

Replication Physical Replication



Copyright

© Postgres Professional, 2017–2025

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov, Alexey Beresnev

Photo: Oleg Bartunov (Phu Monastery and Bhrikuti Peak, Nepal)

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 Use Cases
How Physical Replication Works
WAL Record Delivery
Limitations and Considerations
Synchronous and Asynchronous Replication
Replication Monitoring
Common Issues and Troubleshooting

Replication Use Cases



Replication

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

A fundamental mechanism serving multiple goals

reliability	when a failure occurs, the system must remain operational
scalability	load balancing across servers

3

A single server managing databases may not meet the requirements of the system. There are two main challenges that require multiple servers to address.

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, it's preferable to scale horizontally rather than vertically, distributing the load across multiple servers.

This can be due to both hardware costs and the need for different configurations tailored to different workload types (such as OLTP and reporting).

In PostgreSQL, distributed systems follow a "shared nothing" architecture: multiple servers operate independently without sharing memory or storage. Therefore, when servers need to work with the same data, that data must be kept synchronized between them. This synchronization mechanism is called replication.

Leader → Replica

WAL entries are transmitted to the replica and applied there

- data flows in one direction only

- the entire cluster is replicated; selective replication is not supported

The replica is an exact copy of the leader

- the same major server version

- fully compatible architectures and platforms

Replica is read-only

During physical replication, servers have assigned roles: one is the leader and others are replicas.

The leader sends write-ahead log records to the replica, which then applies them to its data files. The replica does not “understand” the changes, just applies them mechanically without any interpretation of what the changes mean, but it is faster than re-executing the original SQL statements.

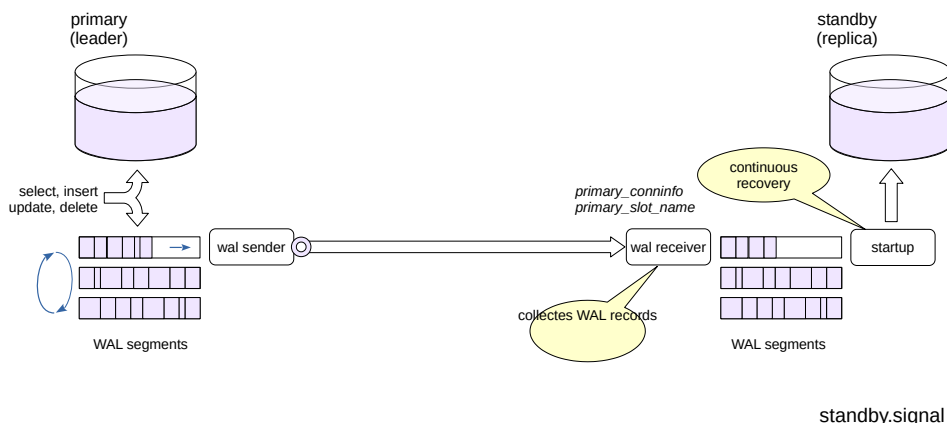
However, binary compatibility between servers is critical: they must run on the same hardware and software and use matching major versions of PostgreSQL.

<https://postgrespro.com/docs/postgresql/16/hot-standby#HOT-STANDBY-ADMIN>

Since the WAL is shared across the entire cluster, only the cluster as a whole can be replicated. There's no way to filter which log entries get replicated.

Replica does not generate records, just applies them. Therefore, replicas are read-only: no data modification operations allowed.

Streaming Replication



If there is a `recovery.signal` file, the server is restored from the backup: replays the WAL records, restores consistency and completes the recovery. Starting the replica is similar, but the recovery does not complete: the replica continues to read and apply records from the leader. The signal for this is the `standby.signal` file.

In the topic "Base Backup," we considered two ways to deliver WAL records from the server to the client: streaming and file-based. Replication uses the same two mechanisms. Streaming replication is the most common approach by far.

The `wal receiver` process on the replica connects to the leader over the replication protocol, using the connection string from the `primary_conninfo` parameter, and reads the stream of WAL records. With streaming replication, the replica lag is minimal and can be reduced to zero if needed.

The received records are written to disk as WAL segments, just as they are on the leader. Next, the `startup` process – the same one that handles crash recovery – applies records to data files, only now it works continuously.

The leader starts a `wal sender` process to handle this connection.

Streaming replication can use a replication slot when necessary. The slot name is stored in the `primary_slot_name` parameter.

Streaming Replication Configuration

Since our configuration will not include a WAL archive, it is essential to use a replication slot at every stage – otherwise, given sufficient delay, the primary may delete the required file segments before they are committed, forcing you to take a new backup and redo the entire setup.

Create a slot:

```
α=> SELECT pg_create_physical_replication_slot('replica');

pg_create_physical_replication_slot
-----
(replica,)
(1 row)
```

Examine the slot we just created:

```
α=> SELECT * FROM pg_replication_slots \gx

-[ RECORD 1 ]-----+-----
slot_name          | replica
plugin             |
slot_type          | physical
datoid             |
database           |
temporary          | f
active             | f
active_pid         |
xmin              |
catalog_xmin       |
restart_lsn        |
confirmed_flush_lsn|
wal_status         |
safe_wal_size      |
two_phase          | f
conflicting        |
```

The slot is not initialized right away (restart_lsn and wal_status are empty).

Take a standalone backup into the beta server's data directory using the slot we created. When using the -R option, the tool adds the files needed to start the future replica to the backup.

```
student$ sudo pg_ctlcluster 16 beta stop
```

Cluster is not running.

```
postgres$ rm -rf /var/lib/postgresql/16/beta
```

```
postgres$ pg_basebackup \
--checkpoint=fast \
--pgdata=/var/lib/postgresql/16/beta \
-R --slot=replica
```

Check the slot again:

```
α=> SELECT * FROM pg_replication_slots \gx

-[ RECORD 1 ]-----+-----
slot_name          | replica
plugin             |
slot_type          | physical
datoid             |
database           |
temporary          | f
active             | f
active_pid         |
xmin              |
catalog_xmin       |
restart_lsn        | 0/20000000
confirmed_flush_lsn|
wal_status         | reserved
safe_wal_size      |
two_phase          | f
conflicting        |
```

Once the backup completes, the slot is initialized and the primary retains all WAL files starting from the beginning of the backup (restart_lsn, wal_status).

Compare with the line in backup_label:

```
postgres$ head -n 1 /var/lib/postgresql/16/beta/backup_label
```

```
START WAL LOCATION: 0/2000028 (file 00000001000000000000000002)
```

The postgresql.auto.conf file was generated by pg_basebackup because of the -R option. This file contains the connection information for the primary and the replication slot name:

```
postgres$ cat /var/lib/postgresql/16/beta/postgresql.auto.conf
```

```
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
primary_conninfo = 'user=postgres passfile='/var/lib/postgresql/.pgpass'
channel_binding=prefer host='/var/run/postgresql' port=5432 sslmode=prefer
sslnegotiation=postgres sslcompression=0 sslcertmode=allow sslsni=1
ssl_min_protocol_version=TLsv1.2 gssencmode=prefer krbsrvname=postgres gssdelegation=0
target_session_attrs=any load_balance_hosts=disable'
primary_slot_name = 'replica'
```

By default, the replica will be a hot standby, meaning it can serve queries during recovery. If this is not necessary, the replica can be warm instead (hot_standby = off).

