wiki.postgresql.org

Binary Replication Tutorial - PostgreSQL wiki

Welcome to the new PostgreSQL 9 replication and standby databases guide. This new set of features implements possibly the longest awaited functionality in PostgreSQL's history. As a result, a lot of people are going to be trying to deploy standby databases for the first time, and find the process rather unintuitive. This guide is here to help.

Work in progress: only 40% complete

NB: there is some duplication with the page on Streaming Replication

This is the easiest way to set up replication between a master and standby. It requires shutting down the master; other methods are detailed later in this guide.

What we're going to do is shut down the master and copy the files we need over to the slave server, creating a cloned copy of the master. Because the master is shut down, we don't have to worry about changes being made to it.

Note: Both the '5 minutes' instructions and the '10 minutes' version which follows do not deal with the complications that arise with a database that uses tablespaces, specifically what to do about the pg_tblspc directory and its contents.

Prerequisites

You must have the right setup to make this work:

- 2 servers with similar operating systems (e.g both Linux 64-bit).
- The same release of PostgreSQL 9.0 installed on both servers.
- PostgreSQL superuser shell access on both servers.
- Knowledge of how to start, stop and reload Postgres.
- PostgreSQL 9.0 running on Server1.
- A database created and loaded on Server1.
- A postgres user or root user who has network

See the full documentation for more information:

- 9.0 Replication Documentation
- 9.1 Replication Documentation

Binary Replication in 7 Steps

This 6-step guide, and all of the examples in this tutorial, assume that you have a master server at 192.168.0.1 and a standby server at 192.168.0.2 and that your database and its configuration files are installed at /var/lib/postgresql/data. Replace those with whatever your actual server addresses and directories are.

1. Edit postgresql.conf on the master to turn on streaming replication. Change these settings:

```
listen_addresses = '*'
wal_level = hot_standby
max_wal_senders = 3
```

2. Edit pg_hba.conf on the master in order to let the standby connect.

```
host  replication   all   192.168.0.2/32      trust
```

3. Edit postgresql.conf on the standby to set up hot standby. Change this line:

```
hot_standby = on
```

4. Create or edit recovery.conf on the standby to set up replication and standby mode. Save a file in the standby's **data directory**, called recovery.conf, with the following lines:

```
standby_mode = 'on'
primary_conninfo = 'host=192.168.0.1'
```

5. Shut down both the master and standby, and copy the files. You want to copy most but not all files between the two servers, excluding the configuration files and the pg_xlog directory. An example rsync script would be:

```
rsync -av --exclude pg_xlog --exclude postgresql.conf data/*
192.168.0.2:/var/lib/postgresql/data/
```

6. Start the standby first, so that they can't get out of sync. (Messages will be logged about not being able to connect to the primary server, that's OK.)

7. Start the master.

Starting Replication with only a Quick Master Restart

Is taking down the master for long enough to copy the files too long? Then you need the 10-minute version.

What we're going to do this time is similar to what we did before, cloning the database by copying the files from the master to the slave server. However, because the database is only going to be shut down for a short period of time, long enough to activate the changes in the configuration file, after we've copied the data files we will need to copy additional files so that the slave will be an up-to-date copy of the master.

So, we will tell the master we're running a backup, copy the data files (not quite the same set of files as before), tell the master the backup is complete, then copy the WAL files in the pg_xlog directory so that when the slave comes up it can make all the changes that were committed to the master database after the backup was started.

First, start with the same prerequisites as above.

1. Set the postgresql.conf variables the same in step (1) as above.

2. Don't close the file yet. You'll need to set two other variables which control the size of your write-ahead-log (WAL). The first is wal_keep_segments, the second is checkpoint_segments. Unless you've already done so, you're going to need to increase these, which is usually a good idea for performance anyway. You want the WAL to be big enough to not get used up in 15 or 20 minutes. If you don't have a clear idea of that, here's some reasonable values, based on how busy and how large your database is. Also, a database with large blob objects may require a much larger setting. Remember, these logs will take up disk space, so make sure that you have enough available - space requirements are below.

```
checkpoint_segments = 8
wal_keep_segments = 8
# light load       500MB

checkpoint_segments = 16
wal_keep_segments = 32
# moderately busy  1.5GB

checkpoint_segments = 64
wal_keep_segments = 128
# busy server      5GB
```

You don't *have* to increase checkpoint_segments in order to increase wal_keep_segments, but it's generally a good idea. Now save the file.

