

Replication Overview



Copyright

© Postgres Professional, 2015–2022

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov

Translated by Alexander Meleshko

Use of course materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

Feedback

Please send your feedback, comments and suggestions to:

edu@postgrespro.ru

Disclaimer

Postgres Professional assumes no responsibility for any damages and losses, including loss of income, caused by direct or indirect, intentional or accidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

Replication purposes and types

Physical replication

Logical replication

Replication use cases

Replication purposes

Replication

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

Purposes

reliability	if one of the servers fails, the system must maintain availability (performance degradation is acceptable)
scalability	load distribution between servers

A single database server may not meet all the requirements.

First, reliability. One physical server is a possible point of failure. If the server fails, the system becomes unavailable.

Secondly, performance. One server may not be able to handle the load. Often, the ability to scale and distribute the load between multiple servers is preferable to increasing single server capacity.

The solution is to have multiple servers managing the same databases. Replication refers to the process of synchronizing these servers.

Replication types

Physical

- primary-replica: data flow in one direction only
- delivery of WAL records or files
- binary server compatibility is required
- only the cluster as a whole can be replicated

Logical

- publication-subscription: data flow is possible in both directions
- row level information (*log level* = logical)
- protocol-level compatibility is required
- can replicate individual tables

There are multiple ways to set up synchronization between servers. The two main venues available in PostgreSQL are physical and logical replication.

During physical replication, one server is assigned the main server role and the other becomes a replica. The main server transfers WAL records to a replica (in the form of files or a stream of records). The replica applies these records to its data files. The WAL record application is purely mechanical, without “understanding the meaning” of the changes, so binary compatibility between servers is necessary (the same platforms and major PostgreSQL versions). Since the WAL is shared across the entire cluster, only the cluster as a whole can be replicated.

During logical replication, higher-level information is added to the WAL, allowing the replica to sort out changes at the row level (requires the parameter *wal_level* = logical). This sort of replication does not require binary compatibility, it only needs the replica to be able to understand the incoming WAL information. Logical replication allows, if necessary, to replicate only the changes made to individual tables.

Logical replication was introduced in PostgreSQL 10. Before, you had to use the `pg_logical` extension or set up trigger-based replication.

- How physical replication works
- WAL transmission modes
- Replica usage
- Switching to replica and back
- Replica configuration and use cases

Let's discuss physical replication first.

It works by translating changes to the replica in the form of WAL records. This is a very efficient mechanism, but it requires binary compatibility between servers (the major version of the server, the operating system, the hardware platform).

Physical replication is one-way only: while there may be any number of replicas, there is always only one main server.

Backup

- base backup via `pg_basebackup`
- WAL files archive

Continuous recovery

- deploy the backup
- set configuration parameters
- create a `standby.signal` file
- launch the server
- the server restores consistency and continues to apply incoming logs
- delivery by replication protocol stream or WAL archive
- connections (read-only) are allowed immediately after consistency is restored

Setting up replication is very similar to setting up a physical backup. The difference is that the backup deploys immediately, without waiting for the main server to crash, and works in *continuous recovery mode*: it continuously reads and applies new WAL segments coming from the main server. To tell the replica to start in this mode, a `standby.signal` file is created instead of `recovery.signal`.

This way, the replica is constantly maintained in an almost up-to-date state and if the main server fails, the replica is ready to take over.

By default, the replica operates in the “hot standby” mode. This means that during the recovery process, it allows connections to read data (as soon as consistency is restored). You can prohibit these connections (this is called “warm standby”).

Unlike backup, replication does not allow you to recover to an arbitrary point in the past. In other words, replication cannot be used to correct an error (although it is possible to configure a replica so that it lags behind the main server by a certain amount of time).

Allowed

- read-only queries (select, copy to, cursors)
- setting server parameters (set, reset)
- transaction management (begin, commit, rollback...)
- creating a backup (pg_basebackup)

Not allowed

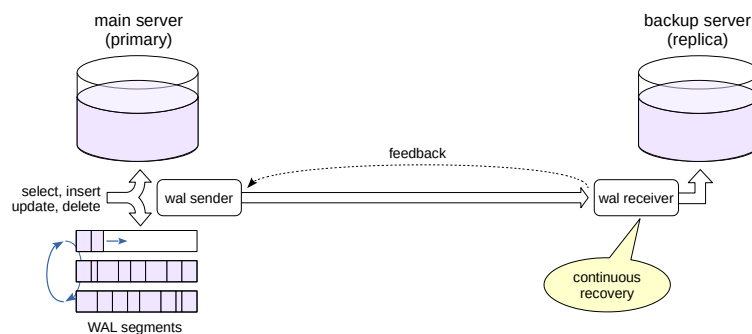
- any changes (insert, update, delete, truncate, nextval...)
- locks expecting changes (select for update...)
- DDL commands (create, drop...), including creating temporary tables
- maintenance commands (vacuum, analyze, reindex...)
- access control (grant, revoke...)
- triggers and advisory locks are disabled

In hot standby mode, no data changes (including sequences), locks, DDL commands, service commands such as VACUUM and ANALYZE, or access control commands are allowed on the replica. Basically, anything that changes the data in any way is prohibited.

The replica can still process read-only queries. Changing server parameters and executing transaction management commands is allowed. For example, you can start a (reading) transaction with a specific isolation level.

In addition, the replica can also be used for making backups (of course, taking into account the possible lag behind the main server).

Streaming replication



There are two ways to deliver WALs from the primary server to the replica. The one used more commonly in production is streaming replication.

In this case, the replica connects to the primary server via the replication protocol and receives the WAL record stream. This minimizes the replica lag and can even eliminate it entirely (in synchronous mode).

