

Backup WAL Archive



16

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.

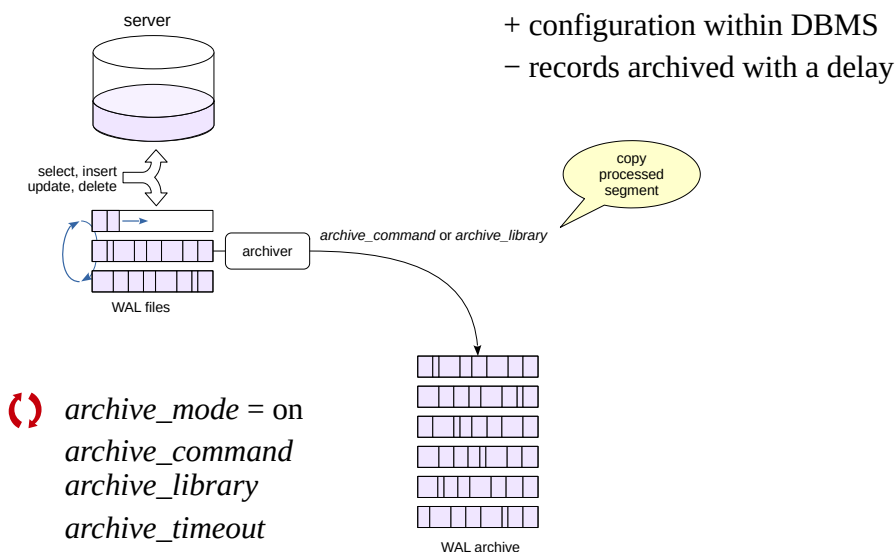
File-Based Archiving – Continuous Archiving

Stream Archiving – pg_receivewal

Recovery from Archive

Archive Cleanup

Continuous Archiving



3

The base backup discussed in the previous topic contains all the WAL records needed to restore data consistency. The resulting backup is called **standalone** because it contains all the data necessary for recovery up to the point in time when the backup was created.

If you add new WAL files generated by the server to the standalone backup, you can restore the cluster to an arbitrary point in time. In practice, the WAL segments are not copied to the backup file itself, but to a separate **archive**. This is done by the archiver process (if `archive_mode = on`).

There are two ways to archive a WAL segment:

1. Use an OS command. When switching to the next WAL segment, the process executes the command stored in the `archive_command` parameter.
2. Use the archiving module functions (PostgreSQL 15 and up). The module name is set in the `archive_library` parameter.
<https://postgrespro.com/docs/postgresql/16/archive-modules>

If the server load is light, the `archive_timeout` parameter is commonly used to set the maximum time that can pass before switching to a new WAL segment. Switching can also be done manually using the `pg_switch_wal` function.

<https://postgrespro.com/docs/postgresql/16/continuous-archiving>

An operating system command that archives completed WAL segments

- copies file%p to archive named%f,
- the archive structure can be organized in any way you choose.
- must return 0 only if successful
- must not overwrite files that already exist in the archive
- must ensure data is written to persistent storage
- conveniently supports custom scripts

Monitoring

- pg_stat_archiver view shows current status
- can enable the message collector (*logging_collector* = on) and write messages to the server log

4

The archive command must copy the specified file to an archive. The archive can be organized in any way you choose. For example, this could be a file system on a separate server.

The command must return a zero exit status only when it succeeds. In this case, the copied segment can be deleted by the checkpoint process. With a non-zero status, the segment that failed to copy and subsequent segments are not deleted, and the server periodically retries the archive command until it receives 0.

The command should never overwrite existing files, as this likely indicates an error, and overwriting a file could invalidate the archive.

To ensure reliability, the command must guarantee that the file is written to persistent storage; otherwise, if the archiving command fails, you could lose a file segment if the PostgreSQL server deletes it in the meantime.

The current archiving status is shown in the pg_stat_archiver view. The message collector process *logging_collector* can conveniently write error and other messages from *archive_command* execution to the server log.

If complex logic is required, it can be scripted in a separate file, and the archiving command can be set to run it.

<https://postgrespro.com/docs/postgresql/16/runtime-config-wal#GUC-ARCHIVE-COMMAND>

Setting up continuous archiving

We will store the archive in the `/var/lib/postgresql/archive` directory. It must be accessible to the PostgreSQL owner user.

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

In practice, you can keep the archive on a separate server or disk system accessible over the network.

Let's enable archiving mode and set the command for copying completed WAL segments.

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

```
ALTER SYSTEM
```

In `archive_command`, we first check whether a file with the specified name already exists in the archive and only copy it if it's not there.

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

```
ALTER SYSTEM
```

You can specify any command in `archive_command`, as long as it returns 0 only when successful. For example, you can set up compression of archived segments:

```
SET archive_command =  
'test ! -f /var/lib/postgresql/archive/%f && gzip <%p >/var/lib/postgresql/archive/%f';
```

Ideally, the archiving command should perform a `sync` to ensure the file is safely written to persistent storage. Otherwise, in case of a failure, the file might be lost from the archive while the server has already deleted it from the `pg_wal` directory.

It's usually convenient to put all the necessary archiving logic in a separate script and call the entire script:

```
SET archive_command = 'archive.sh "%f" "%p"';
```

Changing `archive_mode` requires a server restart.

