Authors: Egor Rogov, Pavel Luzanov

Feedback
Send feedback, comments and suggestions to:
edu@postgrespro.ru

Topics

Replication tasks and types

Physical replication

Logical replication

Replication use cases

Replication tasks

Replication

the process of synchronizing multiple copies of a database cluster on different servers

Tasks

availability	the system must maintaine availability when one (or more) server is down (performance degradation is possible)
scalability	load balancing between servers to increase throughput

A single database server cannot always meet some requirements.

First, availability. A single physical server is a possible point of failure. If the server goes down (either due to a failure or for maintenance purposes), the system becomes inaccessible. Fault tolerance is very similar concept, but is narrowed down to failures.

Second, performance. If a single server cannot handle the workload, there are two possibilities: server upgrade or adding new servers and managing load balancing between them. The former is much simpler, but the latter if often less expensive.

Thus, it is a question of having several servers working on the same data. Replication is the process of synchronizing data between these servers.

Note that the *database cluster* in PostgreSQL refers to the set of databases inside a single server. This is confusing, as the term is usually refers to a set of servers.

Types of replication

Physical

- primary-standby: data flow is unidirectional
- WAL shipping or streaming replication
- binary compatibility between servers required
- entire database cluster is replicated

Logical

- publisher-subscriber: bidirectional data flow is possible
- requires additional information in WAL (the logical level)
- possible between different major versions and architectures
- individual tables can be replicated

The two approaches to synchronize data available in PostgreSQL are physical and logical replication.

During physical replication, servers have assigned roles: primary and standby. The primary server sends WAL records to the standby (as WAL files or by replication protocol). The standby server applies these records to the data files. WAL records are designed to be just binary patches which can be applied to pages mechanically, without any understanding of their meaning. Therefore binary compatibility between servers is vital (the same hardware architecture, the same major version of PostgreSQL). Since the log is shared between all databases in a cluster, the entire cluster is replicated.

For logical replication some additional information is added to the log, allowing the server to «understand» the changes and decode them in terms of table rows changes. Thus, *wal_level* = logical is required. For such replication, binary compatibility is no longer needed. Logical replication uses the publisher-subscriber model: one server publishes its changes, and others may subscribe to them. The same server may be both publishing and subscribing. Logical replication allows to replicate not all changes, but only those relating to individual tables.

Logical replication is available starting from version 10; earlier versions should have used the `pg_logical` extension, or use trigger-based replication instead.

Physical replication

How replication works

WAL delivery methods

Standby usage restrictions

Switchover to standby (and back)

Replication use cases

Replication

Physical backup

- base backup by the pg_basebackup utility
- WAL files: continuous archiving or streaming archive

Continuous recovery

- place backup on the standby
- create the recovery.conf control file (*standby_mode* = on) and start the server
- server performs recovery to a consistent state and continues to apply incoming log records
- delivery of WAL records by means of WAL archive or replication protocol
- connections (read-only) are accepted immediately after reaching a consistent state

Setting up physical replication is very similar to setting up a physical backup. The difference is that the standby server starts up immediately, without waiting for the primary server to fail, and it doesn't leave recovery mode (*standby_mode* = on): it continuously reads all new incoming WAL records from the primary server and applies them. Thus, the standby is constantly maintained in an almost-up-to-date state, and in the case of a failure, we have a ready-to-go server.

If the standby server does not allow client connections, it is called a *warm standby*. It is possible to set up a *hot standby*, which allows connections for reading data (as soon as the data consistency is restored). Writes on the standby are not allowed.

Unlike backup, replication does not allow the point-in-time recovery. In other words, replication cannot be used to correct a user error (although it is possible to set up replication so that it lags behind the primary for a certain time).

