_set_origin != getLocalNodeId('_schemadoc') then
    raise exception 'Slony-I: subscribeSet() must be called on origin';
  end if;

  -- ---
  -- Verify that the provider is either the origin or an active subscriber
  -- Bug report #1362
  -- ---
  if v_set_origin <> p_sub_provider then
    if not exists (select 1 from sl_subscribe
```

```
      where sub_set = p_sub_set and
                              sub_receiver = p_sub_provider and
            sub_forward and sub_active) then
      raise exception 'Slony-I: subscribeSet(): provider % is not an active forwarding node ↩
          for replication set %', p_sub_provider, p_sub_set;
    end if;
  end if;

  -- ----
  -- Create the SUBSCRIBE_SET event
  -- ----
  v_ev_seqno :=  createEvent('_schemadoc', 'SUBSCRIBE_SET',
      p_sub_set::text, p_sub_provider::text, p_sub_receiver::text,
      case p_sub_forward when true then 't' else 'f' end,
      case p_omit_copy when true then 't' else 'f' end
                       );

  -- ----
  -- Call the internal procedure to store the subscription
  -- ----
  perform subscribeSet_int(p_sub_set, p_sub_provider,
      p_sub_receiver, p_sub_forward, p_omit_copy);

  return v_ev_seqno;
end;
```

13.137 subscribeset_int(p_omit_copy integer, p_sub_forward integer, p_sub_receiver integer, p_sub_provider boolean, p_sub_set boolean)

Function Properties
Language: PLPGSQL, *Return Type:* integer subscribeSet_int (sub_set, sub_provider, sub_receiver, sub_forward, omit_copy)
Internal actions for subscribing receiver sub_receiver to subscription set sub_set.

```
declare
  v_set_origin    int4;
  v_sub_row     record;
begin
  -- ----
  -- Provider change is only allowed for active sets
  -- ----
  if p_sub_receiver = getLocalNodeId('_schemadoc') then
    select sub_active into v_sub_row from sl_subscribe
        where sub_set = p_sub_set
        and sub_receiver = p_sub_receiver;
    if found then
      if not v_sub_row.sub_active then
        raise exception 'Slony-I: subscribeSet_int(): set % is not active, cannot change  ↩
            provider',
            p_sub_set;
      end if;
    end if;
  end if;

  -- ----
  -- Try to change provider and/or forward for an existing subscription
  -- ----
  update sl_subscribe
      set sub_provider = p_sub_provider,
        sub_forward = p_sub_forward
```

```
      where sub_set = p_sub_set
      and sub_receiver = p_sub_receiver;
  if found then
    -- ----
    -- Rewrite sl_listen table
    -- ----
    perform RebuildListenEntries();

    return p_sub_set;
  end if;

  -- ----
  -- Not found, insert a new one
  -- ----
  if not exists (select true from sl_path
      where pa_server = p_sub_provider
      and pa_client = p_sub_receiver)
  then
    insert into sl_path
        (pa_server, pa_client, pa_conninfo, pa_connretry)
        values
        (p_sub_provider, p_sub_receiver,
        '<event pending>', 10);
  end if;
  insert into sl_subscribe
      (sub_set, sub_provider, sub_receiver, sub_forward, sub_active)
      values (p_sub_set, p_sub_provider, p_sub_receiver,
        p_sub_forward, false);

  -- ----
  -- If the set origin is here, then enable the subscription
  -- ----
  select set_origin into v_set_origin
      from sl_set
      where set_id = p_sub_set;
  if not found then
    raise exception 'Slony-I: subscribeSet_int(): set % not found', p_sub_set;
  end if;

  if v_set_origin = getLocalNodeId('_schemadoc') then
    perform createEvent('_schemadoc', 'ENABLE_SUBSCRIPTION',
        p_sub_set::text, p_sub_provider::text, p_sub_receiver::text,
        case p_sub_forward when true then 't' else 'f' end,
        case p_omit_copy when true then 't' else 'f' end
        );
    perform enableSubscription(p_sub_set,
        p_sub_provider, p_sub_receiver);
  end if;

  -- ----
  -- Rewrite sl_listen table
  -- ----
  perform RebuildListenEntries();

  return p_sub_set;
end;
```

13.138 tablestovacuum()

Function Properties
Language: PLPGSQL, *Return Type:* SET OF vactables Return a list of tables that require frequent vacuuming. The function is used so that the list is not hardcoded into C code.