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

```
student$ psql
```

The archiver process now exists:

```
α=> SELECT pid, backend_type, backend_start  
FROM pg_stat_activity  
WHERE backend_type='archiver';
```

pid	backend_type	backend_start
71779	archiver	2026-09-16 23:56:16.851718+03

(1 row)

Let's verify that archiving is working. Create a database and a table.

```
α=> CREATE DATABASE backup_archive;
```

```
CREATE DATABASE
```

```
α=> \c backup_archive
```

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

```
α=> CREATE TABLE t(s text);
```

```
CREATE TABLE
```

```
α=> INSERT INTO t VALUES ('Row #1');
```

```
INSERT 0 1
```

Here's the WAL segment currently in use:

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

pg_walfile_name
000000010000000000000001

(1 row)

To fill out the file, we would need to perform a large number of operations. For testing purposes, it's easier to force a segment switch:

```
α=> SELECT pg_switch_wal();
```

```
pg_switch_wal
-----
0/1FB6108
(1 row)
```

```
α=> INSERT INTO t VALUES ('Another row');
```

```
INSERT 0 1
```

The segment has switched:

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

```
pg_walfile_name
-----
000000010000000000000002
(1 row)
```

The previous one should have made it to the archive. Check:

```
student$ sudo ls -lh /var/lib/postgresql/archive
```

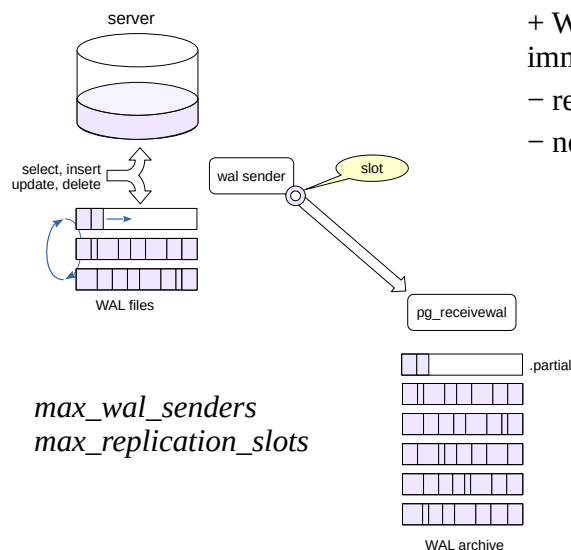
```
total 16M
-rw----- 1 postgres postgres 16M Sep 16 23:56 000000010000000000000001
```

The current archiving status is shown in the pg_stat_archiver view:

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

```
-[ RECORD 1 ]-----+-----
archived_count   | 1
last_archived_wal | 000000010000000000000001
last_archived_time | 2026-09-16 23:56:24.776632+03
failed_count     | 0
last_failed_wal  | 
last_failed_time | 
stats_reset      | 2026-07-15 23:46:12.428787+03
```

File-based archiving is now configured and working.



- + WAL records are archived immediately
- requires a separate tool
- need a replication slot

Alternatively, you can populate the archive using the replication protocol and the `pg_receivewal` utility.

Typically, the utility runs on a separate server that stores the archive. It generates files in the same way as the main server itself, and writes them to the specified directory. Incomplete segments have the suffix `.partial`. Cache synchronization with the file system occurs only when the WAL segment is closed, but you can make it synchronous with the `--synchronous` parameter.

The utility can compress WAL segments. Use the parameter `--compress` with the algorithm (`gzip` or `lz4`) and the compression level as arguments.

When running the utility for the first time (when the archive is empty), archiving starts from the beginning of the current WAL segment.

Otherwise, it starts from the beginning of the segment following the last completed segment in the archive.

When using a replication slot, this ensures that there are no gaps in the archive, even if the utility was disabled for a while.

By default, the utility receives records indefinitely, but you can set it to terminate when receiving a record with a specific LSN: `--endpos = lsn`.

Note that the utility does not start automatically as a service and cannot be a daemon.

<https://postgrespro.com/docs/postgresql/16/app-pgreceivewal>

Streaming Archive

To set up archive population via the streaming replication protocol, let's first stop file-based archiving.

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

```
ALTER SYSTEM
```

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

```
ALTER SYSTEM
```

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

The current state of the archive.

```
student$ sudo ls -lh /var/lib/postgresql/archive
```

```
total 32M
-rw----- 1 postgres postgres 16M Sep 16 23:56 00000001000000000000000001
-rw----- 1 postgres postgres 16M Sep 16 23:56 00000001000000000000000002
```

First, let's have the `pg_receivewal` utility create a slot to ensure all WAL records are received.

```
postgres$ /usr/lib/postgresql/16/bin/pg_receivewal --create-slot --slot=archive
```

Start it in the background in archiving mode. Upon detecting existing files in the archive, the utility will request the next segment from the server to ensure there are no gaps in the archive.

```
postgres$ /usr/lib/postgresql/16/bin/pg_receivewal -D /var/lib/postgresql/archive --slot=archive &
```

Let's add a ton of rows to the table and then delete them.

```
student$ psql -d backup_archive
```

```
α=> INSERT INTO t SELECT 'Hello again.' FROM generate_series(1,200_000);
```

```
INSERT 0 200000
```

```
α=> DELETE FROM t WHERE s = 'Hello again.';
```

```
DELETE 200000
```

```
α=> VACUUM t;
```

```
VACUUM
```

New files have appeared in the archive.

