PostgreSQL Installation and Maintenance

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Informatikdienste

Operating System: Red Hat Enterprise Linux 5 / 6 / 7
Database: PostgreSQL 9.0 / 9.1 / 9.2 / 9.3 / 9.4
Date: 2015-05-06

Table of Contents

Introduction

This document gives an overview of installation and regular maintenance of the PostgreSQL database server. It covers installation, basic configuration, version upgrades, backup and recovery of databases, as well as giving some hints for

tuning. It focuses on major version 9.4, which is the most recent productive version as of the time of writing this document. But most of the described features were also tested with versions 9.3, 9.2 and 9.1.
While PostgreSQL runs on many operating systems, including Linux, FreeBSD, Solaris, Mac OS X and Windows, this document focuses on Redhat Enterprise Linux 7 wherever it needs to be specific to an operating system.
While the bulk of the document is independent of any particular application that is run on PostgreSQL, in a few places we refer to the openBIS system and how it uses the PostgreSQL database.

Organization of databases on disk

PostgreSQL has the concept of a *database cluster*. A database cluster is a collection of databases that is stored at a common file system location (the "data area"). It is possible to have multiple database clusters, as long as they use different data areas and different communication ports.[1]

Installation, Initialization and Client Authentication

To install PostgreSQL using `yum` is straightforward. But you might need to add an external repository. Follow these instructions to prevent yum update from getting an old PostgreSQL from the distribution's repository:

Edit `/etc/yum/pluginconf.d/rhnplugin.conf [main]` section, adding this line to the bottom of the section:

```
exclude=postgresql*
```

Select the appropriate repository configuration file for your operating system and PostgreSQL major version:
http://yum.postgresql.org/

Do a `wget` of the appropriate version, for example:

```
# wget http://yum.postgresql.org/9.4/redhat/rhel-7-x86_64/pgdg-
redhat94-9.4-1.noarch.rpm
```

Next install the rpm configuration file with:

```
rpm -ivh pgdg-redhat94-9.4-1.noarch.rpm
```

Now we can perform the actual installation:

```
# yum install \
postgresql94.x86_64 \
postgresql94-contrib.x86_64 \
postgresql94-devel.x86_64 \
postgresql94-libs.x86_64 \
postgresql94-server.x86_64
```

Configure the PostgreSQL server to be started on system startup:

```
# chkconfig postgresql-9.4 on
```

Create a new PostgreSQL database cluster:

```
# service postgresql-9.4 initdb
```

Or alternatively as user postgres:

```
$ usr/pgsql-9.4/bin/initdb
```

Creating a database cluster consists of creating the directories in which the database data will live, generating the shared catalog tables (tables that belong to the whole cluster rather than to any particular database), and creating the `template0`, `template1` and `postgres` databases. When you later create a new database, by default everything in the `template1` database is copied. (Therefore, anything installed in `template1` is automatically copied into each database created later.) The copy command, however, can also choose to copy `template0`, which is a minimal template that provides full control over character encoding and enabled database languages.[2] The postgres database is a default database meant for use by users, utilities and third party applications that do not care about having their own database.
Although `initdb` will attempt to create the specified data directory, it might not have permission if the parent directory of the desired data directory is `root`-owned. To initialize in such a setup, create an empty data directory as `root`, then use `chown` to assign ownership of that directory to the database user account (which is `postgres` on Redhat Enterprise Linux 7), then `su` to become the database user to run `initdb`.

`initdb` must be run as the user that will own the server process, because the server needs to have access to the files and directories that `initdb` creates. Since the server cannot be run as `root`, you must not run `initdb` as `root` either. (It will in fact refuse to do so.)

Client authentication is controlled by a configuration file, which traditionally is named `pg_hba.conf` and is stored in the database cluster's data directory. (HBA stands for host-based authentication.) A default `pg_hba.conf` file is installed when the data directory is initialized by `initdb`.

[2] The openBIS database setup routine uses `template0` and enables the necessary database languages by itself. `template1` is irrelevant for openBIS.

You might want to add `/var/lib/pgsql/9.4/data/pg_hba.conf` and add the IPs you allow access to the PostgreSQL database:

```
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD

# "local" is for Unix domain socket connections only
local   all         all                               trust
# IPv4 local connections:
host    all         all         127.0.0.1/32          trust
# IPv4 network connections
hostssl all         all         129.132.228.224/27      md5
hostssl all         all         129.132.228.144/32      md5
# IPv6 local connections:
host    all         all         ::1/128               trust
```

Start the server and add DB-User and create password:

```
# service postgresql-9.4 start
# sudo -u postgres createuser --no-createdb --no-createrole \
  --no-superuser openbis
# sudo -u postgres psql -c "\password openbis"
```

```
# sudo -u postgres createdb -Upostgres -E 'UNICODE' -T template0 -O
openbis openbis_productive;
```

The following command enables PL/pgSQL for the specified `<database>`:

```
$ createlang -U postgres plpgsql <database>
```

where user `postgres` is assumed to be the database super user.

To list all enabled languages of a database use the following command:

```
$ createlang -U postgres -l <database>
```

By enabling PL/pgSQL for the database `template1` it will be available for all newly created databases because a new database is just a clone of `template1`, if they do not specify a different template.