3. Edit pg_hba.conf as in (2) in the "Six Steps" above.

4. Now you need to restart the master. Given the interruption in service, you should probably plan this ahead.

5. Edit postgresql.conf and recovery.conf on the standby as in (3) above.

6. Now, we're going to need to copy the files from the master and start the standby. Unlike in the 6-step version, this needs to be done quickly or the standby will fail to sync and you'll need to try again. First step, you need to tell the master you're starting a backup (see below for a more detailed explanation of this). Log in to psql as the database superuser.

```
psql -U postgres
# select pg_start_backup('clone',true);
```

Note that the string you use as a backup label doesn't matter; use any string you want.

7. Now, quickly copy all the database files. This rsync is slightly different from the 6-step version:

```
rsync -av --exclude pg_xlog --exclude postgresql.conf --exclude postgresql.pid \
   data/*  192.168.0.2:/var/lib/postgresql/data/
```

8. As soon as that's done you need to stop the backup on the master:

```
# select pg_stop_backup();
```

9. As soon as that completes, you need to quickly copy the WAL files from the master to the standby.

```
rsync -av data/pg_xlog 192.168.0.2:/var/lib/postgresql/data/
```

10. Now, start the standby.

If you've done this quickly enough, then the standby should catch up with the master and you should be replicating. If not, you'll get this message:

```
(Future Revisions note:  Message needs to go here)
```

... which means you need to try again, possibly with checkpoint_segments and wal_keep_segments higher. If that still doesn't work, you're going to need to use the even more complex archiving method described below.

Now, the rest of the guide will explain how to deal with more complex situations, such as archive logs, handling security, and maintaining availability, failover and standby promotion.

Binary replication is also called "Hot Standby" and "Streaming Replication" which are two separate, but complimentary, features of PostgreSQL 9.0 and later. Here's some general information about how they work and what they are for.

What Can You Do With Binary Replication?

- Have a simple and complete replica of your production database, preventing all but a couple seconds of data loss even under catastrophic circumstances.
- Load-balance between your read/write master server and multiple read-only slave servers. (Note: This means that queries that aren't read-only cannot be run on a slave server. A common misconception has to do with finding the 'current' value of a sequence on a slave server, that is not possible.)
- Run reporting or other long-running queries on a replica server, taking them off your main transaction-processing server.
- Replicate all DDL, including table and index changes, and even creating new databases.
- Replicate a hosted multi-tenant database, making no specific requirements for primary keys or database changes of your users.

What Can't You Do With Binary Replication?

- Replicate a specific table, schema, or database. Binary replication is the entire Postgres instance (or "cluster").
- Multi-master replication. Multi-master binary replication is probably technically impossible.
- Replicate between different versions of PostgreSQL, or between different platforms.
- Set up replication without administration rights on the server. Sorry, working on it.
- Replicate data synchronously, guaranteeing zero data loss. And ... this is here since the release of PostgreSQL 9.1!

For the reasons above, we expect that Slony-I, Londiste, Bucardo, pgPool2 and other systems will continue to be used.

Transaction Logs and Log Shipping

Users who are already familiar with the PostgreSQL transaction log and warm standby can skip this section.

An individual "instance", "server", or (confusingly) "cluster" of PostgreSQL (hereafter Server) consists of a single postmaster server process connected to a single initialized PostgreSQL data directory (PGDATA), which in turn contains several databases. Each running Server has a transaction log, located in the PGDATA/pg_xlog directory. This transaction log consists of binary snapshots of data, written to record synchronously each change to all databases' data, in case of unexpected shutdown of the database server (such as in a power failure). This ensures that data is not corrupted and no completed transaction is lost.

You can also use this log to allow a copy of the original database to replicate changes made to a master database. This was first implemented with the PITR feature in PostgreSQL 8.0, and is known as "log shipping". Log shipping is required for most forms of binary replication.

This log consists of 16MB segments full of new data pages (8K segments) of the database, and not of SQL statements. For this reason there is no before and after auditing possible via this log, as you cannot know exactly what has changed. Also, the log is treated as a buffer, being deleted as it is no longer needed for crash recovery. More importantly, the data page format of the log means that log segments can only be applied to a database which is binary-identical to the database which created the log.

PITR, Warm Standby, Hot Standby, and Streaming Replication

For the rest of this tutorial, we will refer to the active read-write instance of the Server which generates transaction logs as the "Master" and the passive, read-only or offline instance (or instances) of the Server which receives transaction logs as the "Standby" (or "Standbys"). The term Master/Standby is equivalent to other terminology which may be used in the database industry, such as Master/Slave, Primary/Secondary or Primary/Replica.