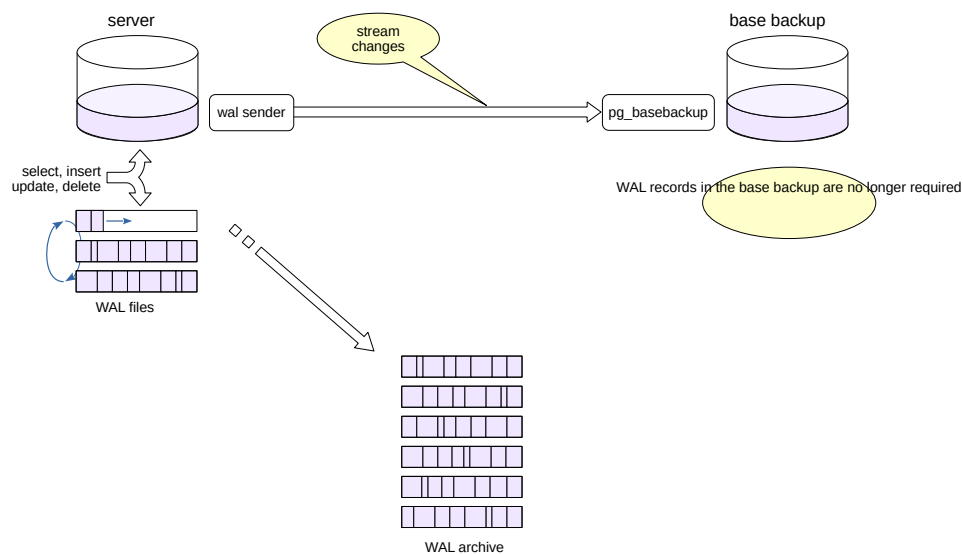
```
student$ sudo ls -lh /var/lib/postgresql/archive
```

```
total 64M
-rw----- 1 postgres postgres 16M Sep 16 23:56 00000001000000000000000001
-rw----- 1 postgres postgres 16M Sep 16 23:56 00000001000000000000000002
-rw----- 1 postgres postgres 16M Sep 16 23:56 00000001000000000000000003
-rw----- 1 postgres postgres 16M Sep 16 23:56 00000001000000000000000004.partial
```

The last file most likely has a `.partial` suffix – it is currently being written to.

Stream archiving is now configured and working.

Base backup



If we have a configured write-ahead log archive, then WAL files become optional in the base backup — since they can be retrieved from the archive during recovery.

Therefore, `pg_basebackup` can be run with `--wal-method=none`.

(But there is no harm in having WAL files in the base copy, except for the space they occupy. In fact, they enable system recovery if the archive becomes unavailable.)

Base Backup

Since the archive is populated automatically, let's instruct `pg_basebackup` not to include WAL files in the backup:

```
student$ pg_basebackup \
--checkpoint=fast \
--wal-method=none \
--pgdata /home/student/tmp/backup
```

NOTICE: WAL archiving is not enabled; you must ensure that all required WAL segments are copied through other means to complete the backup

There are no WAL segments in the backup's `pg_wal` directory:

```
student$ ls -l /home/student/tmp/backup/pg_wal/

total 4
drwx----- 2 student student 4096 Sep 16 23:56 archive_status
```

The archive, however, has gained new files:

```
student$ sudo ls -lh /var/lib/postgresql/archive

total 96M
-rw----- 1 postgres postgres 16M Sep 16 23:56 00000001000000000000000001
-rw----- 1 postgres postgres 16M Sep 16 23:56 00000001000000000000000002
-rw----- 1 postgres postgres 16M Sep 16 23:56 00000001000000000000000003
-rw----- 1 postgres postgres 16M Sep 16 23:56 00000001000000000000000004
-rw----- 1 postgres postgres 16M Sep 16 23:56 00000001000000000000000005
-rw----- 1 postgres postgres 16M Sep 16 23:56 000000010000000000000006.partial
```

Recovery

recovery.signal

signals recovery
from backup

tablespace_map

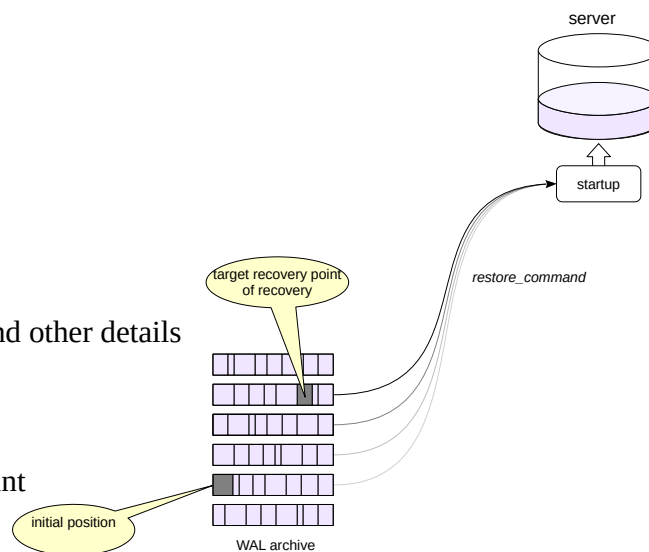
tablespace paths

backup_label

starting position and other details

postgresql.conf

restore_command
target recovery point
etc.



10

The recovery process is controlled by three files and parameters.

The **recovery.signal** file is created manually. If it exists, the server starts in recovery mode. The file's contents are ignored. If **recovery.signal** is missing, PostgreSQL assumes it's performing automatic crash recovery. If the file exists, the server understands that a user-controlled restore from a backup is taking place, and checks two other files that are generated when a base backup is created:

- The **tablespace_map** file contains paths for table spaces. If the symlinks in `pg_tblspc` are missing (which is true for tar format), the server will create them at startup based on the information from **tablespace_map**.

We've already encountered this file in the "Base Backup" topic.

- The **backup_label** file stores the backup name, creation time, and, most importantly, the name of the WAL segment and the position in it to start recovery from.

During recovery, the ``startup`` process reads files from the archive using the command from *restore_command* and applies WAL records from the starting position specified in the label file up to the target recovery position.

