PostgreSQL

Jason Healy, Director of Networks and Systems

Last Updated Mar 18, 2008

Contents

Chapter 1

PostgreSQL

Last updated 2008/03/18

1.1 Introduction

PostgreSQL (pronounced "post-gress-cue-ell", or "postgres" for short) is an open-source relational database engine. The database runs on Unix variants (Linux, BSD, Mac OS X), as well as Windows. Clients can connect to the database using native clients or JDBC. Interaction with the database is performed using SQL. For more information about the features of PostgreSQL, please visit the project's web page:

http://www.postgresql.org/

PostgreSQL is an alternative to other SQL-based database products, such as MySQL, Microsoft SQLServer, Oracle, and FileMaker (note, however, that FileMaker has a built-in GUI tool which most other databases do not have). In terms of free database engines, PostgreSQL "competes" most directly with MySQL. Suffield uses PostgreSQL because it is free, robust, and has some features that MySQL lacks. Additionally, while most MySQL software can be made to run under PostgreSQL, the reverse is not always true.

A full explanation of relational database systems is beyond the scope of this document; it is assumed that the reader is familiar with the basic concepts of relational databases, tables, columns, data types, and SQL queries.

This document describes the basic installation, setup, use, and maintenance of a PostgreSQL database server.

1.2 Installation

Suffield uses Mac OS X and Debian Linux as its primary server platforms. Please follow the instructions for the appropriate platform.

1.2.1 Debian Linux

Debian Linux supports PostgreSQL as a package in their standard distribution. Install the `postgresql` and `postgresql-client` packages for the latest stable version, or include a version number to get a specific revision:

```
apt-get install postgresql-8.2 postgresql-contrib-8.2
```

The packages will be downloaded and installed automatically.

Note that different versions of Debian support different versions of PostgreSQL in their package repositories. Also, depending on the relase dates of Debian and PostgreSQL, you may not be able to get the latest version in your distribution.

To get the latest version in Debian, you can use "apt-pinning" to draw in selected files from unstable distributions of Debian. To do this, add the "backports", or "unstable" (or possibly "experimental") releases of Debian to your `/etc/apt/sources.list`:

```
# only need one of these!
deb http://www.backports.org/debian etch-backports main contrib non-free
deb http://debian.lcs.mit.edu/debian/ unstable main
deb http://debian.lcs.mit.edu/debian/ experimental main
```

Then, set the pin-priority for these new releases in `/etc/apt/preferences` (create the file if it doesn't exist):

```
Package: *
Pin: release a=stable
Pin-Priority: 700

Package: *
Pin: release a=etch-backports
Pin-Priority: 650

Package: *
Pin: release a=unstable
Pin-Priority: 600

Package: *
Pin: release a=experimental
Pin-Priority: 500
```

Note that the "stable" distribution has the highest priority. This means that only packages that cannot be found in the stable distribution will be fetched from the unstable distribution.

Now you can run `apt-get update` and install the version of PostgreSQL you'd like.

1.2.2 Mac OS X

Note that PostgreSQL can be installed on a regular "client" build of Mac OS X; you do **not** need the "server" version of the OS.

Mac OS X supports several methods for installing PostgreSQL: via Fink, DarwinPorts, or PostgreSQL Tools for Mac OS X. The first two are packaged installers using a build or package framework (DarwinPorts is like "ports" for BSD, and Fink is like "apt-get" for Debian). The latter is a native Mac OS X package that installs independently of other software.

However, most of these packaging systems only install the base version of the server. We have since switched to compiling our own binary direct from source, as this gives us access to some necessary extensions for the software. Don't worry; this isn't difficult at all (the software builds cleanly on Mac OS X with no additional dependencies).

Downloading

To obtain PostgreSQL, just visit their website:

http://www.postgresql.org/ftp/source/

That's a direct link to their source section; you can also go to their main site and follow links to their download section.

Download the full source distribution (it's the one that's called `postgresql-X.Y.Z.tar.bz2`). The `base`, `docs`, `opt`, and `test` tarballs are all included in this file, so you don't need to download them separately.

Compiling and Installing PostgreSQL

You'll need to have a compiler available on the machine, which means installing **Apple's Developer Tools CD**. You can get it from:

http://connect.apple.com/

Once the developer tools are installed, you're ready to build PostgreSQL.