Standby usage restrictions

PROFESSIONAL Postgres

Allowed

- querying the data (select, copy to, cursors)
- setting server parameters (set, reset)
- transaction management (begin, commit, rollback...)
- backups (pg_basebackup)

Not allowed

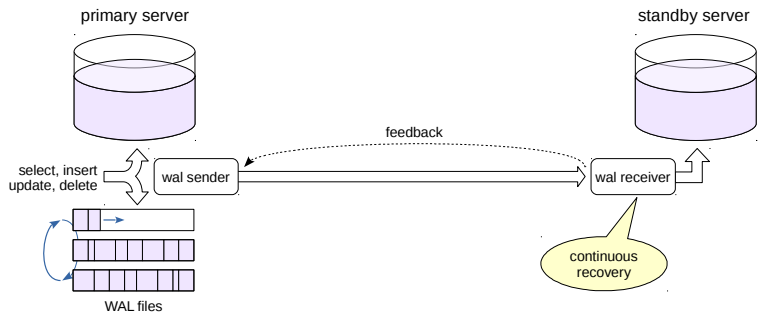
- changing the data (insert, update, delete, truncate, nextval...)
- locks that imply a change (select for update...)
- DDL commands (create, drop...), including creating temporary tables
- maintenance commands (vacuum, analyze, reindex...)
- access control (grant, revoke...)
- triggers will not fire, advisory locks will not work

In the hot standby mode no changes to the data are allowed. This includes not only INSERT, UPDATE, DELETE, TRUNCATE statements, but also sequences, locks, DDL commands, maintenance commands (such as VACUUM and ANALYZE), access control commands.

A standby server can perform requests for reading data (such as SELECT). Setting the server parameters and transaction control commands will also work. For example, you can start a (read-only) transaction with the required isolation level.

In addition, the standby can be used for making backups (of course, taking into account the possible lag from the primary).

Streaming replication

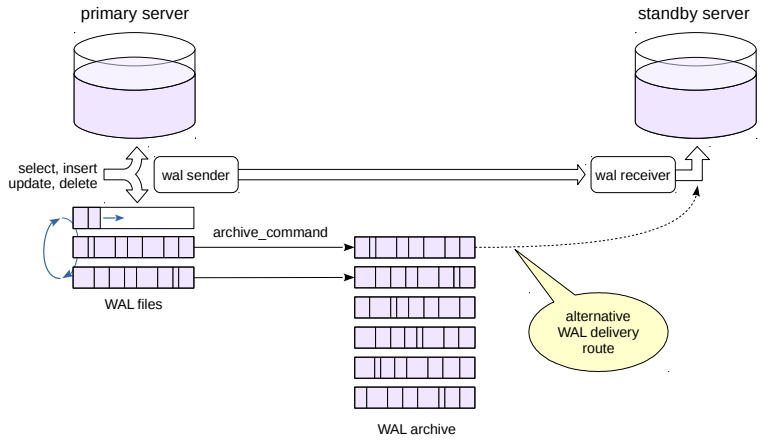


There are two ways to deliver logs from primary to standby. The most widely used one is streaming replication.

In this case, the standby connects to the primary through the replication protocol and reads a stream of WAL records. This ensures minimal lag (and even zero lag in case of synchronous replication mode).

A subtle point is that vaccuming on the primary server can remove the row versions that are needed for the snapshots on the stanby. The affected query on the standy will be canceled. This problem is solved by setting up the standby to send feedback to the primary through the replication protocol. In this case, the primary knows what tuples are needed for the snapshots on the stanby and postpone vacuuming them.

WAL archive



When using streaming replication, there is a danger that the primary server will delete the not-yet-delivered WAL file. To be safe, either replication slot or WAL archive must be used. The latter is often maintained anyway to satisfy the backup policy.

If the standby fails to get the next log record by the replication protocol, it will try to read it from the archive using the *restore_command* from `recovery.conf` file.

Generally speaking, replication can be set up without streaming replication, using the WAL archive alone. But in this case:

- the standby will lag behind the primary for the time of filling the WAL file;

- the primary won't know anything about the existence of the standby (no feedback is possible), so vacuum can delete the tuples required on the standby (standby can delay applying conflicting records, but it is not clear how long should be the timeout).

