Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov
Translated by Alexander Meleshko
Cover photo by Oleg Bartunov (Phu monastery and Bhrikuti peak, Nepal)

Feedback
Please send your feedback, comments and suggestions to:
edu@postgrespro.ru

Agenda

Logical backup

Physical backup

Logical backup

What is logical backup

Table backup

Database backup

Cluster backup

Logical backup

SQL commands to restore data from scratch

+ can backup a separate object or database
+ can recover on a cluster running another major PostgreSQL version
+ can recover on a different architecture
− low speed

There are two types of backup: logical and physical.

Logical backup is a set of SQL commands that restores a cluster (or a database, or a separate object) from scratch.

Such a backup is, in fact, a plain text file, which gives a certain flexibility. For example, you can make a copy of only those objects that are needed; you can edit the file by changing the names or data types, etc.

In addition, SQL commands can be executed on a different version of the DBMS (if there is compatibility at the command level) or on a different architecture (so binary compatibility is not required).

However, for a large database, this mechanism is inefficient, since executing all the commands will take a long time. Moreover, it is possible to restore the system from such a backup only to the moment at which the backup was made.

https://postgrespro.com/docs/postgresql/15/backup-dump

COPY: table backup

Backup

output of a table or a query into a file, console or another program

Recovery

insertion of rows from a file or console into an existing table

Server variant	Client variant
SQL COPY command	psql \COPY command
the file must be accessible to the postgres user on the server	the file must be accessible to the user who has launched psql on the client

If you want to save only the contents of one table, you can use the COPY command.

The command writes a table (or the result of an arbitrary query) either to a file or to the console, or sends it as input to another program. You can specify parameters such as format (plain text, csv or binary), field separators, NULL representation etc.

The alternative variant of the COPY command reads fields from a file or the console and inserts them into a table. The table isn't truncated, the new rows are simply appended to the existing ones.

The COPY command is significantly faster than similar INSERT commands, because the client does not need to access the server repeatedly, and the server does not have to parse the commands multiple times.

https://postgrespro.com/docs/postgresql/15/sql-copy

In psql, there is a client version of the COPY command with a similar syntax. Unlike the server version, which is an SQL command, the client version is a psql command.

The file name in the SQL command corresponds to a file on the database server. The user running PostgreSQL (usually postgres) must have access to this file. In the client version, the file is accessed on the client, and only the content is transmitted to the server.

https://postgrespro.com/docs/postgresql/15/app_psql

COPY

Create a database and a table in it.

```
=> CREATE DATABASE backup_overview;

CREATE DATABASE

=> \c backup_overview

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

=> CREATE TABLE t(id numeric, s text);

CREATE TABLE

=> INSERT INTO t VALUES (1, 'Hello!'), (2, ''), (3, NULL);

INSERT 0 3

=> SELECT * FROM t;

 id |   s
----+--------
  1 | Hello!
  2 |
  3 |
(3 rows)
```

Here is what the COPY command shows:

```
=> COPY t TO STDOUT;

1       Hello!
2
3       \N
```

Note that an empty string and NULL are different things, despite the output not telling us that.

You can input the data:

```
=> TRUNCATE TABLE t;

TRUNCATE TABLE

=> COPY t FROM STDIN;
1       Hi there!
2
3       \N
\.

COPY 3
```

Verify:

```
=> \pset null '<null>'

Null display is "<null>".

=> SELECT * FROM t;

 id |     s
----+-----------
  1 | Hi there!
  2 |
  3 | <null>
(3 rows)
```

pg_dump: database backup

Backup

- outputs either an SQL script or an archive in a custom format with a table of contents to the console or to a file
- supports parallel execution
- can define what objects to backup
  (tables, schemas, only DML or only DDL, etc.)

Recovery

- SQL script via psql
- custom format archive via pg_restore
  (can define what objects to recover and supports parallel execution)
- the new database must be created from the template0 database
- roles and tablespaces must be created in advance

The pg_dump utility creates a full-scale database backup. Depending on the parameters, it provides either an SQL script containing commands that create the required objects, or an archive in a custom format with a table of contents.