Recovery Configuration

Let's start the recovery configuration by creating a signal file.

```
student$ touch /home/student/tmp/backup/recovery.signal
```

When the base backup was created, a backup_label file was generated.

```
student$ cat /home/student/tmp/backup/backup_label
```

```
START WAL LOCATION: 0/5000028 (file 000000010000000000000005)
CHECKPOINT LOCATION: 0/5000060
BACKUP METHOD: streamed
BACKUP FROM: primary
START TIME: 2026-09-16 23:56:35 MSK
LABEL: pg_basebackup base backup
START TIMELINE: 1
```

The key piece of information in this file is the starting point for recovery (START WAL LOCATION). Theoretically, recovery can begin from an earlier position (but not a later one), though this will take more time.

Since our cluster has no additional tablespaces, the tablespace_map file is not necessary. All that remains is to set a restore_command and other parameters in the configuration file.

OS command restoring WAL segment

performs the reverse of *archive_command*
copies file%f from archive to server at path%p
must return 0 only if successful
(can be called for non-existent files)

Monitoring

log_startup_progress_interval = 10s
logging of `startup` process status

The *restore_command* parameter specifies the command to "reverse" the *archive_command*. This command must copy the file from the archive to the *pg_wal* directory and return a status of 0 only when successful. This is very important, since during the recovery process the server can execute a restore command for files that will not be in the archive – this is not an error, but the command must end with non-zero status.

Starting with version PostgreSQL 14, changing the *restore_command* parameter does not require a server restart.

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

In PostgreSQL 15 and higher, the status of the recovery process can be monitored in the server message log with the *log_startup_progress_interval* option enabled. It logs the progress every 10 seconds by default.

In earlier versions, information about the current recovery state was displayed only in the name of the `startup` unix process.

Checkpoint equivalent for recovery

- flush dirty buffers
- write to control file
- remove unnecessary WAL segments

Performed when a control point record is applied

- if at least *checkpoint_timeout* has elapsed since the restartpoint
- or WAL volume approaching *max_wal_size*

Monitoring

- log_checkpoints*
- pg_stat_bgwriter*

When the startup process applies WAL records during recovery, the data in the server's shared memory changes in the same way as in normal operation: dirty buffers appear. WAL records must be retained in case a failure occurs and recovery must restart. To limit the amount of stored records and not start recovery from the very beginning, the server periodically performs *restartpoints*, resetting, as at a checkpoint, dirty buffers of shared memory structures: buffer cache, clog cache and others. If the restartpoint completes successfully, the server writes to the control file and deletes the WAL segments that entirely precede the restartpoint start.

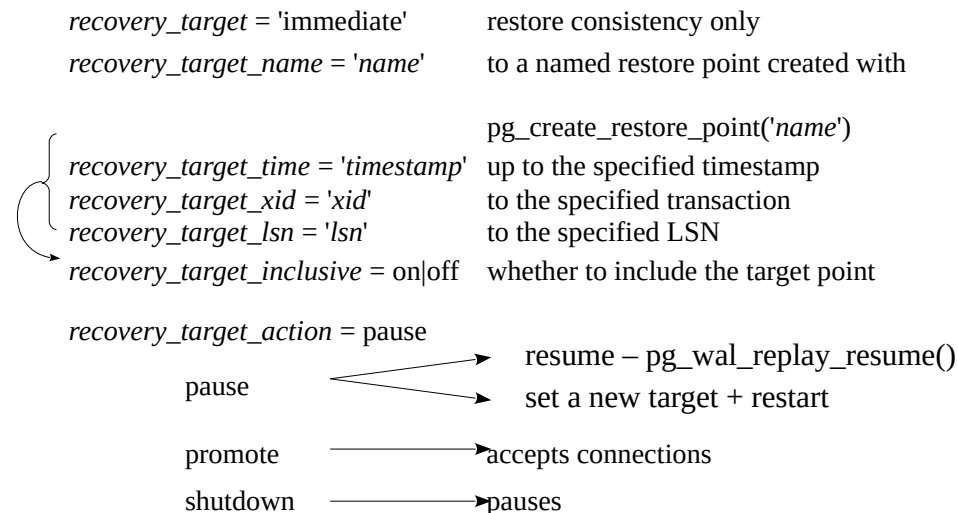
The restartpoint is executed when the startup process encounters a checkpoint record in the WAL, if at least *checkpoint_timeout* time has passed since the last restartpoint, or if the WAL size approaches the *max_wal_size* value. In other words, the restartpoint is triggered by the checkpoint WAL record, but if none of the conditions are met, the record is skipped.

If *log_checkpoints* is enabled, all restartpoints will be recorded in the message log, and the *pg_stat_bgwriter* view will show accumulated statistics.

<https://postgrespro.com/docs/postgresql/16/wal-configuration>

Recovery Target

Point-in-Time Recovery (PITR)



14

If the recovery target is not set, the cluster will be restored as close as possible to the time of failure. However, you can stop the recovery process at any other point by setting a recovery target.

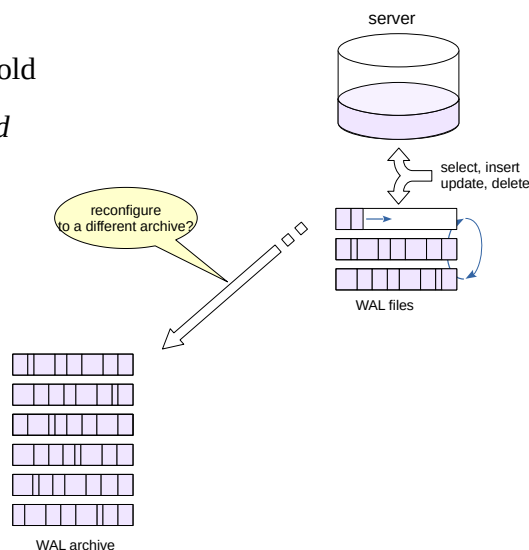
- The `recovery_target = 'immediate'` parameter will stop recovery as soon as consistency is achieved. This is effectively the same as recovering from a base backup without using the archive.
- The `recovery_target_name` parameter lets you specify a named recovery point that was previously created using the `pg_create_restore_point` function.
- The `recovery_target_time` parameter allows you to specify any point in time (timestamp), `recovery_target_xid` allows you to specify any transaction, and `recovery_target_lsn` allows you to specify an LSN. The `recovery_target_inclusive` parameter lets you choose whether to include or exclude the target point itself.