Switching to standby

Switchover

- scheduled shutdown of the primary server for maintenance
- manual mode

Failover

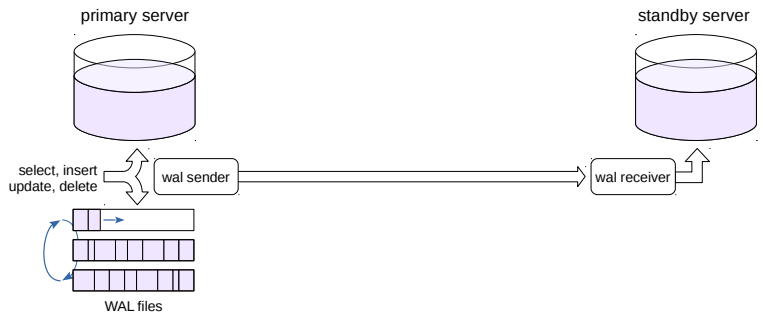
- failover to standby due to primary server failure
- manual mode,
  can be automated using third-party clusterware

There may be different reasons for switching to the standby server.

In case of the need for some maintenance work on the primary server (upgrading, patching etc.) the scheduled *switchover* is performed. In case of a primary server failure, it is necessary to *failover* to the standby server as quickly as possible in order to maximize availability (and minimize the downtime).

Even in the case of a failure, only manual failover is possible since PostgreSQL does not have embedded clusterware (which should monitor the status of the servers and initiate the failover).

Failover



Failover illustrated.

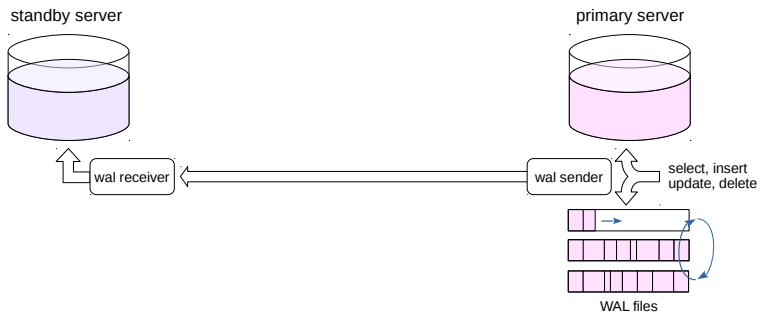
Replication is configured between the primary and the standby servers.

Failover

In case the primary server fails (or when it is down for the maintenance), the stanby is *promoted* to become a new primary.

Of course, there must be a way to redirect clients to the new server; this is done by third-party tools.

Bring old primary back



After the the former primary server is restored, it can be brought back as a new standby server.

Bring old primary back

Cannot simply connect to the new primary server

- some WAL records may not reach the standby when the failure occurs

Restore from backup

- restore a brand new standby in place of the old primary
- may take a lot of time (rsync can help to speedup)

The pg_rewind utility

- «rewinds» lost changes by replacing the corresponding pages on disk with pages from the new primary
- there are a number of limitations

In case of data loss (hardware failure, for example), the only option is to make a completely new standby server from a backup. Otherwise there is a need to quickly bring the old primary back to the system (now as a standby).

Unfortunately, we cannot simply connect the old primary to the new standby by replication protocol. This is not guaranteed to work. The reason is that due to delays in replication, some of the WAL records from the primary might not reach the standby at the moment of a failure. If such records remain on the old primary, then applying WAL records from the new primary will damage the data.