Restoring from an SQL script is as simple as executing it in psql.

https://postgrespro.com/docs/postgresql/15/app-pgdump

Restoring from a custom format is done using the pg_restore tool. It reads the archive and translates it into regular psql commands. The advantage is that it allows you to specify what objects to restore at the recovery stage, not just at the backup stage. Moreover, this type of backup and recovery supports parallel execution.

https://postgrespro.com/docs/postgresql/15/app-pgrestore

The database for recovery must be created from the database template0, since all changes made in template1 will also be backed up. In addition, the necessary roles and tablespaces must be created in advance, since these objects belong to the entire cluster. After recovery, it is recommended to run the ANALYZE command to collect fresh statistics.

pg_dumpall: cluster backup

Backup

- makes a backup of the entire cluster, including roles and tablespaces
- outputs an SQL script to the console or to a file
- parallel execution is not supported, but you can dump only the global objects and then use pg_dump

Recovery

- via psql

pg_dumpall creates a backup of the entire cluster, including roles and tablespaces.

Since pg_dumpall requires access to all objects of all databases, it should be ran by the superuser. pg_dumpall connects to each database in the cluster one by one and makes a backup using pg_dump. In addition, it also stores data related to the cluster as a whole.

The result of pg_dumpall is a script for psql. Other formats are not supported. This means that pg_dumpall does not support parallel execution, which can be a problem for larger clusters. In this case, you can use the --globals-only key to backup only roles and tablespaces, and then backup all the databases using pg_dump.

https://postgrespro.com/docs/postgresql/15/app-pg-dumpall

pg_dump tool

Take a look at pg_dump output in a plain text format. Note the way data from the table is saved.

If any changes were made to template1, they too will make it into the backup. Therefore, when recovering a database, it is best to create one on the target from template0 (the --create key adds the necessary commands automatically).

```
student$ pg_dump -d backup_overview --create

--
-- PostgreSQL database dump
--

-- Dumped from database version 15.1 (Ubuntu 15.1-1.pgdg22.04+1)
-- Dumped by pg_dump version 15.1 (Ubuntu 15.1-1.pgdg22.04+1)

SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
SET xmloption = content;
SET client_min_messages = warning;
SET row_security = off;

--
-- Name: backup_overview; Type: DATABASE; Schema: -; Owner: student
--

CREATE DATABASE backup_overview WITH TEMPLATE = template0 ENCODING = 'UTF8' LOCALE_PROVIDER = libc LOCALE = 'en_US.UTF-8';


ALTER DATABASE backup_overview OWNER TO student;

\connect backup_overview

SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
SET xmloption = content;
SET client_min_messages = warning;
SET row_security = off;

SET default_tablespace = '';

SET default_table_access_method = heap;

--
-- Name: t; Type: TABLE; Schema: public; Owner: student
--

CREATE TABLE public.t (
    id numeric,
    s text
);


ALTER TABLE public.t OWNER TO student;

--
-- Data for Name: t; Type: TABLE DATA; Schema: public; Owner: student
--

COPY public.t (id, s) FROM stdin;
1       Hi there!
2
3       \N
\.


--
-- PostgreSQL database dump complete
--
```

As an example, let’s copy the table into another database.

```
=> CREATE DATABASE backup_overview2;

CREATE DATABASE

student$ pg_dump -d backup_overview --table=t | psql -d backup_overview2

SET
SET
SET
SET
SET
 set_config
------------

(1 row)

SET
SET
SET
SET
SET
SET
CREATE TABLE
ALTER TABLE
COPY 3

student$ psql -d backup_overview2

  => SELECT * FROM t;

   id |     s
  ----+-----------
    1 | Hi there!
    2 |
    3 |
  (3 rows)
```

Physical backup

What is physical backup

Cold and hot backups

Replication protocol

Standalone backup

Continuous WAL archiving

Physical backup

Crash recovery mode: base backup + WAL

+ recovery speed
+ can recover to a certain point in time
− cannot recover a separate database, only the cluster as a whole
− can recover only on the same architecture and major PostgreSQL version

Physical backup uses the crash recovery mechanism. This requires:

- a copy of cluster files (base backup),

- a set of write-ahead logs needed to restore consistency.