The `recovery_target_action` option allows you to select a state in which the server will be on reaching the target point:

- ``pause`` suspends recovery. The administrator can query the data to verify whether the recovery target was chosen correctly, and then either complete the recovery by calling the `pg_wal_replay_resume` function, or set a new target, restart the server, and continue applying WAL records.
- With ``promote``, the server will immediately be able to accept connection requests.
- ``shutdown`` will stop the server. In this case, the `recovery.signal` file is not removed.

Completion of Recovery

```
remove recovery.signal  
backup_label → backup_label.old  
perform recovery_end_command  
problems are diagnosed  
using the server log
```



15

Once recovery is complete, the ``startup`` process terminates, ``postmaster`` launches the remaining service processes needed for the server instance to operate, and the server begins running in normal mode.

The `recovery.signal` file is deleted, and `backup_label` is renamed to `backup_label.old`.

In the `recovery_end_command` parameter, you can specify a command that will be executed at the end of recovery.

These may be file system operations, archive cleanup, notifications, etc.

The server may fail to start. In this case, no processes will be running in the operating system (and the `postmaster.pid` file will be absent), while the `recovery.signal` file will remain in place. The reasons can be found in the server log (for example, invalid configuration parameters). Once the issue has been resolved, restart the server.

Important note: if continuous archiving was set up on the origin server, it must either be disabled on the restored server or redirected to a different archive location.

Recovery from a Base Backup

Let's configure recovery settings on beta. In the simplest case, all you need is a restore command that copies the specified WAL segment from the archive to the specified path:

```
student$ echo "restore_command = 'cp /var/lib/postgresql/archive/%f %p'" \
>> /home/student/tmp/backup/postgresql.auto.conf
```

If no recovery target is specified (via one of the recovery_target* parameters), WAL records from all files in the archive will be applied to the base backup.

Place the backup in the beta server's data directory and start it.

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

Cluster is not running.

```
student$ sudo rm -rf /var/lib/postgresql/16/beta
```

```
student$ sudo mv /home/student/tmp/backup /var/lib/postgresql/16/beta
```

```
student$ sudo chown -R postgres:postgres /var/lib/postgresql/16/beta
```

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

After a successful recovery, the recovery.signal file is removed and the label file is renamed to backup_label.old:

```
student$ sudo ls -l /var/lib/postgresql/16/beta | egrep 'recovery|backup_label'
```

```
-rw----- 1 postgres postgres 225 Sep 16 23:56 backup_label.old
```

Since all WAL records have been applied, the beta server immediately starts accepting connections. Let's verify what was restored in the table:

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

```
| β=> SELECT * FROM t;
```

```
|          s
| -----
| Row #1
| Another row
| (2 rows)
```

The server's cumulative statistics contain information about restart points:

```
| β=> SELECT checkpoints_timed, checkpoints_req, checkpoint_write_time, checkpoint_sync_time, buffers_checkpoint
| FROM pg_stat_bgwriter \gx
```

```
| -[ RECORD 1 ]-----+---
| checkpoints_timed   | 0
| checkpoints_req     | 1
| checkpoint_write_time | 29
| checkpoint_sync_time | 15
| buffers_checkpoint  | 0
```

Timelines

Sequential number, +1 with each recovery

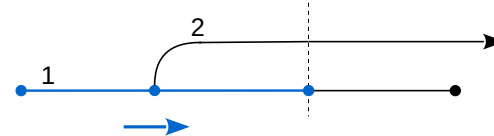
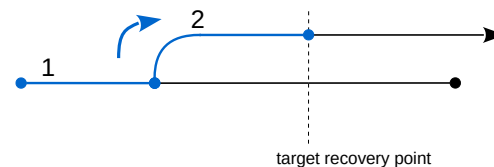
Timeline number N is included in WAL segment name

Historical data is stored in the `pg_wal/N.history` and archived

`recovery_target_timeline = 'latest'`
restore to the last timeline
(by default)

`recovery_target_timeline = 'current'`
restore to the current timeline

`recovery_target_timeline = '1'`
restore to the specified timeline



After restoring to a point in the past, the server begins to generate new WAL records. The record IDs will repeat the record IDs "from the past." To preserve these segments and maintain the ability to recover to different points in time, PostgreSQL introduces the concept of timelines. After each recovery using `recovery.signal`, the timeline number increments, and this number becomes part of the WAL segment numbering.

Timelines form a tree structure, as shown in the diagrams. Timeline 1 is the initial timeline. A recovery was performed to a point in the past, creating timeline 2.

When performing recovery with the `recovery_target_timeline = 'latest'` parameter (the default), WAL replay will follow the second (newer) timeline once it reaches the branch point.

Setting `recovery_target_timeline = '1'` will continue recovery along timeline 1 (the specified timeline), while `recovery_target_timeline = 'current'` will follow the current timeline (which in this case is also timeline 1).

PostgreSQL retrieves branch point information from history files located at `pg_wal/N.history`. Therefore, these files should never be removed from the archive.