PITR

In Point-In-Time Recovery (PITR), transaction logs are copied and saved to storage until needed. Then, when needed, the Standby server can be "brought up" (made active) and transaction logs applied, either stopping when they run out or at a prior point indicated by the administrator. PITR has been available since PostgreSQL version 8.0, and as such will not be documented here.

PITR is primarily used for database forensics and recovery. It is also useful when you need to back up a very large database, as it effectively supports incremental backups, which pg_dump does not.

Warm Standby

In Warm Standby, transaction logs are copied from the Master and applied to the Standby immediately after they are received, or at a short delay. The Standby is offline (in "recovery mode") and not available for any query workload. This allows the Standby to be brought up to full operation very quickly. Warm Standby has been available since version 8.3, and will not be fully documented here.

Warm Standby requires Log Shipping. It is primary used for database failover.

Hot Standby

Hot Standby is identical to Warm Standby, except that the Standby is available to run read-only queries. This offers all of the advantages of Warm Standby, plus the ability to distribute some business workload to the Standby server(s). Hot Standby by itself requires Log Shipping.

Hot Standby is used both for database failover, and can also be used for load-balancing. In contrast to Streaming Replication, it places no load on the master (except for disk space requirements) and is thus theoretically infinitely scalable. A WAL archive could be distributed to dozens or hundreds of servers via network storage. The WAL files could also easily be copied over a poor quality network connection, or by SFTP.

However, since Hot Standby replicates by shipping 16MB logs, it is at best minutes behind and sometimes more than that. This can be problematic both from a failover and a load-balancing perspective.

Streaming Replication

Streaming Replication improves either Warm Standby or Hot Standby by opening a network connection between the Standby and the Master database, instead of copying 16MB log files. This allows data changes to be copied over the network almost immediately on completion on the Master.

In Streaming Replication, the master and the standby have special processes called the walsender and walreceiver which transmit modified data pages over a network port. This requires one fairly busy connection per standby, imposing an incremental load on the master for each additional standby. Still, the load is quite low and a single master should be able to support multiple standbys easily.

Streaming replication does not require log shipping in normal operation. It may, however, require log shipping to start replication, and can utilize log shipping in order to catch up standbys which fall behind.

Cloning a Live Database

If your workload doesn't allow you to take the master down (and whose does?), things get a bit more complicated. You need to somehow take a "coherent snapshot" of the master, so that you don't have an inconsistent or corrupt database on the standby. Now, in some cases this can be done via filesystem snapshotting tools or similar tricks, but as that approach is tricky and platform-dependent, we're not going to cover it here.

Instead, we're going to cover the built-in method, which involves keeping a log of all changes applied to the database which happen during the copying process. The steps are essentially the same, regardless of whether you're planning to use just hot standby, streaming replication, or both. There are two parts:

- Cloning the database files
- Copying the archive logs

Unintuitive as it is, the latter needs to be set up first, so we're going to start with that.

Setting Up Archiving On The Master

Archiving is the process of making an extra copy of each WAL file as it is completed. These log files then need to somehow be accessed by the standby. There are three basic ways to handle this, and you should decide in advance what method you're going to use:

1. Manually
2. Automatic file copying from master to standby using rsync or simiar
3. Writing them to a common shared network file location

The first method is only appropriate if you're archiving logs only to jump-start streaming replication, and you have a fairly low-traffic database or the ability to stop all writes. The third method is probably the easiest to manage if you have an appropriate network share; it can even be used to support multiple standbys with some extra thought and scripting. All of these methods will be explained below.

This needs to be turned on on the master, which if it's never been done before may require a restart (sorry, working on it), and will certainly require a reload. You'll need to set the following parameters:

```
wal_level = hot_standby
archive_mode = on
archive_command = 'some command'
```

What archive command you use depends on which archiving approach you are taking, of course. Here are three examples of commands you might use. Note that you will need to create the "archive" directories.

1. Manual: cp -f %p /var/lib/postgresql/data/archive/%f </dev/null
2. Automatic Copy: rsync -a %p 192.168.0.2:/var/lib/pgsql/data/archive/%f
3. Network Share: cp -f %p /shares/walarchive/archive/%f </dev/null

In these commands, %p is replaced by postgres at invocation time with the full path and name of the WAL file, and %f with the name of the file alone. There are more escapes and parameters dealing with WAL archiving which will be detailed later in the tutorial. Note that, in real production, you are unlikely to want to use any commands as simple as the above. In general, you will want to have archive_command call an executable script which traps errors and can be disabled. Examples of such scripts are available in this tutorial.