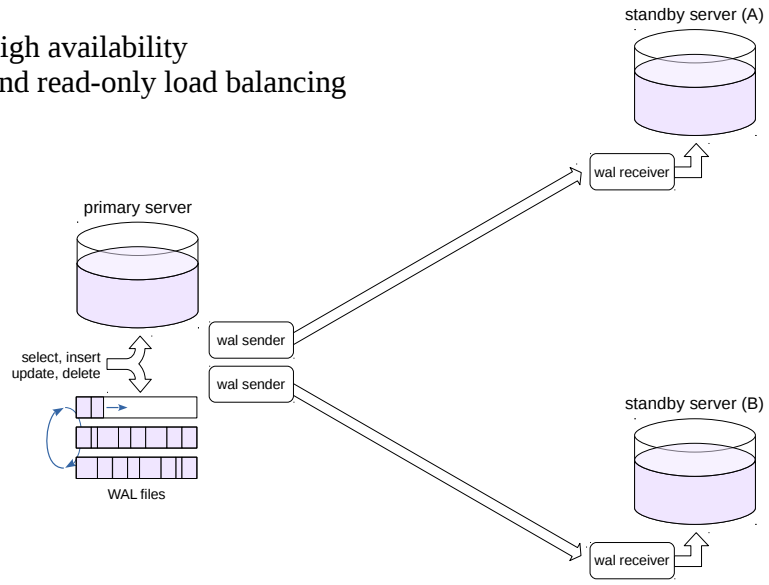
There is always an option to create a completely new standby from a backup. However, for large databases this process can take a lot of time (although It can be speed up with `rsync` utility).

An even faster option is to use the `pg_rewind` utility https://postgrespro.com/docs/postgresql/11/app-pgrewind (available from version 9.5; for 9.3 and 9.4 there is an extension).

The utility identifies WAL records that did not reach the standby (up to the nearest checkpoint), and finds the data pages affected by those records. Found pages are replaced with the pages from the new primary server. After the server is started, it recovers in usual way.

1. Multiple standbys

high availability
and read-only load balancing



The replication allows the system to be designed so that it meets the requirements imposed on it. Consider a few typical usecases.

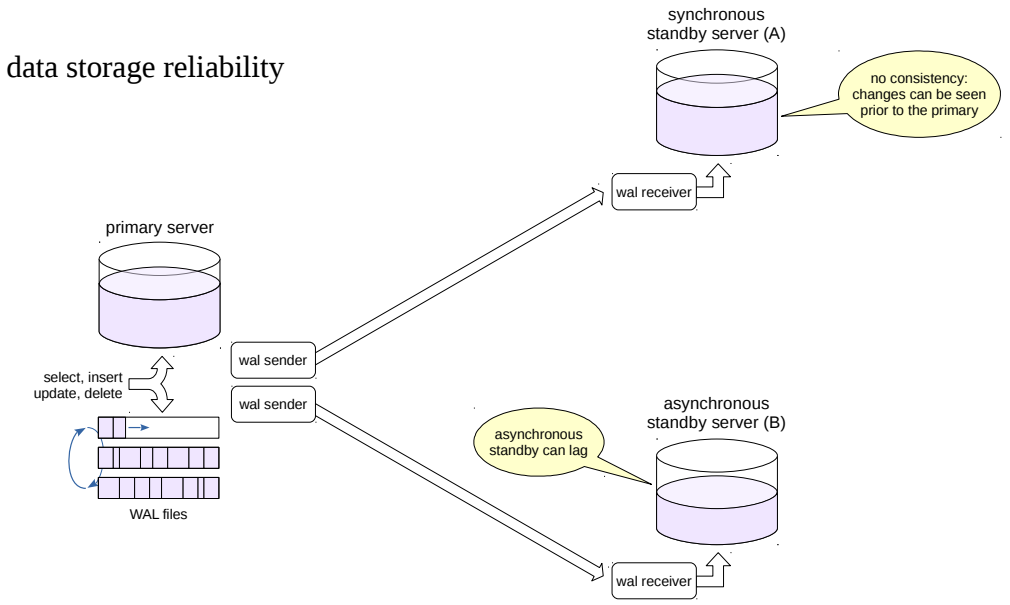
Objective: to ensure high availability and read-only load balancing.