Timelines

What happened to the timeline after recovery?

```
β=> SELECT pg_walfile_name(pg_current_wal_lsn());
```

```
pg_walfile_name
-----
00000002000000000000000006
(1 row)
```

The number incremented by one.

A history file corresponding to this branch has appeared in the pg_wal directory:

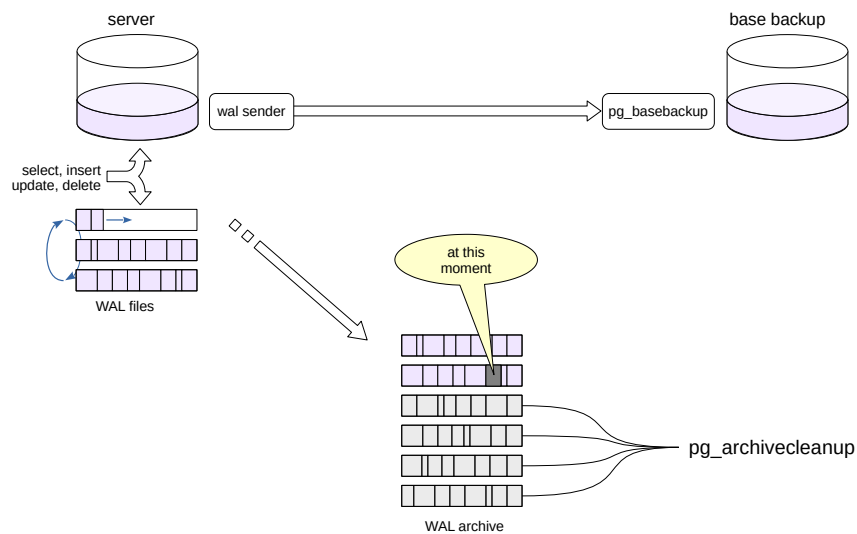
```
postgres$ ls /var/lib/postgresql/16/beta/pg_wal/00000002.history
/var/lib/postgresql/16/beta/pg_wal/00000002.history
```

It contains information about the branch points through which we arrived at this timeline:

```
student$ sudo cat /var/lib/postgresql/16/beta/pg_wal/00000002.history
1          0/6000000          no recovery target specified
```

PostgreSQL uses these files when a timeline is specified in the `recovery_target_timeline` parameter. History files must be archived alongside WAL file segments and should not be removed from the archive.

Archive Cleanup



19

The archive grows over time and needs to be cleaned of unnecessary files every now and then.

The `pg_archivecleanup` tool does just that. It takes a WAL segment name and deletes all preceding segments.

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

The question is which archive position to use as the cutoff point.

If you are only using the archive to restore from a single base backup, you can clean up as soon as you update the backup.

However, if the archive needs to support recovery from multiple base backups, you'll need to either write a custom cleanup script or, better yet, use a third-party backup tool like `pg_probackup` that offers flexible backup retention policies.

Archive Cleanup

The first server's archive currently contains unnecessary files that were added before the base backup was created:

```
student$ sudo ls -lh /var/lib/postgresql/archive
```

```
total 96M
-rw----- 1 postgres postgres 16M Sep 16 23:56 00000001000000000000000001
-rw----- 1 postgres postgres 16M Sep 16 23:56 00000001000000000000000002
-rw----- 1 postgres postgres 16M Sep 16 23:56 00000001000000000000000003
-rw----- 1 postgres postgres 16M Sep 16 23:56 00000001000000000000000004
-rw----- 1 postgres postgres 16M Sep 16 23:56 00000001000000000000000005
-rw----- 1 postgres postgres 16M Sep 16 23:57 00000001000000000000000006.partial
```

The `pg_archivecleanup` utility takes the archive path and the name of the last WAL segment to keep (which can be found in the `backup_label` file).

```
student$ sudo head -n 1 /var/lib/postgresql/16/beta/backup_label.old
```

```
START WAL LOCATION: 0/5000028 (file 00000001000000000000000005)
```

```
student$ sudo pg_archivecleanup /var/lib/postgresql/archive 000000010000000000000005
```

Let's check which files remain in the archive:

```
student$ sudo ls -lh /var/lib/postgresql/archive
```

```
total 32M
-rw----- 1 postgres postgres 16M Sep 16 23:56 00000001000000000000000005
-rw----- 1 postgres postgres 16M Sep 16 23:57 00000001000000000000000006.partial
```

The archive has been cleaned up.

The utility can be called immediately after creating the base backup by placing both commands in a script.

Physical backups:
base backup + WAL archive

Good for

- ongoing periodic backups
- crash recovery with minimal data loss
- point-in-time recovery
- backing up large volumes of data

Not so good for

- long-term storage

Not suitable for

- migration to a different platform

1. On the alpha server, create a database and a data table.
2. Set up continuous archiving.
3. Make a copy of the cluster without log files.
4. Insert a few more rows into the table and verify that the current WAL segment has been archived.
5. Restore the beta server from the backup, specifying only the restore command in the configuration.
6. Restore the beta server again from the same backup, this time with ``immediate`` as the recovery target.

After completing the practice tasks, delete the databases and reset the configuration.

