České vysoké učení technické v Praze
Fakulta elektrotechnická

Bakalářská práce

Vizuální konfigurace replikačního mechanizmu Slony-I v otevřeném nástroji pgAdmin III

Ondřej Čečák

Vedoucí práce: Ing. Michal Valenta, Ph.D.

Studijní program: Elektrotechnika a informatika, strukturovaný, bakalářský

Obor: Informatika a výpočetní technika

červenec 2008

"Has everyone noticed that all the letters of the word 'database' are typed with the left hand? Now the layout of the QWERTYUIOP typewriter keyboard was designed, among other things, to facilitate the even use of both hands. It follows, therefore, that writing about databases is not only unnatural, but a lot harder than it appears."

Acknowledgments

I would like to thank everyone who helped me improving this text for support, many corrections, valuable pieces of advice and patience. I thank namely my parents Šárka and Pavel Čečákovi, my sister Barbora Čečáková, my pampering girlfriend Martina Matějková and my supervisor Michal Valenta. I also thank my friends Honza Houštěk, Aleš Hakl, Jiří Simon and Kateřina Srnská.

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.

Nemám žádný důvod proti užití tohoto školního díla ve smyslu § 60 zákona § 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 10.7.2008 ....

Abstract

Slony-I is one of the most powerful tools, which are currently used for replication of PostgreSQL Database Management System. The database replication, as a process of sharing information so as to ensure the consistency between redundant resources, is important to improve reliability, fault-tolerance or accessibility. Very often used in enterprise environment. This thesis describes PostgreSQL DBMS, its means of replication, Slony-I engine and administration tool pgAdmin III with design and implementation user friendly wizards. They as addition to pgAdmin allow simple creating and managing Slony-I clusters.

Abstrakt

Slony-I je jedním z nejschopnějších nástrojů, které se v současné době používají pro replikaci SŘDB PostgreSQL. Replikace databází, ve smyslu způsobu sdílení informací pro zajištění konzistence redundatních zdrojů, je důležitá pro zvýšení spolehlivosti, odolnosti nebo dostupnosti. Velmi často se používá ve firemním prostředí. Tato práce popisuje SŘDB PostgreSQL, jeho způsoby replikace, mechanizmus Slony-I a administrační nástroj pgAdmin III spolu s návrhem a implementací uživatelsky přívětivých průvodců (wizards), které jako doplněk k pgAdmin dovolují jednoduché vytváření a správu klastrů Slony-I.

Contents

List of Figures

1 Introduction

This bachelor thesis was written with focus on Slony-I, an enterprise level replication system for PostgreSQL Database Management System. Its main target is addition to pgAdmin III (PostgreSQL administration and management tool) which allows administration of Slony-I replication engine in a user friendly way. All software projects mentioned above are developed as free and open-source software.

2 Problem Description and Target Specification

This chapter presents the basic facts about PostgreSQL and main targets of this work.

2.1 PostgreSQL: The World's Most Advanced Open-Source Database

PostgreSQL is an object-relational database management system (ORDBMS) [1] – it provides a management system which allows developers to integrate a database with their own custom data types and methods.

PostgreSQL Database Management System source code is released under the BSD license and thus is a free open-source software. As many other open-source projects, PostgreSQL is not controlled by any single company, it is developed by a worldwide community.

BSD license is probably the most liberal open-source license. It allows to use, modify and distribute source code and other parts in any form, open or closed source. Any further modifications, enhancements, or changes belong to their author.

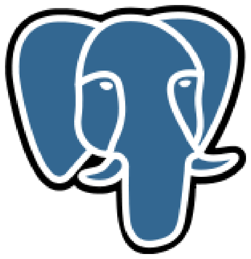


Figure 2.1: The blue/white elephant logo of the PostgreSQL RDBMS.

2.1.1 Brief History

The history of software called PostgreSQL is quite long. The name itself was changed several times – starting with Ingres, or more precisely POSTGRES (derived from post-Ingres) to PostgreSQL which originated as a result of discussion between the community and PostgreSQL Core Team and lasts until now. The current name still contains letters SQL despite the ubiquitous support of the SQL Standard.

Everything started with the Ingres project at UC Berkeley [3] and project leader Michael Stonebraker, who after his return to the Academia in 1985, started another project called post-Ingres. From this time on, the source code is fully separated out from Ingres.

The original papers describing the basics of the system were released in 1986. In 1988, the project team had a running prototype version. There are four major POSTGRES versions, the first version 1 was released in June 1989 to quite a small number of users while the last version 4 in 1993 (primarily meant as a cleanup) closed the project with a huge number of users.

Although the original project was officially ended, the BSD license allowed the community to continue the development. In 1994, two Berkeley students Andrew Yu and Jolly Chen added an SQL language interpreter to the original code and created Postgres95 which was consequently released to the web.

The first non-university development server was provided by Marc Fournier at Hub.Org Networking Services in July 1996. Roughly at the same time, Bruce Momjian and Vadim B. Mikheev started stabilizing the Berkeley code. The first open-source version was released in August 1996.

In the same year, the project was renamed again to the current name PostgreSQL.

The PostgreSQL project still honours the open-source model and makes yearly major releases (the current stable version at the beginning of July 2008 was 8.3.3).

2.1.2 Features

PostgreSQL is presented as an enterprise class database, it is fully ACID-compliant, has full support for foreign keys, joins, views, triggers and stored procedures. It includes the most of SQL92 and SQL99 data types. It also includes support of binary large objects storage. It has a native programming interfaces for C/C++, Java, .NET and many others.

As an enterprise class database, PostgreSQL has many advanced features such as Multi-Version Concurrency Control (MVCC), point in time recovery, tablespaces, nested transactions (savepoints), online/hot backups, a sophisticated query planner/optimizer and write ahead logging for fault tolerance. It is highly scalable.

One of the advanced features is a support of compound, unique, partial, and functional indexes which can use any of B-tree, R-tree, hash, or GiST storage methods. Other features include table inheritance, rules systems and database events.

Aside from many features, the documentation of PostgreSQL is truly exceptional.

2.2 Aim of the Thesis

This work is focused on Slony-I replication mechanism engine and its support in pgAdmin III administrative tool. Its main goals are:

- To describe replication system of Slony-I project.
- To familiarize with concept of open-source administration tool pgAdmin III, particularly with the current support of Slony-I replication mechanism.
- To design and implement a set of replication scenarios which will be provided through a form of wizard-based user friendly interface. Requested scenarios are: configuration of replication cluster, add/remove cluster node, add/remove object.
- Design and implementation should be done with possible code integration into the main branch of pgAdmin III project in mind.

3 PostgreSQL Replication

3.1 Replication

In Wikipedia [4], replication (in computer science) is defined as the process of sharing information so as to ensure the consistency between redundant resources, such as software or hardware components, to improve reliability, fault-tolerance or accessibility.

In our case, we would like to provide replication with all characteristics mentioned above: improve reliability with fault-tolerance (e.g. a hardware fault on one cluster node will not be a problem in the database due redundant data distribution) and improve accessibility and performance (with load balancing between two cluster nodes etc.).

Nevertheless, the replication itself should be fully transparent to its users.

3.1.1 Database Replication

Database replication on database management systems have several main concepts. Above all:

- The master/slave replication.
- Multi-master replication.

The master/slave replication is probably the most common. In this scenario usually all operations are performed on the master node and the data changes are transparently transfered to the slave node which works as a hot backup – if something goes wrong with the master node, the slave takes over his role.

With multi-master replication we could perform data update on any of cluster master nodes. That usually requires a sophisticated transactional system which is necessary to avoid the data change conflicts (the system should prevent it or solve it).

3.1.2 Replication Schemes

Both of the cluster nodes could be in a local network or geographically dispersed (then we talk about the geographical cluster).

Another type of division could be synchronous and asynchronous replication. The synchronous replication guarantees no possible data loss by the means of atomic write operations (the data change completes on both sides, or not at all). Asynchronous replication considers write as finished after the operation on the main (master) node is completed. The remote node is updated as well, but with possible latency.

3.2 Main PostgreSQL Replication Mechanisms

3.2.1 Indirect Ways

At the beginning, we should notice that there are some other ways, how to archive PostgreSQL replication in “indirect ways“. For example using lower layers than application layer like filesystem replication (e.g. GFS [5] or DRBD [6] are quite popular). Even if this is a fully functional possibility, it is not interesting from our point of view, so we cat omit it.