Unpack the tarball into a directory that's stable (we want to keep the compiled sources around for extensions, so don't put the sources anywhere that's temporary):

```
tar -jxf postgresql-X.Y.Z.tar.bz2

cd postgresql-X.Y.Z
```

Now it's time to run the `configure` script. Nearly all the options that PostgreSQL uses are supported out-of-the-box by Mac OS X, so we enable most of the options. We tell the configure script to place the resulting software in `/opt/postgresql-X.Y.Z`, though you may choose to put it almost anywhere (`/Applications`, `/Library/PostgreSQL`, *etc.*):

```
./configure --prefix=/opt/postgresql-X.Y.Z --with-perl --with-tcl \
            --with-python --with-krb5 --with-pam --with-ldap \
            --with-bonjour --with-openssl
```

Once the configure script has completed, you're ready to build the software. Run make:

```
make
```

And wait for the compilation to finish. Assuming there are no errors, you may now install the software:

```
sudo make install
```

You now have a base installation of PostgreSQL installed.

Compiling and Installing Extensions

In addition to the base install, we install a few extensions that are useful, or required by other software that uses PostgreSQL. These programs live in the `contrib` directory in the source folder, so we must move there first:

```
cd contrib
```

First, we build the `tsearch2` full-text search engine:

```
cd tsearch2
make
sudo make install
cd ..
```

We also use the xml2 extensions:

```
cd xml2
make
sudo make install
cd ..
```

The full suite of software is now installed.

Account Creation

You must create a `postgres` user account on your machine, if one does not exist already. You can do this using the usual account creation tools (**System Preferences** or **Workgroup Manager**), or you can do it from the command line using this script:

```
#!/bin/bash

NAME="postgres"
HOME="/opt/postgresql-X.Y.Z"

# Check to see if the user exists or not
nicl . -read /users/${NAME} >/dev/null 2>&1
if [ $? != 0 ]; then

    echo "User ${NAME} does not exist; creating..."

    # delete any pre-existing group with our name (shouldn't happen)
    nicl . -delete /groups/${NAME} >/dev/null 2>&1

    # Find the next shared UID/GID that's between 100 and 500
    lastid=`(nireport . /users uid ; nireport . /groups gid) | sort -n | uniq | egrep -v '^([0-9]{1,2}|[5-9][0-9]{2,}|[0
    id="`expr $lastid + 1`"

    # in the case that there is no ID over 100 to come after, just start at 200
    if [ ${id} -lt 100 ]; then
        id=200
    fi

    # create new group
    echo "Creating group ${NAME} with GID $id"
    nicl . -create /groups/${NAME}
    nicl . -createprop /groups/${NAME} gid ${id}
    nicl . -createprop /groups/${NAME} passwd '*'
    echo "niutil: group '${NAME}' added."

    echo "Creating user ${NAME} with UID ${id}"
    nicl . -create /users/${NAME}
    nicl . -createprop /users/${NAME} uid ${id}
    nicl . -createprop /users/${NAME} gid ${id}
    nicl . -createprop /users/${NAME} passwd '*'
    nicl . -createprop /users/${NAME} change 0
    nicl . -createprop /users/${NAME} expire 0
    nicl . -createprop /users/${NAME} realname "PostgreSQL Database"
    nicl . -createprop /users/${NAME} home "${HOME}"
    nicl . -createprop /users/${NAME} shell '/usr/bin/false'
    nicl . -createprop /users/${NAME} _writers_passwd "${NAME}"
    echo "niutil: user '${NAME}' added."
else
    echo "User '${NAME}' already exists; not creating it again"
fi
```

Note that the password is blanked out for security purposes. If you create the script using a GUI tool, you should disable the password.

Finally, you'll need to create two directories and own them to the `postgres` user:

```
mkdir /opt/postgresql-X.Y.Z/data
mkdir /opt/postgresql-X.Y.Z/log

chown -R ${NAME}:${GROUP} /opt/postgresql-X.Y.Z/data
chown -R ${NAME}:${GROUP} /opt/postgresql-X.Y.Z/log

chmod 700 /opt/postgresql-X.Y.Z/data
```

Cluster Creation

A new database cluster must be **initialized** before it can be used. This is done by running the `initdb` command as the `postgres` user:

```
sudo -u postgres /opt/postgresql-X.Y.Z/bin/initdb -D /opt/postgresql-X.Y.Z/data
```

The database is now initialized and ready to run.

PATH setup

By default, the server and client binaries are installed in `/opt/postgresql-X.Y.Z/bin`. To enable local machine users to use these binaries without having to specify the full path, you should add a stanza similar to the following to your `/etc/profile` (for system-wide effect), or to your `~/.profile` (single user):

```
if [ -d /opt/postgresql-X.Y.Z/bin ]; then
    PATH=${PATH}:/Library/PostgreSQL8/bin
    export PATH
fi
```

LaunchDaemon Startup

If you've installed on a machine running Mac OS X 10.4 or later, you may wish to use `launchd` to start and stop the database engine. To do so, follow these steps:

1. Copy our LaunchDaemon plist to the `/Library/LaunchDaemons/` directory on the database server.

2. Customize the plist file to include the correct path to the server (replace `X.Y.Z` with your actual version of PostgreSQL).