There's a notable possible issue with reading from a replica. While a query on a replica takes a MVCC snapshot, the primary server may vacuum the row versions required for the snapshot. The affected query on the replica will have to be terminated in this case. With streaming replication, the issue is resolved by the *feedback mechanism*. It lets the primary server know if any transaction IDs are in use by the replica so that it can delay the vacuuming.

Physical replication. Primary server backup

The replication protocol can run with the default server configuration:

- wal_level = replica,
- max_wal_senders = 10,
- connection permission in pg_hba.conf.

Create a standalone backup. The -R key tells pg_basebackup to set all necessary configuration parameters for the replication.

```
student$ sudo rm -rf /home/student/basebackup
student$ pg_basebackup --pgdata=/home/student/basebackup -R
Make sure that the second server is stopped and copy the backup into its data directory. The user postgres must be the owner of the backup files.
student$ sudo pg_ctlcluster 13 replica status
pg_ctl: no server running
student$ sudo rm -rf /var/lib/postgresql/13/replica
student$ sudo mv /home/student/basebackup/ /var/lib/postgresql/13/replica
student$ sudo chown -R postgres:postgres /var/lib/postgresql/13/replica
```

Replica

pg_basebackup has added the connection parameters for the main server into postgresql.auto.conf:

```
student$ sudo cat /var/lib/postgresql/13/replica/postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
primary_conninfo = 'user=student passfile='''/home/student/.pgpass'' channel_binding=prefer host='''/var/run/postgresql'' port=5432 sslmode=prefer sslcompression=0 sslsnr=1 ssl_min_proto
```

It also created an empty standby.signal file that will tell the server to start in continuous recovery mode.

```
student$ sudo ls -l /var/lib/postgresql/13/replica/standby.signal
-rw----- 1 postgres postgres 0 Mar  7 13:51 /var/lib/postgresql/13/replica/standby.signal
```

We can start the server now.

```
student$ sudo pg_ctlcluster 13 replica start
```

Check what processes are running on the replica.

```
student$ sudo head -n 1 /var/lib/postgresql/13/replica/postmaster.pid
```

144247

```
student$ sudo ps -o pid,command --ppid 144247
      PID COMMAND
144248 postgres: 13/replica: startup waiting for 00000001000000000000000000
144249 postgres: 13/replica: checkpointer
144250 postgres: 13/replica: background writer
144251 postgres: 13/replica: stats collector
144252 postgres: 13/replica: walreceiver
```

The walreceiver process reads the WAL stream, the startup process applies the changes.

Compare them to the processes on the primary server.

```
student$ sudo head -n 1 /var/lib/postgresql/13/main/postmaster.pid
```

129559

```
student$ sudo ps -o pid,command --ppid 129559
      PID COMMAND
129561 postgres: 13/main: checkpointer
129562 postgres: 13/main: background writer
129563 postgres: 13/main: walwriter
129564 postgres: 13/main: autovacuum launcher
129565 postgres: 13/main: stats collector
129566 postgres: 13/main: logical replication launcher
143991 postgres: 13/main: student student [local] idle
144253 postgres: 13/main: walsender student [local] idle
```

Here, a process called walsender exists.

Replication verification

The state of the replication can be checked on the primary server:

```
student$ psql -p 5432
=> SELECT * FROM pg_stat_replication \gx
-[ RECORD 1 ]-----+
pid                | 144253
usesysid           | 16384
username           | student
application_name   | 13/replica
client_addr        |
client_hostname    |
client_port        | -1
backend_start      | 2024-03-07 13:51:58.613524+03
backend_xmin       |
state              | streaming
sent_lsn            | 0/80000000
write_lsn           | 0/80000000
flush_lsn           | 0/80000000
replay_lsn          | 0/80000000
write_lag           | 00:00:00.100474
flush_lag           | 00:00:00.100474
replay_lag          | 00:00:00.100474
sync_priority       | 0
sync_state          | async
reply_time          | 2024-03-07 13:51:58.715715+03
```

Run several commands on the primary server:

```
=> CREATE DATABASE replica_overview;
```

CREATE DATABASE

```
=> \c replica_overview;
```

You are now connected to database "replica_overview" as user "student".

```
=> CREATE TABLE test(id integer PRIMARY KEY, descr text);
```

CREATE TABLE

Check the replica:

```
student$ psql -p 5433 -d replica_overview
```

```
| => SELECT * FROM test;
```

```
|   id | descr  
|-----+-----  
| (0 rows)
```

```
| => INSERT INTO test VALUES (1, 'One');
```

```
INSERT 0 1
```

```
| => SELECT * FROM test;
```

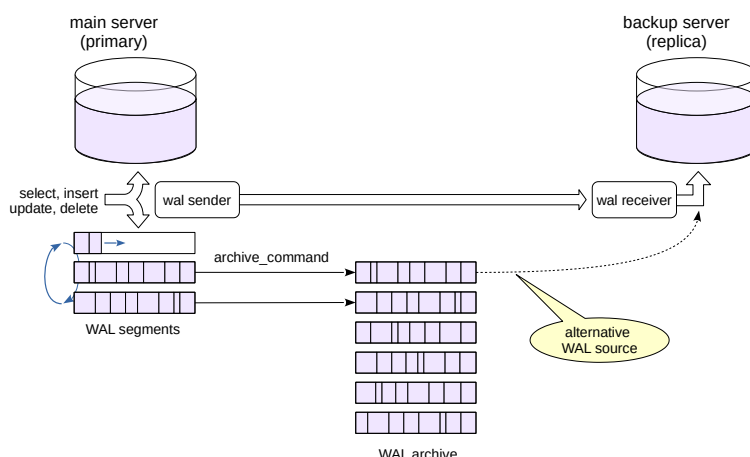
```
|   id | descr  
|-----+-----  
|    1 | One  
| (1 row)
```

No changes can be done on replica directly:

```
| => INSERT INTO test VALUES (2, 'Two');
```

```
| ERROR:  cannot execute INSERT in a read-only transaction
```

Replication via WAL archive



10

During streaming replication, there's a chance that the primary server will delete a WAL segment that hasn't been received by the replica yet. To ensure that it does not happen, you have to use either a replication slot or streaming replication together with a WAL archive (that you need for backups anyway).