3.2.2 Using Backup System

The easiest way to “replicate” is simply to do backup of the master node from time to time using tools directly from PostgreSQL (`pg_dump` and `pg_restore`). Disadvantages of this solution overreach the easiness. The database dump of the master node has to be completed (even if the data changes from last backup was minimal). We have obviously lost all changed data since the last master node dump, thus this is rather theoretical possibility introduced for completeness' sake.

3.2.3 On-line Backup with Point-in-time Recovery

PostgreSQL creates a write ahead log (WAL) at all times, in which is described every change of the database data files. It is primarily maintained for crash-safety purposes – if the system crashs, the data can be restored by replaying separate log entries since the last checkpoint to the consistent state. Though, it could be used also to create hot-stand-by system, if we continuously replay logs on the slave system, we will get the master/slave replication.

Copying the WAL files could be done with a few lines in some shell script, PostgreSQL itself allows to work with the WAL files using defined `archive_command` and `restore_command`, but database recovery using the write ahead log has some difficulties. This method could restore only an entire database cluster (not a subset), requires many local storage for keeping the log files. Operation with non-B-tree indexes (i.e. hash, R-tree and GiST indexes) are not logged in the WAL so this indexes operations have to be replayed manually. Because PostgreSQL has never been designed to have architecture-independent data files, it is not reliably possible to restore the WAL backup created for example on a 32bit machine to a 64bit hot-standby machine.

Probably the most unpleasant difficulty has been solved recently. If we had wanted to recover from the write ahead log on the slave system, we would have had to stop PostgreSQL's `postmaster` process, do some file copying and start postmaster process again. Postmaster would have started in the recovery mode replaying all log entries which on busy systems could be a quite long time.

Solution to this problem was created during the Google Summer of Code 2007, where student Florian G. Pflug with mentor Simon Riggs in project "Implementing support for read-only queries on PITR slaves" [7] wrote few-thousand-line patch [8] to PostgreSQL source code which from the original Summer of Code project proposal [9] implemented above all the support for the on-line WAL replaying. Anyway, there are still some open issues in the code and possible bugs, but this mechanism looks very hopefully.

3.2.4 pgpool-II

pgpool-II is a middleware that works between PostgreSQL servers and a PostgreSQL database client [10]. pgpool-II could be configured as a proxy between database servers (cluster nodes) and clients, which allows to project the SQL operations to master and slaves.

pgpool-II could be also used for load balancing – for the replicated database is possible to divide execution of the SQL `SELECT` statements between more cluster nodes.

pgpool-II is the transparent (for both client and server side) tool for asynchronous replication which is very easy to use (changing existing database structures or application's configuration is not usually necessary). There are some situation where pgpool-II is probably not the best choice – for example if one node of a geographical cluster is on unstable connection line, you will probably have inconsistent data on the mentioned node.

Another problem is common to all query-propagating systems – applications are susceptible to data corruption for any updates which involve a nondeterministic query (e.g. `RANDOM()` in SQL statements), a time-sensitive query (e.g. using `NOW()` in SQL) or a host-sensitive results.

In spite of the fact that there are some minor problems with using pgpool-II as replication mechanism, its concept is good and the program works very well.

3.2.5 PGCluster

PGCluster [11] is a replication system that is synchronous and multi-master, guarantees the data consistency (each transaction is either fully completed or not even started) and allows full-user access to the multiple master nodes.

With little more complicated server infrastructure (a load balancer, Cluster DB and a replication server) makes possible both load balancing and high availability (HA) scenarios.

Although PGCluster supports the latest stable version of PostgreSQL, project pages and development of software itself seems abandoned.

3.2.6 Squoia

Squoia [12] is a middleware that provides a transparent access to the cluster via Java Database Connectivity (JDBC). Beyond some special developers features (like traffic logging or profiling) Squoia offers the failover and performance improving.

Project is a continuation of former C-JDBC project and its development is arched over by company Continuent, Inc. Software is licensed under terms of the Apache License, thus it is the open-source software.

Thanks to JDBC, no code change for using Squoia is required. Data are accessed from application by a virtual database which Squoia provides and which can be replicated to more cluster nodes. JDBC driver forwards all database queries to the Squoia controller.

Replication itself is provided by recovery log of the virtual database and referred as Redundant Array of Inexpensive Databases (RAIDb). This acronym is very similar to the disc arrays known as Redundant Array of Inexpensive Disks (RAID). So is the data replication concept – controller is between backend to database and underlaying resources and it depends only on chosen scheme. While RAIDb-0 provides a performance scalability with full table partitioning, RAIDb-1 offers a full data replication. (Both concepts may be mixed in e.g. cascade configuration.)

3.2.7 Slony-I

Slony-I [13] is a asynchronous, trigger-based replication system originally written by Jan Wieck, software engineer of Afilias company which sponsored the project development.

At that time when the other replication projects had already worked, Slony-I appeard as a new concept. It created a replication mechanism that does not require only one version of PostgreSQL. It also gives the opportunity to start without starting/stopping the database engine or without the need for initial dump & restore of replicated database.

Slony-I supports the cascade replication (master to multiple chained slaves) that is usefull to reduce bandwidth usage or working load on the master system. Replication could be use to create a high availability database cluster or to do no-downtime migration to a new version of PostgreSQL.

Configuration of Slony-I replication system could be changed dynamically, but it does not have any functionality to detect a node failure, that is along with slave node promotion task of some other software.

Because Slony-I is trigger-based, it does not automatically propagate schema changes, thus it is not desirable for situations, when there are uncontrollable scheme changes (e.g. database service providers with independent customers or systems for development). Besides mentioned Data Definition Language (DDL) operations do not directly support the replication of SQL statements `GRANT` and `REVOKE`. It also does not replicate `large_objects`.

More about Slony-I features, limitations and configuration is written in the next chapter. Basically, Slony-I is a more complex way how to replicate the sophisticated databases than easy query-propagating systems, but has some specific issues.

3.2.8 Slony-II

Slony-II [14] is currently more likely a theoretical concept than a real implementation. It should be a synchronous replication mechanism that extends Slony-I and should also support multi-master scenarios (in fact originally Slony-I is only the first approach of main target of whole “Slony replication system family”).

3.2.9 Commercial Projects

There are also some only commercial projects which are focused on the PostgreSQL replication (in fact, PostgreSQL itself has some commercial forks). An example of only commercial project might be Mammoth Replicator [15] from Commnad Prompt, Inc.

4 PostgreSQL Replication with Slony-I

In this chapter the replication with Slony-I is beeing described. The introduction to the project is at the end of the previous chapter.

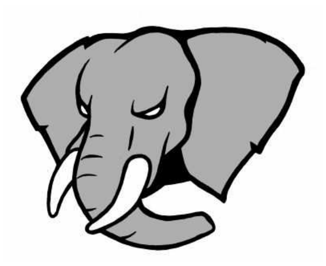


Figure 4.1: Slon, the elephant logo of the Slony-I.

By the way, “slon” is a Russian word for “elephant” which is a good variation to the mascot of PostgreSQL – an elephant as well. Slony is the plural of the word slon (group of elephants) and slonik (which will be introduced later in this chapter) is a Russian word for “little elephant”.

4.1 Principles of Slony-I Replication

Basic structure of the system using Slony-I is (in accordance to the “Initial concept paper”[17]) a master node with one or more slaves nodes. It allows scalability, the bandwitch save for master of backup site (slave modes may be connected in a cascade), and thus many failover scenarios.

Slony-I proclaims that the project is optimized for LAN (it was planned as a system for data centers and backup sites) – it assumes a quite wide bandwitch size between each connected node. There are not applied any operations which can lower size of the replication data package. For example there is no such operation as cancel redundant changes in transaction.

The costs of communication grow in a quadratic fashion in several directions as the number of replication nodes in a cluster increases. The more slaves we add, the more working load is placed on the master node of cluster (this could be alleviated by using cascaded slaves).

4.1.1 Triggers

Slony-I is an `AFTER ROW` trigger based replication system that analyzes the new and old rows to create pieces of SQL statements representing the change to the actual data row. Rows in the log must be identifiable, therefore it must have unique constraint which can be a compound key of any data types. The logs are stored in one of very few rotating log tables. It means that the replication engine can retrieve the actual data for one replication step with very few queries selected from one table only. Please notice that PostgreSQL does not provide the ability to define trigger called on schema changes (as many others database systems).

The replication mechanism uses triggers to collect table updates, where a single data update may be replicated to the multiple nodes. Replication of sequence numbers is also possible (sequences are optimized for concurrency, only guarantees not to generate duplicate ID), since the action sequence is allocated in an `AFTER ROW` trigger. Its ascending order is automatically in non-conflict order.