If the file system is already consistent (the backup was made when the server was stopped correctly), then the WALs are not required.

However, with WALs together with the base backup, you can recover the system state at any point in time. This way, the database can be recovered to the state right before the crash (or at any moment before that, if needed).

High recovery speed and the ability to create a backup on the fly without stopping the server make physical backup the main choice for routine backup needs.

https://postgrespro.com/docs/postgresql/15/backup-file

https://postgrespro.com/docs/postgresql/15/continuous-archiving

Hot or cold

	Cold backup		Hot backup
The file system is backed up when...	the server is off	the server was shut down incorrectly	the server is running special tools required
WAL records are...	not required	required after the last checkpoint stored in the file system	required for the duration of the backup creation server must not delete WALs too soon

Physical backup creates a copy of the file system at some point.

If the copy is created while the server is stopped, it's called a "cold backup". A cold backup either contains consistent data (if the server was shut down correctly), or contains all the logs necessary for recovery (for example, if the OS has done a data snapshot). This simplifies recovery, but requires that the server is stopped.

If a copy is created while the server is running (which requires certain additional actions since you can't copy files just like that), it is called a "hot backup". The process is more complicated, but can be performed without stopping the server.

During a hot backup, the file system will be inconsistent. However, the crash recovery mechanism can also be successfully applied to backup recovery. This will require the WALs for at least the time it takes to back up the OS files.

Standalone backup

A standalone backup contains both data files and WALs

Backup via pg_basebackup

- connects to the server over the replication protocol
- performs a checkpoint
- copies the file system into a specified directory
- saves all WAL segments generated during the copying process

Recovery

- deploy the standalone backup
- start the server

Hot backups are created with the pg_basebackup tool.

First, it performs a checkpoint. Then, the cluster file system is copied.

All WAL files generated by the server during the time from the checkpoint to the end of file copying are also added to the backup. The resulting backup is called standalone because it contains all the data necessary for recovery.

All you need to restore from it is to deploy the backup and start the server. It will use the WALs to recover consistency on startup if necessary, and will be ready to go.

https://postgrespro.com/docs/postgresql/15/app-pgbasebackup

Replication protocol

Protocol

- receiving the WAL stream
- backup and recovery control commands

Managed by the wal_sender process (*max_wal_senders*)

wal_level = replica

Replication slot

- a server object that consumes WAL records
- remembers which record was read last
- WAL segments are not deleted until fully read through the slot

The replication protocol allows processes to connect to the server and collect all WAL files generated during file copying. Despite the name, the protocol is used not only for replication (which will be discussed in the next topic), but also for backup. The protocol can stream WAL entries while data files are being copied.

To prevent the server from deleting WAL files too early, the replication slot can be employed.

Establishing a connection over the replication protocol requires a certain configuration.

First, the initiating role must have the REPLICATION attribute (or be a superuser). This role must also have the necessary permission in the pg_hba.conf configuration file.

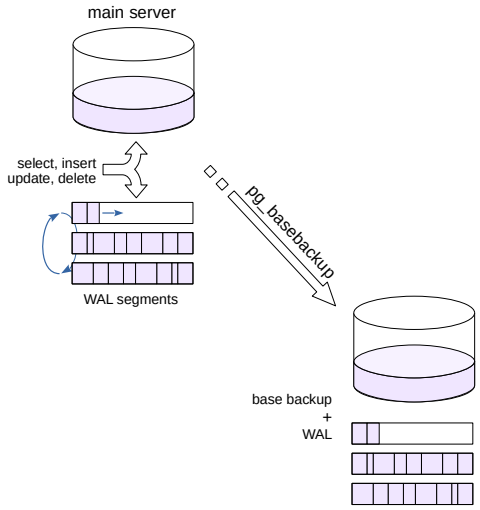
Second, the *max_wal_senders* parameter must be set sufficiently high. This parameter limits the number of simultaneously running wal sender processes serving replication protocol connections.

Third, the *wal_level* parameter, which determines the amount of information in the WAL, must be set to *replica*.