Major and Minor Versions

The first two digits of the version number denote the major version, while the third digit denotes the minor version. In version number 9.4.1, for example, 9.4 denotes the major version while 1 denotes the minor version.

All minor versions of the same major version are guaranteed to use the same on-disk format, and thus an update to a new minor version (like the one from 9.4.0 to 9.4.1) does not require any change to the data area. A PostgreSQL yum repository represents one major version, e.g. PostgreSQL 9.4, and will track minor versions as package updates that get installed by calling '`yum update`'.
Each PostgreSQL major version has a clearly communicated support period[3]. During this period, bugs and security issues are fixed regularly by new minor versions. Major version 9.4 will be supported until September 2019.

Upgrade to a New Major Version

Different major versions in general are using a different on-disk format. Thus, PostgreSQL 9.4 will not be able to work with a PostgreSQL 9.3 data area. The default procedure to upgrade a database cluster to a new major version of PostgreSQL is to *dump* the database cluster with the old major version and to *load* the load file with the new major version. For large databases, this so-called *dump and load cycle* can be a time-consuming operation, which requires a long downtime to ensure that the upgraded database has all the latest changes. Instead of doing the time-consuming dump and load cycle, PostgreSQL offers the possibility to do an upgrade of an existing data area to the on-disk format of a new major version. For this upgrade to work, both versions of PostgreSQL must be installed in separate directories. In general it is not a problem to skip a major version when upgrading, which means upgrading from 9.0 to 9.4 is no problem.

Note: The `pg_upgrade` command is part of the `contrib` rpm package! The binaries should be in /usr/pgsql-9.4/bin/
Note2: The files `postgresql.conf` and `pg_hba.conf` are not migrated and need to be adopted to your needs again.

At least these packages of <u>both</u> PostgreSQL versions need to be installed to be able to use `pg_upgrade` (here shown for the example of 9.3 -> 9.4):

```
postgresql93.x86_64
postgresql93-libs.x86_64
postgresql93-server.x86_64
postgresql94.x86_64
postgresql94-contrib.x86_64
postgresql94-libs.x86_64
postgresql94-server.x86_64
postgresql94-devel.x86_64
```

If these packages are not present, the upgrade will be terminated with an error message, so make sure you have installed them. You can quickly check this with:

[3] See http://www.postgresql.org/support/versioning/

```
# yum list postgresql*
# ls -l /usr/pgsql-9.4/bin/
```

You need to run `initdb` on the database cluster for the new version before performing the upgrade. Then, after shutting down the old database server, start the upgrade to 9.4 as database super user `postgres`:

```
postgres$ cd /tmp
postgres$ /usr/pgsql-9.4/bin/initdb -D /var/lib/pgsql/9.4/data/
postgres$ service postgresql-9.3 stop
postgres$ /usr/pgsql-9.4/bin/pg_upgrade \
-d /var/lib/pgsql/9.3/data/ -D /var/lib/pgsql/9.4/data/ \
-b /usr/pgsql-9.3/bin/ -B /usr/pgsql-9.4/bin/
```

The mandatory flags are the old data directory (-d), the new data directory (-D), the old binary directory (-b) and the new binary directory (-B).

Once started, `pg_upgrade` will verify the two clusters are compatible and then do the migration. You can use '`pg_upgrade --check`' to perform only the checks, even if the old server is still running. '`pg_upgrade --check`' will also outline any manual adjustments you will need to make after the migration. It might needed to specify the port with `--old-port=OLDPORT`. `pg_upgrade` requires write permission to the current directory which will be used for temporary files.
Obviously, no one should be accessing the clusters during the migration, so both database servers need to be shut down. [4]

Speeding up the upgrade using the `--link` option:

```
postgres$ /usr/pgsql-9.4/bin/pg_upgrade \
-d /var/lib/pgsql/9.3/data/ -D /var/lib/pgsql/9.4/data/ \
-b /usr/pgsql-9.3/bin/ -B /usr/pgsql-9.4/bin/ --link
```

If you use link mode, the upgrade will be much faster (no file copying), but you will not be able to access your old cluster once you start the new cluster after the upgrade. Link mode also requires that the old and new cluster data directories be in the same file system. See `pg_upgrade --help` for a full list of options.

Note: Failure when upgrading from PostgreSQL 9.0 to 9.1

On RHEL6 64-bit, we have seen this error message:

```
pg_ctl failed to start the new server
```

[4] http://www.postgresql.org/docs/9.4/interactive/pgupgrade.html

In order to resolve this issue, proceed like this:

1. First stop the postgres server again!

2. Then have a look at the shared library used by postgres:

```
# ldconfig -p | grep pq
 libpqwalreceiver.so (libc6) => /usr/pgsql-9.0/lib/libpqwalreceiver.so
 libpqwalreceiver.so (libc6) => /usr/pgsql-9.1/lib/libpqwalreceiver.so
 libpq.so.5 (libc6) => /usr/pgsql-9.0/lib/libpq.so.5
 libpq.so.5 (libc6) => /usr/pgsql-9.1/lib/libpq.so.5
 libpq.so (libc6) => /usr/pgsql-9.0/lib/libpq.so
 libpq.so (libc6) => /usr/pgsql-9.1/lib/libpq.so
```