4.1.2 Replication Daemon

The replication engine itself is implemented in a node daemon[1] called Slon. This daemon consists of one hybrid program with master/slave functionality (which in fact is not really appropriate anyway, it is only configuration option and could be easily changed, e.g. in failover scenario).

A Slon instance is responsible for splitting the logdata, exchanging messages, confirming events, cleaning up and finally replicating data.

Slon checks in configurable interval if the log action sequence number has changed and if so, it will generate an event, which is created in serializable transaction and lock one object. This guarantees the right order of committed transactions, in which are logdata splitted.

All configuration changes and sync messages are communicated trough the system as events. An event is generated by inserting the event information into the proper table. Every Slon daemon establishes connection to the database, from where it receives events. The replication engine uses the PostgreSQL `LISTEN` and `NOTIFY` mechanism about event generation which creates Slony message-exchanging system.

Confirming events are realised more or less by inserting or deleting a row in one of the control tables (confirmation tables) which are made in same transaction that daemon processes the event.

Since all events are stored in tables, the replication daemon has to rotate (clean) them periodically. The confirmation data are condensed and the old event log data is removed.

Probably the most significant task for Slon is data replication. It means using the mechanism described above to collect log data from the master and apply them to the replicated database.

4.1.3 Replication Events

There are two main sorts of events – configuration events that submit updates to the configuration of the cluster and SYNC (synchronize) events, which update tables that are replicated.

4.2 Project Related Terms

Cluster is a named set of PostgreSQL replicated database instances Slony-I will create for each cluster individual schemas.

Node is defined as named PostgreSQL database. The replication cluster consists of a set of nodes.

Replication set is a set of tables and sequences that will be replicated between nodes in a cluster.

Origin node is the master provider, the main node, where the applications may change replicated data.

Subscribers are nodes which connect to data providers. This could be both the origin node and some node subscribed to origin node (cascade subscription). Origin node could not be a subscriber.

Providers are nodes that provide data for subscribers.

Slon daemons start for each node in the cluster, Slon makes replication itself.

[1]Daemon, a computer program that runs in the background, is spelled with inside "e" letter. Dæmon, from Greek world δαίμων, is usually good being, in contrast to demon, usually malevolent spirit, or fallen angel.

Slonik configuration processor is a tool for processes commands in a special language used to configuration's updates of a cluster.

4.3 Prerequisites

4.3.1 Software Requirements

Any platform that can run PostgreSQL should be able, thanks to the open-source code, to run Slony-I. Project itself was successfully tested on some version of Linux, Solaris and even Windows.

All the cluster nodes must have synchronized time. This can be easily achieved with Network Time Protocol [16] (implemented for example by ISC NTP daemon).

4.3.2 Installation

Slon, the replication daemon, is written in C programming language. To build a binary C compiler and the PostgreSQL sources are needed. All Slony-I source codes could be obtained from homepage `http://slony.info/downloads/`.

At the present time, version 1.2 is available as source codes or binary for Win32 system and RPM for some Linux systems. Slony-I could be also getted directly from distribution package repository – for example Debian GNU/Linux has main packages `postgresql-8.1-slony1` and `slony1-bin`. Installation and system integrations are with using distribution repository just one easy command.

4.4 Basic Configuration Parts

4.4.1 Replication Sets

Each replication set consists of tables, sequences and keys for tables without suitable primary key. Since every row in replicaton logs must be identifiable, each table must have the unique constraint – usually the primary key.

If the table has not the primary key, Slony-I could use some other unique constraint, if possible, for example index on a combination of fields that is both `UNIQUE` and `NOT NULL`. In fact, if we have such a combination in our table, we should really consider declare a primary key. Slony-I could also add the primary key to our table, but this option is not recommended, as this introduces the possibility that updates to this table can fail due to the new unique index.

It is a generally good idea to group tables that are related via foreign key together into a single set. We should be only beware of having all tables in one replication set, the initial copy event for large sets leads to long running transaction (databases with size of many gigabytes takes many hours or even days to copy in one replication set). Other problem may cause `EXECUTE SCRIPT`, Slony-I mechanism to perform DDL operations which request a lock on every single table in the replication set on origin and its subscribers.

We should also notice that replication of sequences requires more indirect costs for replication and its values have to be propagated regularly for each sequence.

4.5 Example

Since the example could do more than a thousands words, let's create first replicated database using Slony-I and `pgbench`, benchmarking tool from PostgreSQL contrib package.

4.5.1 Nodes in Replication Scenario

We should always do some planning before building a replication cluster. In this case the situation will be pretty clear – we will have two nodes running Debian GNU/Linux 4.0 (code-name Etch) on same hardware architecture, one node will be the master, second its subscriber. Database will be created from scratch.

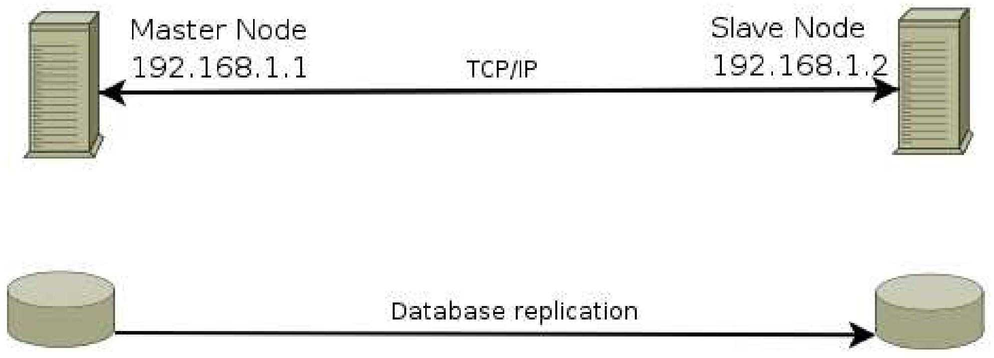


Figure 4.2: The master-to-slave replication scheme.

4.5.2 Preliminary

Assume we have already installed PostgreSQL. Now we add Slony-I packages, i.e. `slony1-bin`, `postgres-8.1-slony1` and `postgres-8.1-contrib` as well. Slon daemons have no Debian initial configurations, so there is no need for shutting down newly installed services.

We have to permit TCP/IP socket for `postmaster` (listen at all network interfaces including default `localhost`) and we also permit the access to database from our network `192.168.1.0/24`. In base directory (`/etc/postgresql/8.1/main/`) we add following line into `postgresql.conf`:

```
listen_addresses = '*'
```

and this line into file `pg_hba.conf`:

```
host    all         all         192.168.1.0/24        password
```

Then we create two database users, `pgbench`, which accesses benchmark database and superuser `slony`:

```
$ createuser -P pgbench
Enter password for new role: xxx
Enter it again: xxx
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n) n
CREATE ROLE

$ createuser -P slony
Enter password for new role: xxx
Enter it again: xxx
Shall the new role be a superuser? (y/n) y
CREATE ROLE
```

All changes must be done at all nodes as well as database creation (name of database, slony users and of course passwords could be different on master and slave):

```
$ createdb -O pgbench -E UNICODE pgbench
CREATE DATABASE
```

Next we fill database on master with `pgbench` in initialize mode, we create 1 000 000 tuples in the accounts table.

```
/usr/lib/postgresql/8.1/bin/pgbench -i -U pgbench -P xxx pgbench
```

Slony-I uses PL/pgSQL procedural language, so we install its support:

```
createlang plpgsql pgbench
```

Since Slony-I does not automatically copy database structure from one node to another, we do it manually:

```
master$ pg_dump -s pgbench > pgbench.sql
slave$ psql -d pgbenchslave -f pgbench.sql
```

4.5.3 Using Slonik to Create Configuration

It is quite common to generate Slony-I configuration instead creating it manually, but we make it in this simple scenario by hand using `slonik`, tool for processing commands in a special language.