Starting from PostgreSQL 10, the default settings already satisfy all these requirements (for a local connection).

https://postgrespro.com/docs/postgresql/15/protocol-replication

Standalone backup



The image on the left shows the main server. It processes incoming queries. At the same time, WAL records are formed and the state of the databases changes (first in the buffer cache, then on disk). WAL segments are cyclically overwritten (that is, old segments are deleted as new ones are created, since the file names are unique).

At the bottom of the picture is a backup archive (usually located on another server). It contains a base copy of the data and a set of WAL files.

Standalone backup

The default configuration is sufficient for replication:

```
=> SELECT name, setting
FROM pg_settings
WHERE name IN ('wal_level','max_wal_senders');
```

```
      name       | setting
-----------------+---------
 max_wal_senders | 10
 wal_level       | replica
(2 rows)
```

The local connection permission for the replication protocol is also on in pg_hba.conf by default (not for all package distributions):

```
=> SELECT type, database, user_name, address, auth_method
FROM pg_hba_file_rules()
WHERE 'replication' = ANY(database);
```

```
 type  |   database    | user_name |  address  |  auth_method
-------+---------------+-----------+-----------+---------------
 local | {replication} | {all}     | <null>    | trust
 host  | {replication} | {all}     | 127.0.0.1 | scram-sha-256
 host  | {replication} | {all}     | ::1       | scram-sha-256
(3 rows)
```

Another database cluster, replica, has been initiated on the port 5433. Ubuntu has a stock tool we can use to verify that the cluster is stopped:

```
student$ pg_lsclusters
```

```
Ver Cluster Port Status Owner    Data directory                 Log file
15  main    5432 online postgres /var/lib/postgresql/15/main    /var/log/postgresql/postgresql-15-main.log
15  replica 5433 down   postgres /var/lib/postgresql/15/replica /var/log/postgresql/postgresql-15-replica.log
```

Create a backup. Use the default format (plain):

```
student$ sudo rm -rf /home/student/basebackup

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

Move the new backup into replica cluster directory (after making sure the cluster is stopped):

```
student$ sudo pg_ctlcluster 15 replica status

pg_ctl: no server running

student$ sudo rm -rf /var/lib/postgresql/15/replica

student$ sudo mv /home/student/basebackup/ /var/lib/postgresql/15/replica
```

The cluster files must belong to the postgres user.

```
student$ sudo chown -R postgres:postgres /var/lib/postgresql/15/replica
```

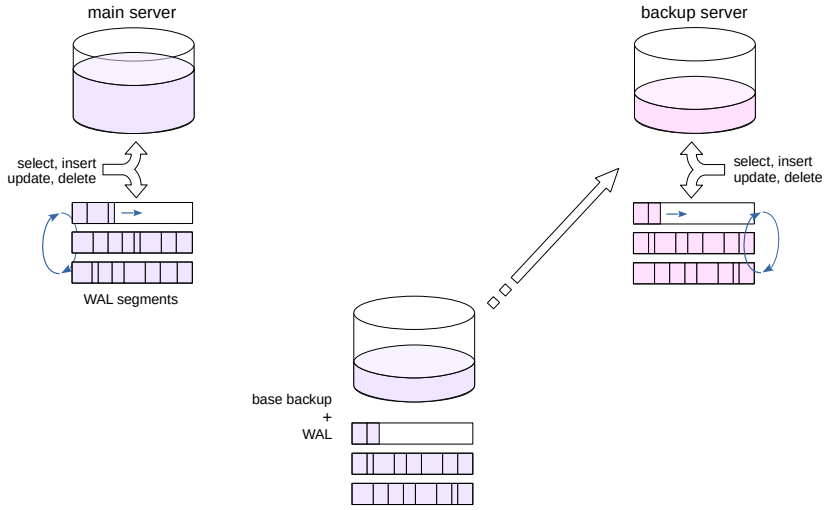
Verify the contents:

```
student$ sudo ls -l /var/lib/postgresql/15/replica
```

```
total 344
-rw------- 1 postgres postgres    225 May  4 19:50 backup_label
-rw------- 1 postgres postgres 268500 May  4 19:50 backup_manifest
drwx------ 8 postgres postgres   4096 May  4 19:50 base
drwx------ 2 postgres postgres   4096 May  4 19:50 global
drwx------ 2 postgres postgres   4096 May  4 19:50 pg_commit_ts
drwx------ 2 postgres postgres   4096 May  4 19:50 pg_dynshmem
drwx------ 4 postgres postgres   4096 May  4 19:50 pg_logical
drwx------ 4 postgres postgres   4096 May  4 19:50 pg_multixact
drwx------ 2 postgres postgres   4096 May  4 19:50 pg_notify
drwx------ 2 postgres postgres   4096 May  4 19:50 pg_replslot
drwx------ 2 postgres postgres   4096 May  4 19:50 pg_serial
drwx------ 2 postgres postgres   4096 May  4 19:50 pg_snapshots
drwx------ 2 postgres postgres   4096 May  4 19:50 pg_stat
drwx------ 2 postgres postgres   4096 May  4 19:50 pg_stat_tmp
```

```
drwx------ 2 postgres postgres   4096 May  4 19:50 pg_subtrans
drwx------ 2 postgres postgres   4096 May  4 19:50 pg_tblspc
drwx------ 2 postgres postgres   4096 May  4 19:50 pg_twophase
-rw------- 1 postgres postgres      3 May  4 19:50 PG_VERSION
drwx------ 3 postgres postgres   4096 May  4 19:50 pg_wal
drwx------ 2 postgres postgres   4096 May  4 19:50 pg_xact
-rw------- 1 postgres postgres     88 May  4 19:50 postgresql.auto.conf
```

Recovery



During recovery, the base backup, including the necessary WAL files, is deployed, for example, on another server (shown on the right).

After the server starts, it restores consistency and is ready to go.
The system is recovered to the point in time when the backup was made.
Of course, the main server can go far ahead in the meantime.

Recovery from backup

When started, the cluster will begin recovery.

```
student$ sudo pg_ctlcluster 15 replica start
```

Now both servers run concurrently and independently.

Main server:

```
=> INSERT INTO t VALUES (4, 'Main server');

INSERT 0 1

=> SELECT * FROM t;

 id |      s
----+-------------
  1 | Hi there!
  2 |
  3 | <null>
  4 | Main server
(4 rows)
```

Server recovered from the backup:

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

```
=> INSERT INTO t VALUES (4, 'Backup');

INSERT 0 1

=> SELECT * FROM t;