There must be a primary server and several standby servers. Standbys can be used to perform read-only queries. If the primary server fails, one of standbys can be promoted with minimal system downtime.

Each standby establishes connection to the primary, which is served by its own wal writer process and, if necessary, its own replication slot.

Load balancing can be arranged by third-party tools.

2. Synchronous replication



Objective: not to lose any data in case of a failure (that is zero RPO, Recovery Point Objective).

The solution is to use synchronous replication. In the case of a single server, a synchronous WAL writing mode (*synchronous_commit* = on) ensures that the data will not be lost in case of a failure. The similar holds true for replication: COMMIT on the primary waits for acknowlegement from the synchronous standby. If necessary, the parameter can be set on per-transaction basis.

Note that synchronous replication does not ensure data consistency between servers: changes may become visible on the primary and on the standby at different times.

Starting from version 9.6, there may be several synchronous standbys. In version 10 a quorum-based synchronous mode is available.

On the illustration, standby B is asynchronous and may lag, standby A is synchronous. When committing changes, the primary server performs the following:

- creates and flushes the WAL record (thus, the change will not be lost upon failure);

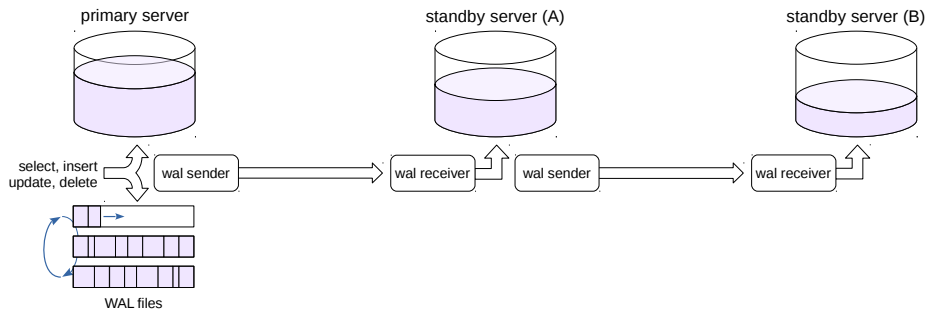
- waities for acknowlegement from the synchronous standby (thus, the change will not be lost upon data loss in the primary server);

- changes transaction state in xact buffer (makes the change visible).

Thus, the query on the synchronous standby can see the changes even earlier than the query on the primary.

3. Cascading replication

several standbys with no additional load on the primary



Objective: to have several standbys, with no additional load on the primary server (both in terms of CPU and memory usage and network throughput).

The requires cascading replication: one standby can transfer WAL records to another standby, and so on.

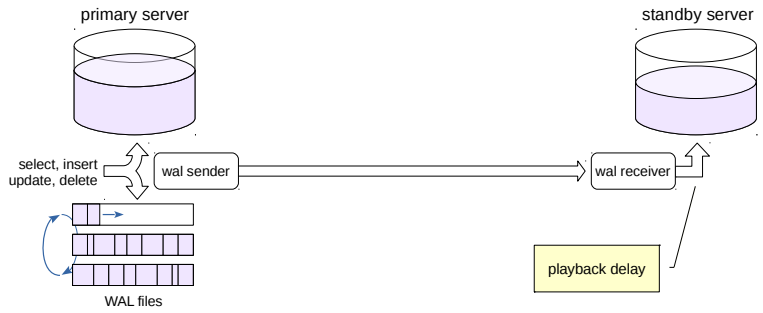
There can be no synchronous cascading standbys. However, the feedback comes back to the primary server from all the downstream standbys.

The standby closest to the primary should be chosen for switchover, as its lag is known to be minimal.

In the illustration: there is only one wal sender process on the primary server; standbys pass WAL records to each other along the chain. The farther from the primary, the greater the lag may be. Monitoring is complicated in this case and involves multiple servers.

4. Delayed replication

time machine
and point-in-time recovery without backups



Objective: to be able to access the data at some point in the past and, if necessary, restore the server at that point.

