PostgreSQL for Alfresco: The practical guide

Contents

Intro

At Alfresco's devcon in 2018 I gave a presentation on a range of PostgreSQL topics applied to Alfresco. I decided to limit my presentation to 3 main topics and prepare demo's for all of them. These are the topics I selected:

- Master - Slave replication
- Continuous backup and Point-in-time recovery
- Upgrading your database without downtime

All these topics are covered in the context of Alfresco. However, the techniques used are applicable to a wide range of applications. This guide should be applicable on PostgreSQL 9.5, 9.6 and 10.

Master - Slave replication

In PostgreSQL, a master-slave setup relies on a concept called *streaming replication*. This concept relies on the transaction logs (WAL) that ensure ACID compliance and copies those over from master to slave, or standby.

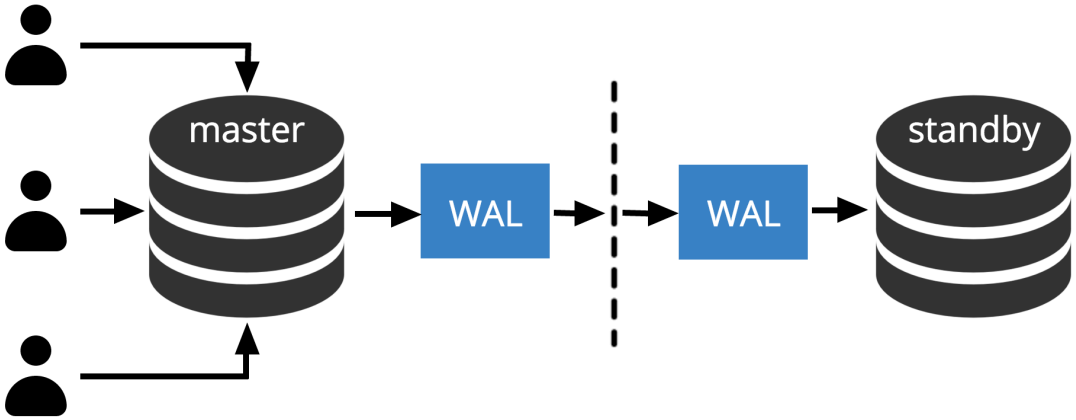

Figure 1: master-slave

Before you setup such a replication there is an important choice to make: Synchronous vs Asynchronous replication.

Synchronous replication

The advantage of synchronous replication is that when the master returns from a commit, you are sure that the commit is also safely written to the slave. In case of a disaster on the master, you don't loose any commits.

However, when you want *high availability*, you need to have at least 2 slaves. Otherwise, the master will block writes, when the slave is down. Using 2 slaves can overcome this, because the master can return to the user, when the commit is written to at least one slave.

Synchronous replication can have a large impact on performance, because it is highly dependent on the network latency, and throughput.

Asynchronous replication

When using asynchronous replication, you don't have these drawbacks, but you have to take into account that there will be a replication lag. It is important to monitor this lag and alarm when the lag is higher than what you think is acceptable for your use case.

In case of disaster, you might loose some transactions. This also means that you will have to restart Alfresco, because you risk having inconsistent caches otherwise.

For simplicity, we continue using asynchronous replication in this article, because it is easier to setup. It will do for demo purposes, but it is up to you to decide what you need for your own production setup.

Configuration

To configure PostgreSQL for streaming replication we have to adapt 2 files in the PostgreSQL data directory:

- postgresql.conf: This is the main configuration file.
- pg_hba.conf: In this file we can configure who has access to what database, from where.

Master

This is how we configure the master:

postgresql.conf:

```
wal_level = hot_standby # or higher
max_wal_senders = 20
```

With the `wal_level` setting, you can control how much information is saved in the transaction logs. `hot_standby` saves enough information so the slave can actually serve read requests.

There is a limit on the amount of connections that send the transaction logs to slaves. This limit can be augmented with the `max_wal_senders` parameter.

pg_hba.conf:

```
host replication repuser 172.18.0.0 255.255.255.0 md5
```

This line gives *replication* access to a user called *repuser* from specific subnet with a password (md5). Please check the PostgreSQL documentation before configuring for production. If not, you might end up with an security risk.

Slave

Configuring the slave is very simple. You just need to run a command, and everything will be copied from the master, including the configuration:

```
pg_basebackup --pgdata=datadir --xlog-method=stream --write-recovery-conf
--progress --dbname="host=pgmaster port=5432 user=repuser password=pwd"
```

When this command has finished, the slave is ready to go. All data is copied from the master and an extra `recovery.conf` file has been added to the data directory, specifying how to connect to the master. All you have to do now, is start the slave.

Alfresco

Alfresco uses a JDBC connection string. You can define multiple servers in a PostgreSQL JDBC connection string:

```
jdbc:postgresql://pgmaster:5432,pgslave:5432/alfresco
```

The connections are made in that is defined here. The slave is only used when the master fails. There are other configuration options for PostgreSQL JDBC connection strings. I only use it for demo and testing purposes. You probably want a tcp loadbalancer like HAProxy that can be configured and reloaded at runtime.