 id |     s
----+-----------
  1 | Hi there!
  2 |
  3 |
  4 | Backup
(4 rows)
```

WAL archive

File archive

- WAL segments are archived as they are filled
- controlled by the server
- archiving happens with a delay

Streaming archive

- a stream of WAL records is continuously recorded into the archive
- external tools required
- delays are minimal

The hot backup concept can be improved upon even further. Since we have a copy of the file system and WALs, then by constantly saving new logs, we will be able to restore the system not only at the time of copying files, but also at any point in time after that.

There are two ways to go about it. The first is to archive old WAL files before reusing them. There's a server setting for that. Unfortunately, with this option, an incomplete WAL file will not be archived until the server switches to writing to another WAL file.

The second way is to continuously read WAL entries using the replication protocol and write them into the same archive. This way, delays are minimal, but a separate tool is needed to receive the data stream and archive it.

WAL file archive

archiver process

Parameters

archive_mode = on

archive_command — a shell command to copy a WAL segment to a separate storage

Algorithm

when switching to a new WAL segment, the *archive_command* command is called for the filled segment

if the command terminates with the status 0, the segment is deleted

if the command returns anything else (in particular, if the command is not specified), the segment remains until the attempt is successful

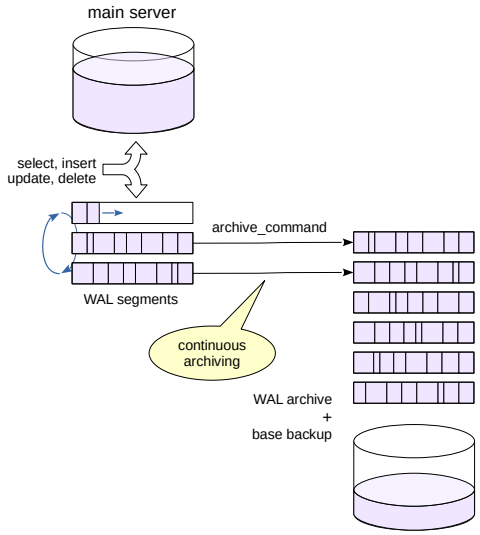
The WAL files archive is managed by the archiver background process.

An arbitrary shell command to be used for copying can be defined in the *archive_command* parameter. The mechanism itself is enabled by the parameter *archive_mode* = on.

The algorithm goes as follows. When a WAL segment is filled, the copy command is called. If it terminates with a 0 status, the segment can be deleted safely. Otherwise, the segment (and the ones following it) will not be deleted, and the server will periodically try to execute the command until it returns 0.

https://postgrespro.com/docs/postgresql/15/continuous-archiving

WAL file archive



This figure shows the main server with continuous archiving set up. Filled WAL segments are copied to a separate archive using the command defined by the *archive_command* parameter. Usually, the archive is located on a separate server, and it also stores the base backup (or several, from multiple points in time).

Streamed WAL archive

pg_receivewal

- connects over the replication protocol (can use a replication slot) and streams WAL records into segment files
- the starting position is the beginning of the segment following the last filled one in the directory, or the start of current segment, if the directory is empty
- unlike the file archive, records are added continuously
- parameters have to be reconfigured when changing servers

Another solution is to use the pg_receivewal utility to write segments to the archive using the stream replication protocol.

pg_receivewal usually runs on a separate "archive" server and connects to the main server with the parameters specified in the command line keys. It can (and should) use the replication slot in order to ensure that records are not lost.

Pg_receivewal generates files in the same way as the server does, and writes them to the specified directory. Segments that have not yet been filled out are written with the .partial prefix.

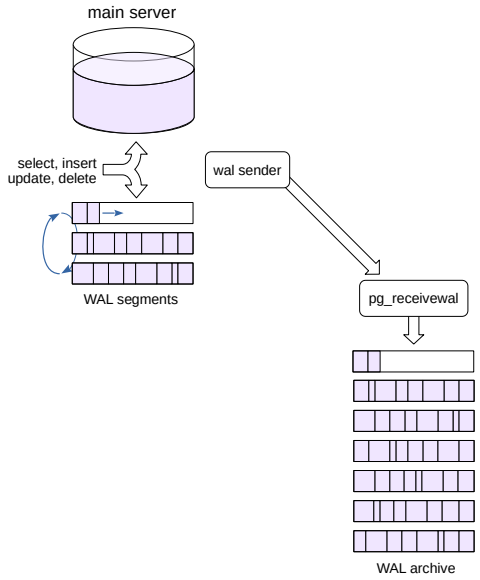
Archiving always starts from the beginning of the segment following the last filled archive segment. If the archive is empty (first run), archiving starts from the beginning of the current segment.

When switching to a new server, pg_receivewal must be stopped and restarted with new parameters.

Note that the utility itself does not start automatically (as a service) and does not run as a daemon.

https://postgrespro.com/docs/postgresql/15/app-pgreceivewal