```
declare
  prec vactables%rowtype;
begin
  prec.nspname := '_schemadoc';
  prec.relname := 'sl_event';
  if ShouldSlonyVacuumTable(prec.nspname, prec.relname) then
    return next prec;
  end if;
  prec.nspname := '_schemadoc';
  prec.relname := 'sl_confirm';
  if ShouldSlonyVacuumTable(prec.nspname, prec.relname) then
    return next prec;
  end if;
  prec.nspname := '_schemadoc';
  prec.relname := 'sl_setsync';
  if ShouldSlonyVacuumTable(prec.nspname, prec.relname) then
    return next prec;
  end if;
  prec.nspname := '_schemadoc';
  prec.relname := 'sl_seqlog';
  if ShouldSlonyVacuumTable(prec.nspname, prec.relname) then
    return next prec;
  end if;
  prec.nspname := '_schemadoc';
  prec.relname := 'sl_archive_counter';
  if ShouldSlonyVacuumTable(prec.nspname, prec.relname) then
    return next prec;
  end if;
  prec.nspname := '_schemadoc';
  prec.relname := 'sl_components';
  if ShouldSlonyVacuumTable(prec.nspname, prec.relname) then
    return next prec;
  end if;
  prec.nspname := 'pg_catalog';
  prec.relname := 'pg_listener';
  if ShouldSlonyVacuumTable(prec.nspname, prec.relname) then
    return next prec;
  end if;
  prec.nspname := 'pg_catalog';
  prec.relname := 'pg_statistic';
  if ShouldSlonyVacuumTable(prec.nspname, prec.relname) then
    return next prec;
  end if;

   return;
end
```

13.139 terminatenodeconnections(p_failed_node integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer terminates all backends that have registered to be from the given node

```
declare
```

```
  v_row     record;
begin
  for v_row in select nl_nodeid, nl_conncnt,
      nl_backendpid from sl_nodelock
      where nl_nodeid = p_failed_node for update
  loop
    perform killBackend(v_row.nl_backendpid, 'TERM');
    delete from sl_nodelock
      where nl_nodeid = v_row.nl_nodeid
      and nl_conncnt = v_row.nl_conncnt;
  end loop;

  return 0;
end;
```

13.140 uninstallnode()

Function Properties
Language: PLPGSQL, *Return Type:* integer Reset the whole database to standalone by removing the whole replication system.

```
declare
  v_tab_row   record;
begin
  raise notice 'Slony-I: Please drop schema "_schemadoc"';
  return 0;
end;
```

13.141 unlockset(p_set_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer Remove the special trigger from all tables of a set that disables access to it.

```
declare
  v_local_node_id   int4;
  v_set_row     record;
  v_tab_row     record;
begin
  -- ----
  -- Check that the set exists and that we are the origin
  -- and that it is not already locked.
  -- ----
  v_local_node_id := getLocalNodeId('_schemadoc');
  select * into v_set_row from sl_set
      where set_id = p_set_id
      for update;
  if not found then
    raise exception 'Slony-I: set % not found', p_set_id;
  end if;
  if v_set_row.set_origin <> v_local_node_id then
    raise exception 'Slony-I: set % does not originate on local node',
        p_set_id;
  end if;
  if v_set_row.set_locked isnull then
    raise exception 'Slony-I: set % is not locked', p_set_id;
  end if;