Point-in-time recovery from a backup basically solves the problem, but it requires some preparatory work and may take a lot of time. And there is no way to build a snapshot of data as of an arbitrary point in the past in PostgreSQL.

The task is solved by creating a standby that applies WAL records after some configurable delay.

Note that clocks have to be synchronized between the servers for this feature to works properly.

When the standby is promoted, it applies the rest of the WAL records without any delay.

Feedback generally should not be used this this feature, since the large delay will cause table bloat on the primary because vacuum will not delete old tuples required by the standby.

Logical replication

Publishers and subscribers

Conflict detection and resolution

Logical replication use cases

Built-in logical replication is available starting from PostgreSQL 10. For earlier versions, similar functionality is available in the `pg_logical` extension.

Logical replication

Publisher

- decodes data changes from WAL and outputs them row by row in commit order (only INSERT, UPDATE, DELETE are replicated)
- decoding requires *wal_level* = logical
- initial synchronization is possible
- always uses logical replication slot

Subscriber

- receives and applies changes
- no parsing, transformation, or planning required
- possibility of conflicts with local data

Logical replication uses the publisher-subscriber model. A publication is created on one server, which may include a number of tables from a single database. Another server can create a subscription to this publication and receive and apply changes.

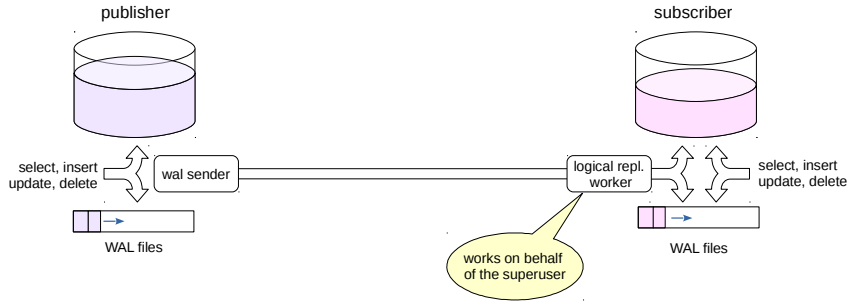
Only modified rows are replicated. DDL is not replicated, that is, the destination tables on the subscriber must be created manually. But it is possible to automatically synchronize tables contents when creating a subscription.

Technically, the information about the modified rows (which is written to WAL), is decoded on the publisher and transferred to the subscriber via replication protocol in a platform-independent format. The logical replication worker process on a subscriber accepts and applies the changes. To ensure the reliability of the transmission (no losses and no repetitions), a logical replication slot (similar to a regular replication slot) is always used.

Changes are applied on low level without executing SQL statements and taking the associated overhead of parsing and planning, so the load on the subscriber is lower than on the publisher.

https://postgrespro.com/docs/postgresql/11/logical-replication

Logical replication



In the illustratino: the logical replication worker process on the subscriber server receives information from the publisher and applies it. At the same time, the server works in the usual way and accepts queries for both reading and writing.

Conflicts

Row identity options

- primary key columns (default)
- columns of the specified unique index with NOT NULL constraint
- all columns
- no identity (default for system catalog)

Conflicts (violation of integrity constraints)