Streamed WAL archive



pg_receivewal connects to the server over the stream replication protocol. The connection is handled by a separate wal sender process (this must be taken into account when setting the *max_wal_senders* parameter).

pg_receivewal records data without waiting for the entire segment to be received.

Backup + archive

Configured continuous WAL archiving

Backup via pg_basebackup

- connects to the server over the replication protocol
- performs a checkpoint
- copies the file system into a specified directory

WAL segments are not required

Recovery

- deploy the backup
- set configuration parameters
  (command to read WAL from the archive, set the target recovery point)
- create a recovery.signal file
- start the server

To create a backup with continuous archiving configured, the same pg_basebackup tool is used, only with a different set of parameters. The only difference is that the WAL files are not saved to the backup, since they are already in the archive.

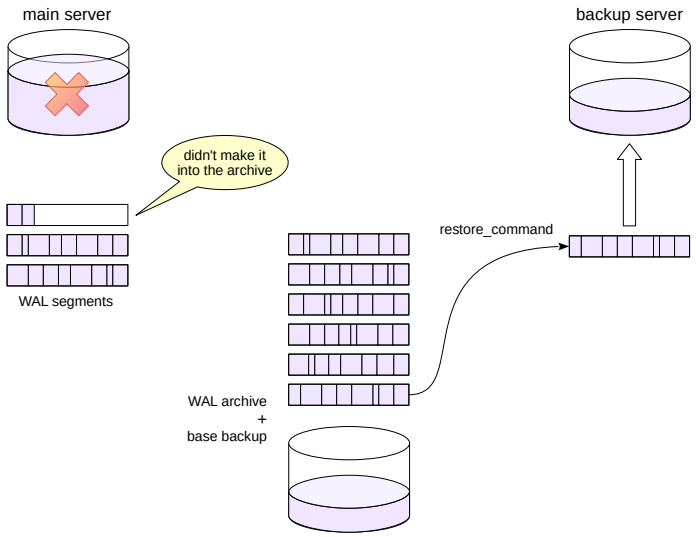
Recovery is more complicated in this case. In addition to deploying the base backup, some recovery settings must be specified:

- *restore_command* parameter (inverse of *archive_command*, copies files from the archive to the server),

- target recovery point.

In addition, a *recovery.signal* file is needed. If present at server startup, the file tells the server to enter the managed recovery mode (the contents of the file are ignored).

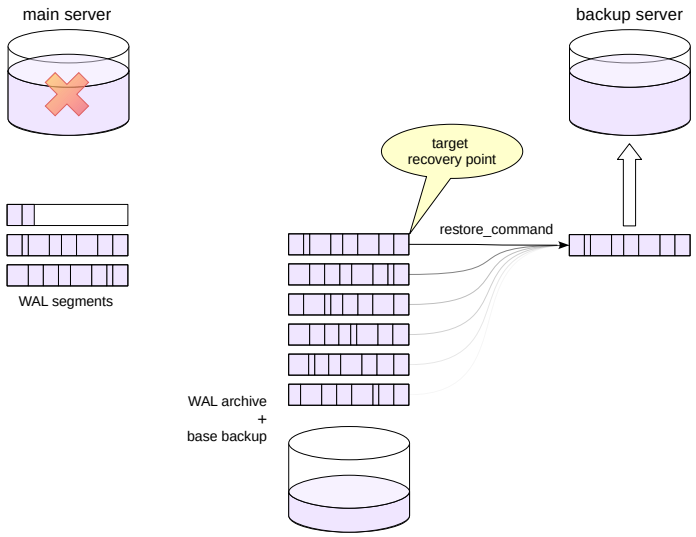
Recovery



The recovery procedure (for example, after main server crash) is performed as follows. A base backup is deployed on another (or the same) server and a recovery.signal file is created. The server starts up and starts reading WAL segments from the archive using *restore_command* and applying them.

Note that during file archiving, the last unfilled WAL segment at the main server will not be archived. However, the segment can be manually added to the pg_wal directory on the backup server, if necessary. Of course, there may be several such unarchived segments, but only in case of some kind of failure during archiving.

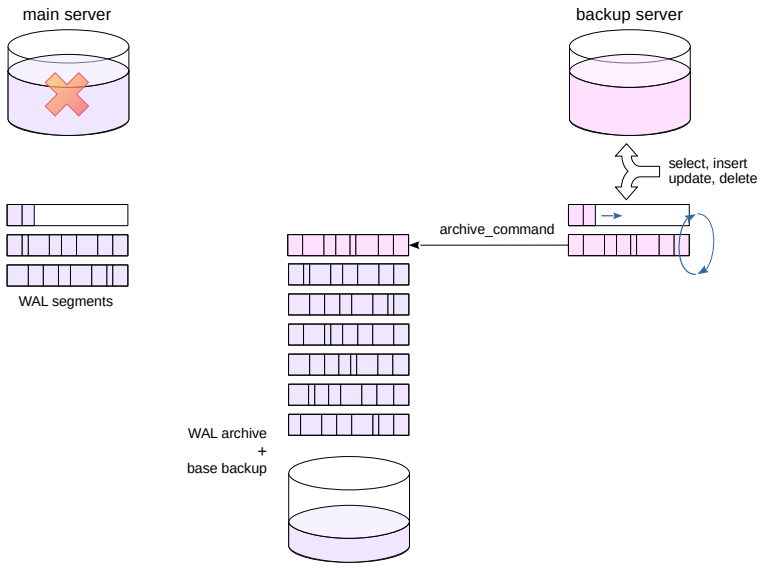
Recovery



The backup server reads WAL segments from the pg_wal directory and applies them (in the absence of a segment, making an attempt to copy it from the archive), ultimately bringing the state of the databases up to date. The maximum possible loss is the last unfilled WAL segment that has not been archived, and only if it cannot be copied manually for some reason.

By default, all available log entries are applied. If a target recovery point is specified, recovery will stop after reaching it.

Recovery



After that, the backup server goes into normal operation: processing incoming queries, archiving new WAL segments, and so on.

The recovered server can act as the primary server from now on, but in this case it should be deployed on sufficiently powerful hardware in the first place to avoid performance degradation.

Takeaways

Logical backup
creates SQL commands to recover the state of database objects

- copy command, pg_dump and pg_dumpall utilities

Physical backup
creates a copy of the cluster files + a set of WAL files

- pg_basebackup utility

WAL segments archive

- file or stream
- can restore the system to an arbitrary point in time

Practice

1. Create a database and a table in it with several rows.
2. Make a logical backup of the database using pg_dump.
   Delete the database and restore it from the backup you made.
3. Make an standalone physical backup of the cluster using pg_basebackup.
   Modify the table.
   Recover into a new cluster from the backup you made and verify that the recovered database does not contain any of the later changes.

Task 3. A replica cluster has already been created in the training VM on port 5433. Use this cluster to recover into.

The cluster directory is located at /var/lib/postgresql/15/replica.

To connect, specify the port number: psql -p 5433

Task 1. Database and table