The order matters here and we rename the old library to overcome this problem:

```
# cd /etc/ld.so.conf.d
# mv postgresql-9.0-libs.conf postgresql-9.old-libs.conf
# ldconfig
# ldconfig -p | grep pq
libpqwalreceiver.so (libc6) => /usr/pgsql-9.1/lib/libpqwalreceiver.so
libpqwalreceiver.so (libc6) => /usr/pgsql-9.0/lib/libpqwalreceiver.so
libpq.so.5 (libc6) => /usr/pgsql-9.1/lib/libpq.so.5
libpq.so.5 (libc6) => /usr/pgsql-9.0/lib/libpq.so.5
libpq.so (libc6) => /usr/pgsql-9.1/lib/libpq.so
libpq.so (libc6) => /usr/pgsql-9.0/lib/libpq.so
```

3. Now `pg_upgrade` works fine and can be started. Renaming the 9.0 configuration file back to its original name and using yum to remove 9.0 is no problem.

Ways of backing up PostgreSQL databases

In general there are three different options for taking a backup of a PostgreSQL database:

- SQL dump
- Continuous Archiving
- File system level backup

In the next three sections we will see how to use all options. Although the first option might be the most well known, the second option is the preferred option against data loss.

Backup: SQL dump using `pg_dump` and `pg_restore`

The tool `pg_dump` allows consistent backups without shutting down the database. It does not block read or write access.

You can create a single file containing the whole database. The file format can be in plain text SQL (default), tar or custom (preferred). The custom format is compressed by default, the most flexible format and a suitable input for the `pg_restore` command.

A simple usage of `pg_dump` using the custom format (-Fc) looks like this:

```
$ export DATE=`/bin/date +%Y-%m-%d_%H%M`
$ export DATABASE=openbis_productive
$ pg_dump -Upostgres -O -Fc ${DATABASE} > ${DATE}_${DATABASE}-db.dmp
```

Now we can restore the database either on the same server or copy the dump to a different server and initiate the restore there. In the first case we need to first drop the faulty database, in this case called `openbis_productive`:

```
$  dropdb openbis_productive
```

In the latter case we need to create an empty database first from a built-in template with Unicode encoding (-E) and openbis as owner (-O):

```
$ createdb -Upostgres -E 'utf8' -T template0 -O openbis \
  openbis_productive
```

Now we can start the actual restore:

```
$ pg_restore -Fc -d openbis_productive -j 4 -Uopenbis \
  2011-06-28_1744_openbis_productive-db.dmp
```

Alternatively to the commands `dropdb` and `createdb` you can use `psql` with the `–c` flag:

```
$ psql -U postgres -c "drop database <dbname>;"
$ psql -U postgres -c "create database <dbname> with owner <owname>
encoding = 'utf8';"
```

Restoring might fail with message:

```
ERROR: role "<role name>" does not exist
```

If this is the case, create the missing role by:

```
$ psql -U postgres -c "create role <owname>"
```

Or with the command `createuser`:

```
$ createuser -Upostgres -S -d –r <owname>
```

The owner in creating the database and restoring the database should be the same. Otherwise you'll probably see the database but you'll not be able to access it. This especially happens when you want to migrate a database dump with openBIS (by just starting the server).

If you are restoring a huge database on a multi core system try to use the -j flag to utilize multiple CPU cores! To find a well performing numbers of jobs, consider the number of CPU cores of the database server:

```
$ cat /proc/cpuinfo
```

Additional commands, which can be handy

The ability of pg_dump and psql to write to or read from pipes makes it possible to dump a database directly from one server to another, for example:

```
$ pg_dump -h host1 <dbname> | psql -h host2 <dbname>
```

Instead of using `pg_dump` for each database there is also a command called `pg_dumpall`, which backs up all databases in a database cluster.

```
pg_dumpall > outfile
```

The dump can be restored via:

```
psql -f infile postgres
```

If you have large databases you might have problems with the file size of a dump or just not enough space to store the dump in a single file. Therefore you can split a compressed dump using a pipe in combination of the `split` command:

```
$ export DATE=`/bin/date +%Y-%m-%d_%H%M`
$ export DATABASE=openbis_productive
$ pg_dump -Fc -Upostgres openbis_productive | split -b 2G - \
${DATE}_${DATABASE}-db.dmp
```

The `-b` flag value may be (or may be an integer optionally followed by) one of following: KB 1000, K 1024, MB 1000*1000, M 1024*1024, and so on for G, T, P, E, Z, Y.

Backup: Continuous Archiving

At all times, PostgreSQL maintains a *write ahead log* (WAL) in the `pg_xlog/` subdirectory of the cluster's data directory. This is comparable to Oracles Redo-logs. By default, these files are rotating and get overwritten. However, WAL logs can be used to perform incremental backups. This method is called *continuous archiving*.