When using a WAL archive, a special archiver process on the primary server archives full WAL segments using the `archive_command` (this mechanism is discussed in the Backup module).

If the replica cannot receive the next WAL entry via the replication protocol, it will try to read it from the archive using the command from the `restore_command` parameter.

In fact, replication can work with just the archive, without streaming replication. But in this case:

- the replica is forced to lag behind the primary server by the time it takes to fill the WAL segment,
- the primary server is not aware of the replica's existence, so vacuuming can delete the row versions needed for replica snapshots (you can set up a delay for applying conflicting records, but it is not always clear how long the delay should it be).

Scheduled switchover

- shutdown of the main server for maintenance without interruption of service
- manual mode

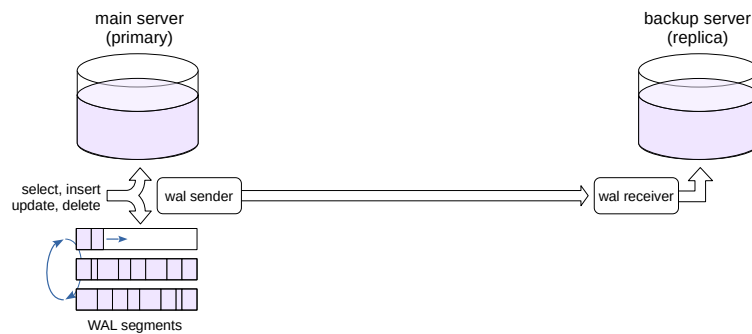
Emergency switchover

- switch to a replica due to a primary server failure
- manual mode, but can be automated with external cluster software

There are different reasons for switching to a backup server. The switchover can be performed routinely at a convenient time to allow for maintenance shutdown of the main server. If it is a main server failure, on the other hand, the switchover has to be performed as quickly as possible to avoid service downtime.

Even an emergency switchover must be performed manually, because PostgreSQL does not come with integrated cluster management software that should monitor the state of the servers and initiate switchovers.

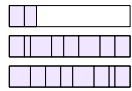
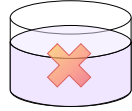
Switching to a replica



The image above illustrates the state of the servers before a switchover. The main server is on the left, the replica on the right, and replication is set up between the two.

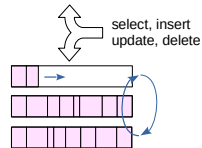
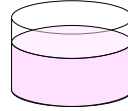
Switching to a replica

former main server



WAL segments

main server
(former replica)



In case of a main server failure or a planned switchover, the replica is given the command to stop recovering and become an independent server, and the former main server is disconnected.

Of course, a way to redirect users to a new server is required, but this is done by means outside of PostgreSQL.

Switchover to replica

Tell the replica to exit recovery mode and start as usual.

```
student$ sudo pg_ctlcluster 13 replica promote
```

```
| => INSERT INTO test VALUES (2, 'Two');
```

```
| INSERT 0 1
```

We have two completely independent servers running at the same time.

Main server recovery



After the former main server is recovered or any maintenance on it is complete, it connects as a replica to the new main server.

Simply restarting the server will not work

WAL records missed by the replica due to delay will be lost

Restoring from a backup “from scratch”

a fresh new replica is deployed at the former primary server
this is time-consuming (rsync can accelerate it somewhat)

`pg_rewind`

“rolls back” the lost WAL records, replacing the corresponding pages on the disk with pages from the new primary server
comes with a number of limitations

If the switchover has occurred due to a hardware failure (disk or server replacement is required) or an operating system failure (OS reinstallation is required), then the only option is to create a completely new replica on the server.

If the switchover was a planned one, the server can be reconnected quickly (now as a replica).

Unfortunately, you can't simply switch the server back on and connect it to the new primary server over the replication protocol. Because of replication delay, some WAL records could have not made it to the replica. If the old primary server has such records and the new primary server doesn't, then applying WALs from the new primary server will ruin the database.

You can always scrap the old primary server data and create a brand new replica instead from a base backup. However, for large databases, this can take a long time. The rsync process can speed this up to an extent.

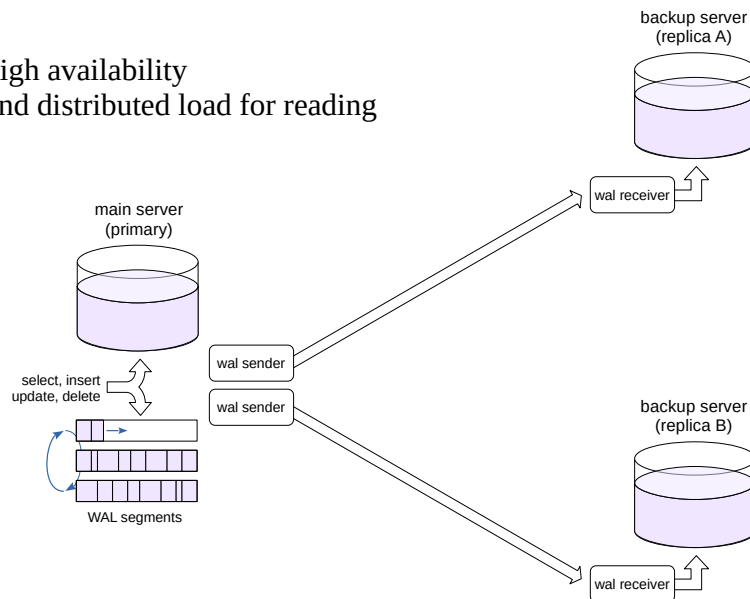
An even faster option is to use `pg_rewind`.