The tool also created the standby.signal file. Its presence instructs the server to enter continuous recovery mode.

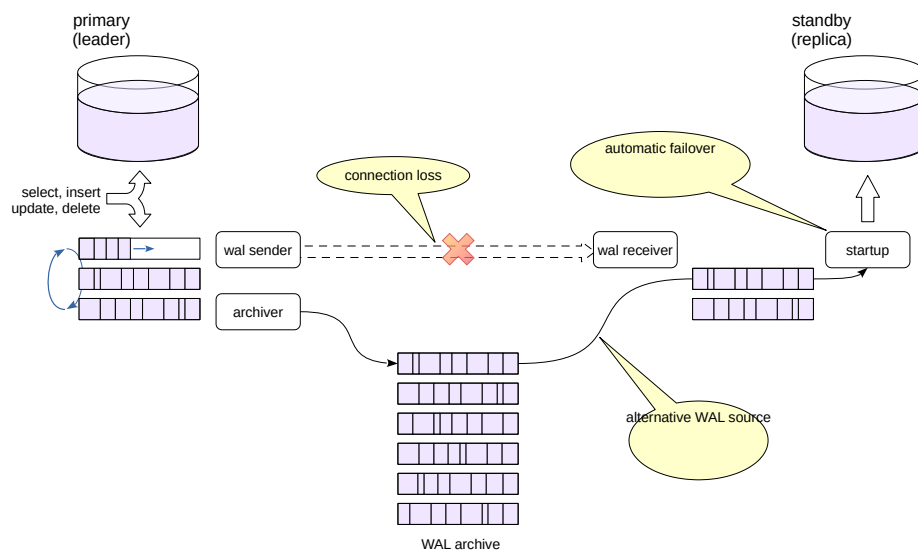
```
postgres$ ls -l /var/lib/postgresql/16/beta/standby.signal
```

```
-rw----- 1 postgres postgres 0 Sep 16 23:57 /var/lib/postgresql/16/beta/standby.signal
```

The WAL records needed to bring the replica to a consistent state will be read from the standalone base backup. Then it will enter continuous recovery mode and begin receiving and replaying the WAL stream.

```
student$ sudo pg_ctlcluster 16 beta start
```

Replication with a WAL archive



Another way to deliver WAL records is to give the replica access to the WAL archive, just as you would during regular recovery from archive (this mechanism is covered in the "Backup" module).

If the replica can't retrieve the next WAL record via the replication protocol for any reason (such as a connection loss), it will attempt to read it from the archive using the command specified in the *restore_command* parameter. When the connection is restored, the replica will automatically switch back to streaming replication.

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

- the replica is forced to lag behind the leader by the time it takes to fill the WAL segment;
- the leader has no awareness of the replica's existence, which can lead to problems in certain situations.

Leader	Replica
wal sender	wal receiver
	startup
archiver	archiver
	(<i>archive_mode</i> = always)
checkpointer (checkpoints)	checkpointer (restartpoints)
background writer	background writer
wal writer	
autovacuum	

Different sets of processes run on the leader and on the replica. Beyond the wal sender (on the leader) and wal receiver and startup processes (on the replica) we've already covered, there are others.

Common processes:

- **background writer** – writes dirty pages from the buffer cache to disk (on the replica, pages are read into the buffer cache in the same way to apply WAL records, then written either in the background or during page eviction).
- **checkpointer** – performs restartpoints (See “WAL Archive”). If a failure occurs during recovery, the replica can resume from the last restartpoint rather than starting over from the beginning.

Processes running only on the leader:

- **wal writer** – the replica doesn't generate WAL records; it only receives them from the leader.
- **autovacuum launcher/worker** — the replica doesn't perform vacuum operations; when vacuum runs on the primary, it's propagated to the replica through WAL.

There is also a process that may or may not be present depending on the configuration:

- **archiver** — runs only on the leader when *archive_mode* = on, but runs on both the leader and replica when *archive_mode* = always (covered in more detail in the "Switchover to Replica" module).

Replica Processes

Compare the processes on the primary:

```
student$ ps f -C postgres -o pid,command | grep alpha
```

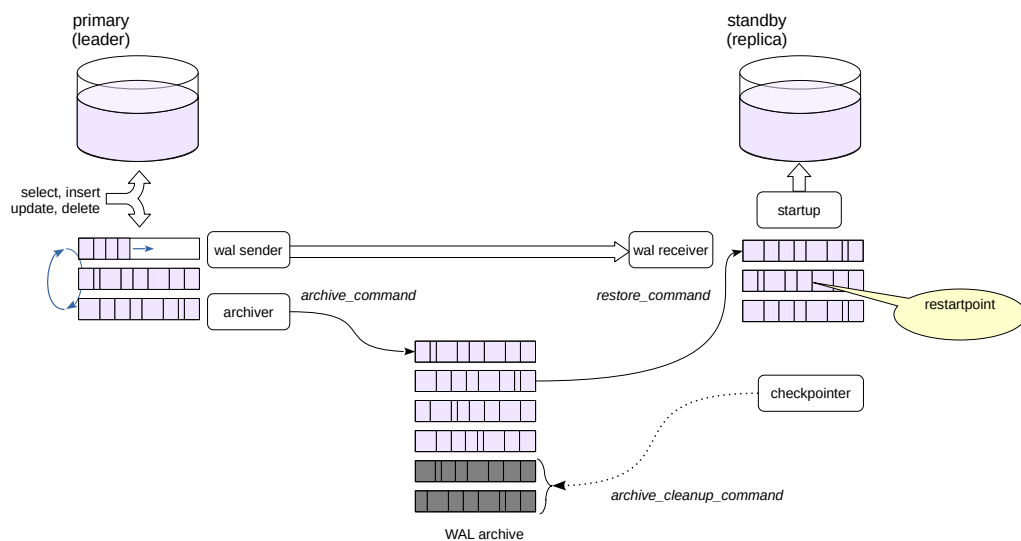
```
73709 /usr/lib/postgresql/16/bin/postgres -D /var/lib/postgresql/16/alpha -c
config_file=/etc/postgresql/16/alpha/postgresql.conf
73710 \_ postgres: 16/alpha: checkpointer
73711 \_ postgres: 16/alpha: background writer
73713 \_ postgres: 16/alpha: walwriter
73714 \_ postgres: 16/alpha: autovacuum launcher
73715 \_ postgres: 16/alpha: logical replication launcher
73758 \_ postgres: 16/alpha: student student [local] idle
74265 \_ postgres: 16/alpha: walsender postgres [local] streaming 0/3000060
```

And on the replica:

```
student$ ps f -C postgres -o pid,command | grep beta
```

```
74260 /usr/lib/postgresql/16/bin/postgres -D /var/lib/postgresql/16/beta -c
config_file=/etc/postgresql/16/beta/postgresql.conf
74261 \_ postgres: 16/beta: checkpointer
74262 \_ postgres: 16/beta: background writer
74263 \_ postgres: 16/beta: startup recovering 0000000100000000000000003
74264 \_ postgres: 16/beta: walreceiver streaming 0/3000060
```

Archive Cleanup



10

At restartpoints, the checkpointer process will delete unneeded WAL segments from the local `pg_wal` directory.