Nevertheless, we do not use configuration scripts, we write configuration in a single file using Bourne Again Shell (BASH). All lines are commented, thus in my opinion the following configuration example is pretty obvious:

```
#!/bin/sh
slonik << EOF
  # namespace, used for name of new Slony-I scheme
  cluster name = first;

  # nodes and its network accessibility
  node 1 admin conninfo = 'dbname=pgbench      host=192.168.1.1 user=slony
   password=xxx';
  node 2 admin conninfo = 'dbname=pgbenchslave host=192.168.1.2 user=slony
   password=xxx';

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

```
    # public.history doesn't have primary key, we add one with Slony-I
    table add key (node id = 1, fully qualified name = 'public.history');

    # create replication set
    create set (id=1, origin=1, comment='All pgbench tables');
    set add table (set id=1, origin=1, id=1, fully qualified name = 'public.accounts',
     comment='accounts table');
    set add table (set id=1, origin=1, id=2, fully qualified name = 'public.branches',
     comment='branches table');
    set add table (set id=1, origin=1, id=3, fully qualified name = 'public.tellers',
     comment='tellers table');
    set add table (set id=1, origin=1, id=4, fully qualified name = 'public.history',
     comment='history table', key = serial);

    # create second node
    store node (id=2, comment = 'Slave node');
    store path (server = 1, client = 2, conninfo='dbname=pgbench
      host=192.168.1.1 user=slony password=xxx');
    store path (server = 2, client = 1, conninfo='dbname=pgbenchslave
      host=192.168.1.2 user=slony password=xxx');
  EOF
```

4.5.4 Replication Daemons

So far we have both of databases fully prepared. Next step is to configure and start up Slon replication daemons. We do it by "The Debian Way", creating the file `slon_tools.conf` `/etc/slony1/` directory:

```
  $CLUSTER_NAME = 'first';
  $LOGDIR = '/var/log/slony1';
  $MASTERNODE = 1;

  add_node(node     => 1,
           host     => '192.168.1.1',
           dbname   => 'pgbench',
           port     => 5432,
           user     => 'slony',
           password => 'xxx');

  add_node(node     => 2,
           host     => '192.168.1.2',
           dbname   => 'pgbenchslave',
           port     => 5432,
           user     => 'slony',
           password => 'xxx');

  1;
```

We choose on each node in file `/etc/default/slony1`, which part of the file to use (which node to start), for example on origin we use:

```
SLON_TOOLS_START_NODES="1"
```

After invoking start (`/etc/init.d/slony-1 start`) we should see debug output in `/var/log/slony1/`. Daemons can now communicate with each other, but replication is not started yet. We have to subscribe node 2 to provider – node 1. We do it again using `slonik`:

```
 #!/bin/bash
 slonik << EOF
  # namespace, used for name of new Slony-I scheme
  cluster name = first;

  # nodes and its network accessibility
  node 1 admin conninfo = 'dbname=pgbench      host=192.168.1.1 user=slony password=xxx';
  node 2 admin conninfo = 'dbname=pgbenchslave host=192.168.1.2 user=slony password=xxx';

  # subscribe node 2
  subscribe set (id = 1, provider = 1, receiver = 2, forward = no);
EOF
```

After Slonik script is started, Slon daemon on node 2 will immediately start to copy data from node 1. When the copy process is finished, Slon on node 2 will start to apply the accumulated replication log.

We may try it with very simple example:

```
pgbench=# SELECT tbalance FROM tellers WHERE tid=1;
tbalance
----------
        0
(1 row)

pgbenchslave=# SELECT tbalance FROM tellers WHERE tid=1;
 tbalance
----------
        0
(1 row)

pgbench=# UPDATE tellers SET tbalance=1337 WHERE tid=1;
UPDATE 1
```

After most highly default 100 milliseconds since `UPDATE` on master node ended, we should see in node 2 Slon's log information about performed SYNC and new value in replicated database:

```
syncThread: new sl_action_seq 1 - SYNC 113
localListenThread: Received event 2,113 SYNC
remoteListenThread_1: queue event 1,123 SYNC
remoteWorkerThread_1: Received event 1,123 SYNC
```

```
remoteWorkerThread_1: SYNC 123 processing
remoteWorkerThread_1: syncing set 1 with 4 table(s) from provider 1
ssy_action_list value:  length: 0
remoteWorkerThread_1: current local log_status is 0
remoteWorkerThread_1_1: current remote log_status = 0
remoteHelperThread_1_1: 0.004 seconds delay for first row
remoteHelperThread_1_1: 0.006 seconds until close cursor
remoteHelperThread_1_1: inserts=0 updates=1 deletes=0
remoteWorkerThread_1: new sl_rowid_seq value: 1000000000000000
remoteWorkerThread_1: SYNC 123 done in 0.021 seconds

pgbenchslave=# SELECT tbalance FROM tellers WHERE tid=1;
 tbalance
----------
     1337
(1 row)
```

4.5.5 Notes

Complete configuration files are stored at the covered CD. Debian `slony1` init script is not fully functional, it does not show status of running Slon daemons or it could not even stop it – it is necessary to use e.g. `killall slon` to stop daemon.

4.6 Interesting Usage Scenarios

Slony-I could be used in many scenarios, we will look at some main types.

4.6.1 Cascade Replication

Cascade replication with Slony-I is a very easy task, it only takes to properly configure the providers:

```
 #!/bin/bash
 slonik << EOF
  # namespace, used for name of new Slony-I scheme
  cluster name = cascade;

  # nodes and its network accessibility
  node 1 admin conninfo = 'dbname=sales host=192.168.1.1 user=slony password=xxx';
  node 2 admin conninfo = 'dbname=sales host=192.168.1.2 user=slony password=xxx';
  node 3 admin conninfo = 'dbname=sales host=192.168.10.1 user=slony password=xxx';
  node 4 admin conninfo = 'dbname=sales host=192.168.10.2 user=slony password=xxx';

  # subscribe nodes in main data center
  subscribe set (id = 1, provider = 1, receiver = 2, forward = no);

  # subscribe nodes in backup data center
  subscribe set (id = 1, provider = 1, receiver = 3, forward = yes);
  subscribe set (id = 1, provider = 3, receiver = 4, forward = no);
EOF
```

In this scenario we have a master node on host `192.168.1.1` which is replicated by hosts `192.168.1.2` and `192.168.10.1` which is placed on remote location and serves as source for host `192.168.10.2`. Queries from remote location could be performed on hosts `192.168.10.1` or `192.168.10.2`. It could save some network bandwitch or work load on the main host. Remote location could be easily promote to main replication node with Slonik `MOVE SET` command.

4.6.2 Upgrades Between Major PostgreSQL Releases

Replication may be used for helping perform upgrades of PostgreSQL servers, it could migrate data between databases without requiring a substantial downtime. The main benefit is a up-to-date copy in the node with a new version of PostgreSQL, while clients could perform data updates in the old version.

We simply create a new replication set likewise example in previous section and after new database is replicated, we stop all data-modifying applications and perform Slonik commands `LOCK SET` and `MOVE SET` for change a replication origin to the new node. This operation might take less than a second even if we have a quite large databases (a few tens of GB), while performing backup & restore (in addition with `INSERT` commands rather than `COPY`) could be very slow, e.g. requiring several hours.

4.6.3 Log Shipping Replication

Slony-I could spool transaction logs in directory which could be copied into remote location by some other mean than Slony direct connection, from file transfer protocols to pigeon post.

Slon daemon have to be started with option `-a archive_directory`. Logs could be applied with tool `slony_logshipper`.

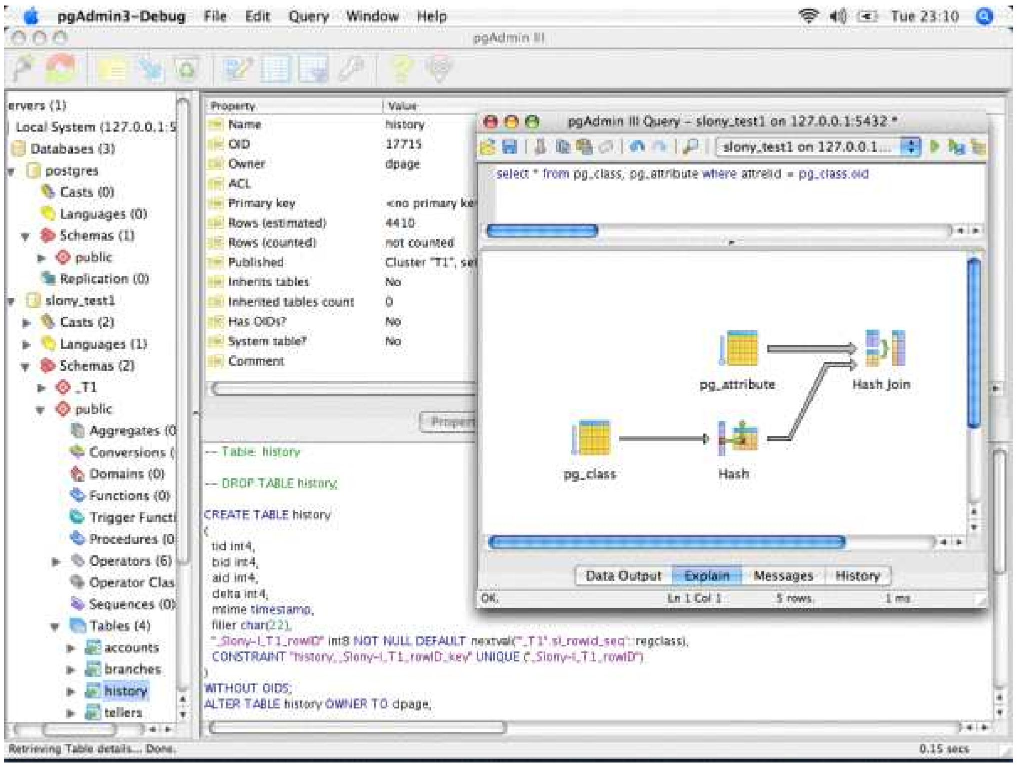
5 pgAdmin III – PostgreSQL Admin Tools

5.1 About pgAdmin III

On program homepage [18], pgAdmin III is described as the most popular and feature rich open-source administration and development platform for PostgreSQL. It is graphical front-end administration tool, available on popular computer platforms (e.g. Microsoft Windows, Linux, Mac OS, Solaris) and in several languages.

First prototype, pgManager, was written in 1998 for PostgreSQL 6.3. Current version of pgAdmin III (1.8.4 in early July 2008) allow to manage PostgreSQL version 7.3 and above as well as PostgreSQL derivatives (EnterpriseDB, Mammoth PostgreSQL etc.) with native data access, so no ODBC layer is needed. It is designed to help PostgreSQL's users, developers and administrators. Main features include syntax highlighting SQL editor, a server-side code and a configuration editor, a job scheduling agent and also extensive documentation.

pgAdmin III is an open-source software licensed under the Artistic License – it is more or less Free Software license. pgAdmin III is developed by a community of PostgreSQL experts around the world.



Property	Value
Name	history
OID	17715
Owner	dpage
ACL	
Primary key	<no primary ke
Rows (estimated)	4410
Rows (counted)	not counted
Published	Cluster "T1", se
Inherits tables	No
Inherited tables count	0
Has OIDs?	No
System table?	No
Comment	