  -- ----
```

```
  -- Drop the lockedSet trigger from all tables in the set.
  -- ----
  for v_tab_row in select T.tab_id,
      slon_quote_brute(PGN.nspname) || '.' ||
      slon_quote_brute(PGC.relname) as tab_fqname
      from sl_table T,
        "pg_catalog".pg_class PGC, "pg_catalog".pg_namespace PGN
      where T.tab_set = p_set_id
        and T.tab_reloid = PGC.oid
        and PGC.relnamespace = PGN.oid
      order by tab_id
  loop
    execute 'drop trigger "_schemadoc_lockedset" ' ||
        'on ' || v_tab_row.tab_fqname;
  end loop;

  -- ----
  -- Clear out the set_locked field
  -- ----
  update sl_set
      set set_locked = NULL
      where set_id = p_set_id;

  return p_set_id;
end;
```

13.142 unsubscribeset(p_sub_receiver integer, p_sub_set integer)

Function Properties
Language: PLPGSQL, *Return Type:* bigint unsubscribeSet (sub_set, sub_receiver) Unsubscribe node sub_receiver from subscription set sub_set. This is invoked on the receiver node. It verifies that this does not break any chains (e.g. - where sub_receiver is a provider for another node), then restores tables, drops Slony-specific keys, drops table entries for the set, drops the subscription, and generates an UNSUBSCRIBE_SET node to publish that the node is being dropped.

```
declare
  v_tab_row     record;
begin
  -- ----
  -- Check that this is called on the receiver node
  -- ----
  if p_sub_receiver != getLocalNodeId('_schemadoc') then
    raise exception 'Slony-I: unsubscribeSet() must be called on receiver';
  end if;

  -- ----
  -- Check that this does not break any chains
  -- ----
  if exists (select true from sl_subscribe
      where sub_set = p_sub_set
        and sub_provider = p_sub_receiver)
  then
    raise exception 'Slony-I: Cannot unsubscribe set % while being provider',
        p_sub_set;
  end if;

  -- ----
  -- Remove the replication triggers.
  -- ----
  for v_tab_row in select tab_id from sl_table
```

```
      where tab_set = p_sub_set
      order by tab_id
  loop
    perform alterTableDropTriggers(v_tab_row.tab_id);
  end loop;

  -- ----
  -- Remove the setsync status. This will also cause the
  -- worker thread to ignore the set and stop replicating
  -- right now.
  -- ----
  delete from sl_setsync
      where ssy_setid = p_sub_set;

  -- ----
  -- Remove all sl_table and sl_sequence entries for this set.
  -- Should we ever subscribe again, the initial data
  -- copy process will create new ones.
  -- ----
  delete from sl_table
      where tab_set = p_sub_set;
  delete from sl_sequence
      where seq_set = p_sub_set;

  -- ----
  -- Call the internal procedure to drop the subscription
  -- ----
  perform unsubscribeSet_int(p_sub_set, p_sub_receiver);

  -- Rewrite sl_listen table
  perform RebuildListenEntries();

  -- ----
  -- Create the UNSUBSCRIBE_SET event
  -- ----
  return  createEvent('_schemadoc', 'UNSUBSCRIBE_SET',
      p_sub_set::text, p_sub_receiver::text);
end;
```

13.143 unsubscribeset_int(p_sub_receiver integer, p_sub_set integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer unsubscribeSet_int (sub_set, sub_receiver) All the REAL work of removing the subscriber is done before the event is generated, so this function just has to drop the references to the subscription in sl_subscribe.