If the archive is used to synchronize just one replica, you can also clean it automatically. The `archive_cleanup_command` parameter specifies the command that will be executed at the end of each restartpoint. For example, you can set it to run the `pg_archivecleanup` utility (the server will set the restartpoint segment name as `%r`):

```
archive_cleanup_command = 'pg_archivecleanup /path/to/archive %r'
```

<https://postgrespro.com/docs/postgresql/16/runtime-config-wal#RUNTIME-CONFIG-WAL-ARCHIVE-RECOVERY>

<https://postgrespro.com/docs/postgresql/16/pgarchivecleanup>

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...)

We've already mentioned that replicas don't allow any data modifications. More specifically, the following operations are not allowed:

- modifications to tables, materialized views, and sequences;
- locks (since these require modifying data pages);
- DDL commands, including creating temporary tables;
- commands such as VACUUM and ANALYZE;
- access control commands.

A replica can execute read queries when *hot_standby* = on. Setting server parameters and transaction management commands will also work. For example, you can start a (reading) transaction with a specific isolation level. Additionally, replicas can be used for creating backups.

Replica usage

Run several commands on the primary server:

```
α=> CREATE DATABASE replica_physical;
```

CREATE DATABASE

```
α=> \c replica_physical
```

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

```
α=> CREATE TABLE test(s text);
```

CREATE TABLE

```
α=> INSERT INTO test VALUES ('Hello, world!');
```

INSERT 0 1

Check the replica:

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

```
| β=> \c replica_physical
```

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

```
| β=> SELECT * FROM test;
```

```
|      s  
| -----  
| Hello, world!  
| (1 row)
```

No changes can be done on replica directly:

```
| β=> INSERT INTO test VALUES ('Replica');
```

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

In general, you can tell a replica apart from the primary using the function:

```
| β=> SELECT pg_is_in_recovery();
```

```
| pg_is_in_recovery  
| -----  
| t  
| (1 row)
```

Isolation and MVCC

- large number of savepoints may slow down query execution
- Serializable isolation level is not supported

Triggers and locks

- triggers and advisory locks do not work

Creating backups on a replica

- the *full_page_writes* parameter must be enabled on the leader beforehand
- delays caused by waiting for checkpoints and segment switches

Order of changes

- tablespace directories – create on replica first
- max_...* parameter values – increase on replica first

Replication has several limitations and considerations.

If the leader is running a transaction with more than 64 nested transactions (that is, it is actively using savepoints), snapshot creation on the replica will be suspended, preventing query execution for a while.

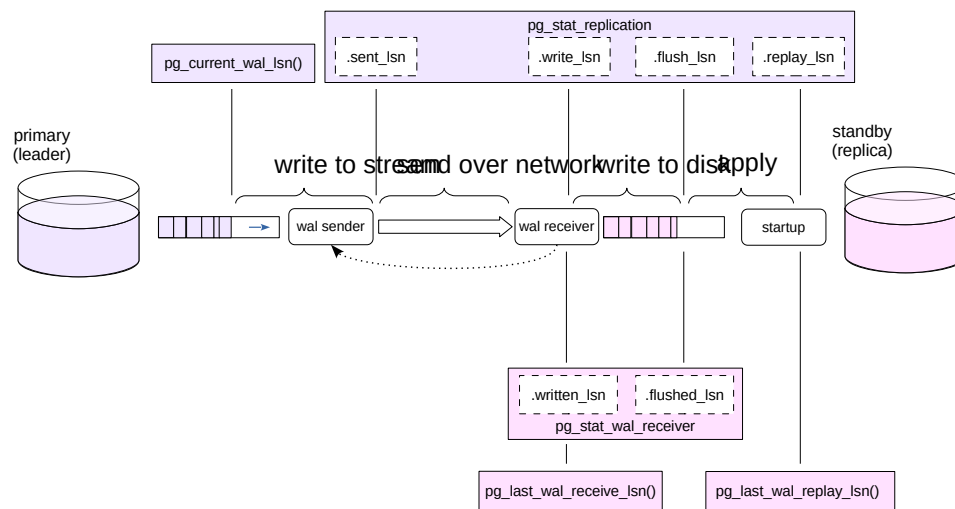
The Serializable isolation level is not supported on replicas.

Triggers won't fire on the replica since they've already executed on the leader: only their results get replicated. Advisory locks won't work either.

When performing backups from a replica, there's no way to fully control the primary server – specifically, you can't trigger checkpoints or enable the *full_page_writes* parameter. Therefore, you need to enable this parameter on the leader in advance, and then wait for a checkpoint to occur, which can significantly delay the process.

Tablespace creation is replicated, but if the OS user who started PostgreSQL does not have the privileges to create the required directory, an error will occur. The best approach is to first create the directories on the leader and all replicas, then execute the CREATE TABLESPACE command.

If you increase a resource allocation parameter on the leader, replication may stop. This affects *max_connections*, *max_prepared_transactions*, *max_locks_per_transaction*, *max_wal_senders* and *max_worker_processes*.



During replication, issues may arise that should be caught by properly configured monitoring. The key data is WAL positions. There are five main steps:

- 1) record created on leader
- 2) record transmitted by wal sender
- 3) record received by wal receiver
- 4) record synchronised to disk (flush)
- 5) record applied by startup process

The leader usually tracks all five steps. The first one is tracked through the `pg_current_wal_lsn()` function, while the others come from the `pg_stat_replication` view. The replica passes replication status to the leader each time it writes to disk, but at least once every `wal_receiver_status_interval` (default is 10 seconds).

Information about the replication slot can be found in the `pg_replication_slots` view.

Some of the replication progress information can also be obtained from the replica:

in the `pg_stat_wal_receiver` view and via `pg_last_wal_receive_lsn()` and `pg_last_wal_replay_lsn()` functions.

Both functions `pg_is_in_recovery()` and `pg_get_wal_replay_pause_state()` return replication status information. The `log_replication_commands` parameter enables logging of replication protocol commands to the server log.

Replication Monitoring

The replica is currently in recovery mode and applying WAL records:

```
β=> SELECT pg_is_in_recovery(), pg_get_wal_replay_pause_state();

 pg_is_in_recovery | pg_get_wal_replay_pause_state 
-----+-----
 t                  | not paused
(1 row)
```

The current max_connections value on the primary:

```
α=> SHOW max_connections;

max_connections
-----
100
(1 row)
```

If the number of users has grown, we'll want to increase this limit:

```
α=> ALTER SYSTEM SET max_connections = 150;
```

```
ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 16 alpha restart
```

How's the replica doing?

```
β=> SELECT pg_is_in_recovery(), pg_get_wal_replay_pause_state();

 pg_is_in_recovery | pg_get_wal_replay_pause_state 
-----+-----
 t                  | paused
(1 row)
```

Replication has stopped. The reason is in the log:

```
student$ tail -n 7 /var/log/postgresql/postgresql-16-beta.log
```

```
2026-09-16 23:58:21.189 MSK [74263] WARNING: hot standby is not possible because of
insufficient parameter settings
2026-09-16 23:58:21.189 MSK [74263] DETAIL: max_connections = 100 is a lower setting
than on the primary server, where its value was 150.
2026-09-16 23:58:21.189 MSK [74263] CONTEXT: WAL redo at 0/3432EE8 for
XLOG/PARAMETER_CHANGE: max_connections=150 max_worker_processes=8 max_wal_senders=10
max_prepared_xacts=0 max_locks_per_xact=64 wal_level=replica wal_log_hints=off
track_commit_timestamp=off
2026-09-16 23:58:21.189 MSK [74263] LOG: recovery has paused
2026-09-16 23:58:21.189 MSK [74263] DETAIL: If recovery is unpaused, the server will
shut down.
2026-09-16 23:58:21.189 MSK [74263] HINT: You can then restart the server after making
the necessary configuration changes.
2026-09-16 23:58:21.189 MSK [74263] CONTEXT: WAL redo at 0/3432EE8 for
XLOG/PARAMETER_CHANGE: max_connections=150 max_worker_processes=8 max_wal_senders=10
max_prepared_xacts=0 max_locks_per_xact=64 wal_level=replica wal_log_hints=off
track_commit_timestamp=off
```

Follow the recommendation and increase the value on the replica.

```
β=> ALTER SYSTEM SET max_connections = 150;

ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 16 beta restart
```

Check the replica:

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

```
β=> SELECT pg_is_in_recovery(), pg_get_wal_replay_pause_state();
```

pg_is_in_recovery		pg_get_wal_replay_pause_state
t		not paused
(1 row)		

Replication continues. To avoid replication stopping, limits should be increased on the replica first.