3. Start the server using `launchd`:

   ```
   sudo launchctl load -w /Library/LaunchDaemons/org.postgresql.postmaster.plist
   ```

4. Confirm that the server is running:

   ```
   ps auxwww | grep postgres
   ```

   You should see several entries for the database processes.

1.3 Configuration

Once the database is installed, you should take a few minutes to configure the database engine. Most of these configuration options have to do with network security and who is allowed to access the database.

Under Mac OS X, the configuration files live in the `data` folder, relative to where you installed the software:

`/opt/postgresql-X.Y.Z/data`

You'll need to be a super-user or the user `postgres` in order to access this location on the hard drive. To become the `postgres` user, execute the following:

```
sudo -u postgres /bin/bash
```

That will give you a login shell as the `postgres` user. You can then enter the directory listed above.

1.3.1 Schemas

A PostgreSQL cluster may contain one or more **databases**. Each one of these databases is logically separated from the others; tables, functions, triggers, and other objects in a database cannot affect others. This is desirable because it allows different applications to have their own databases without conflicting with each other.

However, this also prevents sharing information between two databases. At Suffield, we have several different applications that rely on a database, and sometimes we'd like to tie these data together. For example, we might have a database with user information, and another database with phone accounting records. Ordinarily, the applications that use these databases know nothing about each other. However, we might wish to run a report that correlates phone usage with user names.

In earlier versions of PostgreSQL, the only way to share data was to assign all tables to the same database. This was problematic, because different applications might each want a table with the same name (*e.g.*, "users").

With recent versions of PostgreSQL, we can use the *schema* feature:

http://www.postgresql.org/docs/current/static/ddl-schemas.html

Schemas allow multiple applications to share the same database, but each have their own namespace for tables and other objects. This way, applications can operate on their own data (just like they would in separate databases), but also cross to other schemas to get data from other applications.

To create a schema, just issue the following:

```
CREATE SCHEMA myschema AUTHORIZATION myuser;
```

That creates a schema and assigns all rights to the given user (you may omit the AUTHORIZATION portion if you don't want to assign the privs to another user).

Now you can create tables (or other objects) within the schema:

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

Note the fully-qualified name. If you don't want to specify that for every table, you may set the default schema for a user by executing:

```
ALTER USER myuser SET search_path TO myschema;
```

1.3.2 Tablespaces

PostgreSQL allows you to store different parts of the database on different logical volumes:

http://www.postgresql.org/docs/current/static/manage-ag-tablespaces.html

For systems with multiple disks, this allows you to spread the load to different disks, or to assign different tables to filesystems with different levels of performance, reliability, or size.

To create a new tablespace, make a directory on a logical filesystem and own it to the PostgreSQL database user. Then, issue the following command in the PostgreSQL shell:

```
CREATE TABLESPACE spacename LOCATION '/path/to/spacename/dir';
```

Now, when you create a table in PostgreSQL, supply the tablespace as the final argument. For example:

```
CREATE TABLE foo(bar VARCHAR) TABLESPACE spacename;
```

If you'd like to permanently set the default tablespace for a particular user (so you don't need to specify it for each table), you can issue the following:

```
ALTER USER myuser SET default_tablespace TO spacename;
```

This may be helpful when you wish to set a user to always default to a given tablespace.

1.3.3 Host-based Authentication

Once remote access has been turned on, we PostgreSQL must be told which machines and users may connect to which databases. This is done by editing the `pg_hba.conf` file.

Generally, you should only allow connections from known, trusted machines. Additionally, you should only allow connections from given usernames to specific databases whenever possible. We make an exception for "localhost" so that users can connect directly on the database machine itself.

Below is our sample `pg_hba.conf` file:

```
# Database administrative login by UNIX sockets
```

```
local   all         postgres                          ident sameuser

# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD

# "local" is for Unix domain socket connections only
local   all         all                               ident sameuser
# IPv4 local connections:
host    all         all         127.0.0.1/32          md5
# IPv6 local connections:
host    all         all         ::1/128               md5

# Allow "testuser" to connect from a particular machine
host  suffield  testuser                 172.30.0.40/32  md5

# Reject all others
host    all     all       0.0.0.0      0.0.0.0          reject
```

Here, we allow users on the machine to connect directly without a password, but they cannot specify a different name.