1. Database and Table

```
α=> CREATE DATABASE backup_archive;
CREATE DATABASE
α=> \c backup_archive
You are now connected to database "backup_archive" as user "student".
α=> CREATE TABLE t(s text);
CREATE TABLE
α=> INSERT INTO t VALUES ('Hello, world!');
INSERT 0 1
```

2. Setting Up Continuous Archiving

```
postgres$ mkdir /var/lib/postgresql/archive
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
student$ sudo pg_ctlcluster 16 alpha restart
```

3. Base Backup

Since we'll need to restore from the same backup twice, let's create it in compressed tar format.

```
student$ pg_basebackup \
--checkpoint=fast \
--wal-method=none \
--format=tar \
--gzip \
--pgdata=/home/student/tmp/backup
NOTICE: all required WAL segments have been archived
student$ ls -l /home/student/tmp/backup
total 5328
-rw----- 1 student student 224439 Sep 17 00:08 backup_manifest
-rw----- 1 student student 5229378 Sep 17 00:08 base.tar.gz
```

4. Populating a Table With Rows

```
student$ psql -d backup_archive
α=> INSERT INTO t VALUES ('Another row');
INSERT 0 1
Switch the segment:
α=> SELECT pg_walfile_name(pg_current_wal_lsn()), pg_switch_wal();
      pg_walfile_name      | pg_switch_wal
-----+-----
000000010000000000000003 | 0/3006640
(1 row)
```

Check the archiving status:

```
α=> SELECT last_archived_wal FROM pg_stat_archiver;
      last_archived_wal
-----
000000010000000000000003
(1 row)
```

5. Recovery from a Base Backup

```
student$ sudo pg_ctlcluster 16 beta stop
Cluster is not running.
```

```
postgres$ rm -rf /var/lib/postgresql/16/beta
postgres$ mkdir /var/lib/postgresql/16/beta
postgres$ chmod 700 /var/lib/postgresql/16/beta
student$ sudo tar xzf /home/student/tmp/backup/base.tar.gz -C /var/lib/postgresql/16/beta
```

To make it simple, disable continuous archiving on the standby server and set just the restore command:

```
postgres$ echo "restore_command = 'cp /var/lib/postgresql/archive/%f %p' " \
| tee /var/lib/postgresql/16/beta/postgresql.auto.conf
restore_command = 'cp /var/lib/postgresql/archive/%f %p'
```

Create recovery.signal.

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

Start the server.

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

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

```
| β=> SELECT * FROM t;
```

```
|
|      s
|-----
| Hello, world!
| Another row
| (2 rows)
```

All rows were restored. Since all available WAL records are applied by default, the server immediately switched to normal mode and is ready to accept write queries:

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

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

6. Recovery from Base Backup (immediate)

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

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

```
postgres$ mkdir /var/lib/postgresql/16/beta
```

```
postgres$ chmod 700 /var/lib/postgresql/16/beta
```

```
student$ sudo tar xzf /home/student/tmp/backup/base.tar.gz -C /var/lib/postgresql/16/beta
```

Create recovery.signal, and add the restore command and recovery target to the configuration file:

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

```
postgres$ cat << EOF | tee /var/lib/postgresql/16/beta/postgresql.auto.conf
restore_command = 'cp /var/lib/postgresql/archive/%f %p'
recovery_target = 'immediate'
EOF
```

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

Start the server.

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

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

```
| β=> SELECT * FROM t;
```

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

Only the first row was restored.

This time, only the WAL records required to bring the data to a consistent state were applied. The server is already accepting read queries, and recovery can be continued if needed:

```
| 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)

```

The `pg_get_wal_replay_pause_state()` function returns:

- not paused – no pause was requested
- pause requested – a pause has been requested
- paused – recovery is paused

Since we don't need to apply any further records, we complete the recovery by promoting the server to normal mode (superuser privileges required):

```

β=> SELECT pg_wal_replay_resume();

```

```

pg_wal_replay_resume
-----
(1 row)

```

```

β=> SELECT pg_is_in_recovery();

```

```

pg_is_in_recovery
-----
f
(1 row)

```

To avoid manually promoting the server out of recovery mode, we could have set `recovery_target_action = promote` in advance.

Cleanup

Remove the backup and databases, reset the configuration.

```

student$ rm -rf /home/student/tmp/backup

```

```

β=> ALTER SYSTEM RESET archive_mode;

```

```

ALTER SYSTEM

```

```

β=> ALTER SYSTEM RESET archive_command;

```

```

ALTER SYSTEM

```

```

student$ sudo pg_ctlcluster 16 beta stop

```

```

α=> \c student postgres

```

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

```

α=> DROP DATABASE backup_archive;

```

```

DROP DATABASE

```

```

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 -rf /var/lib/postgresql/archive

```

1. On the alpha server, create a table in which data will be periodically written. Create a physical backup and configure continuous WAL archiving.
2. Insert 100 rows into the prepared table three times (data for January, February and March), marking the time immediately after insertion. Delete some lines between inserts: simulate a situation in which the user accidentally deletes part of the data at an unknown moment, but the problem is not noticed immediately.
3. Recover the deleted data without stopping the alpha server.

3. Use the beta server as an auxiliary.

1. Table and Backup with WAL Archive

```
α=> CREATE DATABASE backup_archive;
```

```
CREATE DATABASE
```

```
α=> \c backup_archive
```

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

Let's prepare the table:

```
α=> CREATE TABLE t (
  id integer PRIMARY KEY,
  month text
);
```

```
CREATE TABLE
```

After setting up the data structure, create a physical backup:

```
postgres$ pg_basebackup \
--checkpoint=fast \
--pgdata /var/lib/postgresql/backup
```

WAL segment archive:

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

Add the archiving parameters file to the supplementary configuration directory.

```
postgres$ cat << EOF | tee /etc/postgresql/16/alpha/conf.d/arch.conf
archive_mode = 'on'
archive_command = 'test ! -f /var/lib/postgresql/archive/%f && cp %p /var/lib/postgresql/archive/%f'
archive_timeout = '30s'
EOF
```

```
archive_mode = 'on'
archive_command = 'test ! -f /var/lib/postgresql/archive/%f && cp %p
/var/lib/postgresql/archive/%f'
archive_timeout = '30s'
```

Restart the server.