This approach is more complex to administer than the previous approach, but it has some significant benefits:

- Since we can combine an indefinitely long sequence of WAL files for replay, continuous backup can be achieved simply by continuing to archive the WAL files, thus it is an incremental backup. This is particularly valuable for large databases, where it might not be convenient to take a full backup frequently.
- It is not necessary to replay the WAL entries all the way to the end. We could stop the replay at any point and have a consistent snapshot of the database as it was at that time. Thus, this technique supports point-in-time recovery (PITR): it is possible to restore the database to its state at any time since your base backup was taken.
- If we continuously feed the series of WAL files to another machine that has been loaded with the same base backup file, we have a warm standby system: at any point we can bring up the second machine and it will have a nearly-current copy of the database.

The backups produced by continuous archiving are only valid within one major version! When you upgrade to a new major version (e.g. 9.3 → 9.4), you won't be able to restore those backups.

Note: We do not need a perfectly consistent file system backup as the starting point. Any internal inconsistency in the backup will be corrected by log replay (this is not significantly different from what happens during crash recovery, e.g. after a power failure). So we do not need a file system snapshot capability, just `tar`, `rsync` or a similar archiving tool.

Switch on Continuous Archiving

Put the scripts `archive_wal.sh` and `full_db_backup.sh` to `/usr/local/bin`. In this section we will assume the data area to be in `/mnt/localssd/pgsql/data` and the backup directory to be in `/mnt/local0/db-backups/full`. Adapt this to your environment.

/usr/local/bin/archive_wal.sh

```
#! /bin/bash

WAL_PATH="$1"
WAL_FILE="$2"

BACKUP_DIR=/mnt/local0/db-backups/pg_xlog

test ! -f ${BACKUP_DIR}/${WAL_FILE} && /bin/cp ${WAL_PATH}
${BACKUP_DIR}/${WAL_FILE}
```

/usr/local/bin/full_db_backup.sh

```
#! /bin/bash


MAIL_LIST="<dbadmin>@<yourdomain>"
BOX=`uname -n`
MAILX="/bin/mail"



PG_DATA_DIR=/mnt/localssd/pgsql/data
BACKUP_DIR=/mnt/local0/db-backups/full

DATE=`/bin/date +%Y-%m-%d_%H%M`
BACKUP_PATH="${BACKUP_DIR}/${DATE}"

/usr/bin/psql -U postgres -c "SELECT pg_start_backup('${BACKUP_PATH}')"

/usr/bin/rsync -a --exclude "pg_xlog/*" ${PG_DATA_DIR} ${BACKUP_PATH}/

/usr/bin/psql -U postgres -c "SELECT pg_stop_backup()"

if [ $? -ne 0 ]; then
    echo -e "PostgreSQL DB backup broken ... :-(" | $MAILX -s
"PostgreSQL backup from $BOX is B R O K E N !" $MAIL_LIST
else
    echo -e "PostgreSQL DB backup ran OK on $BOX :-)" | $MAILX -s
"PostgreSQL Backup from $BOX ran OK" $MAIL_LIST
fi
```

Create the directories given as `$BACKUP_DIR` in the two scripts and give them to user postgres:

```
# mkdir /mnt/local0/db-backups/full
# chown postgres:postgres /mnt/local0/db-backups/full
# mkdir /mnt/local0/db-backups/pg_xlog
# chown postgres:postgres /mnt/local0/db-backups/pg_xlog
```

Add to `/var/lib/pgsql/9.4/data/postgresql.conf` the lines:

```
# minimal, archive, or hot_standby

wal_level = archive

[…]

# range 30s-1h
checkpoint_timeout = 1h

[…]

# allows archiving to be done
archive_mode = on

# command to use to archive a logfile segment
archive_command = '/usr/local/bin/archive_wal.sh %p %f'

# force a logfile segment switch after, number of seconds
# 0 disables
archive_timeout = 60
```

Alternatively in 9.4 there is the possibility to alter the settings of the database cluster dynamically. The main advantages are that the changes are immediately checked and any typo will cause the command to fail, and not later at startup time. And the changes are stored in a separate file called `postgresql.auto.conf`, which will override the settings in the `postgresql.conf`.

More details are described here:
http://www.postgresql.org/docs/9.4/static/sql-altersystem.html

```
postgres=# alter system set checkpoint_segments = 64;
ALTER SYSTEM

postgres=# show checkpoint_segments;
 checkpoint_segments
---------------------
 32
(1 row)

postgres=# \! cat /var/lib/pgsql/9.4/data/postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by ALTER SYSTEM command.
shared_buffers = '1GB'
checkpoint_segments = '64'
checkpoint_timeout = '1h'
maintenance_work_mem = '512MB'
temp_buffers = '64MB'
work_mem = '64MB'
```

Finally, restart the `postmaster` process:

```
# service postgresql-9.4 restart
```

Consider removing old backups by putting a script like this into the cron tab:

/usr/local/bin/delete_old_backups.sh

```
#! /bin/bash

DB_BACKUP_DIR_FULL=/mnt/local0/db-backups/full
RETENTION_DAYS_FULL=30
DB_BACKUP_DIR_INCREMENTAL=/mnt/local0/db-backups/pg_xlog
RETENTION_DAYS_INCREMENTAL=37

/usr/bin/find ${DB_BACKUP_DIR_FULL} -maxdepth 1 -mtime
+${RETENTION_DAYS_FULL} -not -path ${DB_BACKUP_DIR_FULL} -exec /bin/rm
-fR {} \;

/usr/bin/find ${DB_BACKUP_DIR_INCREMENTAL} -maxdepth 1 -mtime
+${RETENTION_DAYS_INCREMENTAL} -not -path ${DB_BACKUP_DIR_INCREMENTAL}
-exec /bin/rm -fR {} \;
```

Here we assume full backups being performed once a week and a retention time of a month. Note that if the file system containing `pg_xlog/` fills up, PostgreSQL will do a PANIC shutdown. No committed transactions will be lost, but the database will remain offline until you free some disk space on this file system.