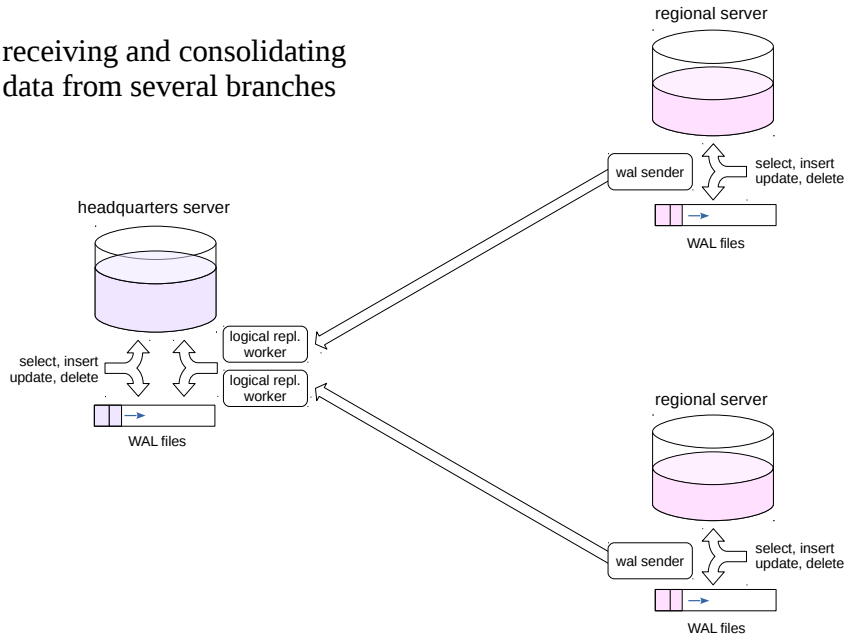
- replication is suspended until the conflict is resolved manually
- either correcting the data
  or skip conflicting transaction

Inserting new rows is quite simple. The situation is more complicated with updates and deletes, in which case the row to be modified must be found. By default, the primary key columns are used for this, but other options can be specified: use a unique index or use all columns. You can also refuse to support replication for some tables at all (by default, the system catalog tables).

Since the tables on the publisher and on the subscriber can change independently from each other, a conflict may occur when modifying the data, that is a violation of integrity constraints. In this case, replication is suspended until the conflict is manually resolved. You can either correct the data on the publisher so that a conflict does not occur, or you can skip the conflicting transaction.

1. Consolidation

receiving and consolidating
data from several branches



Consider several usecases for logical replication.

Suppose there are several regional branches, each runs its own PostgreSQL server. The task is to consolidate part of the data on the HQ server.

The publications of the necessary data are created on the regional servers. The HQ server subscribes to these publications. The obtained data can be processed using triggers on the side of the HQ server (for example, to transform the data to a unified format).

The same scheme, deployed «inside out», allows to transfer reference information from the HQ server to regional ones.

Technical aspect: since replication is based on the WAL, lags in replication (for example due to network problems) may lead to increased space usage and performance problems on the publisher.

From the business logic point of view, there are many other features that require a comprehensive study. In some cases it may be easier to transfer data in batches once a certain time interval.

In the illustration: on the HQ server there are two processes for receiving logs, one for each subscription.

2. Server upgrade

major upgrade with minimal or no downtime

Objective: the upgrade the PostgreSQL server to the other major release with minimal or no downtime.

(Note that the most commonly used way to upgrade is to use the `pg_upgrade` utility, which is out of scope of this course.)

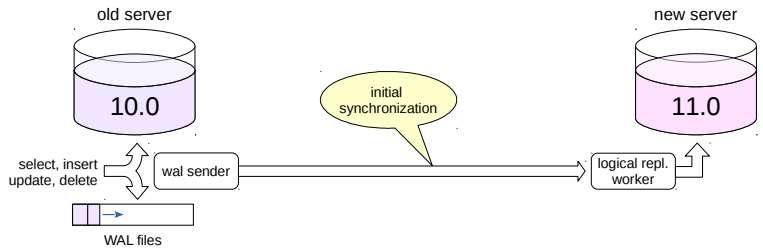
Since there is no binary compatibility between major versions, physical replication does not help. However, logical replication can be used to replicate changes.

As usual, third-party tools are required to redirect clients between servers. Required downtime is determined by this operation.

First, a new server is created using the target PostgreSQL version.

2. Server upgrade

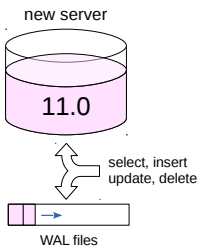
major upgrade with minimal or no downtime



Then the logical replication of all necessary databases is configured and the data is synchronized. This is possible due to the fact that logical replication does not require binary compatibility between servers.