<https://postgrespro.com/docs/postgresql/13/app-pgrewind>

`pg_rewind` detects WAL records that have not reached the replica (starting from the last common checkpoint) and finds the pages affected by these records. The pages (should be just a few) are replaced with pages from the new primary server. In addition, `pg_rewind` copies all service files from the source server (the new primary server). The usual recovery process takes it from there.

1. Multiple replicas

high availability
and distributed load for reading



17

The replication mechanism offers flexible system design options for a variety of applications. Let's consider several typical cases and possible solutions.

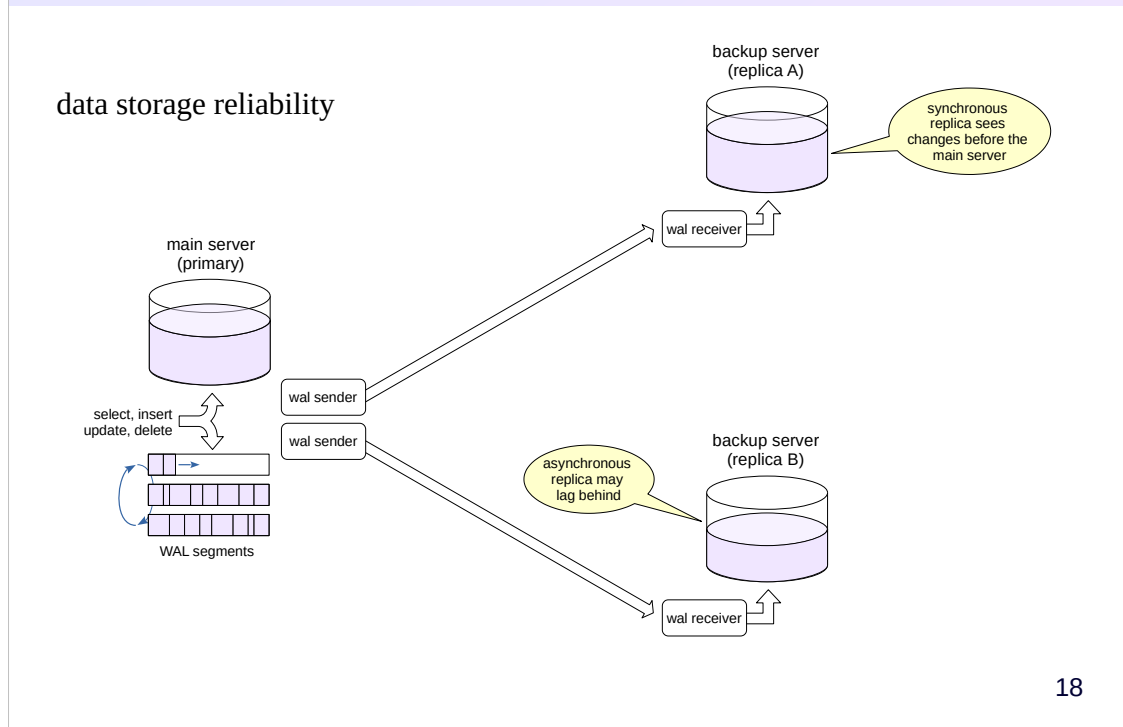
Case: a system with high availability and reading load distribution.

Solution: a primary server and multiple replicas. Replicas can serve read-only queries and can take over immediately if the main server fails.

Each replica will have a dedicated wal sender process on the main server and a replication slot, if necessary.

The reading load distribution between replicas must be done by external software.

2. Synchronous replication



Case: in the event of a main server failure, no data must be lost when switching to a replica.

Synchronous replication is the solution. In a single server environment, synchronous WAL recording ensures that the committed data will not be lost in the event of a failure. Replication works in a similar manner. Changes on the main server are committed only when a confirmation from the replica is received. If necessary, synchronization can be managed at the transaction level.

Synchronous replication does not ensure perfect data consistency between servers. Changes can become visible on the primary server and on the replica at different points in time.

Synchronization can be set up with multiple replicas. In this case, you can also set up quorum-based voting.

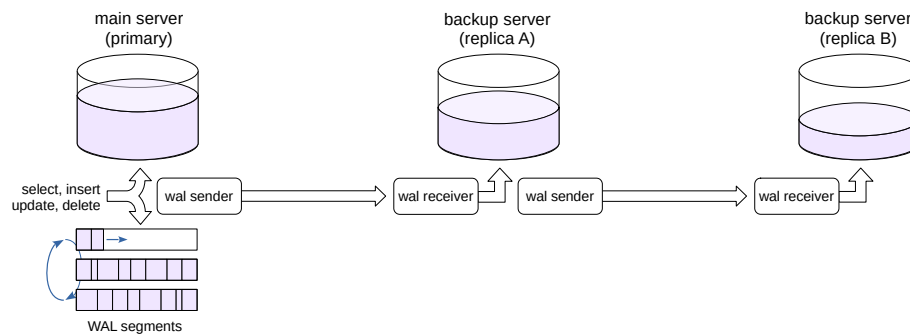
In the image above, replica B is asynchronous and may lag behind; replica A is synchronous. When committing changes, the main server performs following actions:

- makes a WAL record (so that the change is not lost in case of failure),
- waits for a confirmation from the replica that the WAL record was received on its end,
- changes the state of the transaction in the clog buffer.