Users can connect to the local loopback interface and specify a username and password (md5 encrypted). We also include IPv6 addresses for completeness.

We then have a list of trusted hosts (well, just one) that allows a particular user access to a particular database from a particular machine. Note that if you use schemas (see above), the database name will most likely be the same for all users.

Finally, we round out the file with an explicit "reject" statement to prevent all other connections.

1.3.4 Allowing Remote Access

By default, the database only listens on the local loopback interface for network connections. This is fine if all your database applications are hosted on the same server (*e.g.*, you're running the database on the same machine as your web server, and the web server is the only client of the database). However, if you need to allow remote machines access to the database, you'll need to change this behavior.

Edit the configuration file `postgresql.conf`. Find the variable `listen_addresses` and set it to `*`:

```
listen_addresses = '*'
```

1.3.5 Autovacuum

From time to time, a PostgreSQL database must be **vacuumed**. This process reclaims lost space in the database, prevents transaction ID wraparound, and generally keeps the database in good working order.

Starting in version 8.1, you can enable an "autovacuum" process that runs concurrently with the database engine. This prevents the need for a manual vacuum by the database administrator.

In the `postgresql.conf` file, change the following lines:

```
stats_start_collector = on
stats_row_level = on

autovacuum = on
```

You may tweak the other settings as you see fit; we currently use the default timeouts and load factors for autovacuuming.

1.3.6 Logging

PostgreSQL can log to STDERR, Syslog, or to a file. To prevent logs from filling up our system logs, we redirect all log output to files.

In the `postgresql.conf` file, change the following lines:

```
log_destination = 'stderr'
redirect_stderr = on
log_directory = '../log'
log_filename = 'postgresql-%Y-%m-%d_%H-%M-%S.log'
log_rotation_age = 86400
client_min_messages = notice
log_min_messages = notice
log_min_error_statement = error
log_line_prefix = '%t %d %u@%h'
```

1.3.7 Tweaking Kernel Memory Parameters

PostgreSQL works quite well "out of the box" but can benefit from some additional tuning. For high database loads, tuning becomes necessary to keep the database running smoothly.

There are many things to tweak in PostgreSQL; an annotated guide to some of these settings are available at the following links:

http://www.varlena.com/GeneralBits/Tidbits/annotated_conf_e.html http://www.powerpostgresql.com/PerfList/

PostgreSQL benefits from larger amounts of memory being made available to the database engine. Specifically, you may wish to turn up the amount of shared memory available. On Mac OS X, the default amount of shared memory is 4MB; we turn this up to 128MB.

Changing the kernel parameters requires use of the `sysctl` command. To make the changes permanent, you can create (or edit) the `/etc/sysctl.conf` file.

Mac OS X

Under Mac OS X (10.3.9 or later), you should add the following lines to `/etc/sysctl.conf` (the example below assumes 128MB of shared memory):

```
kern.sysv.shmmax=134217728
kern.sysv.shmmin=1
kern.sysv.shmmni=32
kern.sysv.shmseg=8
kern.sysv.shmall=32768
```

Note that you need to add **all five values**, even if you're using the defaults for some of them (in the example above, we are only using non-default values for `shmmax` and `shmall`). Mac OS X triggers on the fifth value to set up the shared memory system, and the changes become permanent (until reboot) after that point. If you don't set all five, the change doesn't take, and will be overwritten by the Mac OS X defaults.

Linux

Under Linux, you should add the following lines to `/etc/sysctl.conf` (the example below assumes 128MB of shared memory):

```
kernel.shmall=134217728
kernel.shmmax=134217728
```

1.3.8 PostgreSQL Shared Memory Parameters

Once you've set the kernel's memory parameters (and rebooted if necessary), you can configure PostgreSQL to take advantage of this memory.

Using our system shared memory amout of 128MB, here are some suggested tweaks to `postgresql.conf`. Note that the settings shown are for PostgreSQL 8.2; if you use an earlier version you may not be able to use the "MB" and "kB" suffixes for values, and must instead convert them to ”pages”. See the comments of the file for page size information.

Allocate all but 32MB of the shared memory to the PostgreSQL shared buffer cache:

```
shared_buffers = 96MB
```

This gives a chunk of shared memory over to PostgreSQL to cache table entries for faster searching.

You may also allocate 128MB (assuming your system has enough free memory) to the ”maintenance_work” memory:

```
maintenance_work_mem = 128MB
```

This setting is used for VACUM, ANALYZE, and CREATE INDEX statements, and so can speed up those operations.