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

```
student$ psql -d backup_archive
```

2. Inserting Data into the Table

Our application inserts 100 rows into the table at the end of each month.

Insert rows for January:

```
α=> INSERT INTO t SELECT g.i, 'January' FROM generate_series(1,100) AS g(i);
```

```
INSERT 0 100
```

Save the timestamp after the insert completes:

```
α=> SELECT now();
```

```
          now
-----
2026-09-17 00:09:53.91657+03
(1 row)
```

A user accidentally deletes some of the data:

```
α=> DELETE FROM t WHERE id BETWEEN 42 AND 111;
```

```
DELETE 59
```

Insert rows for February:

```
α=> INSERT INTO t SELECT g.i, 'February' FROM generate_series(101,200) AS g(i);
```

```
INSERT 0 100
```

Save the timestamp again after the transaction completes:

```
α=> SELECT now();

              now
-----
2026-09-17 00:10:04.242256+03
(1 row)
```

The user runs the same command again for some reason:

```
α=> DELETE FROM t WHERE id BETWEEN 42 AND 111;

DELETE 11
```

The error goes unnoticed for now. Insert another batch of rows:

```
α=> INSERT INTO t SELECT g.i, 'March' FROM generate_series(201,300) AS g(i);

INSERT 0 100
```

Let's record the timestamp once more:

```
α=> SELECT now();

              now
-----
2026-09-17 00:10:14.540194+03
(1 row)
```

The error has been discovered:

```
α=> SELECT month, count(*) FROM t GROUP BY month;

 month | count
-----+-----
February |      89
March   |     100
January  |      41
(3 rows)
```

3. Recovery Using the Archive

Let's try to recover the data without stopping the alpha server.

Since the January and February data is missing, the recovery plan is as follows:

- Restore the beta server from the WAL archive to the point when the January rows finished committing
- Export a logical backup of the January data
- Restore the beta server again, this time to the point when the February rows finished committing
- Export a logical backup of the February data (the January data may be corrupted at this point)
- Load the recovered data into the application table on the alpha server

Stop beta and place the physical backup into its data directory.

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

Cluster is not running.

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

```
postgres$ mv /var/lib/postgresql/backup /var/lib/postgresql/16/beta
```

We will need a recovery.signal file:

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

The plan is to restore the server to a specific point in time – that is, perform point-in-time recovery (PITR). Set the restore command and specify the recovery target as the timestamp created right after the January data finished committing:

```
postgres$ cat << EOF | tee /etc/postgresql/16/beta/conf.d/recovery.conf
restore_command = 'cp /var/lib/postgresql/archive/%f %p'
recovery_target_time = '2026-09-17 00:09:53.91657+03'
EOF

restore_command = 'cp /var/lib/postgresql/archive/%f %p'
recovery_target_time = '2026-09-17 00:09:53.91657+03'
```

Before starting recovery, force a segment switch on alpha to ensure all WAL records make it into the archive. Archiving takes place when writing to the next segment begins.

```
α=> SELECT pg_switch_wal();
```

```
pg_switch_wal
-----
0/301D308
(1 row)
```

```
α=> CHECKPOINT;
```

```
CHECKPOINT
```

Start the beta server.

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

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

Verify that the January data has been restored:

```
| β=> SELECT month, count(*) FROM t GROUP BY month;
|
|  month | count
|-----+-----
| January |    100
| (1 row)
```

The beta server is in recovery mode:

```
| β=> SELECT pg_is_in_recovery();
|
| pg_is_in_recovery
|-----
| t
| (1 row)
```

This doesn't prevent us from exporting a logical backup.

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

```
| β=> COPY (SELECT * FROM t WHERE month = 'January') TO '/var/lib/postgresql/tmp/jan.dump';
| COPY 100
```

Now let's move on to February. Set a new target time:

```
postgres$ sed -i "/recovery_target_time/c\recovery_target_time = '2026-09-17 00:10:04.242256+03'"
/etc/postgresql/16/beta/conf.d/recovery.conf
```

```
postgres$ cat /etc/postgresql/16/beta/conf.d/recovery.conf
```

```
restore_command = 'cp /var/lib/postgresql/archive/%f %p'
recovery_target_time = '2026-09-17 00:10:04.242256+03'
```

Restart beta to continue recovery.

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

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

Verify the recovered data:

```
| β=> SELECT month, count(*) FROM t GROUP BY month;
|
|  month | count
|-----+-----
| February |    100
| January  |     41
| (2 rows)
```

The February rows are where they should be, but the January rows are already corrupted. Export the February data:

```
| β=> COPY (SELECT * FROM t WHERE month = 'February') TO '/var/lib/postgresql/tmp/feb.dump';
| COPY 100
```

Restore the exported data for both months on alpha.

```
α=> DELETE FROM t WHERE month IN ('January', 'February');
```

```
DELETE 130
```

```
α=> COPY t FROM '/var/lib/postgresql/tmp/jan.dump';
```

COPY 100

```
α=> COPY t FROM '/var/lib/postgresql/tmp/feb.dump';
```

COPY 100

It worked:

```
α=> SELECT month, count(*) FROM t GROUP BY month;
```

month	count
February	100
March	100
January	100

(3 rows)

In PostgreSQL 17 and later, you can skip the delete step and use the COPY command with the ON ERROR ignore parameter to quietly skip rows that already exist in the table.

Cleanup

Remove the databases and configuration changes.

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

```
α=> \c postgres
```

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

```
α=> DROP DATABASE backup_archive;
```

DROP DATABASE

```
postgres$ rm /etc/postgresql/16/beta/conf.d/recovery.conf
```

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

Remove any unnecessary files.

```
postgres$ rm -rf /var/lib/postgresql/{tmp,archive,backup}
```