2. Server upgrade

major upgrade with minimal or no downtime

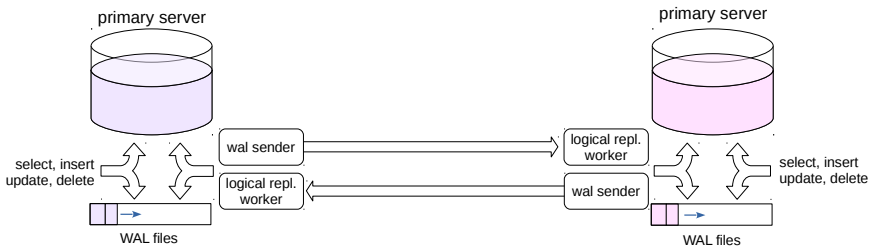


After that, the clients are redirected to the new server, and the old server is turned off.

In fact, the process of upgrading using logical replication is much more complex and faces considerable difficulties. It is discussed in some detail in the DBA2 course.

3. Multimaster configuration

several read-write servers



Objective: several read-write servers. Such configuration won't provide neither write scalability nor consistency, but it increases reliability and availability and makes application development easier (no need to distinguish between read-write and read-only nodes). It also allows to build geographically distributed systems.

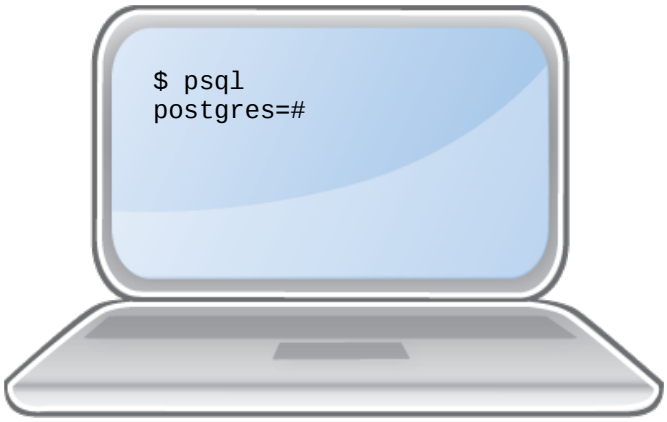
The application system should be designed to avoid conflicts when changing data in the same tables. For example, ensure that different servers work with different key ranges.

The multimaster system built on logical replication does not provide global distributed transactions and hence doesn't guarantees data consistency between servers. Even synchronous replication guarantees reliability but non consistency. Also there are no means for automating failovers, adding or removing servers from the system etc: these tasks requires some third-party tools.

In the illustration: in the multimaster configuration, each server creates a publication and a subscription. There is a bidirectional data flow between the servers.

Note that PostgreSQL 11 still is not capable of bidirectional replication, but sooner or later this feature should appear in the kernel (see the `pg_logical` extension https://www.2ndquadrant.com/en/resources/pglogical/ and BDR https://www.2ndquadrant.com/en/resources/bdr/).

Demonstration



Summary

Replication is based on the transfer of WAL records to another server and replaying them

- WAL files or replication protocol

Physical replication maintains an exact copy of the entire database cluster

- unidirectional
- requires binary compatibility

Logical replication sends the changes made to the individual tables rows

- multidirectional
- no binary compatibility required

Practice

1. Configure physical streaming replication between two servers in synchronous mode.
2. Check that replication works. Make sure that commit does not complete when the standby server is stopped.
3. Promote the standby server.
4. Create two tables on both servers.
5. Configure logical replication of the first table from one server to another, and the second in the opposite direction.
6. Check that replication works.

1. This requires the following parameters to be set on the primary:

- *synchronous_commit* = on,

- *synchronous_standby_names* = 'replica',

and on the standby server add «application_name=replica» to the *primary_conninfo* parameter in the recovery.conf file.