With this setup, a query to a synchronous replica can see changes even earlier than a query to the primary server.

3. Cascading replication

multiple replicas
no additional load on the master server



19

Case: have multiple replicas without creating additional load on the main server.

The solution is to use cascading replication. With this setup, each replica sequentially transfers WAL records to the next one.

Cascading replication does not support synchronization, but the main server still collects feedback from all replicas, so the functionality is there.

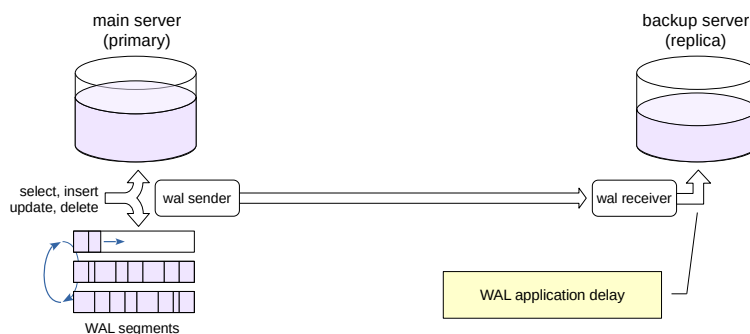
When switching over, the replica closest to the main server in the replication chain should be selected, as it lags behind the least.

The image shows that the main server has only one wal sender process, and replicas transfer WAL records to each other along the chain. The further away a replica from the main server, the more the delay. With this setup, the replication process has to be monitored at multiple servers at once, making monitoring more complicated.

4. Delayed replication

“time machine”

can recover to a specific moment in time without a WAL archive



20

Case: have the ability to view data at and recover to an arbitrary point in time.

The usual archive-based point-in-time recovery mechanism can work here, but it requires a lot of preparation and takes a lot of time. And PostgreSQL itself doesn't allow to make data snapshots for a given moment in the past.

The solution is to have a replica apply WAL records not immediately, but with a certain delay.

In order for the delay to work correctly, clock synchronization between servers is necessary.

If a replica switches from continuous recovery mode into normal operation, the rest of the records will be applied immediately.

Feedback is tricky in this setup. A large delay will cause table bloating on the main server, since vacuuming will not delete old versions of rows that may be needed by the replica as quickly as it usually does.

Publications and subscriptions
Conflict detection and resolution
Replica configuration and usage options

Stock logical replication tools first appeared in PostgreSQL 10.

Row-level logical changes are transmitted over the replication protocol.
Logical replication requires the *wal_level* parameter set to logical.

There is no main server or replica roles in logical replication, so it is possible to set up bidirectional replication.

Publisher

- streams data changes row by row in the order they are committed (replicates INSERT, UPDATE, DELETE and TRUNCATE commands)
- can do initial synchronization
- always uses the logical replication slot
- `wal_level = logical`

Subscriber

- receives and applies changes
- no parsing, rewriting and planning, just blind execution
- possible conflicts with local data

22

Logical replication uses the publish-subscribe pattern. A *publication* is created on one server, which can include a number of tables in a single database. Other servers can *subscribe* to this publication to receive and apply changes to the tables.

Only table row modifications are replicated, not SQL commands. DDL commands are not transmitted, so target tables on the subscriber must be created manually. Initial synchronization can be used to synchronize the tables when a subscription is created.

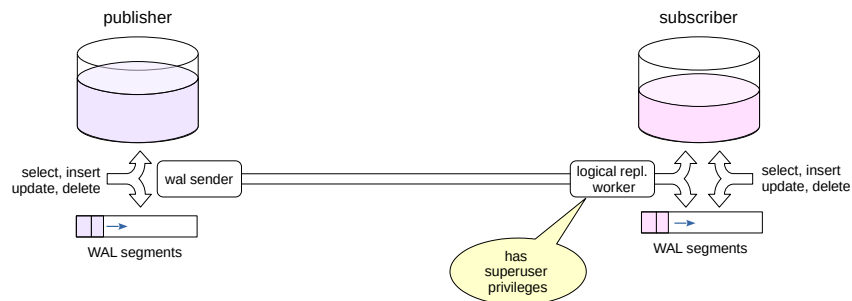
The information about modified rows is extracted and decoded from existing WAL records on the publishing server, and then sent by the wal sender process to the subscriber over the replication protocol. The transmission format is independent of the platform and server version. The log level on the publishing server (the `wal_level` parameter) must be set to “logical” for this all to work.

The logical replication worker process on the subscriber accepts and applies the changes. In order to guarantee transmission reliability (no losses and repetitions), a *logical replication slot* (similar to the physical replication slot) is required.

The changes are applied without executing SQL commands, avoiding the overhead of parsing and planning. On the other hand, a single SQL command can result in multiple one-row changes.

<https://postgrespro.com/docs/postgrespro/13/logical-replication>

Logical replication



The image shows the logical replication worker process on the subscriber that receives data from the publisher and applies it. Meanwhile, the server works normally and accepts both read and write queries.

