Engineer

Replication

keeping copies of a database in step on other servers

What it is

Replication keeps copies of a database on other servers and sends each copy every change made to the original, so that they all stay in step. Where the context leaves room for doubt it is called database replication. The PostgreSQL documentation calls the one server that accepts writes the primary and the servers that follow it standbys. Redis speaks of a master and its replicas.

PostgreSQL has two kinds, and they copy different things.

Physical replication copies the write-ahead log (WAL), the record every change is written to before it reaches the data files. A standby replays that log and becomes a block-for-block copy of the whole cluster. In streaming replication, the standby keeps a connection open to the primary and receives WAL records as they are generated, instead of waiting for finished WAL files to be shipped. A standby that only replays, and accepts no connections until it is promoted, is a warm standby; one that also answers read-only queries is a hot standby.

Logical replication copies changes to rows. A publisher sends on the inserts, updates and deletes made to chosen tables, each row identified by its replica identity, normally the primary key, and a subscriber applies them to its own tables. Because it deals in rows rather than disk blocks, it can carry a few tables instead of the whole cluster, even to a different major version of PostgreSQL.

Streaming replication is asynchronous by default. The primary confirms a commit once the WAL is on its own disk and sends it on afterwards; the documentation puts the delay at typically under one second when the standby can keep up. With synchronous replication switched on, the commit waits until a standby has the WAL as well.

Use the concept in a decision

Put “What is replication?” to work in a real decision. We can help relate the concept to the system you are building or buying. Backups that are tested rather than assumed, and a written plan for the morning a server does not come back on.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Automated 100 critical data backups using Barman — 100 critical backups automated across the databases

Automated 100 critical data backups using Barman Get in touch →

Why it matters

The first reason is surviving the loss of a server. When the primary fails, a standby that already holds the data can be promoted and take the traffic, far sooner than a backup could be restored onto a fresh machine.

The second is spreading the reads. In many applications most queries only read, and a hot standby can answer them, leaving the primary free for the writes, which only it can take.

The third is moving data. Logical replication can copy just the tables a reporting database needs, or carry data across to a server on another major version of PostgreSQL.

Every copy comes with a choice between waiting and losing. Asynchronous replication costs nothing on the commit path, but a standby is always slightly behind: a read from it may miss a write that has just happened, and if the primary dies, commits it had already confirmed may never reach the standby that takes over. Synchronous replication closes that gap and puts a network round trip into every commit.

Replication is also not a backup. A standby faithfully replays an accidental `DELETE` without a `WHERE`, so within moments every copy has the mistake too. Undoing it still takes a backup.

What it looks like in practice

A shop runs its database on one primary at `10.0.0.1` and adds a second server, `db2`, as a streaming replica. On `db2`, with PostgreSQL installed and stopped:

```
pg_basebackup -h 10.0.0.1 -U replicator -D /srv/pgdata \
    -R -C -S db2_slot -P
```

`pg_basebackup` copies the whole running cluster into the directory named by `-D`, which it creates if needed and which must otherwise be empty, and by default streams the WAL generated during the copy alongside it. `-C -S db2_slot` first creates a replication slot on the primary, so that the primary keeps each WAL record until the new standby has received it. `-R` creates a `standby.signal` file and appends the connection settings, slot included, to `postgresql.auto.conf`. `-P` reports progress. The role `replicator` needs the `REPLICATION` attribute or superuser rights, and the primary's `pg_hba.conf` has to allow the replication connection.

When PostgreSQL starts on `db2`, `standby.signal` makes it come up as a standby and stream from the primary, and since `hot_standby` is on by default it takes read-only queries once it is consistent. On the primary, one row per standby shows how it is doing:

```
SELECT application_name, state, sync_state, replay_lsn, replay_lag
FROM   pg_stat_replication;
```

A healthy replica shows `state` as `streaming` and `sync_state` as `async`. `replay_lag` approximates how long recent commits took to become visible there; once the standby has caught up and WAL activity stops, it goes back to NULL rather than showing zero.

The note on optimising the database before you shard it explains why a shop does this. Most of its traffic reads: the product page, the basket, the order history. None of that needs the primary, which has to stay free for writes. A streaming replica takes those reads, and the note points out that adding one is configuration, where a shard is a project. It adds two cautions. A replica runs a little behind, so what a user has just written should be read from the primary. And a long report on a replica can be cancelled when it holds up the changes the replica is replaying.