1.3.9 Speeding Up Disk Access

PostgreSQL uses a write-ahead log (WAL) to commit its changes. To ensure consistency, the database engine writes these log files just before comitting a transaction, and then flushes the log to disk using `fsync`.

While this process preserves data consistency (a good thing), it does slow the database down, as the logs must be flushed.

To speed up this process, there are a few steps we can take to speed up database operations. **Note:** some of these operations are **dangerous** in the sense that they can cause the database to possibly lose data.

- First, mount the `data` directory of the database on its own disk. Ideally, this should be a fast disk (RAID 1+0 is recommended for optimum performance). RAID 0 is fast, but unable to withstand failures. At the same time, RAID 1 is fault-tolerant, but has slow write performance. The risk of doing this is directly linked to the stability of the disk.

  We used to have Apple Xserve hardware, which has a maximum of 3 SCSI drives. Therefore, RAID 1+0 is not possible. We opted for a RAID 1 setup with a write-back cache (and a read-ahead for reads, though this isn't as important). The write-back cache gives reasonable write performance and the RAID 1 gives excellent read performance. In the event of a power failure, we have a UPS and generator to prevent data loss in the cache.

We have since moved to a Linux-based Compaq box with 18 drives, organized into two logical arrays running RAID 1+0. This gives us maximum redundancy with increased speed.

- Next, consider mounting the `pg_xlog` subdirectory of the `data` directory on its own disk. This is the directory that contains the write-ahead logs, so making it independent from the regular database store helps speed access. Again, you must balance the reliability of the drive with the possible speed increases.

- Finally, you can tweak the `fsync` and `wal_sync_method` parameters. These affect how the database writes the WAL out to disk. Note that turning off `fsync` can speed up writes significantly, but can be **very dangerous**. If the WAL files become corrupted, the database may be unable to automatically recover in the event of a system crash or power failure.

1.4 PostgreSQL Usage

In general, interaction with PostgreSQL occurs in three major ways:

1. Server Management commands, used to create or delete entire databases. Creation of users and their permissions is also frequently done through external commands. Finally, full backups and restores using human-readable file formats are performed using external utilities.

2. Manual database interaction, using commands entered at a command prompt via a special database shell. Most often used to test queries, or perform small maintenance tasks.

3. Programatic access using a language that supports direct communication with the database. Usually, these are SQL statements sent from the program to the database, with the program receiving and interpreting the results. Examples of languages that can communicate with PostgreSQL include (but are not limited to) Perl, Java, PHP, and C.

1.4.1 Server Management

Most server management is performed using special binaries that are included with the PostgreSQL distribution. Note that the binaries may reside on a special path on your system (especially under Mac OS X), so if you don't seem to have the binaries installed, be sure to search in the same directory tree as the database itself.

All commands have well-documented manual pages. Copies of the documentation are also available on the PostgreSQL web site.

:`createuser` / `dropuser`

Creates a database user credential. These credentials are frequently used for remote access to a database, and usually include a username/password pair.

The command takes several options, which describe the capabilities of the user. Read the documentation for the command carefully, and only allow the minimum privileges necessary for the user.

:`createdb` / `dropdb`

Creates or destroys a database. A *database* is a related group of tables that are accessed under a common name. It is possible (and highly likely) that you will have several databases run by a single instance of PostgreSQL.

This command supports a few options, including the `--owner` flag, which lets you specify a user who "owns" (has full administrative control over) the database. Usernames must be created using the `createuser` command (described above).

:`pg_dump/pg_dumpall/pg_restore`

Commands to back up and restore a database, or entire cluster of databases. `pg_dump` only dumps a single database, while `pg_dumpall` saves the contents of all the databases in the cluser. `pg_restore` will restore a database from the output produced from dumping the database.

The dumpfiles used by these commands are in plain text (or may be compressed using `gzip`. As such, they are highly portable, but not terribly space-efficient. Also, they are not easily used for incremental or differential backups (see later sections of this document for alternatives). As such, these utilities should be used only for periodic or transition-based full backups.

1.4.2 Manual Interaction

The command `psql` launches a database shell, which allows the user to send SQL statements directly to the database. Additionally, a small subset of PostgreSQL-specific commands are supported (mostly for querying the schema and data definitions).

To launch `psql`, you normally include a database name to connect to, a username to connect with, and a hostname to connect to (though you may omit any of these options if the defaults suffice). Thus, a typical invocation looks like:

```
psql -d mydatabase -U myusername -h localhost
```

Once connected, you can enter SQL statements directly at the prompt, terminated by a semicolon. The statements will execute, and any results (or errors) will be returned directly to the screen. On recent versions, large result sets will be piped to a paging program (such as `less`) for easy viewing.