Logical replication

To set up logical replication between two servers, we need the WAL to store additional information.

```
=> ALTER SYSTEM SET wal_level = logical;
```

```
ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 13 main restart
```

Create a publication on the first server:

```
student$ psql -d replica_overview
```

```
=> CREATE PUBLICATION test_pub FOR TABLE test;
```

```
CREATE PUBLICATION
```

```
=> \dRp+
```

```

              Publication test_pub
 Owner | All tables | Inserts | Updates | Deletes | Truncates | Via root
-----+-----+-----+-----+-----+-----+-----
student | f          | t       | t       | t       | t       | f
Tables:
"public.test"
```

Subscribe to the publication on the second server (disable the initial data copying):

```
=> CREATE SUBSCRIPTION test_sub
CONNECTION 'port=5432 user=student dbname=replica_overview'
PUBLICATION test_pub WITH (copy_data = false);
```

```
NOTICE: created replication slot "test_sub" on publisher
CREATE SUBSCRIPTION
```

```
=> \dRs
```

```

          List of subscriptions
 Name   | Owner | Enabled | Publication
-----+-----+-----+-----
test_sub | student | t       | {test_pub}
(1 row)
```

```
=> INSERT INTO test VALUES (3, 'Three');
```

```
INSERT 0 1
```

```
=> SELECT * FROM test;
```

```

 id | descr
----+-----
  1 | One
  2 | Two
  3 | Three
(3 rows)
```

The following view shows the state of the subscription:

```
=> SELECT * FROM pg_stat_subscription \gx
```

```

-[ RECORD 1 ]-----+-----
subid          | 24618
subname        | test_sub
pid            | 145074
reloid         |
received_lsn   | 0/802BB50
last_msg_send_time | 2024-03-07 13:52:08.864581+03
last_msg_receipt_time | 2024-03-07 13:52:08.865103+03
latest_end_lsn | 0/802BB50
latest_end_time | 2024-03-07 13:52:08.864581+03
```

The logical replication worker process starts (you can see its ID in pg_stat_subscription.pid):

```
student$ sudo ps -o pid,command --ppid 144247
```

```

PID COMMAND
```

144249 postgres: 13/replica: checkpointer
144250 postgres: 13/replica: background writer
144251 postgres: 13/replica: stats collector
144686 postgres: 13/replica: student replica_overview [local] idle
144860 postgres: 13/replica: walwriter
144861 postgres: 13/replica: autovacuum launcher
144862 postgres: 13/replica: logical replication launcher
145074 postgres: 13/replica: logical replication worker for subscription 24618

Identification modes for modifying and deleting rows

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

Conflicts: violation of integrity constraints

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

Inserting new rows is straightforward. Changes and deletions are more complicated. These operations need to somehow identify the old version of the row. By default, primary key columns are used for this, but you can specify other ways (replica identity) when defining a table, i.e. use a unique index or all the table columns. Or you can disable replication for some tables altogether (system catalog tables have it disabled by default).

Since the table on the publisher and the table on the subscriber can change independently of each other, conflicts in the form of integrity constraint violations are possible when inserting new row versions. Whenever this happens, the process of applying records is suspended until the conflict is resolved manually. You can either correct the data on the subscriber to resolve the conflict, or cancel the application of the conflicting records.

Conflicts

Local changes on the subscriber side are allowed. Insert a row into a table on the second server:

```
=> INSERT INTO test VALUES (4, 'Four (local)');  
  
INSERT 0 1
```

If we add a row with the same primary key value on the server, the subscription will have a conflict when trying to apply the change.

```
=> INSERT INTO test VALUES (4, 'Four');  
  
INSERT 0 1  
  
=> INSERT INTO test VALUES (5, 'Five');  
  
INSERT 0 1
```

The subscription is unable to apply the change, replication stops.

```
=> SELECT * FROM pg_stat_subscription \gx  
  
-[ RECORD 1 ]-----+-----  
subid          | 24618  
subname        | test_sub  
pid            |  
relid          |  
received_lsn   |  
last_msg_send_time |  
last_msg_receipt_time |  
latest_end_lsn |  
latest_end_time |  
  
=> SELECT * FROM test;  
  
 id | descr  
----+-----  
  1 | One  
  2 | Two  
  3 | Three  
  4 | Four (local)  
(4 rows)
```

To resolve the conflict, remove the row on the second server and wait a moment...

```
=> DELETE FROM test WHERE id=4;  
  
DELETE 1  
  
=> SELECT * FROM test;  
  
 id | descr  
----+-----  
  1 | One  
  2 | Two  
  3 | Three  
  4 | Four  
  5 | Five  
(5 rows)
```

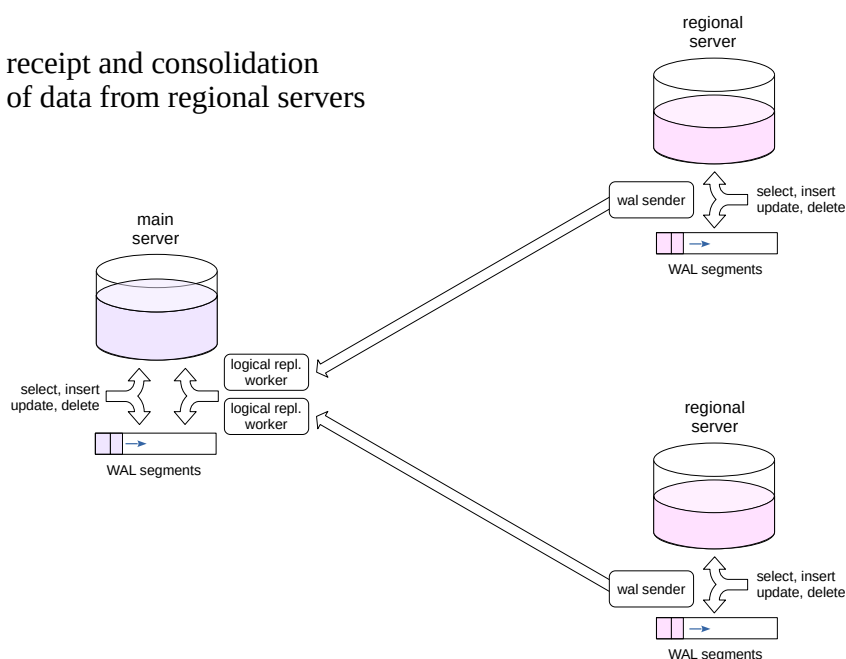
Replication continues.