The state of the replication can be checked on the primary server: To access this information, a user must be a member of the pg_read_all_stats group role or a superuser.

```
student$ psql -d replica_physical
```

```
α=> \drg student|pg_monitor
```

List of role grants			
Role name	Member of	Options	Grantor
pg_monitor	pg_read_all_settings	INHERIT, SET	postgres
pg_monitor	pg_read_all_stats	INHERIT, SET	postgres
pg_monitor	pg_stat_scan_tables	INHERIT, SET	postgres
student	pg_checkpoint	INHERIT, SET	postgres
student	pg_create_subscription	INHERIT, SET	postgres
student	pg_monitor	INHERIT, SET	postgres
student	pg_read_server_files	INHERIT, SET	postgres
student	pg_write_server_files	INHERIT, SET	postgres
(8 rows)			

The pg_monitor role is a member of the pg_read_all_stats role and inherits its privileges.

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

-[RECORD 1]-----	
pid	75323
usesysid	10
username	postgres
application_name	16/beta
client_addr	
client_hostname	
client_port	-1
backend_start	2026-09-16 23:58:27.895802+03
backend_xmin	
state	streaming
sent_lsn	0/3437440
write_lsn	0/3437440
flush_lsn	0/3437440
replay_lsn	0/3437440
write_lag	
flush_lag	
replay_lag	
sync_priority	0
sync_state	async
reply_time	2026-09-16 23:58:30.809823+03

Note the *_lsn and *_lag fields – they show the replica's lag at different stages. All positions are currently in sync, with zero lag.

Now let's insert a large number of rows into the table to observe replication in action.

```
α=> INSERT INTO test SELECT 'Just a line' FROM generate_series(1,1_000_000);
```

```
INSERT 0 1000000
```

```
α=> SELECT *, pg_current_wal_lsn() from pg_stat_replication \gx
```

```

-[ RECORD 1 ]-----+-----
pid          | 75323
usesysid     | 10
username     | postgres
application_name | 16/beta
client_addr  |
client_hostname |
client_port  | -1
backend_start | 2026-09-16 23:58:27.895802+03
backend_xmin  |
state        | streaming
sent_lsn     | 0/7640000
write_lsn    | 0/75E0000
flush_lsn    | 0/75E0000
replay_lsn   | 0/75806F8
write_lag    | 00:00:00.098318
flush_lag    | 00:00:00.098318
replay_lag   | 00:00:00.196214
sync_priority | 0
sync_state   | async
reply_time   | 2026-09-16 23:58:35.839864+03
pg_current_wal_lsn | 0/79151A8

```

A small lag has appeared.

Check the replica:

```

β=> SELECT count(*) FROM test;

 count
-----
1000001
(1 row)

```

All rows have been delivered successfully.

Check the replication status one more time:

```
α=> SELECT *, pg_current_wal_lsn() from pg_stat_replication \gx
```

```

-[ RECORD 1 ]-----+-----
pid          | 75323
usesysid     | 10
username     | postgres
application_name | 16/beta
client_addr  |
client_hostname |
client_port  | -1
backend_start | 2026-09-16 23:58:27.895802+03
backend_xmin  |
state        | streaming
sent_lsn     | 0/79171B8
write_lsn    | 0/79171B8
flush_lsn    | 0/79171B8
replay_lsn   | 0/79171B8
write_lag    | 00:00:00.000609
flush_lag    | 00:00:00.008241
replay_lag   | 00:00:00.009082
sync_priority | 0
sync_state   | async
reply_time   | 2026-09-16 23:58:36.077635+03
pg_current_wal_lsn | 0/79171B8

```

All positions are now in sync.

Replication Modes

Asynchronous

`synchronous_commit` = off
local

Synchronous

commit with callback from the synchronous replica
provides reliability but not consistency

`synchronous_commit` = remote_write
= on
remote_apply
`synchronous_standby_names` = replica list
FIRST N (replica list)
ANY N (replica list)

16

There are two transaction commit modes. The faster asynchronous mode carries a risk of losing already committed changes if a crash occurs. In synchronous mode, the COMMIT command doesn't complete until the record reaches disk.

If you have multiple servers, you can still improve reliability by waiting for the replica to confirm that it has committed a record. In this case, the data will not be lost even if the disk fails.

Synchronous replication is enabled using the same `synchronous_commit` parameter that controls synchronous commits. Different values of this parameter let you control the level of reliability. With `remote_write`, the leader waits for confirmation that the replica has received the record (though there's still a chance of losing changes if the replica crashes before writing the data to disk).

With `on`, the leader waits for confirmation that the record has been written to the replica's disk (this is the reliable mode; however, an application querying the replica might not yet see the changes).

Finally, the `remote_apply` value makes the primary wait until the replica has actually applied the record. Each next mode introduces progressively longer delays.

The leader can synchronize with one or multiple replicas. You can specify a list of replicas in priority order or set up quorum-based synchronization, where the leader waits for confirmation from any N of the available replicas.

The leader might delete a WAL file that the replica still needs
if the replica didn't collect it yet
(i.e. because it was down for a while)

Solution 1. Replication slot

makes the leader depend on the replica, requires monitoring
`max_slot_wal_keep_size = -1`

Solution 2. WAL archive

no slot required

Several challenges can arise during replication, each with different possible solutions.

PostgreSQL periodically removes WAL files that are no longer needed for recovery. The leader might delete a file containing data that hasn't yet been transmitted to the replica.

This isn't an issue when a write-ahead log archive is available, since the replica can simply read the required data from the archive if streaming replication fails, then switch back to receiving data from the stream once it catches up with the leader.

Without the archive, the replica won't be able to continue recovery and will need to be reconstructed from a backup.

To prevent this, you can use a replication slot. However, this creates a dependency. If the replica stops collecting WAL records, they will accumulate on the leader, potentially flooding the file system. To prevent this, you can limit the accumulated volume by setting the `max_slot_wal_keep_size`. If this limit is exceeded, the slot will stop, and the server will be able to delete the WAL files again.

In any case, slots are yet another item you will have to monitor.