Performing the Backup

To make use of the backup, you will need to keep all the WAL segment files generated during and after the file system backup. To aid you in doing this, the `pg_stop_backup` function creates a *backup history file* that is immediately stored into the WAL archive area. This file is named after the first WAL segment file that you need for the file system backup. For example, if the starting WAL file is `0000000100001234000055CD` the backup history file will be named something like `0000000100001234000055CD.007C9330.backup`. The script `/usr/local/bin/full_db_backup.sh` performs all necessary steps.

Restoring the Backup

Make sure that the old PostgreSQL server process is stopped:

```
# service postgresql-9.4 stop
```

Get the old `pgsql/data` directory out of the way:

```
# mv /mnt/localssd/pgsql/data /mnt/localssd/pgsql/data.sv
```

Be careful to check that there is enough space for two copies of the database on the partition if you move it to a place on the same partition.

Copy the latest full backup to `pgsql/data`:

```
# rsync -a /mnt/local0/db-backups/full/<last>/data \
/mnt/localssd/pgsql/
```

If there are any usable WAL segments, copy them over:

```
# rsync -a /mnt/localssd/pgsql/data.sv/pg_xlog/*
/mnt/localssd/pgsql/data/pg_xlog/
```

Create `recovery.conf`:

/mnt/localssd/pgsql/data/recovery.conf

```
restore_command = '/bin/cp /mnt/local0/db-backups/pg_xlog/%f %p'
```

Start the postgres process for recovery:

```
# service postgresql-9.4 start
```

Monitor the logs:

```
# tail -f /mnt/localssd/pgsql/data/pg_log/postgresql-<last>.log
```

Normally, recovery will proceed through all available WAL segments, thereby restoring the database to the current point in time (or as close as possible given the available WAL segments). Therefore, a normal recovery will end with a "file not found" message, the exact text of the error message depending upon your choice of `restore_command`. You may also see an error message at the start of recovery for a file named something like `00000001.history`. This is also normal and does not indicate a problem in simple recovery situations.

If you want to recover to some previous point in time (say, right before someone messed up the database), just specify the required stopping point in `recovery.conf`[5]. You can specify the stop point, known as the "recovery target", either by date/time or by completion of a specific transaction ID. As of this writing only the date/time option is very usable, since there are no tools to help you identify with any accuracy which transaction ID to use.

```
recovery_target_time = ''      # e.g. '2004-07-14 22:39:00 EST'
```

See more about Point-In-Time Recovery (PITR) here:
http://www.postgresql.org/docs/9.4/interactive/archive-recovery-settings.html

[5] e.g: http://pgpool.projects.postgresql.org/pgpool-II/doc/recovery.conf.sample

Backup: File system level backup

For the sake of completeness, we mention also the simplest but also most limited form of backup, which is tar-ing the data area:

```
# tar -cf backup.tar /usr/local/pgsql/data
```

File system level backups have the same limitations as continuous archiving backups, namely:

- File system backups only work for complete backup and restoration of an entire database cluster
- File system backups only work with the major version running at the time of the backup

However, it has a third restriction:

- The database server must be shutdown during the backup in order to get a consistent and usable backup

This last restriction in particular is not acceptable in most productive environments, therefore this option is not further discussed here. Please refer to the PostgreSQL documentation for further details.

Trouble shooting on pg_xlog archiving failures

The most common problem is that either the local hard drive or the archiving destination folder is running out of space. For the first case the DB is clever enough to no longer operate and wait until there is some space left on the device, so no inconsistent state will occur. For the latter case it might be needed to clean the pg_xlog folder. Here are two options shown which help to get the DB up and running again.

Reset the pg_xlogs

```
sudo /etc/init.d/postgresql stop

du -sh /var/lib/pgsql/9.4/data/pg_xlog/
985M     /var/lib/pgsql/9.4/data/pg_xlog/

/usr/pgsql-9.4/bin/pg_controldata /var/lib/pgsql/9.4/data/
...
Latest checkpoint's NextXID:          1/2718420992
Latest checkpoint's NextOID:          496696
...
```

```
sudo -u postgres /usr/pgsql-9.4/bin/pg_resetxlog -o 496696 -x 2718420992 -f \
      /var/lib/pgsql/9.4/data/

Transaction log reset

du -sh /var/lib/pgsql/9.4/data/pg_xlog/
17M     /var/lib/pgsql/9.4/data/pg_xlog/

sudo /etc/init.d/postgresql start
```

After running this command, it should be possible to start the server, but bear in mind that the database might contain inconsistent data due to partially-committed transactions. You should immediately dump your data, run `initdb`, and reload. After reload, check for inconsistencies and repair as needed.