Scenario

When you have configured all the components, and have Alfresco up and running, you can stop the master. Connections from Alfresco to the master will then be broken. Because Alfresco will try to make new connections, those will now go to the slave, because the master is not available.

The scenario that I explained and performed is only for demo purposes. If you want to set it up in production, I suggest you take a look at the tooling that is available. We use *Patroni* to streamline this process.

Continuous backup and Point-in-time recovery

Readers that are familiar with PostgreSQL will know `pg_dump` without a doubt. While backing up a database with `pg_dump` is easy to use, it has some disadvantages when dealing with bigger databases. That's where *wal archiving* comes in the picture.

The concept of *wal archiving* is very simple: the wal files are copied to another location. These copied wal files can then be used when we want to restore a backup. This method has (at least) 2 advantages:

- You don't need to schedule a nightly backup.
- The backup is done continuously, meaning that the time window of data loss, in case of disaster, is very small.
- Upon restore, you can decide to restore up until a certain time or transaction. If you made an error, like accidentally dropping a table, you can restore just before you made the mistake.

Setting it up

Just like a master-slave setup, we start with a base backup. This is the starting point for replaying the wal files in case of a restore. From then on, the PostgreSQL can start the `wal archiving` process.

Wal Archiving

To start the `wal archiving` we need to add the following lines in the postgresql.conf:

```
archive_mode = on
archive_command = 'test ! -f /wal_archive/%f && cp %p /wal_archive/%f'
```

The `archive_command` parameter uses 2 arguments:

- `%f`: The name of the wal file
- `%p`: The path of the wal file to copy

The example configuration will copy the wal files to a folder called `/wal_archive`. Knowing the arguments, you can make your own `archive_command` according to your setup.

By default, an wal file will be copied when PostgreSQL switches to a new file. This happens when it reaches 16MB of file size. If you want to control the window of data loss, you can set the parameter `archive_timeout`. This setting forces PostgreSQL to switch wal files after a configured amount of seconds. You can also force to switch a wal file manually with the `SELECT pg_switch_xlog()` statement.

Point-in-time Recovery

These backups can now be used to perform a point-in-time recovery. Before starting the recovery, we need to restore the last base backup, taken before the time we want to recover. This base backup should become the new data directory.

In that new data directory you need to put a `recovery.conf`:

```
restore_command = 'cp /wal_archive/%f %p'
recovery_target_time = '2018-01-09 09:39:50'
```

The `restore_command` is inverse of the `archive_command` from the postgresql.conf. It is the command that tells PostgresSQL how to restore the wal files it needs for playback.

The `recovery_target_time` is self explanatory. It contains the *point-in-time* to which you want to restore.

Once the configuration is done, you can start your PostgreSQL with the configured data directory. The logs should indicate that it is doing a recovery up until the configured *point-in-time*. When done, the database can accept read only connections. You can decide if you want to promote it, or even to continue the recovery process.

Bookkeeping

If you don't *redo* a base backup from time to time, a restore can become a time consuming operation because you have to *replay* a big amount of wal files. It is also not safe in case of data corruption. There are tools around to help you manage the process I described above, like Barman and Wal-e.

Upgrading without downtime

The easiest way to perform a database upgrade is using `pg_dump` to make a logical backup and to restore that backup on the new version of the database. The problem with this approach is that you need some time to perform these operations, and while doing so, your database should be down, or in a read-only state. This can be a problem for big databases with high figure SLA's.

A classic master-slave replication looks like viable solution. This way you could add a slave of the new version and switch over when all transactions are replicated. However, because the replication is based on a binary format, that changes between version, you can only use physical streaming replication between the same versions of PostgreSQL.

With the help of an extension pglogical, we can overcome this problem. Pglogical translates the physical statements to a version independent format, so they can be applied on another version of PostgreSQL. As from version 10 of PostgreSQL, logical replication is built in. Since we are upgrading from a 9.x version to 10, we need to use the extension.

Execution

Before starting the upgrade, you need to have setup your old and new versions of PostgreSQL with the pglogical extension installed. We will be working with a 9.6 version, and upgrade it to 10. The procedure should work, starting from PostgreSQL 9.5. You need to do a small extra to make it work for 9.4.

Old Version

The following script prepares the 9.6 alfresco database for logical replication:

```
#dump the alfresco database, schema only
pg_dump -Fc -s -U alfresco alfresco -f /share/alfresco.dump

#create extension pglogical on 96
psql -c 'create extension pglogical;' -U alfresco alfresco

#create provider
psql -c "select pglogical.create_node(node_name := 'provider', dsn := 'host
=postgresql96 port=5432 dbname=alfresco');" -U alfresco alfresco
#add all tables to replication set
psql -c "select pglogical.replication_set_add_all_tables('default', ARRAY['
public']);" -U alfresco alfresco
#add all sequences to replication set
psql -c "select pglogical.replication_set_add_all_sequences('default',
ARRAY['public']);" -U alfresco alfresco
```