Impact of the Replication Slot

Stop the replica.

```
student$ sudo pg_ctlcluster 16 beta stop
```

Limit the WAL size to two segments.

```
α=> ALTER SYSTEM SET min_wal_size = '32MB';
```

```
ALTER SYSTEM
```

```
α=> ALTER SYSTEM SET max_wal_size = '32MB';
```

```
ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 16 alpha reload
```

The slot is inactive, but it retains the LSN of the last record received by the replica:

```
α=> SELECT active, restart_lsn, wal_status FROM pg_replication_slots \gx
```

```
-[ RECORD 1 ]-----  
active      | f  
restart_lsn | 0/79171B8  
wal_status  | reserved
```

Let's insert more rows into the table.

```
α=> INSERT INTO test SELECT 'Just a line' FROM generate_series(1,1_000_000);
```

```
INSERT 0 1000000
```

The LSN in the slot has not changed:

```
α=> SELECT active, restart_lsn, wal_status FROM pg_replication_slots \gx
```

```
-[ RECORD 1 ]-----  
active      | f  
restart_lsn | 0/79171B8  
wal_status  | extended
```

Create a checkpoint, which should clean up the WAL since its size exceeds the allowed limit.

```
α=> CHECKPOINT;
```

```
CHECKPOINT
```

What is the WAL size now?

```
α=> SELECT pg_size_pretty(sum(size)) FROM pg_ls_waldir();
```

```
pg_size_pretty  
-----  
96 MB  
(1 row)
```

The checkpoint did not remove the segments – they are being retained by the slot, even though the size limit has been exceeded. If a slot is left unattended, it can exhaust all available disk space.

In most cases, keeping the main server running smoothly is more important than keeping the replica in sync. To prevent WAL from growing out of control, you can limit the amount retained by the slot:

```
α=> ALTER SYSTEM SET max_slot_wal_keep_size = '16MB';
```

```
ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 16 alpha reload
```

```
α=> CHECKPOINT;
```

```
CHECKPOINT
```

Now the checkpoint cleans up the WAL:

```
α=> SELECT pg_size_pretty(sum(size)) FROM pg_ls_waldir();
```

```
pg_size_pretty
-----
32 MB
(1 row)
```

But the slot no longer guarantees the availability of WAL records (wal_status=lost):

```
α=> SELECT active, restart_lsn, wal_status FROM pg_replication_slots \gx
```

```
-[ RECORD 1 ]-----
active       | f
restart_lsn  |
wal_status   | lost
```

And the replica cannot synchronize:

```
student$ sudo pg_ctlcluster 16 beta start
```

```
student$ sudo tail -n 2 /var/log/postgresql/postgresql-16-beta.log
```

```
2026-09-16 23:58:46.203 MSK [76622] FATAL:  could not receive data from WAL stream:
ERROR:  requested WAL segment 00000001000000000000000007 has already been removed
2026-09-16 23:58:46.204 MSK [76619] LOG:  waiting for WAL to become available at 0/7002000
```

In such cases, you can synchronize the replica using the archive by setting the `restore_command` parameter. If that's not an option, you'll need to reconfigure replication from scratch by creating a new base backup.

WAL records incompatible with queries on the replica

- vacuum removes tuples that are still needed by queries on the replica
- exclusive locks on objects
- buffer pinning

Monitoring

- `pg_stat_database_conflicts`

Another problem occurs when when the replica is used to run queries. The leader can perform an action that conflicts with a query running on the replica:

- Vacuum may remove a tuple that's being queried, or it may need to acquire a lock that conflicts with the query.
- A WAL record may acquire an exclusive lock on an object (for example, when dropping a table).
- Buffer pinning may cause conflicts sometimes.

You can monitor such conflicts using the `pg_stat_database_conflicts` view.

Solution 0: Avoid conflicts

- avoid running commands that acquire locks
- disable file truncation at end of vacuuming: retention setting
`vacuum_truncate = off`

Solution 1: Delay applying conflicting records

- `max_standby_streaming_delay`
- `max_standby_archive_delay`
- except deadlocks, DROP TABLESPACE, DROP DATABASE

One way to handle conflicts is to avoid operations on the leader that would result in exclusive locks. For example, you can avoid running commands like DROP TABLE, TRUNCATE, LOCK, DROP INDEX, DROP TRIGGER, and similar operations, or run them only when you know there are no active queries on the replica.

Some implicit locks can be avoided by disabling the corresponding functionality on the leader. For instance, if vacuum frees up pages at the end of a file, shrinking the file size requires an exclusive lock on the table. Then, you can disable file truncation by setting the table storage parameter `vacuum_truncate = off`. However, vacuum cannot be disabled or postponed indefinitely, so this approach won't help avoid conflicts when vacuum removes tuples that are still needed by queries.

Another approach is to delay the application of conflicting records on the replica. This method works with both streaming replication and archive-based replication; it delays the application of records related to both vacuum operations and exclusive locks. You can set the maximum application delay (measured from when the record is received). If the conflicting query hasn't finished by that time, it will be terminated.

However, conflicts caused by rare causes (database or tablespace deletion, deadlocks) are not delayed and are resolved immediately by terminating the query.

Solution 2: feedback via the replication protocol

a snapshot on the replica will prevent tuple removal
in the same manner as

hot_standby_feedback = on

only for vacuum conflicts with queries on the replica

Monitoring

`pg_stat_replication.backend_xmin`

`pg_replication_slots.xmin`

Another approach is to defer conflicting operations on the leader. To enable this, enable replication protocol feedback.

This method only works for vacuum operations (exclusive locks cannot be deferred) and only with streaming replication (if the replica switches to retrieving records from the archive, it essentially becomes invisible to the leader).

Each data snapshot includes a transaction horizon – a transaction ID value that marks the boundary between tuples that are no longer needed and those that must be retained. The minimum horizon across all snapshots on the replica determines what can be vacuumed. This topic is covered in detail in the "Data Snapshots" section of the DBA2 course.

With feedback enabled, the leader knows which tuples the replica needs. In this case, from the vacuum perspective, a query running on the replica is treated exactly the same as a query running on the leader itself.

Feedback allows you to track the vacuum horizon on the leader side. When a replication slot is not in use, the horizon can be monitored through the `pg_stat_replication.backend_xmin` field, while with a slot it appears in `pg_replication_slots.xmin`.

Keep in mind that when feedback is enabled, even an inactive slot will prevent vacuum from proceeding, which can cause bloating. The practice assignments explore this effect.

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

- primary mode: streaming replication
- can be supplemented with WAL archiving

The replica is an exact copy of the leader

- does not generate its own WAL records and operates in read-only mode

A complex mechanism that requires proper configuration and monitoring

- configurations vary significantly depending on your specific requirements

1. Configure streaming replication between two servers in synchronous mode, with a slot, without feedback.
2. Verify that replication works as intended. Make sure that when the replica is stopped, commits on the leader are not completed.
3. Simulate a scenario where a query running on the replica gets terminated because the leader has vacuumed tuples that the query still needs.
4. Disable deferral of conflicting changes and verify that queries are canceled immediately.
5. Enable feedback and verify that vacuum on the leader doesn't interrupt query execution.
6. Stop the replica and verify that the replication slot prevents vacuum from removing tuples. Drop the slot to allow vacuum to resume.

1. For synchronous mode, set on the leader:

- `synchronous_commit = on`
- `synchronous_standby_names = "'16/beta'"`

3. Start a transaction with Repeatable Read isolation level on the replica and periodically query the table. Meanwhile, on the leader, modify that table and run vacuum.

Keep in mind that the `max_standby_streaming_delay` parameter defaults to 30 seconds.