```
begin
  -- ----
  -- All the real work is done before event generation on the
  -- subscriber.
  -- ----
  delete from sl_subscribe
      where sub_set = p_sub_set
        and sub_receiver = p_sub_receiver;

  -- Rewrite sl_listen table
  perform RebuildListenEntries();

  return p_sub_set;
end;
```

13.144 updaterelname(p_only_on_node integer, p_set_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* integer updateRelname(set_id, only_on_node)

```
declare
        v_no_id                 int4;
        v_set_origin            int4;
begin
        -- ----
        -- Grab the central configuration lock
        -- ----
        lock table sl_config_lock;

        -- ----
        -- Check that we either are the set origin or a current
        -- subscriber of the set.
        -- ----
        v_no_id := getLocalNodeId('_schemadoc');
        select set_origin into v_set_origin
                        from sl_set
                        where set_id = p_set_id
                        for update;
        if not found then
                raise exception 'Slony-I: set % not found', p_set_id;
        end if;
        if v_set_origin <> v_no_id
                and not exists (select 1 from sl_subscribe
                        where sub_set = p_set_id
                        and sub_receiver = v_no_id)
        then
                return 0;
        end if;

        -- ----
        -- If execution on only one node is requested, check that
        -- we are that node.
        -- ----
        if p_only_on_node > 0 and p_only_on_node <> v_no_id then
                return 0;
        end if;
        update sl_table set
                tab_relname = PGC.relname, tab_nspname = PGN.nspname
                from pg_catalog.pg_class PGC, pg_catalog.pg_namespace PGN
                where sl_table.tab_reloid = PGC.oid
                        and PGC.relnamespace = PGN.oid;
        update sl_sequence set
                seq_relname = PGC.relname, seq_nspname = PGN.nspname
                from pg_catalog.pg_class PGC, pg_catalog.pg_namespace PGN
                where sl_sequence.seq_reloid = PGC.oid
                and PGC.relnamespace = PGN.oid;
        return p_set_id;
end;
```

13.145 updatereloid(p_only_on_node integer, p_set_id integer)

Function Properties
Language: PLPGSQL, *Return Type:* bigint updateReloid(set_id, only_on_node) Updates the respective reloids in sl_table and sl_seqeunce based on their respective FQN

```
declare
        v_no_id                 int4;
        v_set_origin            int4;
  prec      record;
begin
        -- ----
        -- Check that we either are the set origin or a current
        -- subscriber of the set.
        -- ----
        v_no_id := getLocalNodeId('_schemadoc');
        select set_origin into v_set_origin
                        from sl_set
                        where set_id = p_set_id
                        for update;
        if not found then
                raise exception 'Slony-I: set % not found', p_set_id;
        end if;
        if v_set_origin <> v_no_id
                and not exists (select 1 from sl_subscribe
                        where sub_set = p_set_id
                        and sub_receiver = v_no_id)
        then
                return 0;
        end if;

        -- ----
        -- If execution on only one node is requested, check that
        -- we are that node.
        -- ----
        if p_only_on_node > 0 and p_only_on_node <> v_no_id then
                return 0;
        end if;

  -- Update OIDs for tables to values pulled from non-table objects in pg_class
  -- This ensures that we won't have collisions when repairing the oids
  for prec in select tab_id from sl_table loop
    update sl_table set tab_reloid = (select oid from pg_class pc where relkind <> 'r' and  ↩
        not exists (select 1 from sl_table t2 where t2.tab_reloid = pc.oid) limit 1)
    where tab_id = prec.tab_id;
  end loop;

  for prec in select tab_id, tab_relname, tab_nspname from sl_table loop
          update sl_table set
                  tab_reloid = (select PGC.oid
                  from pg_catalog.pg_class PGC, pg_catalog.pg_namespace PGN
                  where slon_quote_brute(PGC.relname) = slon_quote_brute(prec.tab_relname)
                          and PGC.relnamespace = PGN.oid
        and slon_quote_brute(PGN.nspname) = slon_quote_brute(prec.tab_nspname))
    where tab_id = prec.tab_id;
  end loop;

  for prec in select seq_id from sl_sequence loop
    update sl_sequence set seq_reloid = (select oid from pg_class pc where relkind <> 'S'  ↩
        and not exists (select 1 from sl_sequence t2 where t2.seq_reloid = pc.oid) limit 1)
    where seq_id = prec.seq_id;
  end loop;

  for prec in select seq_id, seq_relname, seq_nspname from sl_sequence loop
          update sl_sequence set
                  seq_reloid = (select PGC.oid
                  from pg_catalog.pg_class PGC, pg_catalog.pg_namespace PGN
```