Please have a look in the documentation for further description of the command:
http://www.postgresql.org/docs/9.4/static/app-pgresetxlog.html

Tweak the archiving command

Another option is to fake an archive command. But this is only possible of the DB is still up and running:

1. Change the archive command to a dummy:

   ```
   archive_command = '/bin/true'
   ```

2. and reload the config:

   ```
   psql -c "SELECT pg_reload_conf()"
   ```

3. The ready files should change to done. Stop the cluster and put the correct archive command back in place.
4. Start the cluster and immediately take a full backup!
5. Any WAL files that have a ".ready" file in there can be moved to archive, and then you delete the .ready
   file, and you're good to go.
6. Or you could try to create a dummy file which has the file name of the missing WAL log. (Not tried)

Postgres Foreign data wrapper (postgres_fdw)

The foreign data wrapper module allows you to query other data sources than the databases you are connected to. It is basically the new version of the older dblink module. To be able to query a different source you have to specify the source. If you want more details please refer to the links at the end of this chapter.

```
$ psql -U postgres -d openbis_productive

ALTER ROLE openbis WITH PASSWORD 'openBIS4good';
CREATE DOMAIN file_path AS character varying(1000);

CREATE EXTENSION postgres_fdw;

CREATE SERVER app_db
  FOREIGN DATA WRAPPER postgres_fdw
  OPTIONS (dbname 'pathinfo_prod', host 'localhost');

CREATE USER MAPPING for openbis
  SERVER app_db
  OPTIONS (user 'openbis', password 'openBIS4good');

CREATE FOREIGN TABLE data_sets_fdw
  (
    id bigint,
    code code,
    location file_path
  )
  SERVER app_db OPTIONS (table_name 'data_sets');


CREATE FOREIGN TABLE data_set_files_fdw
  (
    id bigint,
    dase_id tech_id,
    parent_id tech_id,
    relative_path file_path,
    file_name file_path,
    size_in_bytes bigint,
    checksum_crc32 integer,
    is_directory boolean_char,
    last_modified time_stamp
  )
  SERVER app_db OPTIONS (table_name 'data_set_files');

ALTER TABLE data_sets_fdw OWNER TO openbis;
ALTER TABLE data_set_files_fdw OWNER TO openbis;

\q
```

You also need to adopt the pg_hba.conf on the source server to restrict logins only with passwords:

```
$ sudo vi /var/lib/pgsql/9.4/data/pg_hba.conf:

# IPv4 local connections:
host    all             all             127.0.0.1/32            md5

$ sudo service postgresql-9.4 restart
```

Once configured, one can now do query joins between the openBIS DB and the Pathinfo DB for example.

More about this topic:
http://www.postgresql.org/docs/9.4/static/postgres-fdw.html
http://www.craigkerstiens.com/2013/08/05/a-look-at-FDWs/
http://interdbconnect.sourceforge.net/pgsql_fdw/pgsql_fdw-en.html

Configuration Files

Important files:

File / Folder	Description
/var/lib/pgsql/9.x/	Data directory, can be set as $PGDATA
/var/lib/pgsql/9.x/pgstartup.log	Startup log file
/var/lib/pgsql/9.x/postgresql.conf	Main configuration file
/var/lib/pgsql/9.4/data/postgresql.auto.conf	Since 9.4: Dynamic Configuration file, can be modified via ALTER SYSTEM
/var/lib/pgsql/9.x/pg_hba.conf	Host based authentication file

postgresql.conf

This is a selection of 'interesting' settings from the main configuration file.
For an exhaustive description of each file have a look here:
http://www.postgresql.org/docs/9.4/interactive/runtime-config.html

Memory

`shared_buffers (integer)`
Sets the amount of memory the database server uses for shared memory buffers. The default is typically 32 megabytes (32MB), but might be less if your kernel settings will not support it (as determined during initdb). This setting must be at least 128 kilobytes. (Non-default values of `BLCKSZ` change the minimum.) However, settings significantly higher than the minimum are usually needed for good performance. Several tens of megabytes are recommended for production installations. This parameter can only be set at server start.

`effective_cache_size (integer)`
Sets the planner's assumption about the effective size of the disk cache that is available to a single query. This is factored into estimates of the cost of using an

index; a higher value makes it more likely index scans will be used, a lower value makes it more likely sequential scans will be used. When setting this parameter you should consider both PostgreSQL's shared buffers and the portion of the kernel's disk cache that will be used for PostgreSQL data files. Also, take into account the expected number of concurrent queries on different tables, since they will have to share the available space. This parameter has no effect on the size of shared memory allocated by PostgreSQL, nor does it reserve kernel disk cache; it is used only for estimation purposes. The default is 128 megabytes (128MB).

