PostgreSQL Replication

Second Edition

Leverage the power of PostgreSQL replication to make your databases more robust, secure, scalable, and fast

Hans-Jürgen Schönig
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 4 'Setting Up Asynchronous Replication'
- A synopsis of the book’s content
- More information on *PostgreSQL Replication Second Edition*
Hans-Jürgen Schönig has 15 years of experience with PostgreSQL. He is the CEO of a PostgreSQL consulting and support company called Cybertec Schönig & Schönig GmbH (www.postgresql-support.de). It has successfully served countless customers around the globe.

Before founding Cybertec Schönig & Schönig GmbH in 2000, he worked as a database developer at a private research company focusing on the Austrian labor market, where he primarily focused on data mining and forecast models. He has also written several books about PostgreSQL.
Preface

Since the first edition of PostgreSQL Replication many new technologies have emerged or improved. In the PostgreSQL community, countless people around the globe have been working on important techniques and technologies to make PostgreSQL even more useful and more powerful.

To make sure that readers can enjoy all those new features and powerful tools, I have decided to write a second, improved edition of PostgreSQL Replication. Due to the success of the first edition, the hope is to make this one even more useful to administrators and developers alike around the globe.

All the important new developments have been covered and most chapters have been reworked to make them easier to understand, more complete and absolutely up to date.

I hope that all of you can enjoy this book and benefit from it.

What this book covers

This book will guide you through a variety of topics related to PostgreSQL replication. We will present all the important facts in 15 practical and easy-to-read chapters:

Chapter 1, Understanding the Concepts of Replication, guides you through fundamental replication concepts such as synchronous, as well as asynchronous, replication. You will learn about the physical limitations of replication, which options you have and what kind of distinctions there are.

Chapter 2, Understanding the PostgreSQL Transaction Log, introduces you to the PostgreSQL internal transaction log machinery and presents concepts essential to many replication techniques.
Preface

Chapter 3, Understanding Point-in-time Recovery, is the next logical step and outlines how the PostgreSQL transaction log will help you to utilize Point-in-time Recovery to move your database instance back to a desired point in time.

Chapter 4, Setting Up Asynchronous Replication, describes how to configure asynchronous master-slave replication.

Chapter 5, Setting Up Synchronous Replication, is one step beyond asynchronous replication and offers a way to guarantee zero data loss if a node fails. You will learn about all the aspects of synchronous replication.

Chapter 6, Monitoring Your Setup, covers PostgreSQL monitoring.

Chapter 7, Understanding Linux High Availability, presents a basic introduction to Linux-HA and presents a set of ideas for making your systems more available and more secure. Since the first edition, this chapter has been completely rewritten and made a lot more practical.

Chapter 8, Working with PgBouncer, deals with PgBouncer, which is very often used along with PostgreSQL replication. You will learn how to configure PgBouncer and boost the performance of your PostgreSQL infrastructure.

Chapter 9, Working with pgpool, covers one more tool capable of handling replication and PostgreSQL connection pooling.

Chapter 10, Configuring Slony, contains a practical guide to using Slony and shows how you can use this tool fast and efficiently to replicate sets of tables.

Chapter 11, Using SkyTools, offers you an alternative to Slony and outlines how you can introduce generic queues to PostgreSQL and utilize Londiste replication to dispatch data in a large infrastructure.

Chapter 12, Working with Postgres-XC, offers an introduction to a synchronous multimaster replication solution capable of partitioning a query across many nodes inside your cluster while still providing you with a consistent view of the data.

Chapter 13, Scaling with PL/Proxy, describes how you can break the chains and scale out infinitely across a large server farm.

Chapter 14, Scaling with BDR, describes the basic concepts and workings of the BDR replication system. It shows how BDR can be configured and how it operates as the basis for a modern PostgreSQL cluster.

Chapter 15, Working with Walbouncer, shows how transaction log can be replicated partially using the walbouncer tool. It dissects the PostgreSQL XLOG and makes sure that the transaction log stream can be distributed to many nodes in the cluster.
Setting Up Asynchronous Replication

After performing our first PITR, we are ready to work on a real replication setup. In this chapter, you will learn how to set up asynchronous replication and streaming. The goal is to make sure that you can achieve higher availability and higher data security.

In this chapter, we will cover the following topics:

- Configuring asynchronous replication
- Understanding streaming
- Combining streaming and archives
- Managing timelines

At the end of this chapter, you will be able to easily set up streaming replication in a couple of minutes.