If replication is no longer needed, the subscription has to be deleted manually, otherwise the publisher will keep the open replication slot.

```
=> DROP SUBSCRIPTION test_sub;  
  
NOTICE:  dropped replication slot "test_sub" on publisher  
DROP SUBSCRIPTION
```

1. Consolidation

receipt and consolidation
of data from regional servers



27

Let's discuss some logical replication use cases.

Suppose there are several regional branches, each of which runs on its own PostgreSQL server. The goal is to consolidate some of the data on a central server.

First, publications of the necessary data are created on regional servers. The central server subscribes to these publications. The received data can be processed using triggers on the central server (for example, unifying the data format).

Inverted, the setup allows, for example, to transfer reference information from the central server to regional ones.

Note that the replication relies on maintaining logical replication slots, and the slots require a stable connection. If the connection breaks, the main server will be forced to save its WAL files on disk.

An existing business logic may apply additional constraints on the system. In some cases, it may be easier to transmit data in batches every now and again.

The image shows two WAL receivers running on the central server, one for each subscription.

2. Rolling out server updates

updating the PostgreSQL version
without interruption of service



Case: update the PostgreSQL major version on the server without interrupting the service.

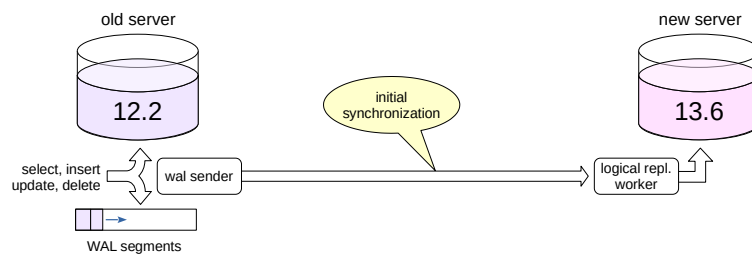
The two major versions don't have binary compatibility, so physical replication will not work. However, logical replication can solve the problem.

As usual, external tools are required to switch users between servers.

First, a new server is created with the desired PostgreSQL version.

2. Rolling out server updates

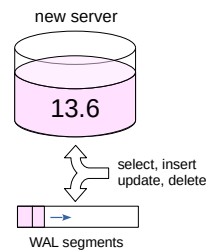
updating the PostgreSQL version
without interruption of service



Then, logical replication of all required databases is set up between the servers, and the servers are synchronized. This is possible because logical replication does not require binary compatibility between servers.

2. Rolling out server updates

updating the PostgreSQL version
without interruption of service

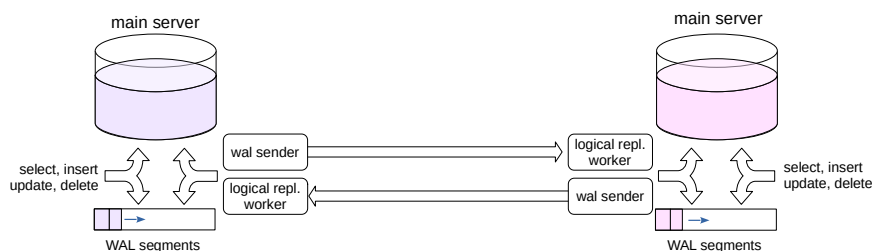


After that, clients switch to the new server, and the old one shuts down.

In practice, the process of updating major server versions using logical replication is much more complicated and difficult. It is discussed in more detail in the the topic “Server Update” in the DBA2 course.

3. Primary-primary

a cluster where
multiple servers can modify data



31

Case: provide reliable data storage on multiple servers with the ability to change the data on any node.

Regular physical replication allows you to change data only on the primary server. Logical replication makes it possible to change data simultaneously on multiple servers. This requires that the applications working with the cluster are built with certain considerations in mind in order to avoid conflicts when modifying data in the same table. One of those is to ensure that different servers work with different ranges of keys.

Keep in mind that the primary-primary setup with logical replication will not support global distributed transactions. With synchronous replication, reliability can be ensured, but consistency of data between servers cannot. In addition, PostgreSQL does not offer any tools for automatic failure processing, connecting or removing nodes from the cluster, etc. These tasks must be solved by external means.

The image shows a primary-primary setup. Each of the servers creates a publication and a subscription, establishing a bidirectional exchange of WAL records. PostgreSQL 13 does not support such replication just yet, but this feature is bound to appear sooner or later. See extensions `pg_logical` <https://www.2ndquadrant.com/en/resources-old/pglogical/> and BDR <https://www.2ndquadrant.com/en/resources-old/postgres-bdr-2ndquadrant/>

The replication mechanism works by delivering WAL records to the replica and applying them there

- streaming WAL records or transferring files

Physical replication creates an exact copy of the entire cluster

- unidirectional

- requires binary compatibility

Logical replication streams individual row changes

- multidirectional

- requires protocol-level compatibility

1. Set up physical streaming replication between the two servers in synchronous mode.
2. Verify that replication works as intended. Make sure that when the replica is stopped, commits on the primary server could not be completed.
3. Exit recovery mode on the replica.
4. Create two tables on both servers.
5. Set up logical replication for the first table from the first server to the second, and the other way around for the second table.
6. Verify that replication works as intended.

1. To do this, set the parameters on the main server:

- *synchronous_commit* = on,
- *synchronous_standby_names* = 'replica',

and on the replica in the postgresql.auto.conf file, add
"application_name=replica" to the *primary_conninfo* parameter.