`work_mem (integer)`
Specifies the amount of memory to be used by internal sort operations and hash tables before switching to temporary disk files. The value defaults to one megabyte (1MB). Note that for a complex query, several sort or hash operations might be running in parallel; each one will be allowed to use as much memory as this value specifies before it starts to put data into temporary files. Also, several running sessions could be doing such operations concurrently. So the total memory used could be many times the value of `work_mem`; it is necessary to keep this fact in mind when choosing the value. Sort operations are used for `ORDER BY`, `DISTINCT`, and merge joins. Hash tables are used in hash joins, hash-based aggregation, and hash-based processing of `IN` subqueries.

`maintenance_work_mem (integer)`
Specifies the maximum amount of memory to be used in maintenance operations, such as VACUUM, CREATE INDEX, and `ALTER TABLE ADD FOREIGN KEY`. It defaults to 16 megabytes (16MB). Since only one of these operations can be executed at a time by a database session, and an installation normally doesn't have many of them running concurrently, it's safe to set this value significantly larger than `work_mem`. Larger settings might improve performance for vacuuming and for restoring database dumps.

Note that when autovacuum runs, up to autovacuum_max_workers times this memory may be allocated, so be careful not to set the default value too high.

Write ahead logs

Already mentioned in the backup chapter.

Appendix A: Links and References

General:

http://www.postgresql.org/about/featurematrix/

http://www.postgresql.org/docs/9.4/interactive/index.html http://www.postgresql.org/docs/9.4/interactive/functions-info.html

http://www.postgresql.org/docs/9.4/interactive/functions-admin.html

Installation, Initialization and Client Authentication:

http://www.postgresql.org/docs/9.4/interactive/creating-cluster.html

http://www.postgresql.org/docs/9.4/interactive/auth-pg-hba-conf.html

Upgrade to a New Major Version

http://www.postgresql.org/docs/9.4/interactive/pgupgrade.html

Backup

http://www.postgresql.org/docs/9.4/interactive/backup.html

http://www.pgbarman.org/

Monitoring

http://www.postgresql.org/docs/9.0/interactive/monitoring-stats.html

Appendix B: Background information on using write ahead logs

Quick Introduction to Checkpoint Timing

As you generate transactions, PostgreSQL puts data into the write-ahead log (WAL). The WAL is organized into segments that are typically 16MB each. Periodically, after the system finishes a checkpoint, the WAL data up to a certain point is guaranteed to have been applied to the database. At that point the old WAL files aren't needed anymore and can be reused. Checkpoints are generally caused by one of two things happening:

- `checkpoint_segments` worth of WAL files have been written
- more than `checkpoint_timeout` seconds have passed since the last checkpoint

The system doesn't stop working while the checkpoint is happening; it just keeps creating new WAL files. As long as the checkpoint finishes in advance of what the next one is required things should be fine.

In the 8.2 model, processing the checkpoint occurs as fast as data can be written to disk. All of the dirty data is written out in one burst, then PostgreSQL asks the operating system to confirm the data has been written via the fsync call (see

Tuning PostgreSQL WAL Synchronization for lots of details about what fsync does). 8.3 let the checkpoint occur at a more leisurely pace.

pg_stat_bgwriter sample analysis

Here is an example from a more busy server than the earlier example, courtesy of pgsql-general, and what advice they were given based on these statistics:

```
db=# select * from pg_stat_bgwriter;
checkpoints_timed   |   checkpoints_req   |   buffers_checkpoint   |
buffers_clean | maxwritten_clean | buffers_backend | buffers_alloc --
-----------------+------------------+--------------------+------------
--+-----------------+----------------+---------------
118  |                 435  |               1925161  |          126291  |
7  |          1397373  |        2665693
```

Here in a well formatted table:

checkpoints_timed	118
checkpoints_req	435
buffers_checkpoint	1,925,161
buffers_clean	126,291
maxwritten_clean	7
buffers_backend	1,397,373
buffers_alloc	2,665,693

You had 118 checkpoints that happened because of **checkpoint_timeout** passing. 435 of them happened before that; typically those are because checkpoint_segments was reached. This suggests you might improve your checkpoint situation by increasing checkpoint_segments, but that's not a bad ratio. Increasing that parameter and spacing checkpoints further apart helps give the checkpoint spreading logic of checkpoint_completion_target more room to work over, which reduces the average load from the checkpoint process.

During those checkpoints, 1,925,161 8K buffers were written out. That means on average, a typical checkpoint is writing 3481 buffers out, which works out to be 27.2MB each. Pretty low, but that's an average; there could have been some checkpoints that wrote a lot more while others wrote nothing, and you'd need to sample this data regularly to figure that out.
The background writer cleaned 126,291 buffers (cleaned=wrote out dirty ones) during that time. 7 times, it wrote the maximum number it was allowed to before meeting its other goals. That's pretty low; if it were higher, it would be obvious you could gain some improvement by increasing **bgwriter_lru_maxpages**.
Since last reset, 2,665,693 8K buffers were allocated to hold database pages. Out of those allocations, 1,397,373 times a database backend (probably the client itself) had to write a page in order to make space for the new allocation. That's not awful, but it's not great. You might try and get a higher percentage written by the background writer in advance of when the backend needs them by increasing **bgwriter_lru_maxpages**, **bgwriter_lru_multiplier**, and decreasing **bgwriter_delay** - making the changes in that order is the most effective strategy.