Setting up streaming replication

In the previous chapter, we recovered data from simple 16 MB XLOG files. Logically, the replay process can only replay 16 MB at a time. This can lead to latency in your replication setup, because you have to wait until the 16 MB have been created by the master database instance. In many cases, this kind of delay might not be acceptable.

Missing the last XLOG file, which has not been finalized (and thus not sent to the archive and lost because of the crash), is often the core reason that people report data loss in the case of PITR.
In this scenario, streaming replication will be the solution to your problem. With streaming replication, the replication delay will be minimal and you can enjoy an extra level of protection for your data.

Let's talk about the general architecture of the PostgreSQL streaming infrastructure. The following diagram illustrates the basic system design:

![Diagram of PostgreSQL streaming infrastructure]

You have already seen this type of architecture. What we have added here is the streaming connection. It is basically a normal database connection, just as you would use in any other application. The only difference is that in the case of a streaming connection, the connection will be in a special mode so as to be able to carry the XLOG.

**Tweaking the config files on the master**

The question now is: how can you make a streaming connection come into existence? Most of the infrastructure has already been made in the previous chapter. On the master, the following must be set:

- The `wal_level` parameter must be set to `hot_standby`
- The `max_wal_senders` parameter must be at a reasonably high value to support enough slaves

How about `archive_mode` and `archive_command`? Many people use streaming replication to make their systems replicate more data to a slave as soon as possible. In addition to that, file-based replication is often utilized to make sure that there is an extra layer of security. Basically, both mechanisms use the same techniques; just the source of XLOG differs in the cases of streaming-based and archive-based recovery.
Now that the master knows that it is supposed to produce enough XLOG, handle XLOG senders, and so on, we can move on to the next step.

For security reasons, you must configure the master to enable streaming replication connections. This requires changing \texttt{pg\_hba\_conf} as shown in the previous chapter. Again, this is needed to run \texttt{pg\_basebackup} and the subsequent streaming connection. Even if you are using a traditional method to take the base backup, you still have to allow replication connections to stream the XLOG, so this step is mandatory.

Once your master has been successfully configured, you can restart the database (to make \texttt{wal\_level} and \texttt{max\_wal\_senders} work) and continue working on the slave.

**Handling \texttt{pg\_basebackup} and \texttt{recovery.conf}**

So far, you have seen that the replication process is absolutely identical to performing a normal PITR. The only different thing so far is \texttt{wal\_level}, which has to be configured differently for a normal PITR. Other than that, it is the same technique; there’s no difference.

To fetch the base backup, we can use \texttt{pg\_basebackup}, just as was shown in the previous chapter. Here is an example:

```
$ pg_basebackup -D /target_directory \
  -h sample.postgresql-support.de \
  --xlog-method=stream
```

Now that we have taken a base backup, we can move ahead and configure streaming. To do so, we have to write a file called \texttt{recovery.conf} (just like before). Here is a simple example:

```
standby_mode = on
primary_conninfo= ' host=sample.postgresql-support.de port=5432 '
```

Note that from PostgreSQL 9.3 onwards, there is a \texttt{-R} flag for \texttt{pg\_basebackup}, which is capable of automatically generating \texttt{recovery.conf}. In other words, a new slave can be generated using just one command.

We have two new settings:

- \texttt{standby\_mode}: This setting will make sure that PostgreSQL does not stop once it runs out of XLOG. Instead, it will wait for new XLOG to arrive. This setting is essential in order to make the second server a standby, which replays XLOG constantly.
Setting Up Asynchronous Replication

- **primary_conninfo**: This setting will tell our slave where to find the master. You have to put a standard PostgreSQL connect string (just like in libpq) here. The primary_conninfo variable is central and tells PostgreSQL to stream XLOG.

For a basic setup, these two settings are totally sufficient. All we have to do now is to fire up the slave, just like starting a normal database instance:

```
iMac:slavehs$ pg_ctl -D / start
server starting
LOG:  database system was interrupted; last known up
at 2015-03-17 21:08:39 CET
LOG:  creating missing WAL directory
    "pg_XLOG/archive_status"
LOG:  entering standby mode
LOG:  streaming replication successfully connected
to primary
LOG:  redo starts at 0/2000020
LOG:  consistent recovery state reached at 0/3000000
```

The database instance has successfully started. It detects that normal operations have been interrupted. Then it enters standby mode and starts to stream XLOG from the primary system. PostgreSQL then reaches a consistent state and the system is ready for action.