```
select * from pg_class, pg_attribute where attrelid = pg_class.oid
```

```
-- Table: history

-- DROP TABLE history;

CREATE TABLE history
(
  tid int4,
  bid int4,
  aid int4,
  delta int4,
  mtime timestamp,
  filler char(22),
  "_Slony-I_T1_rowID" int8 NOT NULL DEFAULT nextval('"_T1".sl_rowid_seq'::regclass),
  CONSTRAINT "history__Slony-I_T1_rowID_key" UNIQUE ("_Slony-I_T1_rowID")
)
WITHOUT OIDS;
ALTER TABLE history OWNER TO dpage;
```

Figure 5.1: pgAdmin III, screenshot from Mac OS X.

5.2 Other PostgreSQL Administration Tools

There are two common PostgreSQL front-ends beside pgAdmin III – psql and phpPgAdmin, but as compared with pgAdmin, provide only simple and less powerful interface.

5.2.1 psql

The primary front-end shipped with PostgreSQL is a simple command-line utility `psql`. It can be used to execute the SQL queries from shell or file and makes administration or development tasks easy – for example it completes names or SQL syntax constructions using tab like populars shells.

5.2.2 phpPgAdmin

phpPgAdmin is more advanced, web-based administration tool. It partially reminds very popular web-based tool phpMyAdmin for MySQL database engine. It allows easy manipulation with data and could manages PostgreSQL objects. phpPgAdmin is quite popular for web hosting providers.

5.3 Current Slony-I Support in pgAdmin III

As referenced on Slony-I with pgAdmin online documentation [19], currently pgAdmin recognize Slony-I objects which are integrated into pgAdmin's main object tree browser, shows the status of the replication queue on the statistics tab and allows monitoring of Slony clusters.

5.3.1 Create Slony-I Structures

pgAdmin III allows through the use of simple tools to create the cluster on the master node, add slave nodes to the cluster, setup paths on each node to all other nodes, setup listens on each node to all other nodes, create replication sets, add tables and sequences to the sets and subscribe the slaves to the set. Nevertheless, cluster setup requires manually follow whole set of this steps, as we will see in the next example.

Goal is same as simple replication scenario in the previous chapter. We have two cluster nodes, master and slave, with two databases – our replication targets. PostgreSQL and Slony-I are installed and prepared. Now we have to go through this following steps:

- **Configure pgAdmin III.**

  Configuration of pgAdmin III is quite easy. All we have to do is to let pgAdmin know, where cluster creation SQL scripts are located (slony modules xxid and slony1_funcs, e.g. on Debian files `xxid.v80.sql` and `slony1_funcs.sql`). In our case we add directory `/usr/share/slony1/` into "Slony-I path" in File – Options ... – General.

- **Create the Slony-I cluster.**

  We create new cluster on the master node via New Slony-I Cluster dialog. Our master node will be named as "Master Node" with ID 1 and we also add the administrative node ID 99 "pgAdmin Node".

- **Add slave node to the cluster.**

  After the master node has been successfully created, slave node may be configured using the dialog New Slony-I Cluster with option "Join existing cluster".

  Slave node ID 2 will be named as "Slave Node", administrative node will be same as master's (selected from the combobox).

- **Setup node paths.**

  On each node, we have to create a path to every other node, so they can communicate properly. In our example case, this means create a path to the slave node on the master and a path to the master node on the slave.

Paths may be created using the New path dialog, we have to choose the host server and provide the connection string as specified in the PostgreSQL documentation [20], for example:

```
host=192.168.1.2 port=5432 dbname=sales2 user=slony password=xxx
```

We are using Slony-I version 1.2.1, so listens for nodes are created automatically. On versions before 1.1 is necessary to create listens (combination of origins and providers for the Slon process), later versions generate listens when paths are defined.

- **Create a replication set.**

  Next step is to create a replication set in the cluster, we have to do it on the master node via the New Replication Set dialog. First we create a new set, next we add tables to it. pgAdmin requires an unique index for each table (it does not add it automatically), because Slony-I requires that the each row in table must be uniquely identifiable.

- **Start slon processes.**

  Since we are using Linux, we start the Slon process manually using the same procedure by "The Debian Way", as described in the first example. We create a new file `/etc/slony/slon_tools.conf` with this configuration:

```
$CLUSTER_NAME = 'cluster';
$LOGDIR = '/var/log/slony1';
$MASTERNODE = 1;

add_node(node     => 1,
         host     => '192.168.1.1',
         dbname   => 'sales',
         port     => 5432,
         user     => 'slony',
         password => 'xxx');

add_node(node     => 2,
         host     => '192.168.1.2',
         dbname   => 'sales2',
         port     => 5432,
         user     => 'slony',
         password => 'xxx');

1;
```

  Next we choose the active node in `/etc/default/slony1` and start up the Slon daemon using init script:

```
/etc/init.d/slony1 start
```

- **Subscribe a replication set.**

  The final step is to subscribe a replication set. This operation is performed on the master (on Slony-1 versions before 1.1 on the slave node) using the New Subscription dialog.

Now we can finally test it with a simple example:

```
sales=# SELECT tbalance FROM tellers WHERE tid=1;
tbalance
----------
        0
(1 row)

sales2=# SELECT tbalance FROM tellers WHERE tid=1;
 tbalance
----------
        0
(1 row)

sales=# UPDATE tellers SET tbalance=1337 WHERE tid=1;
UPDATE 1
```

Slon performs SYNC operation and replicates data:

```
syncThread: new sl_action_seq 1 - SYNC 37
localListenThread: Received event 2,37 SYNC
remoteListenThread_1: queue event 1,49 SYNC
remoteWorkerThread_1: Received event 1,49 SYNC
remoteWorkerThread_1: SYNC 49 processing
remoteWorkerThread_1: syncing set 1 with 3 table(s) from provider 1
ssy_action_list value:  length: 0
remoteWorkerThread_1: current local log_status is 0
remoteWorkerThread_1_1: current remote log_status = 0
remoteHelperThread_1_1: 0.002 seconds delay for first row
remoteHelperThread_1_1: 0.009 seconds until close cursor
remoteHelperThread_1_1: inserts=0 updates=1 deletes=0
remoteWorkerThread_1: new sl_rowid_seq value: 1000000000000000
remoteWorkerThread_1: SYNC 49 done in 0.023 seconds
remoteWorkerThread_1: forward confirm 2,37 received by 1

sales=# SELECT tbalance FROM tellers WHERE tid=1;
 tbalance
----------
     1337
(1 row)
```

Cluster configuration was successful.

5.3.2 Execute DDL Scripts.

Since Slony-I trigger-based replication does not support the DDL operations (e.g. schema changes, `CREATE` ... and `ALTER` ... SQL operations), pgAdmin III offers a tool to apply the demanded change on all selected nodes.

5.3.3 Maintenance Tasks.

As we defined before, the replication set is a set of replicated tables and sequences. We could perform several Slony-I maintenance actions from pgAdmin III:

Restart node remotely restarts a Slon process on a node, re-initializes it and makes it reload its configuration.

Lock set disables replicated updates. This task is necessary for clean switch over the replication origin.

Unlock set enables previously locked replicated updates.

Merge set joins two sets, originating from the same node and subscribed by the same nodes into one.

Move set makes the target node the new source of replication set, for example to switch master role from one node to another.

6 Wizards Design

A wizard is part of a user interface where the user is led through a series of steps, performing tasks in a specific sequence.

6.1 Cluster Setup Wizard

The Cluster setup wizard will allow user to select objects on the master node, specify slave nodes and then do everything necessary to setup the cluster across those nodes.

Since cluster creation is a complex operation, wizard will have to perform these separated tasks:

- Ask user for input data (name of a new cluster, selection of cluster nodes, connection strings for setting paths, name of replication set(s) and its objects),
- create Slony-I cluster on master,
- add slave nodes,
- setup node paths (connects nodes to each other),
- create a replication set (from provided tables and sequences),
- subscribe a replication set,
- inform user about Slon daemon (which is part of Slony-I replication mechanism).

6.2 Wizard for Add New Node to the Cluster

Add a new node to the existing cluster consists from these steps:

- Ask user for input data (new node ID, comment and node connection string, list of cluster nodes available to connect),
- add slave node (join from slave node using provided connection string),
- setup node paths (for new node and for each selected node).

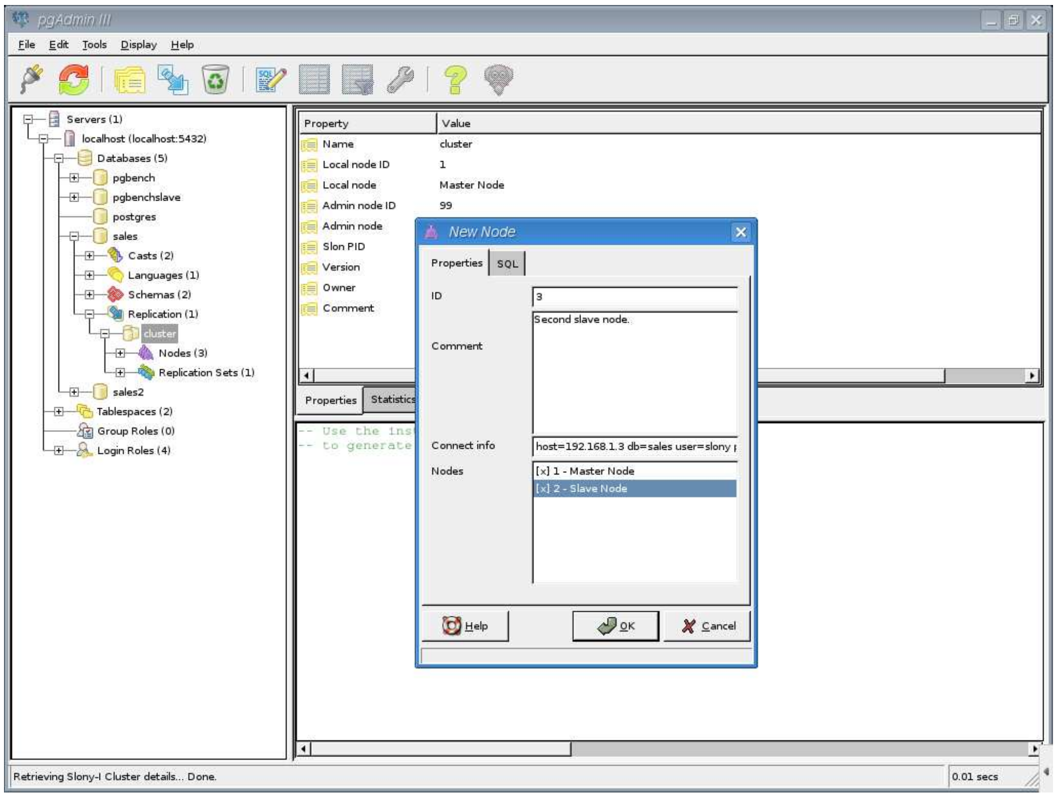
6.3 Wizard for Remove Node from Cluster

Wizard allowing remove the existing node from cluster will not be implemented – this function is already contained in the current version of pgAdmin III, node from cluster could be easily removed from the tree view by a simple tool.

6.4 Wizard for Add One or More Objects to the Replication Set

Addition of new objects (tables or sequences) are quite complex too. This mean to create a whole new set, since after a set has been subscribed, its table and sequence definition cannot be changed any more. So wizard will have to do:

- Ask user for input data (current replicaton set, new tables),
- create new a set with additional tables and sequences,
- subscribe exactly the same receiver nodes to it (as original set is),
- use Merge set action to merge both sets into the new one.



Property	Value
Name	cluster
Local node ID	1
Local node	Master Node
Admin node ID	99
Admin node	
Slon PID	
Version	
Owner	
Comment	

```
-- Use the ins
-- to generate
```

ID	3
Comment	Second slave node.
Connect info	host=192.168.1.3 db=sales user=slony p
Nodes	[x] 1 - Master Node [x] 2 - Slave Node

Retrieving Slony-I Cluster details... Done.

Figure 6.1: Add New Node wizard, screenshot from Linux, KDE.

6.5 Wizard for Remove One or More Objects from the Replication Set

Like wizard for removing node from cluster, tool for removing object from the replication set is already implemented, object removing is simple possible through the project tree.

7 Realization

pgAdmin III is written in C++ programming language, using cross-platform wxWidgets framework and XML-based resource system (known as XRC) with user interface elements stored in text files and loaded at run-time.

Addition to pgAdmin means from C++ point of view to integrate wizards classes into menu, register objects on menu values, fill in objects with methods for SQL generation and performing actions and create XRC templates. I created them using XRCed [21].

7.1 Slony-I Operations

For performing a separated wizard task, we need to know, how to configurate slony. In userspace, we would do it as in first example using the `slonik` command, in pgAdmin we will use directly SQL functions.

7.1.1 Cluster Setup

Creating a cluster from scratch means to initialize the first node at the same time. For this we use a set of files which are distributed with slony and contains declaration of the basic replication schema, declaration of replication support functions and SQL script for loading the transaction ID compatible datatype. All scripts have marks (e.g. `@NAMESPACE@`) which could be easily replaced with a specific name.

Master node could be simple initialized using functions from created schema. (For this examples I will use same "relative marks" with `@` instead specific values).

```
-- Initialize local node
SELECT @NAMESPACE@.initializelocalnode(@MASTER_NODE_ID@, '@MASTER_NODE_NAME@');
SELECT @NAMESPACE@.enablenode(@MASTER_NODE_ID@);
```

We could also create in this step the admin node.