Making a commit wait for a standby takes one setting on the primary. Each standby is named by the `application_name` in its `primary_conninfo`, and the primary lists the names it may wait for:

```
synchronous_standby_names = 'ANY 1 (db2, db3)'
```

Each commit now waits until its WAL is on disk on the primary and on at least one of `db2` and `db3`. `synchronous_commit` can be set per user, per database or per transaction, so the order that takes payment can wait while a page-view counter commits at local speed. The note on selling the last ticket exactly once shows why that matters for stock. With asynchronous replication, a primary can confirm a sale and die before its replica has it; the replica is promoted, the sale is gone, and the same unit is sold again. The note's remedy is a synchronous standby or a store built on Raft, at the price of added latency on every commit.

Logical replication takes two statements. The publisher needs `wal_level = logical`, and the tables must already exist on the subscriber, because their definitions are not copied:

```
-- on the publisher
CREATE PUBLICATION shop_pub FOR TABLE orders, customers;

-- on the subscriber
CREATE SUBSCRIPTION shop_sub
    CONNECTION 'host=10.0.0.1 dbname=shop user=replicator'
    PUBLICATION shop_pub;
```

The subscription first copies the rows already in those tables, then applies every change made after that.

Where it goes wrong

A user cannot see what they just wrote. The application writes to the primary and reads back from a replica that is still a moment behind, so a saved setting seems to revert. Route a user's reads to the primary for a short time after they write. Where data must be visible on the standby the moment the commit returns, `synchronous_commit = remote_apply` waits until the synchronous standbys have replayed the transaction, at a higher cost per commit.

Long queries on the replica are cancelled. The standby has to replay changes that clash with a running query: vacuum removing row versions the query can still see, or the exclusive lock taken by `DROP TABLE` and many forms of `ALTER TABLE`. Replay waits up to `max_standby_streaming_delay`, 30 seconds by default, then

cancels the query. A reporting replica can be given a longer delay, and falls further behind while it waits. `hot_standby_feedback = on` stops the cancellations caused by vacuum, but the documentation warns it can bloat the primary. `pg_stat_database_conflicts` on the standby counts the cancellations by cause, and the application should retry a cancelled query.

A forgotten slot fills the primary's disk. A slot holds WAL until its standby has received it, even when that standby is switched off or long gone, and `max_slot_wal_keep_size` is -1 by default, meaning no limit. Check the slots regularly:

```
SELECT slot_name, active, wal_status, safe_wal_size
FROM   pg_replication_slots;
```

Drop a slot whose standby is gone. Setting `max_slot_wal_keep_size` caps what a slot may hold; a standby that falls further behind finds its slot invalidated, with `wal_status` showing `lost`. `idle_replication_slot_timeout`, off by default, invalidates slots left inactive for longer than the time given.

Writes stop when a synchronous standby does. With one standby named and that standby down, every commit on the primary waits, and may never complete. Name more candidates than the number required, as in `ANY 1 (db2, db3)`. Failing that, lower the number or empty `synchronous_standby_names` and reload the configuration.

Two servers both believe they are the primary. PostgreSQL promotes a standby when asked, with `pg_ctl promote` or `SELECT pg_promote()`, but has no software of its own to detect a failed primary, and nothing stops the old one coming back and taking writes. The documentation warns this leads to "confusion and ultimately data loss". Fence the old primary off, by powering it down or cutting its network, before promoting, and rebuild it as a standby before it rejoins.

Logical replication leaves things behind. Schema changes are not replicated, so a column added on the publisher stops replication until the subscriber has it too; apply additive changes to the subscriber first. Sequences are not replicated either: rows arrive with their IDs, but the subscriber's sequences still stand at their start values, and inserts after a switchover fail on duplicate keys. Before switching, set each sequence past the highest value in use.

A Redis master restarts empty and takes its replicas with it. If persistence is off on the master and something restarts it automatically, it comes back with an empty dataset and its replicas copy that, deleting their own data. Keep persistence on, or do not let the master restart automatically.

Further reading

- High availability — the PostgreSQL overview of primary and standby servers, synchronous and asynchronous replication, and failover.
- Log-Shipping Standby Servers — streaming replication, replication slots and how to configure synchronous replication.
- Hot Standby — read-only queries on a standby, the conflicts that cancel them, and the settings that trade lag against cancellations.
- pg_basebackup — copying a running cluster to start a standby, with every option used above.
- Restrictions — what logical replication does not carry: schema, sequences, large objects and anything other than tables.
- Redis replication — how Redis replicas follow a master, why replication is asynchronous, and the danger of restarting without persistence.



Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is replication?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/replication/