4. Set the `max_standby_streaming_delay` parameter to zero.

5. Enable the `hot_standby_feedback` parameter on the replica. The maximum notification interval is controlled by the `wal_receiver_status_interval` parameter (defaults to 10 seconds).

6. After completing the tasks, delete the database you created and reset the configuration.

Remember to disable the synchronous commit mode, otherwise a stopped replica will prevent transactions from committing.

Don't set `synchronous_commit` to off. Instead, set `synchronous_commit = 'local'` or reset the `synchronous_standby_names` parameter.

1. Synchronous Physical Streaming Replication

```
α=> CREATE DATABASE replica_physical;
```

CREATE DATABASE

Create and start the replica the same way as in the demonstration.

```
α=> \c replica_physical
```

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

Backup:

```
student$ sudo pg_ctlcluster 16 beta stop
```

Cluster is not running.

```
postgres$ rm -rf /var/lib/postgresql/16/beta
```

```
postgres$ pg_basebackup \
--pgdata /var/lib/postgresql/16/beta \
--checkpoint=fast \
-R --create-slot --slot=replica
```

```
student$ sudo pg_ctlcluster 16 beta start
```

For the main server to find the replica, the cluster_name parameter has already been set in the main configuration:

```
student$ psql -p 5433 -c 'SHOW cluster_name;'
```

```
cluster_name
-----
16/beta
(1 row)
```

Another option is to add the application_name attribute to the connection string:

```
primary_conninfo='user=student port=5432 application_name=16/beta'
```

Now enable synchronous replication mode on the primary.

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

ALTER SYSTEM

```
α=> ALTER SYSTEM SET synchronous_standby_names = '"16/beta"';
```

ALTER SYSTEM

If all replicas are synchronous, you can set a value that matches any name:

```
synchronous_standby_names = '*'
```

However, the primary cannot distinguish a replica from, say, the pg_receivewal utility. With this configuration, it will attempt to synchronize with any client connected via the replication protocol, which can result in commits waiting indefinitely. It's better to list names explicitly.

```
student$ sudo pg_ctlcluster 16 alpha reload
```

2. Physical Replication Verification

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

```

-[ RECORD 1 ]-----+-----
pid          | 98849
usesysid     | 10
username     | postgres
application_name | 16/beta
client_addr  |
client_hostname |
client_port   | -1
backend_start | 2026-09-17 00:11:28.62954+03
backend_xmin  |
state        | streaming
sent_lsn     | 0/3000C50
write_lsn    | 0/3000C50
flush_lsn    | 0/3000C50
replay_lsn   | 0/3000C50
write_lag    |
flush_lag    |
replay_lag   |
sync_priority | 1
sync_state   | sync
reply_time   | 2026-09-17 00:11:31.677795+03

```

- sync_state: sync means that replication is running in the synchronous mode:
- application_name: 16/beta – the name by which the primary finds the replica.

Stop the replica.

```
student$ sudo pg_ctlcluster 16 beta stop
```

```
α=> BEGIN;
```

```
BEGIN
```

```
α=> CREATE TABLE test(id integer);
```

```
CREATE TABLE
```

```
α=> COMMIT;
```

The commit waits for the synchronous replica...

```
student$ sudo pg_ctlcluster 16 beta start
```

After the replica starts, the commit completes.

```
COMMIT
```

3. Query Cancellation Due to Vacuum on the Primary

```
α=> INSERT INTO test VALUES (1);
```

```
INSERT 0 1
```

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

When applying a WAL record on the replica is delayed beyond deadlock_timeout due to a conflict with a running query, we want a message to be written to the server log.

```

β=> \dconfig deadlock_timeout

List of configuration parameters
Parameter      | Value
-----+-----
deadlock_timeout | 1s
(1 row)

```

To do this, set the log_recovery_conflict_waits parameter.

```
β=> ALTER SYSTEM SET log_recovery_conflict_waits = on;
```

```
ALTER SYSTEM
```

Also reduce the maximum time streaming replication can be paused before a query on the replica is canceled.

```
β=> ALTER SYSTEM SET max_standby_streaming_delay = '9s';
```

```
ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 16 beta reload
```

Start a transaction with the Repeatable Read isolation level: the first statement will create a data snapshot that all statements in the transaction will use.

```

β=> BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;

BEGIN

β=> SELECT * FROM test;

   id
----
   1
(1 row)

```

On the primary, update the only row and run vacuum. The first tuple will be removed in the process.

```

α=> UPDATE test SET id = 2;

UPDATE 1

α=> VACUUM test;

VACUUM

```

After 5 seconds, the query on the replica still completes – the replica is delaying the application of the conflicting WAL record:

```

β=> SELECT * FROM test;

   id
----
   1
(1 row)

```

And the server log already contains a message about waiting for the conflict to be resolved:

```

postgres$ tail -3 /var/log/postgresql/postgresql-16-beta.log

2026-09-17 00:11:39.371 MSK [99295] LOG:  recovery still waiting after 1024.875 ms:
recovery conflict on snapshot
2026-09-17 00:11:39.371 MSK [99295] DETAIL:  Conflicting process: 99381.
2026-09-17 00:11:39.371 MSK [99295] CONTEXT:  WAL redo at 0/301B8A0 for Heap2/PRUNE:
snapshotConflictHorizon: 738, nredirected: 1, ndead: 0, nunused: 0, redirected: [1->2],
dead: [], unused: []; blkref #0: rel 1663/16409/16410, blk 0

```

Five seconds later, the same query is aborted – the tuple from the snapshot no longer exists.

```

β=> SELECT * FROM test;

FATAL:  terminating connection due to conflict with recovery
DETAIL:  User query might have needed to see row versions that must be removed.
HINT:  In a moment you should be able to reconnect to the database and repeat your
command.
server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
connection to server was lost

```

4. Disable the Delay of Conflicting WAL Record Application

Set the delay for applying conflicting changes to zero.

```

student$ psql -p 5433 -d replica_physical

β=> ALTER SYSTEM SET max_standby_streaming_delay = 0;

ALTER SYSTEM

student$ sudo pg_ctlcluster 16 beta reload

```

Start a new transaction...

```

β=> BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;

BEGIN

β=> SELECT * FROM test;

   id
----
   2
(1 row)

```

On the primary, update the row and run vacuum...

```
α=> UPDATE test SET id = 3;
```

```
UPDATE 1
```

```
α=> VACUUM test;
```

```
VACUUM
```

The query on the replica is canceled immediately:

```
| β=> SELECT * FROM test;
```

```
| FATAL: terminating connection due to conflict with recovery
| DETAIL: User query might have needed to see row versions that must be removed.
| HINT: In a moment you should be able to reconnect to the database and repeat your
| command.
| server closed the connection unexpectedly
|       This probably means the server terminated abnormally
|       before or while processing the request.
| connection to server was lost
```

5. Feedback

Set up feedback and, to avoid long waits, a short notification interval.

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

```
| β=> ALTER SYSTEM SET hot_standby_feedback = on;
```

```
| ALTER SYSTEM
```

```
| β=> ALTER SYSTEM SET wal_receiver_status_interval = '1s';
```

```
| ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 16 beta reload
```

Start a new transaction...

```
| β=> BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

```
| BEGIN
```

```
| β=> SELECT * FROM test;
```

```
| id
| ----
| 3
| (1 row)
```

On the primary, update the row and run vacuum...