To disconnect from the shell, type `^D` (control-d).

1.4.3 Programatic Access

Programatic access usually requires a library call from within the application. Generally, the application must connect to the database, issue SQL commands, retrieve the results of these commands, and then disconnect from the database. Each library handles this process differently, so a full treatment is beyond the scope of this document.

In general, you will need the following information in order to connect to PostgreSQL from an application:

- The name of the database
- A username to connect under
- The password that goes with the username specified
- The hostname to connect to

As a **very basic** example, here is a small Perl program that connects to a fictitious database and executes a small query:

```
use DBI;

# Connect to the DB
my $dbh = DBI->connect('DBI:Pg:dbname=mydatabase;host=127.0.0.1',
                       'myusername', 'mypassword',
                        { PrintError => 0, RaiseError => 0 } )
      or die ("Could not connect to database: " . $DBI::errstr);

# Prepare the query
$sth = $dbh->prepare("SELECT * FROM widgets")
      or die ("Could not prepare statement: " . $dbh->errstr);

# Execute the query
$sth->execute()
      or die ("Could not execute database query: " . $dbh->errstr);

# Fetch the results into a perl hashref
my $results = $sth->fetchall_arrayref({});

# Disconnect when done
$dbh->disconnect();
```

Other languages will use different syntax and library calls, but the general format remains similar to what we've described above. Consult your language's documentation for more information.

1.5 Backing Up

PostgreSQL contains utilities for backing up and restoring an entire database: `pg_dump`, `pg_dumpall`, and `pg_restore`. However, these utilities are focused on backing up entire databases in order to perform complete restores of a database. While these kinds of backups are helpful, they are time-consuming and quickly become out-of-date on a busy database.

PostgreSQL also supports a "hot" version of backup, by relying on its Write-Ahead Logging (WAL) capabilities. For consistency, PostgreSQL always writes any transaction to a special log file before changing the database. In the event of a system crash, this log file can be "replayed" so as to complete any in-progress transactions.

We can use the WAL as a form of incremental backup; if we take a **base** backup of the entire database and store every WAL file generated from that point onwards, we essentially have a collection of data that represents the entire database.

Because the WAL files are generated regularly, we can back each one up as it is created. Since the files are small, we can back each up as it is created, ensuring a near-exact copy of the database at all times.

Suffield uses a hybrid approach for database backups: we take regular full dumps of the database to ensure easy restoration. Additionally, we use the WAL to keep up-to-the-minute backups of the database to prevent data loss.

1.5.1 Suffield Overview

Here at Suffield, we have a single master PostgreSQL server. This server has a spare disk drive in it, which we use for "warm" backups of the database. Additionally, we spool these backups to a remote host on a daily basis for added peace of mind.

Here is a basic overview of how the backups work on our machine. The following process is executed once daily (usually late at night):

- We take a full dump backup of the database to our secondary disk on the database server. This file is also backed up to a remote host.

- We start a new PITR checkpoint backup, archiving the old PITR backup remotely.
- The database archives WAL files one at a time as they are rolled over. As part of the WAL archive, the files are backed up remotely at the same time.

1.5.2 The “postgresql_archive“ Script

To support our PostgreSQL backups, we have written a Perl script called `postgresql_backup`. It takes care of all the different types of backups, and includes a mode that runs all the backups in the correct order. Once a day, we invoke this mode to perform the full database backups. The rest of the time, the script is called by the database to archive the WAL files.

The following section describes the setup and use of this script.

Preparation

The script requires a small amount of preparation in order to work properly.

1. First, you must set aside space on a local filesystem to store database backup files. Ideally, this should be on a separate disk from the database to prevent simple hardware failures from corrupting the backups. The space must be readable and writable by the user performing the backups (we suggest using the `postgres` user, as it has full access to the database).

   ```
   sudo mkdir /Volumes/Space/PostgreSQL-Backup
   sudo mkdir /Volumes/Space/PostgreSQL-Backup/dump
   sudo mkdir /Volumes/Space/PostgreSQL-Backup/pitr
   sudo touch /Volumes/Space/PostgreSQL-Backup/lockfile
   sudo chown -R postgres:postgres /Volumes/Space/PostgreSQL-Backup
   ```

   The lines above show the creation of the directory structure, along with a lockfile (necessary for the proper operation of the script). The final line owns the directories to the specified user so they can write to them.

2. Next, you must create space on your remote host to back up the files. We use our own `rsync_snapshot` scripts to perform the remote archiving, so our directory structure is set up to work with those scripts. If you use a different archiving scheme (for example, just a pure `rsync` call), you can use whatever paths work for you.