### Making the slave readable

So far, we have only set up streaming. The slave is already consuming the transaction log from the master, but it is not readable yet. If you try to connect to the instance, you will face the following scenario:

```
iMac:slavehs$ psql -l
FATAL:  the database system is starting up
psql: FATAL:  the database system is starting up
```

This is the default configuration. The slave instance is constantly in backup mode and keeps replaying XLOG.

If you want to make the slave readable, you have to adapt `postgresql.conf` on the slave system; `hot_standby` must be set to on. You can set this straightaway, but you can also make this change later on and simply restart the slave instance when you want this feature to be enabled.
iMac:slavehs$ pg_ctl -D ./target_directory restart
waiting for server to shut down....
LOG: received smart shutdown request
FATAL: terminating walreceiver process due to administrator command
LOG: shutting down
LOG: database system is shut down
done
server stopped
server starting
LOG: database system was shut down in recovery at 2015-03-17 21:56:12 CET
LOG: entering standby mode
LOG: consistent recovery state reached at 0/3000578
LOG: redo starts at 0/30004E0
LOG: record with zero length at 0/3000578
LOG: database system is ready to accept read only connections
LOG: streaming replication successfully connected to primary

The restart will shut down the server and fire it back up again. This is not too much of a surprise; however, it is worth taking a look at the log. You can see that a process called walreceiver is terminated.

Once we are back up and running, we can connect to the server. Logically, we are only allowed to perform read-only operations:

test=# CREATE TABLE x (id int4);
ERROR: cannot execute CREATE TABLE in a read-only transaction

The server will not accept writes, as expected. Remember, slaves are read-only.

**The underlying protocol**

When using streaming replication, you should keep an eye on two processes:

- wal_sender
- wal_receiver

The wal_sender instances are processes on the master instance that serve XLOG to their counterpart on the slave, called wal_receiver. Each slave has exactly one wal_receiver parameter, and this process is connected to exactly one wal_sender parameter on the data source.
Setting Up Asynchronous Replication

How does this entire thing work internally? As we have stated before, the connection from the slave to the master is basically a normal database connection. The transaction log uses more or less the same method as a `COPY` command would do. Inside the `COPY` mode, PostgreSQL uses a little micro language to ship information back and forth. The main advantage is that this little language has its own parser, and so it is possible to add functionality fast and in a fairly easy, non-intrusive way. As of PostgreSQL 9.4, the following commands are supported:

- **IDENTIFY_SYSTEM**: This requires the server to identify itself. The server replies with four fields (`systemid`, `timeline`, `xlogpos`, `dbname`).
- **TIMELINE_HISTORY tli**: This requests the server to send the timeline history file for a given timeline. The response consists of the filename and content.
- **CREATE_REPLICATION_SLOT slot_name {PHYSICAL | LOGICAL output_plugin}**: This creates a replication slot (physical or logical). In the case of a logical replication slot, an output plugin for formatting the data returned by the replication slot is mandatory.
- **START_REPLICATION [SLOT slot_name] [PHYSICAL] xxx/xxx [TIMELINE tli]**: This tells the server to start WAL streaming for a given replication slot at a certain position for a certain timeline.
- **START_REPLICATION SLOT slot_name LOGICAL XXX/XXX [(option_name [option_value] [, ... ] )]**: This starts logical streaming from a certain position onwards.
- **DROP_REPLICATION_SLOT slot_name**: This drops a replication slot.
- **BASE_BACKUP [LABEL 'label'] [PROGRESS] [FAST] [WAL] [NOWAIT] [MAX_RATE rate]**: This performs a base backup, given certain optional parameters.

What you see is that the protocol level is pretty close to what `pg_basebackup` offers as command-line flags.

### Configuring a cascaded replication

As you have already seen in this chapter, setting up streaming replication is really easy. All it takes is setting a handful of parameters, taking a base backup, and enjoying your replication setup.

In many cases, however, the situation is a bit more delicate. Let's assume for this example that we want to use a master to spread data to dozens of servers. The overhead of replication is actually very small (common wisdom says that the overhead of a slave is around 3 percent of overall performance—however, this is just a rough estimate), but if you do something small often enough, it can still be an issue.
It is definitely not very beneficial for the master to have, say, 100 slaves.

An additional use case is as follows: having a master in one location and a couple of slaves in some other location. It does not make sense to send a lot of data over a long distance over and over again. It is a lot better to send it once and dispatch it to the other side.

To make sure that not all servers have to consume the transaction log from a single master, you can make use of cascaded replication. Cascading means that a master can stream its transaction log to a slave, which will then serve as the dispatcher and stream the transaction log to further slaves.