1. Synchronous physical streaming replication

Instead of using the -R key, create our own postgresql.auto.conf manually, because we need to specify some non-standard parameters.

```
student$ pg_basebackup --pgdata=/home/student/basebackup
```

Make sure the second server is stopped and push the backup:

```
student$ sudo pg_ctlcluster 13 replica status
```

pg_ctl: no server running

```
student$ sudo rm -rf /var/lib/postgresql/13/replica
```

```
student$ sudo mv /home/student/basebackup/ /var/lib/postgresql/13/replica
```

```
student$ sudo chown -R postgres:postgres /var/lib/postgresql/13/replica
```

Configure and start the replica:

```
student$ echo "primary_conninfo = 'user=student port=5432 application_name=replica'" | sudo tee /var/lib/postgresql/13/replica/postgresql.auto.conf
```

primary_conninfo = 'user=student port=5432 application_name=replica'

```
student$ sudo touch /var/lib/postgresql/13/replica/standby.signal
```

```
student$ sudo pg_ctlcluster 13 replica start
```

Main server configuration:

```
student$ psql
```

```
=> ALTER SYSTEM SET synchronous_commit = on;
```

ALTER SYSTEM

```
=> ALTER SYSTEM SET synchronous_standby_names = 'replica';
```

ALTER SYSTEM

```
student$ sudo pg_ctlcluster 13 main reload
```

2. Physical replication verification

```
student$ psql
```

```
=> CREATE DATABASE replica_overview;
```

CREATE DATABASE

```
=> \c replica_overview
```

You are now connected to database "replica_overview" as user "student".

```
=> CREATE TABLE t(n integer);
```

CREATE TABLE

```
=> INSERT INTO t VALUES (1);
```

INSERT 0 1

```
=> SELECT * FROM pg_stat_replication \gx
```

```
-[ RECORD 1 ]-----+-----
pid           | 162643
usesysid      | 16384
username      | student
application_name | replica
client_addr   |
client_hostname |
client_port   | -1
backend_start  | 2024-03-07 13:55:51.773678+03
backend_xmin   |
state         | streaming
sent_lsn       | 0/1401A5C8
write_lsn      | 0/1401A5C8
flush_lsn      | 0/1401A5C8
replay_lsn     | 0/1401A5C8
write_lag      | 00:00:00.000166
flush_lag      | 00:00:00.00269
replay_lag     | 00:00:00.003026
sync_priority  | 1
sync_state     | sync
reply_time     | 2024-03-07 13:55:55.33892+03
```

sync_state: sync means that replication is running in the synchronous mode.

```
student$ psql -p 5433
```

```
| => \c replica_overview
```

```
| You are now connected to database "replica_overview" as user "student".
```

```
| => SELECT * FROM t;
```

```
|      n
| ----
|      1
| (1 row)
```

```
student$ sudo pg_ctlcluster 13 replica stop
```

```
=> BEGIN;
```

```
BEGIN
```

```
=> INSERT INTO t VALUES (2);
```

```
INSERT 0 1
```

```
=> COMMIT;
```

The commit waits for the synchronous replica.

```
student$ sudo pg_ctlcluster 13 replica start
```

```
COMMIT
```

```
student$ psql -p 5433 -d replica_overview
```

```
| => SELECT * FROM t;
```

```
|      n  
|      ---  
|      1  
|      2  
| (2 rows)
```

3. Completion of recovery

```
student$ sudo pg_ctlcluster 13 replica promote
```

The synchronous mode on the first server needs to be disabled now:

```
=> ALTER SYSTEM RESET synchronous_standby_names;
```

```
ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 13 main reload
```

4. Tables to verify logical replication

```
=> CREATE TABLE a(id integer);
```

```
CREATE TABLE
```

```
=> CREATE TABLE b(s text);
```

```
CREATE TABLE
```

```
| => CREATE TABLE a(id integer);
```

```
| CREATE TABLE
```

```
| => CREATE TABLE b(s text);
```

```
| CREATE TABLE
```

5. Logical replication configuration

```
=> ALTER SYSTEM SET wal_level = logical;
```

```
ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 13 main restart
```

```
| => ALTER SYSTEM SET wal_level = logical;
```

```
| ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 13 replica restart
```

```
student$ psql -d replica_overview
```

```
=> CREATE PUBLICATION a_pub FOR TABLE a;
```

```
CREATE PUBLICATION
```

```
student$ psql -p 5433 -d replica_overview
```

```
| => CREATE PUBLICATION b_pub FOR TABLE b;
```

```
| CREATE PUBLICATION
```

```
=> CREATE SUBSCRIPTION b_sub
```

```
CONNECTION 'port=5432 user=student dbname=replica_overview'  
PUBLICATION b_pub;
```

```
NOTICE: created replication slot "b_sub" on publisher  
CREATE SUBSCRIPTION
```

```
| => CREATE SUBSCRIPTION a_sub
```

```
CONNECTION 'port=5432 user=student dbname=replica_overview'  
PUBLICATION a_pub;
```

```
| NOTICE: created replication slot "a_sub" on publisher  
| CREATE SUBSCRIPTION
```

6. Logical replication verification

```
=> INSERT INTO a VALUES (1);
```

```
INSERT 0 1
```

```
| => SELECT * FROM a;
```

```
|      id  
|      ---  
|      1  
| (1 row)
```

```
| => INSERT INTO b VALUES ('Pa3');
```

```
| INSERT 0 1
```

```
=> SELECT * FROM b;
```

```
 s  
-----  
Pa3  
(1 row)
```

Remove the subscriptions as they are no longer needed:

```
=> DROP SUBSCRIPTION b_sub;
```

```
NOTICE: dropped replication slot "b_sub" on publisher  
DROP SUBSCRIPTION
```

```
| => DROP SUBSCRIPTION a_sub;
```

```
| NOTICE: dropped replication slot "a_sub" on publisher  
DROP SUBSCRIPTION
```