```
α=> UPDATE test SET id = 4;
```

```
UPDATE 1
```

```
α=> VACUUM VERBOSE test;
```

```
INFO: vacuuming "replica_physical.public.test"
INFO: finished vacuuming "replica_physical.public.test": index scans: 0
pages: 0 removed, 1 remain, 1 scanned (100.00% of total)
tuples: 0 removed, 2 remain, 1 are dead but not yet removable
removable cutoff: 740, which was 1 XIDs old when operation ended
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers
removed
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 16 hits, 0 misses, 0 dirtied
WAL usage: 1 records, 0 full page images, 188 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

Thanks to feedback, vacuum cannot remove the old tuple (tuples: 0 removed, 2 remain, 1 are dead but not yet removable).

```
| β=> SELECT * FROM test;
```

```
| id
| ----
| 3
| (1 row)
```

```
| β=> COMMIT;
```

```
| COMMIT
```

```
α=> VACUUM VERBOSE test;
```

```
INFO: vacuuming "replica_physical.public.test"
INFO: finished vacuuming "replica_physical.public.test": index scans: 0
pages: 0 removed, 1 remain, 1 scanned (100.00% of total)
tuples: 1 removed, 1 remain, 0 are dead but not yet removable
removable cutoff: 741, which was 0 XIDs old when operation ended
new relfrozenxid: 740, which is 1 XIDs ahead of previous value
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers
removed
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 17 hits, 0 misses, 0 dirtied
WAL usage: 3 records, 0 full page images, 308 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

Once the transaction on the replica completes, the primary can run vacuum (tuples: 1 removed, 1 remain, 0 are dead but not yet removable). The WAL records are replicated, but they won't cause any conflicts.

6. Effect of a Replication Slot on Vacuum When the Replica Is Stopped

Cancel synchronization and stop the replica:

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

```
ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 16 alpha reload
```

```
student$ sudo pg_ctlcluster 16 beta stop
```

Update the table row once more:

```
α=> UPDATE test SET id = 5;
```

```
UPDATE 1
```

The slot is inactive, but it retains the minimum xmin of the replica's data snapshots:

```
α=> SELECT active, xmin FROM pg_replication_slots;
```

```
 active | xmin
-----+-----
f       | 741
(1 row)
```

This xmin value is less than the minimum xmin of local data snapshots:

```
α=> SELECT min(backend_xmin::text::numeric) FROM pg_stat_activity;
```

```
 min
-----
742
(1 row)
```

Therefore, the slot will hold back vacuum (tuples: 0 removed, 2 remain, 1 are dead but not yet removable).

```
α=> VACUUM VERBOSE test;
```

```
INFO: vacuuming "replica_physical.public.test"
INFO: finished vacuuming "replica_physical.public.test": index scans: 0
pages: 0 removed, 1 remain, 1 scanned (100.00% of total)
tuples: 0 removed, 2 remain, 1 are dead but not yet removable
removable cutoff: 741, which was 1 XIDs old when operation ended
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers
removed
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 16 hits, 0 misses, 0 dirtied
WAL usage: 1 records, 0 full page images, 188 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

Delete the slot.

```
α=> SELECT pg_drop_replication_slot('replica');
```

```
pg_drop_replication_slot
```

```
(1 row)
```

Now vacuum will run successfully (tuples: 1 removed).

```
α=> VACUUM VERBOSE test;
```

```
INFO:  vacuuming "replica_physical.public.test"
INFO:  finished vacuuming "replica_physical.public.test": index scans: 0
pages: 0 removed, 1 remain, 1 scanned (100.00% of total)
tuples: 1 removed, 1 remain, 0 are dead but not yet removable
removable cutoff: 742, which was 0 XIDs old when operation ended
new relfrozenxid: 741, which is 1 XIDs ahead of previous value
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers
removed
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 17 hits, 0 misses, 0 dirtied
WAL usage: 3 records, 0 full page images, 308 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

Cleanup

Drop the database and reset configuration.

```
α=> \c postgres
```

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

```
α=> DROP DATABASE replica_physical;
```

```
DROP DATABASE
```

```
α=> ALTER SYSTEM RESET synchronous_commit;
```

```
ALTER SYSTEM
```

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

```
ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 16 alpha restart
```

1. Configure streaming physical replication together with WAL archiving, using the virtual machine's network interface for the replication stream.
2. Verify that streaming replication is working through the interface.
3. Simulate a network failure by disabling the virtual machine's network interface.
4. Ensure that streaming replication has stopped and the replica is receiving changes from the WAL archive, but with lag.
5. Restart the virtual machine's network interface, verifying that streaming replication has automatically recovered.

1. You can check the status of network interfaces with the command:

```
ip a
```

You can check the availability of the network address with the command:

```
ping -c1 <IP address>
```

Edit the `pg_hba.conf` file to allow connections to the alpha server through any network interfaces, and configure it to listen on all network interfaces.

The `wal_receiver` process ends if there is no network connection within the `wal_receiver_timeout` time. Specify a small value.

3. Simulate network failure by turning off the network interface:

```
sudo ip l set enp0s3 down
```

4. Verify that after switching to the next WAL segment, the replica receives the changes made before switching.

5. You can switch the network interface back on with the command:

```
sudo ip l set enp0s3 up
```

After completing the exercises, drop the database you created and reset the configuration parameters to their defaults.

1. Physical Streaming Replication and WAL Archive

The primary virtual machine network interface is enp0s3:

```
student$ ip a
```

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host noprefixroute
        valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 08:00:27:4f:a4:65 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic noprefixroute enp0s3
        valid_lft 78573sec preferred_lft 78573sec
    inet6 fd17:625c:f037:2:a7c5:796c:1d04/64 scope global temporary dynamic
        valid_lft 86031sec preferred_lft 14031sec
    inet6 fd17:625c:f037:2:a00:27ff:fe4f:a465/64 scope global dynamic mngtmpaddr
        valid_lft 86031sec preferred_lft 14031sec
    inet6 fe80::a00:27ff:fe4f:a465/64 scope link
        valid_lft forever preferred_lft forever
```

Verify that the interface is accessible.

```
student$ ping -c1 10.0.2.15
```

```
PING 10.0.2.15 (10.0.2.15) 56(84) bytes of data.
64 bytes from 10.0.2.15: icmp_seq=1 ttl=64 time=0.031 ms
```

```
--- 10.0.2.15 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.031/0.031/0.031/0.000 ms
```

Allow connections to the alpha server from IPv4 addresses belonging to the server's interface networks.

```
postgres$ sed -i.bak 's/127\.0\.\.1/32/samenet/' /etc/postgresql/16/alpha/pg_hba.conf
```

Enable the server to listen on all interfaces and configure WAL archiving.

```
α=> ALTER SYSTEM SET listen_addresses = '*';
```

```
ALTER SYSTEM
```

```
α=> ALTER SYSTEM SET archive_mode = on;
```

```
ALTER SYSTEM
```

```
α=> ALTER SYSTEM SET archive_command =
'test ! -f /var/lib/postgresql/archive/%f && cp %p /var/lib/postgresql/archive/%f';
```

```
ALTER SYSTEM
```

Directory for the WAL archive.

```
postgres$ mkdir /var/lib/postgresql/archive
```

```
student$ sudo pg_ctlcluster 16 alpha restart
```

Backup:

```
student$ sudo pg_ctlcluster 16 beta stop
```

Cluster is not running.

```
postgres$ rm -rf /var/lib/postgresql/16/beta
```

```
postgres$ pg_basebackup --pgdata=/var/lib/postgresql/16/beta --checkpoint=fast
```