To use cascaded replication, you need at least PostgreSQL 9.2.

The following diagram illustrates the basic architecture of cascaded replication:

The slaves to the far right of the diagram could serve as dispatchers again. With this very simple method, you can basically create a system of infinite size.

The procedure to set things up is basically the same as that for setting up a single slave. You can easily take base backups from an operational slave (postgresql.conf and pg_hba.conf have to be configured just as in the case of a single master).

Be aware of timeline switches; these can easily cause issues in the event of a failover. Check out the Dealing with the timelines section to find out more.
Setting Up Asynchronous Replication

Turning slaves into masters
A slave can be a wonderful thing if you want to scale up reads or get a backup of your data. But a slave might not always have to remain a slave. At some point, you might need to turn a slave into a master. In classic cases, this happens when the original master crashes or the hardware has to be changed.

Be careful when promoting a slave. It cannot easily be demoted anymore. Once a slave has turned into a master, it can be a slave again only after performing a complete resync or after running `pg_rewind`, which will be available by default in PostgreSQL 9.5.

PostgreSQL offers some simple ways to do this. The first way, and most likely the most convenient way, to turn a slave into a master is by using `pg_ctl`:

```
 McIntosh:slavehs$ pg_ctl -D /target_directory promote
 server promoting
 McIntosh:slavehs$ psql test
 psql (9.2.4)
 Type "help" for help.
 test=# CREATE TABLE sample (id int4);
 CREATE TABLE
```

The `promote` command will signal the postmaster and turn your slave into a master. Once this is complete, you can connect and create objects.

If you've got more than one slave, make sure that those slaves are manually repointed to the new master before the promotion.

In addition to the `promote` command, there is a second option to turn a slave into a master. Especially when you are trying to integrate PostgreSQL with high-availability software of your choice, it can be easier to create a simple file than to call an `init` script.

To use the file-based method, you can add the `trigger_file` command to your `recovery.conf` file:

```
 trigger_file = '/some_path/start_me_up.txt'
```

In our case, PostgreSQL will wait for a file called `/some_path/start_me_up.txt` to come into existence. The content of this file is totally irrelevant; PostgreSQL simply checks whether the file is present, and if it is, PostgreSQL stops recovery and turns itself into a master.
Creating an empty file is a rather simple task:

```
IMac:slavehs$ touch /some_path/start_me_up.txt
```

The database system will react to the new file, `start_me_up.txt`:

```
FATAL: terminating walreceiver proceed fire up:
LOG: trigger file found: /some_path/start_ss due to administrator command
LOG: redo done at 0/50000E0
LOG: selected new timeline ID: 2
LOG: archive recovery complete
LOG: database system is ready to accept connections
LOG: autovacuum launcher started
```

PostgreSQL will check for the file you have defined in `recovery.conf` every 5 seconds. For most cases, this is perfectly fine, and fast enough by far.

### Mixing streaming-based and file-based recovery

Life is not always just black or white. Sometimes, there are also some shades of gray. For some cases, streaming replication might be just perfect. In some other cases, file-based replication and PITR are all you need. But there are also many cases in which you need a bit of both. One example would be like this: when you interrupt replication for a long period of time, you might want to resync the slave using the archive again instead of performing a full base backup. It might also be useful to keep an archive around for some later investigation or replay operation.

The good news is that PostgreSQL allows you to actually mix file-based and streaming-based replication. You don't have to decide whether streaming-based or file-based is better. You can have the best of both worlds at the very same time.

How can you do that? In fact, you have already seen all the ingredients; we just have to put them together in the right way.

To make this easier for you, we have compiled a complete example.
The master configuration

On the master, we can use the following configuration in `postgresql.conf`:

```
wal_level = hot_standby
archive_mode = on
    # allows archiving to be done
    # (change requires restart)
archive_command = 'cp %p /archive/%f'
    # command to use to archive a logfile segment
    # placeholders: %p = path of file to archive
    #               %f = file name only
max_wal_senders = 5
    # we used five here to have some spare capacity
```

In addition to this, we have to add some configuration lines to `pg_hba.conf` to allow streaming. Here is an example:

```
# Allow replication connections from localhost, by a user with the
# replication privilege.
local   replication     hs   trust
host    replication     hs   127.0.0.1/32      trust
host    replication     all  192.168.0.0/16    md5
```

In our case, we have simply opened an entire network to allow replication (to keep the example simple).

Once we have made these changes, we can restart the master and take a base backup as shown earlier in this chapter.