```
-- Create admin node
SELECT @NAMESPACE@.storeNode(@ADMIN_NODE_ID@, '@ADMIN_NODE_ID@', false);
SELECT @NAMESPACE@.storepath(@MASTER_NODE_ID@, @ADMIN_NODE_ID@, '@MASTER_NODE_CONSTR@', 0);
```

7.1.2 Addition Node Setup

We have two options to add a new node into the cluster. First, we could carefully dump & restore a cluster schema from master. Second, we initilize node using procedure as presented in the previous example.

After initialization we have setup paths from and to the new node:

```
-- Create paths
SELECT @NAMESPACE@.storepath(@NEW_NODE_ID@, @EXISTING_NODE_ID@, '@NEW_NODE_CONSTR@, 10);
SELECT @NAMESPACE@.storepath(@EXISTING_NODE_ID@, @NEW_NODE_ID@, 'EXISTING_NODE_CONSTR', 10);
```

7.1.3 Replication Set Setup

A replication set could be easily created using:

```
  -- Create replication set
  SELECT @NAMESPACE@.storeset(@REPSET_ID@, '@REPSET_NAME@');
```

7.1.4 Addition Objects into Replication Set

Objects in the replication set could be tables or sequences. To add them we will use:

```
  -- Add table into replication set
  SELECT _@NAMESPACE@.setaddtable(@REPSET_ID@, (SELECT COALESCE(MAX(tab_id), 0) + 1
    FROM @NAMESPACE@.sl_table), '@TABLE@', '@TABLE_PKEY@', '@REPTABLE_NAME@');

  -- Add sequence into replication set
  SELECT @NAMESPACE@.setaddsequence(@REPSET_ID@, (SELECT COALESCE(MAX(seq_id), 0) + 1
    FROM @NAMESPACE@.sl_sequence), '@SEQUENCE@', '@REPTABLE_NAME');
```

7.1.5 Subscribe Set

Set is subscribed on the master using:

```
  -- Subscribe set
  SELECT @NAMESPACE@.subscribeset(@ORIGIN@, @PROVIDER@, @RECIEVER@, @WILL_FORWARD@);
```

7.1.6 Merge Sets into One

Merge sets into one is possible in some (previously described) conditions. Merge operation is called via statement:

```
  -- Perform MERGE
  SELECT @NAMESPACE@.mergeset(@TARGET_REPSET_ID@, @SECOND_REPSET_ID@);
```

7.2 Concept Example

A good example is the best sermon. Let's have a look at basic concept and implementation of one wizard.

7.2.1 pgAdmin III SQL Table

pgAdmin III have a tab showing SQL statements which will be performed after user ends dialog with the OK button. Wizards show SQL statements as well, but in more informative way – some parts are performed on the other PostgreSQL servers, so they are only mentioned in comments.

In pgAdmin, content of the SQL tab is an object returned by `GetSql()` method. This example is describing first part of addition node into the cluster, result will be SQL statements which will be performed on the new node.

```
wxString dlgRepNodeAdd::GetSql() {
  // init slave node of cluster
  wxString sql;
wxString createScript;

AddScript(createScript, wxT("xxid.v74.sql"));
AddScript(createScript, wxT("slony1_base.sql"));
AddScript(createScript, wxT("slony1_funcs.sql"));
AddScript(createScript, wxT("slony1_funcs.v74.sql"));
```

We could init a node cluster using slony modules `xxid` and `slony1_funcs`. `AddScript()` method takes content of files from disk, using directory provided with `settings->GetSlonyPath()`. Than we have to replace marks in file with a proper context (cluster name and schema prefix). Method `ReplaceString()` is used, because `wxString::Replace()` from framework is slow on large strings.

```
sql = wxT("-- NOTICE: This SQL commands may be performed on more than one host.\n\n");

sql += wxT("CREATE SCHEMA ")
    + ReplaceString(cluster->GetSchemaPrefix(), wxT("."), wxT("")) + wxT(";\n\n")
    + ReplaceString(createScript, wxT("@NAMESPACE@"),
      ReplaceString(cluster->GetSchemaPrefix(), wxT("."), wxT("")));

sql = ReplaceString(sql, wxT("@CLUSTERNAME@"),
      ReplaceString(cluster->GetSchemaPrefix(),
      wxT("_"), wxT("")));
```

Now we will have all necessary data and functions on a new node, we could use it to initialization.

```
// initialize new node
sql += wxT("\n\n -- Initialize new node.");
sql += wxT("SELECT ")
       + cluster->GetSchemaPrefix() +
       wxT("initializelocalnode(")
       + txtID->GetValue() +
       wxT(", '")
       + txtComment->GetValue() +
       wxT("');\n")
       wxT("SELECT ")
       + cluster->GetSchemaPrefix() +
       wxT("enablenode_int(")
       + txtID->GetValue() +
       wxT(");\n\n")
       wxT("-- In addition, the configuration is copied from
            the existing cluster.\n\n");
```

Next wizard task is to setup paths on new node – one path for each checked node. But to do this, we have to set up a new PostgreSQL connection, so we show user only comment.

```
  // setup paths on and to new node
  sql += wxT("\n-- Last part of SQL automatically create path on and to selected
               nodes and path from selected nodes to new node.\n")
         wxT("-- e.g. with
      SELECT _cluster.storepath(1, 2, 'host=localhost ...', 10\n");

  return sql;
}
```

7.2.2 Actions

After user press OK button, wizard performs prepared SQL and additional actions. Let's look through the code.

```
wxWindow *slonyNodeAddFactory::StartDialog(frmMain *form, pgObject *obj) {
  dlgRepNodeAdd *dlg=new dlgRepNodeAdd(&nodeFactory, form, (slCluster*)obj);
  dlg->InitDialog(form, obj);
  dlg->CreateAdditionalPages();
```

At first, we have to add our action part at a proper place. I chose to set the wizard window modal (via call of method `Go()` with `true` parameter). From method returned code could be decided, if user canceled dialog or not.

```
  if (dlg->Go(true) != wxID_CANCEL) {
```

Next we perform the prepared SQL statement on a new node. I added a new constructor of `pgConn()`, accepting now the PostgreSQL connection string instead a bunch of parameters. If connection fails, user is warned with the message box with exclamation mark icon. We do not delete `connectionToNewNode` (closes connection to the new node), because we use it again later.

```
    wxString sql;
    // we have already generated here:
    // * init slave node of cluster
    // * initialize new node
    sql = dlg->GetSql();

    pgConn* connectionToNewNode = new pgConn(true, dlg->GetConnstring());
    if ( !connectionToNewNode || connectionToNewNode->GetStatus() != PGCONN_OK ) {
      wxMessageBox(_("Problem when connection to database,
                     check provided connection string!"),
                     _("Database connection error!"),
                     wxICON_EXCLAMATION | wxOK);
      delete connectionToNewNode;
    } else {
      connectionToNewNode->ExecuteVoid(sql);
    }
```

Now we setup paths on the new node and to the new node. First part go in cycle trough nodes in the check list box and looks if any of them is checked.

```
// setup paths on/to new node
slCluster* cluster = (slCluster*)obj;
pgDatabase* database = obj->GetDatabase();

// for each selected node add path to new node
for (int i = 0; i < dlg->GetNodeCount(); i++) {
  if ( dlg->IsNodeChecked(i) ) {
```

We have to find connection string (`connstr`) for every node user chose. We get them from slony structures and used them for set up a new server connection.

```
    sql = wxT("SELECT pa_conninfo FROM ")
  + cluster->GetSchemaPrefix() +
  wxT("sl_path WHERE pa_server = ")
        + NumToStr(dlg->GetNodeID(i)) +
        wxT(";");
    wxString connstr = database->ExecuteScalar(sql);

    pgConn* connection = new pgConn(true, connstr);
    if ( !connection || connection->GetStatus() != PGCONN_OK ) {
      wxMessageBox(_("Problem when connection to database,
                     check provided connection string!"),
                   _("Database connection error!"),
                   wxICON_EXCLAMATION | wxOK);
    } else {
```

Connection string and information from user is all we need for setup paths. We do it via our doublet of connections.

```
      // path on new node
      sql = wxT("SELECT ")
          + cluster->GetSchemaPrefix() +
            wxT("storepath(")
          + NumToStr(dlg->GetNodeID(i)) +
            wxT(", ")
          + dlg->GetID() +
            wxT(", '")
          + connstr +
            wxT("', 10);\n");
      connectionToNewNode->ExecuteVoid(sql);

      // path to new node
      sql = wxT("SELECT ")
          + cluster->GetSchemaPrefix() +
           wxT("storepath(")
           + dlg->GetID() +
                wxT(", ")
                + NumToStr(dlg->GetNodeID(i)) +
                wxT(", '")
                + dlg->GetConnstring() +
                wxT("', 10);\n");
          connection->ExecuteVoid(sql);
```

After all SQL queries are executed, we close initialized connections and return.