Source: http://www.westnet.com/~gsmith/content/postgresql/chkp-bgw-83.htm

To reset the values shown in the `pg_stat_bgwriter` table just issue the following command:

```
postgres=# select pg_stat_reset_shared('bgwriter');
```

Appendix C: Helpful commands

This is an incomplete list of useful command during day-to-day operations.

Getting help:

```
postgres=# \?
[…]
```

Show me all databases in this cluster with additional details (like size on disk):

```
postgres=# \l+
[…]
```

Run a psql command from the shell:

```
$ psql -U postgres -d openbis_productive -c "SELECT * FROM samples
where pers_id_registerer=5;"
[…]
```

If not connected to a database, but you want to connect now:

```
postgres=# \c openbis_productive
You are now connected to database "openbis_productive".
openbis_productive=#
```

What is the database size (human readable?):

```
postgres=# select
pg_size_pretty(pg_database_size('openbis_productive'));
 pg_size_pretty
----------------
 96 MB
(1 row)

postgres=#
SELECT
    pg_database.datname,
    pg_size_pretty(pg_database_size(pg_database.datname)) AS size
    FROM pg_database;
```

Where is my database cluster directory?

```
postgres=# show data_directory;
     data_directory
-------------------------
 /var/lib/pgsql/9.4/data
(1 row)

postgres=#
```

You can also get all parameters currently used with the following command:

```
postgres=# show all;
[…]
```

The output is not shown due to space saving.

Force a pg_xlog switch:

```
openbis_productive=# checkpoint;select pg_switch_xlog();
```

This is comparable to an Oracle Redo log-file switch (ALTER SYSTEM SWITCH logfile;)

Run an SQL script:

```
postgres=# \i myscript.sql

$ psql -i myscript.sql -d openbis_productive
```

Appendix D: Guidelines for PostgreSQL server tuning

The following recommendations are taken from:
http://www.packtpub.com/postgresql-90-high-performance/book

New server tuning

There are a few ways to combine all of this information into a process for tuning new server. Which is the best is based on what else you expect the server to be doing, along with what you're looking to adjust yourself versus taking rule of thumb estimates for.

Dedicated server guidelines

Initial server tuning can be turned into a fairly mechanical process:

1. Adjust the logging default to be more verbose in `postgresql.conf`
2. Determine how large to set `shared_buffers` to. Start at 25 percent of system memory. Considering adjusting upward if you're on a recent PostgreSQL version with spread checkpoints and know your workload benefits from giving memory directory to the buffer cache. If you're on a platform where this parameter is not so useful, limit its value or adjust downward accordingly.
3. Estimate your maximum connections generously, as this is a hard limit; clients will be refused connection once it's reached.
4. Start the server with these initial parameters. Note how much memory is still available for the as filesystem cache.
5. Adjust `effective_cache_size` based on `shared_buffers` plus the OS cache.
6. Divide the OS cache size by `max_connections`, then by two. This gives you an idea of a maximum reasonable setting for `work_mem`. If your application is not dependent on sort performance, a much lower value than that would be more appropriate.
7. Set `maintenance_work_mem` to around 50 MB per GB of RAM.
8. Increase `checkpoint_segments` to at least 10. If you have server-class hardware with a battery-backed write cache, a setting of 32 would be a better default. At the same time you should change the `checkpoint_completion_target` to a value of 0.9 to allow spreading out checkpoint writing further.
9. If you're using a platform where the default `wal_sync_method` is unsafe, change it to one that is.
10. Increase `wal_buffers` to 16 MB.
11. For PostgreSQL versions before 8.4, consider increases to both `default_statistics_target` (to 100, the modem default) and `max_fsm_pages` based on what you know about the database workload.
    Once you've setup some number of servers running your type of applications, you should have a better idea what kind of starting values make sense to begin with. The values for `checkpoint_segments` and `work_mem` in particular can end up being very different from what's suggested here.

Shared server guidelines

If your database server is sharing hardware with another use, particularly the common situation where a database-driven application is installed on the same system, you cannot be nearly as aggressive in your tuning as described in the last section. An exact procedure is harder to outline. What you should try to do is use tuning values for the memory-related values on the low side of recommended practice:

- Only dedicate 10 percent of RAM to `shared_buffers` at first, even on platforms where more would normally be advised.
- Set `effective_cache_size` to 50 percent or less of system RAM, perhaps less if you know your application is going to be using a lot of it.
- Be very stingy about increases to `work_mem`.

The other suggestions in the above section should still hold - using larger values for `checkpoint_segments` and considering the appropriate choice of `wal_sync_method`, for example, are no different on a shared system than on a dedicated one.

Then, simulate your application running with a full-sized workload, and then measure available RAM to see if more might be suitable to allocate toward the database. This may be an iterative process, and it certainly should be matched with application-level benchmarking if possible. There's no sense in giving memory to the database on a shared system if the application, or another layer of caching such as at the connection pooler level, would use it more effectively. That same idea – get reasonable starting settings and tune iteratively based on monitoring – works well for a dedicated server, too.