The slave configuration

Once we have configured our master and taken a base backup, we can start configuring our slave system. Let's assume for the sake of simplicity that we are using only a single slave; we will not cascade replication to other systems.

We only have to change a single line in `postgresql.conf` on the slave:

```
hot_standby = on     # to make the slave readable
```
In the next step, we can write a simple `recovery.conf` file and put it into the main data directory:

```plaintext
restore_command = 'cp /archive/%f %p'
standby_mode = on
primary_conninfo = ' host=sample.postgresql-support.de port=5432 '
trigger_file = '/some_path/start_me_up.txt'
```

When we fire up the slave, the following will happen:

1. PostgreSQL will call `restore_command` to fetch the transaction log from the archive.
2. It will do so until no more files can be found in the archive.
3. PostgreSQL will try to establish a streaming connection.
4. It will stream if data exists.

You can keep streaming as long as necessary. If you want to turn the slave into a master, you can again use `pg_ctl promote` or the `trigger_file` file defined in `recovery.conf`.

**Error scenarios**

The most important advantage of a dual strategy is that you can create a cluster that offers a higher level of security than just plain streaming-based or plain file-based replay. If streaming does not work for some reason, you can always fall back on the files.

In this section, we will discuss some typical error scenarios in a dual strategy cluster.

**Network connection between the master and slave is dead**

If the network is dead, the master might not be able to perform the `archive_command` operation successfully anymore. The history of the XLOG files must remain continuous, so the master has to queue up those XLOG files for later archiving. This can be a dangerous (yet necessary) scenario, because you might run out of space for XLOG on the master if the stream of files is interrupted permanently.

If the streaming connection fails, PostgreSQL will try to keep syncing itself through the file-based channel. Should the file-based channel also fail, the slave will sit there and wait for the network connection to come back. It will then try to fetch XLOG and simply continue once this is possible again.
Keep in mind that the slave needs an uninterrupted stream of XLOG. It can continue to replay XLOG only if no single XLOG file is missing, or if the streaming connection can still provide the slave with the XLOG that it needs to operate.

Rebooting the slave
Rebooting the slave will not do any harm as long as the archive has XLOG required to bring the slave back up. The slave will simply start up again and try to get the XLOG from any source available. There won't be corruption or any other problem of this sort.

Rebooting the master
If the master reboots, the situation is pretty non-critical as well. The slave will notice through the streaming connection that the master is gone. It will try to fetch the XLOG through both channels, but it won't be successful until the master is back. Again, nothing bad, such as corruption, can happen. Operations can simply resume after the box reboots.

Corrupted XLOG in the archive
If the XLOG in the archive is corrupted, we have to distinguish between two scenarios:

1. In the first scenario, the slave is streaming. If the stream is okay and intact, the slave will not notice that some XLOG file somehow became corrupted in the archive. The slaves never need to read from the XLOG files as long as the streaming connection is operational.

2. If we are not streaming but replaying from a file, PostgreSQL will inspect every XLOG record and see whether its checksum is correct. If anything goes wrong, the slave will not continue to replay the corrupted XLOG. This will ensure that no problems can propagate and no broken XLOG can be replayed. Your database might not be complete, but it will be the same and consistent up to the point of the erroneous XLOG file.

Surely, there is a lot more that can go wrong, but given those likely cases, you can see clearly that the design has been made as reliable as possible.
Making streaming-only replication more robust

The first thing a slave has to do when connecting to a master is to play catch up. But can this always work? We have already seen that we can use a mixed setup consisting of a streaming-based and a file-based component. This gives us some extra security if streaming does not work.

In many real-world scenarios, two ways of transporting the XLOG might be too complicated. In many cases, it is enough to have just streaming. The point is that in a normal setup, as described already, the master can throw the XLOG away as soon as it is not needed to repair the master anymore. Depending on your checkpoint configuration, the XLOG might be around for quite a while or only a short time. The trouble is that if your slave connects to the master, it might happen that the desired XLOG is not around anymore. The slave cannot resync itself in this scenario. You might find this a little annoying, because it implicitly limits the maximum downtime of your slave to your master's checkpoint behavior.