```
                  where slon_quote_brute(PGC.relname) = slon_quote_brute(prec.seq_relname)
                  and PGC.relnamespace = PGN.oid
      and slon_quote_brute(PGN.nspname) = slon_quote_brute(prec.seq_nspname))
    where seq_id = prec.seq_id;
  end loop;

  return 1;
end;
```

13.146 upgradeschema(p_old text)

Function Properties
Language: PLPGSQL, *Return Type:* text Called during "update functions" by slonik to perform schema changes

```
declare
  v_tab_row record;
  v_query text;
  v_keepstatus text;
begin
  -- If old version is pre-2.0, then we require a special upgrade process
  if p_old like '1.%' then
    raise exception 'Upgrading to Slony-I 2.x requires running slony_upgrade_20';
  end if;

  perform upgradeSchemaAddTruncateTriggers();

  -- Change all Slony-I-defined columns that are "timestamp without time zone" to " ↩
     timestamp *WITH* time zone"
  if exists (select 1 from information_schema.columns c
            where table_schema = '_schemadoc' and data_type = 'timestamp without time zone'
      and exists (select 1 from information_schema.tables t where t.table_schema = c. ↩
          table_schema and t.table_name = c.table_name and t.table_type = 'BASE TABLE')
    and (c.table_name, c.column_name) in (('sl_confirm', 'con_timestamp'), ('sl_event', ' ↩
        ev_timestamp'), ('sl_registry', 'reg_timestamp'),('sl_archive_counter', ' ↩
        ac_timestamp')))
  then

    -- Preserve sl_status
    select pg_get_viewdef('sl_status') into v_keepstatus;
    execute 'drop view sl_status';
    for v_tab_row in select table_schema, table_name, column_name from information_schema. ↩
        columns c
            where table_schema = '_schemadoc' and data_type = 'timestamp without time zone'
      and exists (select 1 from information_schema.tables t where t.table_schema = c. ↩
          table_schema and t.table_name = c.table_name and t.table_type = 'BASE TABLE')
    and (table_name, column_name) in (('sl_confirm', 'con_timestamp'), ('sl_event', ' ↩
        ev_timestamp'), ('sl_registry', 'reg_timestamp'),('sl_archive_counter', ' ↩
        ac_timestamp'))
    loop
    raise notice 'Changing Slony-I column [%.%] to timestamp WITH time zone', v_tab_row. ↩
        table_name, v_tab_row.column_name;
    v_query := 'alter table ' || slon_quote_brute(v_tab_row.table_schema) ||
                  '.' || v_tab_row.table_name || ' alter column ' || v_tab_row.column_name ↩
                      ||
                  ' type timestamp with time zone;';
    execute v_query;
    end loop;
    -- restore sl_status
    execute 'create view sl_status as ' || v_keepstatus;
```

```
        end if;

  if not exists (select 1 from information_schema.tables where table_schema = '_schemadoc'  ↩
      and table_name = 'sl_components') then
     v_query := '
create table sl_components (
  co_actor   text not null primary key,
  co_pid     integer not null,
  co_node    integer not null,
  co_connection_pid integer not null,
  co_activity   text,
  co_starttime    timestamptz not null,
  co_event    bigint,
  co_eventtype    text
) without oids;
';
       execute v_query;
  end if;
  if not exists (select 1 from information_schema.tables t where table_schema = '_schemadoc ↩
      ' and table_name = 'sl_event_lock') then
     v_query := 'create table sl_event_lock (dummy integer);';
     execute v_query;
        end if;
  return p_old;
end;
```