Prepare the configuration for the replica. The wal_receiver_timeout parameter defines how long the replica waits before terminating an idle connection. This allows the replica to detect a dropped network connection more quickly.

```
postgres$ cat << EOF | tee /etc/postgresql/16/beta/conf.d/rest.conf
restore_command = 'cp /var/lib/postgresql/archive/%f %p'
primary_conninfo = 'user=postgres password=postgres host=10.0.2.15 port=5432'
wal_receiver_timeout = '2s'
EOF
```

```
restore_command = 'cp /var/lib/postgresql/archive/%f %p'
```

```
primary_conninfo = 'user=postgres password=postgres host=10.0.2.15 port=5432'
wal_receiver_timeout = '2s'
```

```
postgres$ touch /var/lib/postgresql/16/beta/standby.signal
```

```
student$ sudo pg_ctlcluster 16 beta start
```

2. Replication Check

```
student$ psql
```

```
α=> CREATE DATABASE replica_physical;
```

```
CREATE DATABASE
```

```
α=> \c replica_physical
```

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

```
α=> CREATE TABLE test(s text);
```

```
CREATE TABLE
```

```
α=> INSERT INTO test VALUES ('Hello, world!');
```

```
INSERT 0 1
```

The replica receives WAL records via the stream...

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

```
β=> SELECT pid, status, written_lsn, flushed_lsn, latest_end_lsn, sender_host
FROM pg_stat_wal_receiver \gx
```

```
-[ RECORD 1 ]--+-+-----
pid           | 101744
status        | streaming
written_lsn   | 0/3486380
flushed_lsn   | 0/3486380
latest_end_lsn | 0/3486380
sender_host   | 10.0.2.15
```

...and reports received records to the primary without delay:

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

```
-[ RECORD 1 ]-----+-----
pid           | 101745
usesysid      | 10
username      | postgres
application_name | 16/beta
client_addr   | 10.0.2.15
client_hostname |
client_port   | 55376
backend_start | 2026-09-17 00:12:33.618001+03
backend_xmin   |
state         | streaming
sent_lsn      | 0/3486380
write_lsn     | 0/3486380
flush_lsn     | 0/3486380
replay_lsn    | 0/3486380
write_lag     |
flush_lag     |
replay_lag    |
sync_priority  | 0
sync_state    | async
reply_time    | 2026-09-17 00:12:40.460627+03
```

```
β=> \c replica_physical
```

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

The table has been replicated:

```
β=> SELECT * FROM test;
```

```
      s
-----
Hello, world!
(1 row)
```

3. Network Failure

Disable the network interface to simulate a network failure.

```
student$ sudo ip l set enp0s3 down
```

The network is unavailable.

```
student$ ping -c1 10.0.2.15
```

ping: connect: Network is unreachable

The wal receiver process has stopped:

```
| β=> SELECT * FROM pg_stat_wal_receiver \gx
|
| (0 rows)
```

On the primary, insert a row...

```
α=> INSERT INTO test VALUES ('Network failure.');
```

```
INSERT 0 1
```

...but the replica doesn't receive it:

```
| β=> SELECT * FROM test;
|
|          s
|-----
| Hello, world!
| (1 row)
```

Streaming replication has stopped:

```
α=> SELECT sent_lsn, write_lsn FROM pg_stat_replication \gx
```

```
-[ RECORD 1 ]-----
sent_lsn   | 0/34863F0
write_lsn  | 0/3486380
```

4. Archiving

Archiving on the primary, however, continues to function. Switch the WAL segment:

```
α=> SELECT pg_walfile_name(pg_current_wal_lsn()), pg_switch_wal();
```

```
      pg_walfile_name      | pg_switch_wal
-----+-----
000000010000000000000003 | 0/3486458
(1 row)
```

```
α=> INSERT INTO test VALUES ('WAL segment switched.');
```

```
INSERT 0 1
```

The segment has been archived:

```
α=> SELECT * FROM pg_stat_archiver \gx
```

```
-[ RECORD 1 ]-----+-----
archived_count      | 4
last_archived_wal   | 000000010000000000000003
last_archived_time  | 2026-09-17 00:12:50.518266+03
failed_count        | 0
last_failed_wal     |
last_failed_time    |
stats_reset         | 2026-07-15 23:46:12.428787+03
```

Now the replica can apply the records from the new segment, and we will see the row that was inserted before the switch.

```
| β=> SELECT * FROM test;
```

```

      s
-----
Hello, world!
Network failure.
(2 rows)

```

5. Network Recovery

Let's insert another row into the table on the primary.

```
α=> INSERT INTO test VALUES ('Network still down...');
```

```
INSERT 0 1
```

It doesn't reach the replica.

```
β=> SELECT * FROM test;
```

```

      s
-----
Hello, world!
Network failure.
(2 rows)

```

Now restore the network interface: the outage has been resolved.

```
student$ sudo ip l set enp0s3 up
```

```
student$ ping -c1 10.0.2.15
```

```

PING 10.0.2.15 (10.0.2.15) 56(84) bytes of data.
64 bytes from 10.0.2.15: icmp_seq=1 ttl=64 time=0.018 ms

```

```

--- 10.0.2.15 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.018/0.018/0.018/0.000 ms

```

Streaming replication is working again:

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

```

-[ RECORD 1 ]-----+-----
pid          | 103734
usesysid     | 10
username     | postgres
application_name | 16/beta
client_addr  | 10.0.2.15
client_hostname |
client_port  | 49664
backend_start | 2026-09-17 00:12:59.483625+03
backend_xmin  |
state        | streaming
sent_lsn     | 0/4000150
write_lsn    | 0/4000150
flush_lsn    | 0/4000150
replay_lsn   | 0/4000150
write_lag    | 00:00:00.095116
flush_lag    | 00:00:00.102971
replay_lag   | 00:00:00.103039
sync_priority | 0
sync_state   | async
reply_time   | 2026-09-17 00:12:59.616749+03

```

On the primary, add another row.

```
α=> INSERT INTO test VALUES ('Network restored!');
```

```
INSERT 0 1
```

The replica has received all the rows.

```
β=> SELECT * FROM test;
```

```
      5
-----
Hello, world!
Network failure.
WAL segment switched.
Network still down...
Network restored!
(5 rows)
```

Cleanup

Drop the database we created and reset configuration.

```
α=> \c postgres
```

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

```
α=> DROP DATABASE replica_physical;
```

DROP DATABASE

```
| β=> \c postgres
```

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

```
| β=> ALTER SYSTEM RESET listen_addresses;
```

```
| ALTER SYSTEM
```

```
| β=> ALTER SYSTEM RESET archive_mode;
```

```
| ALTER SYSTEM
```

```
| β=> ALTER SYSTEM RESET archive_command;
```

```
| ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 16 beta stop -m immediate --skip-systemctl-redirect
```

```
α=> ALTER SYSTEM RESET listen_addresses;
```

ALTER SYSTEM

```
α=> ALTER SYSTEM RESET archive_mode;
```

ALTER SYSTEM

```
α=> ALTER SYSTEM RESET archive_command;
```

ALTER SYSTEM

```
student$ sudo pg_ctlcluster 16 alpha restart
```

```
postgres$ rm -f /etc/postgresql/16/beta/conf.d/rest.conf
```