Two choices are available to solve the problem:

- **wal_keep_segments**: Keep some XLOG files on the master
- Physical replication slots: Teach the master to recycle the XLOG only when it has been consumed

**Using wal_keep_segments**

To make your setup much more robust, we recommend making heavy use of `wal_keep_segments`. The idea of this `postgresql.conf` setting (on the master) is to teach the master to keep more XLOG files around than theoretically necessary. If you set this variable to 1000, it essentially means that the master will keep 16 GB of more XLOG than needed. In other words, your slave can be gone for 16 GB (in terms of changes to the master) longer than usual. This greatly increases the odds that a slave can join the cluster without having to completely resync itself from scratch. For a 500 MB database, this is not worth mentioning, but if your setup has to hold hundreds of gigabytes or terabytes, it becomes an enormous advantage. Producing a base backup of a 20 TB instance is a lengthy process. You might not want to do this too often, and you definitely won’t want to do this over and over again.

If you want to update a large base backup, it might be beneficial to incrementally update it using `rsync` and the traditional method of taking base backups.
What are the reasonable values for \texttt{wal\_keep\_segments}? As always, this largely depends on your workloads. From experience, we can tell that a multi-GB implicit archive on the master is definitely an investment worth considering. Very low values for \texttt{wal\_keep\_segments} might be risky and not worth the effort. Nowadays, pace is usually cheap. Small systems might not need this setting, and large ones should have sufficient spare capacity to absorb the extra requirements. Personally, I am always in favor of using at least some extra XLOG segments.

\textbf{Utilizing replication slots}

With the introduction of PostgreSQL 9.4, a more sophisticated solution to the problem of deleted XLOG has been introduced—physical replication slots. As already outlined earlier in this book, replication slots make sure that the master deletes XLOG only when it has been safely consumed by the replica. In the case of the cleanup problem outlined in this section, this is exactly what is needed here.

The question now is: how can a replication slot be used? Basically, it is very simple. All that has to be done is create the replication slot on the master and tell the slave which slots to use through \texttt{recovery.conf}.

Here is how it works on the master:

```
postgres=# SELECT * FROM pg_create_physical_replication_slot('repl_slot');
     slot_name    | xlog_position
---------------+---------------
      repl_slot  |---------------
```

```
postgres=# SELECT * FROM pg_replication_slots;
     slot_name    | slot_type | datoid | database | active | xmin | restart_lsn
---------------+-----------+--------+----------+--------+------+------------
      repl_slot  | physical  |        |          | f      |      |------------
(1 row)
```

Once the base backup has happened, the slave can be configured easily:

```
standby_mode = 'on'
primary_conninfo = 'host=master.postgresql-support.de port=5432 user=hans password=hanspass'
primary_slot_name = 'repl_slot'
```
The configuration is just as if there were no replication slots. The only change is that the `primary_slot_name` variable has been added. The slave will pass the name of the replication slot to the master, and the master knows when to recycle the transaction log. As mentioned already, if a slave is not in use anymore, make sure that the replication slot is properly deleted to avoid trouble on the master (running out of disk space and other troubles). The problem is that this is incredibly insidious. Slaves, being optional, are not always monitored as they should be. As such, it might be a good idea to recommend that you regularly compare `pg_stat_replication` with `pg_replication_slots` for mismatches worthy of further investigation.

**Efficient cleanup and the end of recovery**

In recent years, `recovery.conf` has become more and more powerful. Back in the early days (that is, before PostgreSQL 9.0), there was barely anything more than `restore_command` and some setting related to `recovery_target_time`. More modern versions of PostgreSQL already offer a lot more and give you the chance to control your replay process in a nice and professional way.

In this section, you will learn what kind of settings there are and how you can make use of those features easily.

**Gaining control over the restart points**

So far, we have archived the XLOG indefinitely. Just like in real life, infinity is a concept that causes trouble. As John Maynard Keynes stated in his famous book, *The General Theory of Employment, Interest, and Money*:

"In the long run, we are all dead."

What applies to Keynesian stimulus is equally true in the case of XLOG archiving; you simply cannot keep doing it forever. At some point, the XLOG has to be thrown away.

To make cleanup easy, you can put `archive_cleanup_command` into `recovery.conf`. Just like most other commands (for example, `restore_command`), this is a generic shell script. The script you will put in here will be executed at every restart point. What is a restart point? Every time PostgreSQL switches from file-based replay to streaming-based replay, you face a restart point. In fact, starting streaming again is considered to be a restart point.

You can make PostgreSQL execute some cleanup routine (or anything else) as soon as the restart point is reached. It is easily possible to clean the older XLOG or trigger some notifications.
Setting Up Asynchronous Replication

The following script shows how you can clean any XLOG that is older than a day:

```bash
#!/bin/sh
find /archive -type f -mtime +1 -exec rm -f {} \;
```

Keep in mind that your script can be of any kind of complexity. You have to decide on a proper policy to handle the XLOG. Every business case is different, and you have all the flexibility to control your archives and replication behavior.