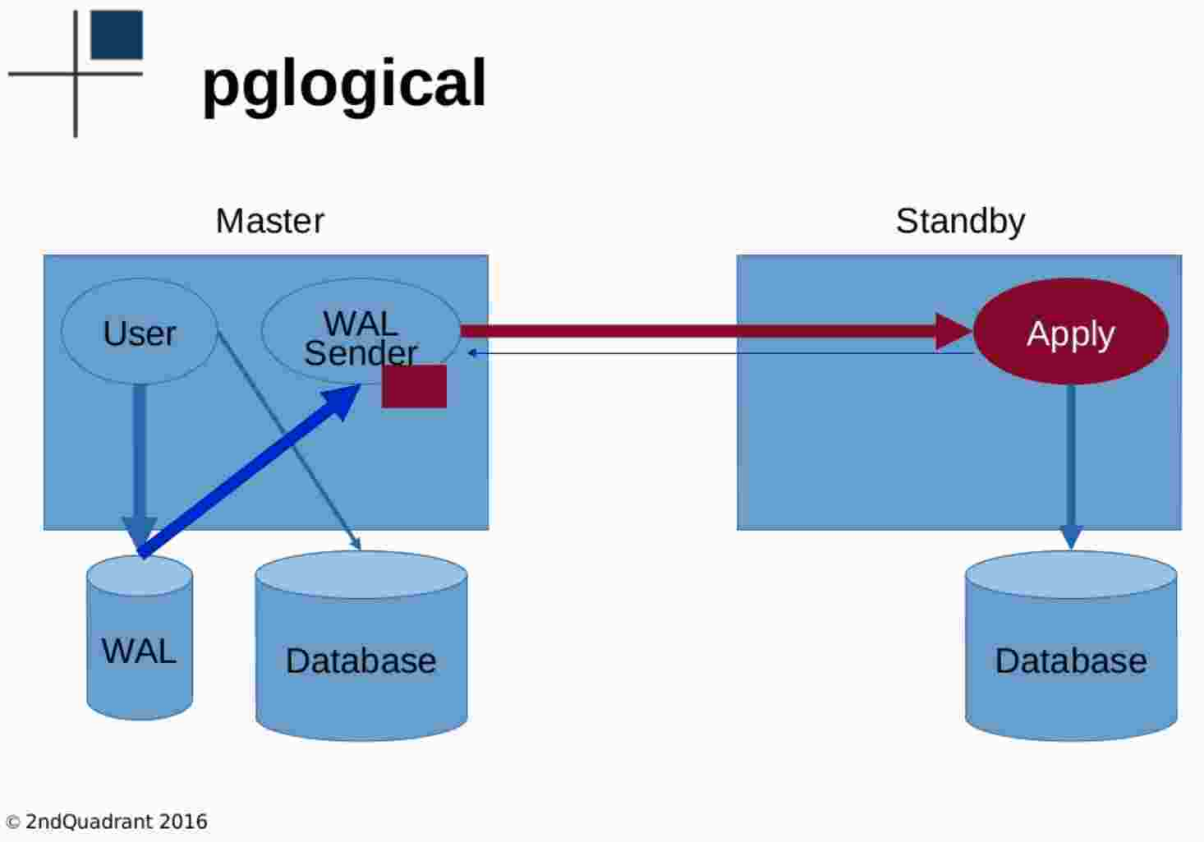

Figure 2: pglogical

Now, the 9.6 database is ready to accept logical replication connections.

New Version

Now can start replicating to the 10 version.

```
#restore the dump on postgresql 10
pg_restore -s -U alfresco -d alfresco /share/alfresco.dump

#create extension
psql -c 'create extension pglogical;' -U alfresco alfresco

#create subscriber
psql -c "select pglogical.create_node(node_name := 'subscriber', dsn := '
host=postgresql10 port=5432 dbname=alfresco');" -U alfresco alfresco
#subscribe
psql -c "select pglogical.create_subscription(subscription_name := '
subscription', provider_dsn := 'host=postgresql96 port=5432 dbname=alfresco
');" -U alfresco alfresco
#check subscription (only for inspection purposes)
psql -c  "select pglogical.show_subscription_status('subscription');"  -U
alfresco alfresco
```

Now, all the database content should be replicated, and new changes are also replicated from 9.6 to 10.

Sequences

There is a small problem when switching over: The sequences, that are responsible for auto-increments, are not always up to date on the slave. Luckily, we can force a synchronization on the 9.6 just before switching over:

```
#sync sequences
psql -c "SELECT pglogical.synchronize_sequence(c.relname::regclass) FROM
pg_class c WHERE c.relkind = 'S';" -U alfresco alfresco
```

Switchover

Now you can do a switchover in the same way you can do that for a classic master-slave. Make sure that the JDBC-driver for Alfresco is working for your PostgreSQL version.

Sources

https://blog.2ndquadrant.com/evolution-of-fault-tolerance-in-postgresql-replication-phase/

https://www.postgresql.org/docs

https://rosenfeld.herokuapp.com/en/articles/infrastructure/2017-11-10-upgrading-postgresql-from-9-6-to-10-with-minimal-downtime

https://www.2ndquadrant.com/en/resources/pglogical/pglogical-docs/