Now, if archive_mode was originally "off" or if you had to change wal_level, you're going to need to restart the master (sorry, this will be fixed in a later version). If you just needed to change the archive_command, however, only a reload is required.

Once you've restarted or reloaded, check the master's logs to make sure archiving is working. If it's failing, the master will complain extensively. You might also check that archive log files are being created; run the command "SELECT pg_switch_xlog();" as the superuser to force a new log to be written.

Setting Up Archiving on the Standby

The standby needs to be configured to consume logs. This is simpler than the master's setup, and doesn't really change no matter what archive copying strategy you're using.

Recovery.conf

On the standby, replication configuration is controlled through a file called, for historical reasons, recovery.conf. If this file is present in PostgreSQL's data directory when PostgreSQL is started, that server will assume it is a standby and attempt to obey it. Generally, there is an example file installed with the other PostgreSQL shared docs. However, that example file covers all of the various replication options at once, so it's often simpler to write your own file, from scratch. Any change to recovery.conf requires a restart of the standby.

In recovery.conf, you need to add a command to copy the archived WAL files to the standby's on pg_xlog directory. This is the mirror image of the archive_command on the master. Generally, a simple cp command is sufficient:

```
restore_command = 'cp -f /var/lib/postgresql/data/archive/%f %p </dev/null'
restore_command = 'cp -f /shares/walarchive/%f %p </dev/null'
```

Again, you might want to use a simple shell script which traps error messages, and, importantly, deletes archive files which are no longer needed. If you will be doing only hot standby and not using streaming replication, you probably want to compile the pg_standby binary provided in PostgreSQL's additional modules or "contrib", and use it instead:

```
restore_command = 'pg_standby /shares/walarchive %f %p %r'
```

More detail on pg_standby is in its documentation.

Cloning a Snapshot of the Master

Once you have archiving working, you're ready to clone the master database. At this point, it's a simple process:

1. As superuser, issue the command "SELECT pg_start_backup('backup');" on the master.
2. Copy all of the database files to the standby.
3. Start the standby database.
4. Issue the command "SELECT pg_stop_backup();" on the master.

Of course, each of those steps deserves a little more elaboration. pg_start_backup and pg_stop_backup are special commands you issue on the master in order to create, hold open, and close, a "snapshot" which is how we make sure your copy of the database is not inconsistent. They also write special files to the archive log which tell the standby when it has a complete snapshot.

If you are using the "manual" method of synching the archive logs, immediately after step 4 you need to do one last rsync or copy of the archive logs to the standby.

When you're done with the cloning, you should see output similar to the below:

This means that you're up and replicating, and should now be able to run queries on the standby.

Failing Over To The Standby

Of course, one of the major reasons to have a standby is in case something (planned or unplanned) causes the master server to shut down. Then you want to "fail over", or stop replication and change the standby to a full read-write master.

The recommended method is the same regardless of the type of replication or standby: via "trigger file". First, you need to set a configuration option in recovery.conf on the standby:

```
trigger_file = '/var/lib/postgresql/data/failover'
```

Then, when it's time to fail over, you just create an empty file with that name, such as by using the "touch" command. The standby will notice the file, attempt to apply any remaining WAL records or files it has received, and then switch to read-write or "master" mode. When this happens, you will see a message like this in the Postgres log:

PostgreSQL will also rename the recovery.conf file to recovery.done in order to prevent having the new master fail on restart. For this reason, the recovery.conf file should be owned by the same user which the server runs as (usually "postgres").

The alternative to using a trigger file is to failover manually, by deleting or renaming the recovery.conf file and restarting the standby. This method is inferior because it requires a restart which would interrupt any read-only connections to the standby currently in use.

In a high-availability system, the above activity should be managed automatically in order to avoid downtime. PostgreSQL itself supplies no tools to do this, but numerous third-party utilities such as "Linux heartbeat" are compatible with PostgreSQL replication.

It's important to prevent the original master from restarting after failover, lest you end up with a "split brain" problem and data loss. There is a substantial body of literature on this, and third-party tools, so we won't discuss them here at this time.

Load Balancing

Managing Archive Logs

Tuning and Configuration of Binary Replication

Monitoring Replication