Tweaking the end of your recovery

The `recovery_end_command` parameter serves purposes similar to `archive_cleanup_command`. It triggers some script execution when your recovery (or XLOG streaming) has completed.

Again, you can use this to clean the old XLOG, send out notifications, or perform any other kind of desired action.

Conflict management

In PostgreSQL, the streaming replication data flows in one direction only. The XLOG is provided by the master to a handful of slaves, which consume the transaction log and provide you with a nice copy of the data. You might wonder how this could ever lead to conflicts. Well, there can be conflicts.

Consider the following scenario: as you know, data is replicated with a very small delay. So, the XLOG ends up at the slave after it has been made on the master. This tiny delay can cause the scenario shown in the following diagram:
Let's assume that a slave starts to read a table. It is a long read operation. In the meantime, the master receives a request to actually drop the table. This is a bit of a problem, because the slave will still need this data to perform its `SELECT` statement. On the other hand, all the requests coming from the master have to be obeyed under any circumstances. This is a classic conflict.

In the event of a conflict, PostgreSQL will issue the **Terminating connection due to conflict with recovery** error message.

There are two ways to solve the problem:

- Don't replay the conflicting transaction log before the slave has terminated the operation in question
- Kill the query on the slave to resolve the problem

The first option might lead to ugly delays during the replay process, especially if the slave performs fairly long operations. The second option might frequently kill queries on the slave. The database instance cannot know by itself what is best for your application, so you have to find a proper balance between delaying the replay and killing queries.

To find this delicate balance, PostgreSQL offers two parameters in `postgresql.conf`:

```bash
max_standby_archive_delay = 30s
# max delay before canceling queries
# when reading WAL from archive;
# -1 allows indefinite delay
max_standby_streaming_delay = 30s
# max delay before canceling queries
# when reading streaming WAL;
# -1 allows indefinite delay
```

The `max_standby_archive_delay` parameter will tell the system how long to suspend the XLOG replay when there is a conflicting operation. In the default setting, the slave will delay the XLOG replay for up to 30 seconds if a conflict is found. This setting is valid if the slave is replaying the transaction log from the files.
The `max_standby_streaming_delay` parameter tells the slave how long it should suspend the XLOG replay if the XLOG is coming in through streaming. If the time has expired and the conflict is still there, PostgreSQL will cancel the statement due to a problem with recovery, causing a problem in the slave system and resuming the XLOG recovery to catch up. These settings cover a cumulative delay. That is, if there are ten queries pending, they don't get 30 seconds each to delay replication. So, a query might run for 10 milliseconds and get canceled because it was unlucky to be at the end of an existing delay, causing the user to wonder what happened.

In the previous example, we have shown that a conflict may show up if a table is dropped. This is an obvious scenario; however, it is by no means the most common one. It is much more likely that a row is removed by `VACUUM` or `HOT-UPDATE` somewhere, causing conflicts on the slave.

Conflicts popping up once in a while can be really annoying and trigger bad behavior of your applications. In other words, if possible, conflicts should be avoided. We have already seen how replaying the XLOG can be delayed. These are not the only mechanisms provided by PostgreSQL. There are two more settings we can use.

The first, and older one of the two, is the setting called `vacuum_defer_cleanup_age`. It is measured in transactions, and tells PostgreSQL when to remove a line of data. Normally, a line of data can be removed by `VACUUM` if no more transactions can see the data anymore. The `vacuum_defer_cleanup_age` parameter tells `VACUUM` to not clean up a row immediately but wait for some more transactions before it can go away. Deferring cleanups will keep a row around a little longer than needed. This helps the slave to complete queries that are relying on old rows. Especially if your slave is the one handling some analytical work, this will help a lot in making sure that no queries have to die in vain.

One more method of controlling conflicts is by making use of `hot_standby_feedback`. The idea is that a slave reports transaction IDs to the master, which in turn can use this information to defer `VACUUM`. This is one of the easiest methods of avoiding cleanup conflicts on the slave.

Keep in mind, however, that deferring cleanups can lead to increased space consumption and some other side effects, which have to be kept in mind under any circumstances. The effect is basically the same as running a long transaction on the master.
Dealing with timelines

Timelines are an important concept you have to be aware of, especially when you are planning a large-scale setup.