```
=> CREATE DATABASE backup_overview;

CREATE DATABASE

=> \c backup_overview

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

=> CREATE TABLE t(n integer);

CREATE TABLE

=> INSERT INTO t VALUES (1), (2), (3);

INSERT 0 3
```

Task 2. Logical backup

Create a backup:

```
student$ pg_dump -f ~/backup_overview.dump -d backup_overview --create
```

Delete the database and restore it from the backup:

```
=> \c postgres

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

=> DROP DATABASE backup_overview;

DROP DATABASE

student$ psql -f ~/backup_overview.dump

SET
SET
SET
SET
SET
 set_config
------------

(1 row)

SET
SET
SET
SET
CREATE DATABASE
ALTER DATABASE
You are now connected to database "backup_overview" as user "student".
SET
SET
SET
SET
SET
 set_config
------------

(1 row)

SET
SET
SET
SET
SET
SET
CREATE TABLE
ALTER TABLE
COPY 3

=> \c backup_overview

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

=> SELECT * FROM t;
```

```
 n
---
 1
 2
 3
(3 rows)
```

Task 3. Physical standalone backup

Create a backup:

```
student$ sudo rm -rf /home/student/basebackup
```

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

Make sure that the second server is stopped, then push the backup:

```
student$ sudo pg_ctlcluster 15 replica status
```

```
pg_ctl: no server running
```

```
student$ sudo rm -rf /var/lib/postgresql/15/replica
```

```
student$ sudo mv /home/student/basebackup/ /var/lib/postgresql/15/replica
```

```
student$ sudo chown -R postgres:postgres /var/lib/postgresql/15/replica
```

Change the table:

```
=> DELETE FROM t;
```

```
DELETE 3
```

Start the server from backup:

```
student$ sudo pg_ctlcluster 15 replica start
```

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

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

```
 n
---
 1
 2
 3
(3 rows)
```