   If you're using SSH keys to allow the remote logins, you should create those keys at this time and install them properly on the client and server machines.

Our setup also includes `postgresql-*.conf` files that match our remote host and path information. These files are read by the `rsync_snapshot` scripts to perform the backups.

Configuration

Confirm that any external scripts or configuration files (such as those that go with `rsync`, or our `rsync_snapshot` scripts) are correctly installed and configured.

Execution

Copy the LaunchDaemon plist for PostgreSQL backups to the `/Library/LaunchDaemons/` directory on the database server. Once copied, schedule the job for execution using `launchd`:

```
sudo launchctl load -w /Library/LaunchDaemons/org.suffieldacademy.postgresql-backup.plist
```

The job should run at the time specified in the plist file. You can confirm that it is running by checking either its generated log file (logated in `$PGDIR/log`) or the system log (where the script logs by default).

1.6 Restoring From Backup

If the worst should ever happen, you'll need to restore your PostgreSQL database from your backups. Depending on the method(s) you use to back up the database, restoration will either be to the latest full SQL dump of the cluster, or a point-in-time recovery.

1.6.1 Point-In-Time Recovery (PITR)

PITR requires a raw backup of the database cluster files, along with a copy of the write-ahead log (WAL) files generated since the raw backup. **Note:** the WAL files are stored in an architecture-specific format (e.g., PowerPC, Intel x86, *etc*), so you **cannot** restore accross platforms using this method. If you have to move to a different machine, you must use a raw SQL dump (see below).

With PITR, it is possible to restore the database to a specific point in time (rather than just a full backup). For example, if you accidentally blew away an important section of the database, you could recover it up until (but not including) the bad command. The instructions below do not cover this use of PITR, but the procedure is largely the same. Follow the instructions below, and when the time comes to create a `recovery.conf` file, add the options for the last transaction date you would like to restore (see the official documentation for more information).

The instructions in this section are adopted from the offical PostgreSQL documentation on the subject. For more information, please visit the official documentation site:

http://www.postgresql.org/docs/8.1/static/backup-online.html#BACKUP-PITR-RECOVERY

1. Stop the backup script and database, if they're running:

   ```
   sudo launchctl unload -w /Library/LaunchDaemons/org.suffieldacademy.postgresql-backup.plist
   sudo launchctl unload -w /Library/LaunchDaemons/org.postgresql.postmaster.plist
   ```

2. If you have space, move the current database cluster directory to a safe location (in case you need to get back to it):

   ```
   sudo mv /Library/PostgreSQL8/data /Library/PostgreSQL8/broken-data
   ```

   If you don't have room to keep the whole cluster, you should keep a copy of the `pg_xlog` subdirectory (in the `data` directory), as you may need the WAL files it contains

3. Restore the raw database files (the `data` directory), and ensure that they have the correct file permissions. You will need to copy the database files off of your backup server (or disk) using `rsync`, `scp`, `cp`, or another transfer tool. For example:

   ```
   sudo rsync -auvze ssh root@backup-server:/Snapshot/Backups/postgresql/data/ \
   /Library/PostgreSQL8/data/
   ```

   (The above line should be changed to use your actual host names and paths.)

   Once you have them on the server, own them to the database user:

   ```
   sudo chown -R postgres:postgres /Library/PostgreSQL8/data/
   ```

4. Next, you must create a `recovery.conf` file, which PostgreSQL interprets as a signal to recover from a previous backup. Normally, one must give a recovery command to find the archived WAL files. In our case, we back these files up with the base backup, and so they do not need to be "found" at all. Therefore, we specify a bogus recovery command. Place the following in the file `/Library/PostgreSQL8/data/recovery.conf`:

```
# use your path to the "false" command
restore_command = '/usr/bin/false'
```

Make the file writable by the database owner (the database renames the file when it's done with it):

```
sudo chown postgres /Library/PostgreSQL8/data/recovery.conf
```

1. Optionally, you may wish to modify the `pg_hba.conf` file to disallow logins to the database. This will prevent users from getting access to the data before the restore is complete.

2. Start PostgreSQL:

   ```
   sudo launchctl load -w /Library/LaunchDaemons/org.postgresql.postmaster.plist
   ```

   The postmaster will go into recovery mode and proceed to read through the archived WAL files it needs. Upon completion of the recovery process, the postmaster will rename recovery.conf to recovery.done (to prevent accidentally re-entering recovery mode in case of a crash later) and then commence normal database operations.

3. Inspect the contents of the database to ensure you have recovered to where you want to be. If not, return to the beginning of the recovery instructions and try again. If all is well, let in your users by restoring `pg_hba.conf` to normal.