```
        }
        delete connection;
      }
    }
    delete connectionToNewNode;
  }

  return 0;
}
```

8 Concept of Tests and Verifications

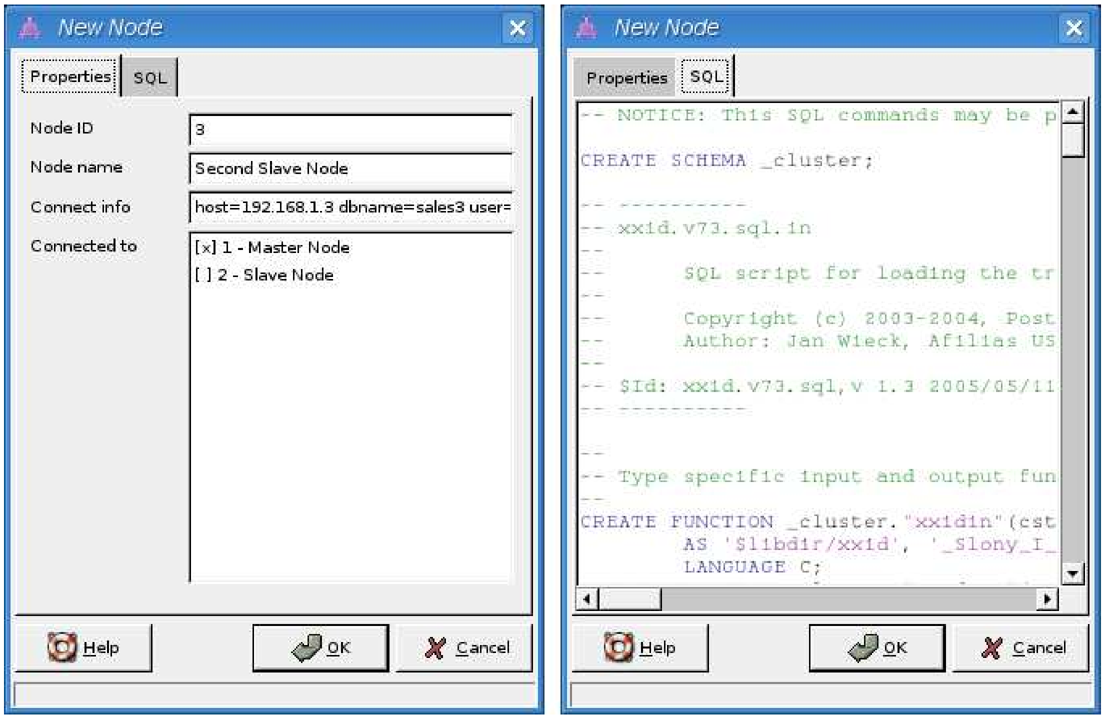
As Douglas Adams said, we should first see, later think and then test. But always see first. Otherwise we will only see what we are expecting.

We will show testing concept on example from the previous chapter.

8.1 Replication Scenario

We will use a simple scenario, where we have two existing nodes in replication cluster. Our concept example is used to add node into a cluster, so we will do it.

To add a new node to the cluster, we start with the wizard dialog using Menu – Tools – Replication – Wizards – Add node wizard.



Node ID: 3
Node name: Second Slave Node
Connect info: host=192.168.1.3 dbname=sales3 user=
Connected to: [x] 1 - Master Node
[ ] 2 - Slave Node

```
-- NOTICE: This SQL commands may be p
CREATE SCHEMA _cluster;

-- ----------
-- xxid.v73.sql.in
--
--      SQL script for loading the tr
--
--      Copyright (c) 2003-2004, Post
--      Author: Jan Wieck, Afilias US
--
-- $Id: xxid.v73.sql,v 1.3 2005/05/11
-- ----------


--
-- Type specific input and output fun
--
CREATE FUNCTION _cluster."xxidin"(cst
        AS '$libdir/xxid', '_Slony_I_
        LANGUAGE C;
```

Figure 8.1: New pgAdmin III wizard, main form, SQL table.

After we fill in all required values, main part of prepared SQL will be shown in "SQL" tab. Cancel button closes the dialog, OK button performs prepared SQL and additional actions.

8.2 Result

The result of prepared action could be verified in database mentioned in connection string and in databases of selected nodes. We could do it using simple SQL queries, testing the right preparation of `_cluster` scheme and right path selection.

```
sales3=# SELECT last_value FROM _cluster.sl_local_node_id;
 last_value
------------
          3

(1 row)
```

```
sales3=# SELECT * FROM _cluster.sl_path;
 pa_server | pa_client | pa_conninfo
-----------+-----------+--------------------------------------------------
         1 |         3 | host=192.168.1.1 port=5432 dbname=sales user=slony
           |           |  password=xxx
         3 |         1 | host=192.168.1.3 port=5432 dbname=sales3 user=slony
           |           |  password=xxx
(2 rows)
```

Values are right, test was successful. All new functions were tested using this tests' concept.

9 Conclusion

Slony-I, the PostgreSQL replication system, is a very powerful engine. I learned its principles of operation and configuration ways, which are described as an introduction in this thesis. I familiarized myself with administration and management tool pgAdmin III and I introduced its basic Slony-I support. I designed and implemented a set of user friendly graphical wizards, proof of concept leading to upstream pgAdmin III enhancement (code acception is in proceeding). Therefore, I consider main goals of this thesis as achieved. I suggest the possible extension of this work – to design and implement some other, in this thesis described, PostgreSQL replication engine, e.g. pgpool-II. It will be widely welcomed and as well as Slony-I wizards addition, it will help people to save time and money.

A Bibliography

[1] *PostgreSQL Homepage*
`http://www.postgresql.org/`
(last fetched on July, 5th 2008)

[2] *PostgreSQL*
Wikipedia, the free encyclopedia
`http://en.wikipedia.org/wiki/Postgres`
(last fetched on January, 4th 2008)

[3] *Ingres Frequently Asked Questions*
`http://www.bizyx.com/ingres/faq.htm`
(last fetched on January, 4th 2008)

[4] *Replication (computer science)*
Wikipedia, the free encyclopedia
`http://en.wikipedia.org/wiki/Replication_%28computer_science%29`
(last fetched on January, 4th 2008)

[5] *Red Hat Global Filesystem*
`http://www.redhat.com/gfs/`
(last fetched on January, 4th 2008)

[6] *DRBD Homepage*
`http://www.drbd.org/`
(last fetched on January, 4th 2008)

[7] *Implementing support for read-only queries on PITR slaves*
Google Summer of Code 2007 Application Information
`http://code.google.com/p/google-summer-of-code-2007-postgres/downloads/list`
(last fetched on January, 4th 2008)

[8] *Support for running readonly queries on PITR slaves*
Patch for PostgreSQL source.
`http://code.google.com/soc/2007/postgres/appinfo.html?csaid=6545828A8197EBC6`
(last fetched on January, 4th 2008)

[9] *Updated proposal for read-only queries on PITR slaves (SoC 2007)*
`http://archives.postgresql.org/pgsql-hackers/2007-03/msg00050.php`
(last fetched on January, 4th 2008)

[10] *pgpool-II Homepage*
`http://pgpool.projects.postgresql.org/`
(last fetched on January, 4th 2008)

[11] *PgCluster Homepage*
`http://pgcluster.projects.postgresql.org/`
`http://www.pgcluster.org/`
(last fetched on January, 4th 2008)

[12] *Sequoia Homepage*
`http://sequoia.continuent.org/`
(last fetched on January, 4th 2008)

[13] *Slony-I Homepage*
`http://www.slony.info/`
(last fetched on January, 4th 2008)

[14] *Slony-II Homepage*
`http://www.slony2.org/`
(last fetched on January, 4th 2008)

[15] *Mammoth Replicator Homepage*
`http://www.commandprompt.com/products/mammothreplicator/`
(last fetched on January, 4th 2008)

[16] *ntp.org: Home of the Network Time Protocol*
`http://www.ntp.org/`
(last fetched on January, 4th 2008)

[17] *Slony-I, A replication system for PostgreSQL – Concept*
Jan Wieck, Afilias USA, Inc.
`http://developer.postgresql.org/~wieck/slony1/Slony-I-concept.pdf`
(last fetched on July, 5th 2008)

[18] *pgAdmin III – PostgreSQL administration and management tools*
`http://www.pgadmin.org/`
(last fetched on July, 5th 2008)

[19] *Slony-I with pgAdmin III overview*
`http://www.pgadmin.org/docs/dev/slony-overview.html`
(last fetched on July, 5th 2008)

[20] *Database Connection Control Functions*
`http://www.postgresql.org/docs/8.1/static/libpq.html#LIBPQ-CONNECT`
(last fetched on July, 6th 2008)

[21] *XRCed Homepage*
`http://xrced.sourceforge.net/`
(last fetched on July, 8th 2008)

B Selected Content of Covered CD

Most important content/directories of attached CD-ROM:

- `build/`, original and additional sources of pgAdmin III, among with custom built Debian "Etch" packages,
- `first-example/`, support scripts and configuration file for setup a simple Slony-I cluster on Debian GNU/Linux. Directory `examples/` is example configuration file from Debian package,
- `images/`, source of images used in this thesis,
- `slony/`, Slony-I binaries and sources,
- `slony-latex/`, LaTeXsources of this thesis with complete PDF version.

pgAdmin III is licensed under the Artistic License. Slony-I is licensed under the BSD License.