So what is a timeline? In fact, it is a certain branch of the XLOG. Normally, a database instance that has been freshly set up utilizes timeline number 1. Let's assume that we are starting to replicate our master database to a slave system. The slave will also operate in timeline 1. At some point, your master might die and your slave will be promoted to a new master. This is the time when a timeline switch happens. The new master will create a transaction log of its own now. Logically, we want to make sure that its XLOG is not mixed with some other XLOG made in the good old times.

How can we figure out that the timeline has advanced? Let's take a look at the XLOG directory of a system that was just turned into a master:

```
00000002.history
000000020000000000000006
000000020000000000000007
000000020000000000000008
```

The first part of the XLOG files is an interesting thing. You can observe that so far, there was always a 1 near the beginning of our filename. This is not so anymore. By checking the first part of the XLOG filename, you can see that the number has changed over time (after turning the slave into a master, we have reached timeline number 2).

It is important to mention that (as of PostgreSQL 9.4) you cannot simply pump the XLOG of timeline 5 into a database instance that is already at timeline 9. It is simply not possible.

In PostgreSQL 9.3, we are able to handle these timelines a little more flexibly. This means that timeline changes will be put to the transaction log, and a slave can follow a timeline shift easily.

Timelines are especially something to be aware of when cascading replication and working with many slaves. After all, you have to connect your slaves to some server if your master fails.
Delayed replicas
So far, two main scenarios have been discussed in this book:

- **Point-in-time Recovery (PITR):** Replaying the transaction log as soon as something nasty has happened
- **Asynchronous replication:** Replaying the transaction log as soon as possible

Both scenarios are highly useful and can serve people's needs nicely. However, what happens if databases, and especially change volumes, start being really large? What if a 20 TB database has produced 10 TB of changes and something drastic happens? Somebody might have accidentally dropped a table or deleted a couple of million rows, or maybe somebody set data to a wrong value. Taking a base backup and performing a recovery might be way too time consuming, because the amount of data is just too large to be handled nicely.

The same applies to performing frequent base backups. Creating a 20 TB base backup is just too large, and storing all those backups might be pretty space consuming. Of course, there is always the possibility of getting around certain problems on the filesystem level. However, it might be fairly complex to avoid all of those pitfalls in a critical setup.

Since PostgreSQL 9.4, the database platform provides an additional, easy-to-use feature. It is possible to tell PostgreSQL that the slave is supposed to stay a couple of minutes/hours/days behind the master. If a transaction commits on the master, it is not instantly replayed on the slave but applied some time later (for example, 6 hours). The gap between the master and the slave can be controlled easily, and if the master crashes, the administrator has a convenient 6-hour window (in my example) to roll forward to a desired point in time. The main advantage is that there is already a base backup in place (the lagging standby in this case), and in addition to that, the time frame, which has to be recovered, is fairly small. This leads to less downtime and faster recovery. Replaying only a small time frame is way more desirable than having to fiddle around with large base backups and maybe even a larger amount of XLOG. Having a slave that lags behind is of course no substitute for a proper backup; however, it can definitely help.

To configure a slave that lags behind, `recovery_min_apply_delay` can be added to `recovery.conf`. Just use replication slots and set the desired value of `recovery_min_apply`. Then your system will work as expected.
Keep in mind that normal *Hot-Standby* is definitely a wise option. The purpose of a lagging slave is to protect yourself against unexpected `DROP TABLE` statements, accidental deletions of data, and so on. It allows users to jump back in time when really needed without having to touch too much data. A lagging slave can be seen as a form of backup that constantly updates itself.

### Handling crashes

It is generally wise to use a transaction log archive when using a lagging slave. In addition to that, a crash of the master itself has to be handled wisely. If the master crashes, the administrator should make sure that they decide on a point in time to recover to. Once this point has been found (which is usually the hardest part of the exercise), `recovery_target_time` can be added to `recovery.conf`. Once the slave has been restarted, the system will recover to this desired point in time and go live. If the time frames have been chosen wisely, this is the fastest way to recover a system.

In a way, `recovery_min_apply_delay` is a mixture of classical *PITR* and *Hot-Standby-Slaves*.

### Summary

In this chapter, you learned about streaming replication. We saw how a streaming connection can be created, and what you can do to configure streaming to your needs. We also briefly discussed how things work behind the scenes.

It is important to keep in mind that replication can indeed cause conflicts, which need proper treatment.

In the next chapter, it will be time to focus our attention on synchronous replication, which is logically the next step. You will learn to replicate data without potential data loss.
Where to buy this book
You can buy PostgreSQL Replication Second Edition from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet
toolbook retailers.
Click here for ordering and shipping details.