4. Don't forget to start the backup process again if you stopped it:

   ```
   sudo launchctl load -w /Library/LaunchDaemons/org.suffieldacademy.postgresql-backup.plist
   ```

At this point, your database should be back up and running just as before. Double-check your log files to ensure that operation appears reliable.

1.6.2 SQL Dump Recovery

If you do not (or cannot) use PITR, you can restore the database from a pure SQL dump produced by the `pg_dumpall` command. This procedure is far simpler than PITR, in that you only need to issue a single command. However, there are a few drawbacks to keep in mind:

- The SQL dump is only current to the time it was taken. Depending on the time of the backup, the database may have changed significantly.
- You cannot recover a SQL dump to a particular point in time; you must restore the entire file or nothing at all.

- You cannot perform an incremental restore; the entire database must be emptied and then fully restored from the file.

A SQL dump does have the main advantage that it is portable accross platform architectures, and frequently portable accross versions of PostgreSQL. If you need to move a database from one machine to another, or between different versions of PostgreSQL, you should use the dump method.

Preparing for Restoration

As mentioned above, you cannot restore over an existing database. Therefore, you will need to destroy (or move) the current database before attempting recovery.

1. Stop the backup script and database, if they're running:

   ```
   sudo launchctl unload -w /Library/LaunchDaemons/org.suffieldacademy.postgresql-backup.plist
   sudo launchctl unload -w /Library/LaunchDaemons/org.postgresql.postmaster.plist
   ```

2. If you have space, move the current database cluster directory to a safe location (in case you need to get back to it):

   ```
   sudo mv /Library/PostgreSQL8/data /Library/PostgreSQL8/broken-data
   ```

   If you don't have room to back up the entire cluster, you **must** save any configuration files that contain non-default values. This includes (but is not limited to) `pg_hba.conf` and `postgresql.conf`.

1. Initialize a new database cluster directory:

   ```
   sudo -u postgres /Library/PostgreSQL8/bin/initdb -D /Library/PostgreSQL8/data
   ```

You now have an empty database cluster, and are ready for the next phase of the restore.

Restoring Configuration Files

To restore the cluster to a working state, you must first restore the configuration files from the old cluster directory. This includes files like `pg_hba.conf` and `postgresql.conf`, and may include others as well.

Before restoring, you may wish to limit access to the database by modifying `pg_hba.conf`. This will prevent users from interacting with the database while it is being restored.

Once you've got the configuration files in place, you should start the database engine:

```
sudo launchctl load -w /Library/LaunchDaemons/org.postgresql.postmaster.plist
```

Restoring the Cluster

This section is based heavily on the official PostgreSQL documentation. If you need more information, please consult the following URL:

http://www.postgresql.org/docs/8.1/static/backup.html#BACKUP-DUMP-RESTORE

Your configuration files should be restored at this point, and the database engine should be running. You're now ready to read in the SQL file containing all your data.

Note: loading large amounts of data can take a long time, especially if there are indices, foreign constraints, or other time-consuming consistency checks. You may wish to review the PostgreSQL documentation on large data loads; it discusses performance tips for loading large data sets. It specifically has this to say about using `pg_dump`:

> By default, pg_dump uses COPY, and when it is generating a complete schema-and-data dump, it is careful to load data before creating indexes and foreign keys. So in this case the first several guidelines are handled automatically. What is left for you to do is to set appropriate (i.e., larger than normal) values for maintenance_work_mem and checkpoint_segments before loading the dump script, and then to run ANALYZE afterwards.

For more information, please refer to the documentation:

http://www.postgresql.org/docs/8.1/static/populate.html

When you're ready to perform the load, you must run the restore command as the `postgres` user. You can do this using the `sudo` command:

```
sudo -u postgres psql -f database_backup_file.sql postgres
```

The last `postgres` in the database to initially connect to. Since the cluster is empty at this point, the only database you can connect to is the `postgres` database (which is created automatically as part of the `initdb` command).

The data load may take a long time, depending on the amount of data and the speed of your system. Wait for the load to fully complete before moving on to the next step.

Once the load is complete, you should log in to the database and make sure that the data appear to be restored correctly. At this point, the database is ready for use, though some additional steps should be taken to improve performance. You may allow clients back into the database at this time.

To help with query performance, you should perform a `ANALYZE` on each database to update all the housekeeping information in the database. You can do this with the following command *on every database in the cluster*:

```
sudo -u postgres psql -c 'ANALYZE' <dbname>
```

Substitute the datbase name for <`dbname`>, running the command as many times as is necessary for each database in the cluser.

Your database should now be restored, and ready to go!