

Backup Base Backup



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.

What is Physical Backup

Cold and Pot Backup

Replication Protocol and Replication Slots

Cluster Cloning and Backup

Integrity Verification

Backup API

Physical base copy – copy of all cluster files

- + faster than logical backup
- + restores statistics
- can restore only to compatible systems and major PostgreSQL versions
- whole cluster backup only, no partial backups

Physical backup involves copying all files related to the database cluster (both data and service files), that is, creating a full binary copy.

Copying files is faster than dumping all SQL commands, as you do for a logical backup. Likewise, starting a server from a physical backup is a matter of minutes, while recovery from a logical backup takes much longer.

Statistical data is also restored from the physical copy, so you don't need to rebuild it.

But there are shortcomings. A physical backup can be used to restore the system only on a compatible platform (that has the same OS, architecture, etc.) with the same PostgreSQL major version installed.

Additionally, you cannot create a physical backup of individual databases within a cluster – only the entire cluster can be backed up.

Server is off

if the checkpoint is completed, you can just copy the files
the backup can be deployed on a different server, regardless of the PGDATA location

- + simplicity
- requires service interruption

File system snapshot

Similar to an unexpected shutdown: recovery required on startup, but the backup has all necessary WAL files

all data, including tablespaces, must be captured in a single snapshot

- + no server downtime required
- file systems with snapshot support are generally slower

A cold backup is made when the data on disk is in a consistent state: that is, all pages are stored on disk and are up to date.

Restoring from such a backup is straightforward: just extract the files, start the server, and it's good to go.

But to create such a backup, you must perform a checkpoint and a clean server shutdown first.

This is a major drawback: shutdowns can take a while, especially with large volumes of data.

The downtime can be reduced by running rsync (or a similar tool) beforehand while the server is still running. After the server is shut down, rsync will only need to copy the changes (which should be relatively few). However, some downtime is still unavoidable.

Another option is to make an inconsistent copy, which will include pages saved at different times. This inconsistency is inevitable when the server is shut down incorrectly or when creating a snapshot of the file system. In the latter case, all cluster files should fall into one snapshot. This may not be possible if the files are on different file systems.

When recovering from inconsistent data, the server will need to restore consistency at startup. This is the standard automatic crash recovery procedure. This isn't a problem since the necessary WAL files are guaranteed to be backed up, though the recovery process can take some time.

<https://postgrespro.com/docs/postgresql/16/backup-file>

Cold File-Based Backup

You can copy the files of a stopped cluster and start a second server with them. It doesn't matter whether the server was shut down cleanly or not.

Create a database and a table.

```
student$ psql
```

```
α=> CREATE DATABASE backup_base;
```

```
CREATE DATABASE
```

```
α=> \c backup_base
```

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

```
α=> CREATE TABLE t(s text);
```

```
CREATE TABLE
```

```
α=> INSERT INTO t VALUES ('This row was inserted on α...');
```

```
INSERT 0 1
```

Crash the server and copy the files to beta:

```
student$ sudo kill -QUIT $(sudo head -n 1 /var/lib/postgresql/16/alpha/postmaster.pid)
```

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

Cluster is not running.

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

```
student$ sudo cp -a /var/lib/postgresql/16/alpha /var/lib/postgresql/16/beta
```

Beta starts and restores consistency:

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

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

```
2026-09-16 23:54:12.254 MSK [70228] LOG:  database system was not properly shut down;
automatic recovery in progress
2026-09-16 23:54:12.272 MSK [70228] LOG:  redo starts at 0/1B82F10
2026-09-16 23:54:12.344 MSK [70228] LOG:  invalid record length at 0/1FB5138: expected at
least 24, got 0
2026-09-16 23:54:12.344 MSK [70228] LOG:  redo done at 0/1FB5110 system usage: CPU: user:
0.00 s, system: 0.00 s, elapsed: 0.07 s
2026-09-16 23:54:12.483 MSK [70226] LOG:  checkpoint starting: end-of-recovery immediate
wait
2026-09-16 23:54:15.888 MSK [70226] LOG:  checkpoint complete: wrote 932 buffers (5.7%);
0 WAL file(s) added, 0 removed, 0 recycled; write=0.041 s, sync=3.293 s, total=3.421 s;
sync files=308, longest=0.026 s, average=0.011 s; distance=4296 kB, estimate=4296 kB;
lsn=0/1FB5138, redo lsn=0/1FB5138
2026-09-16 23:54:15.913 MSK [70225] LOG:  database system is ready to accept connections
```

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

```
| β=> SELECT * FROM t;
```

```
|          s
|-----
| This row was inserted on α...
| (1 row)
```

Server is running

- pages are not consistent, as in case of incorrect shutdown
- pages may change while being copied
(the non-atomic read and write problem)
- restoring consistency requires all WAL records
from the last checkpoint till the backup completion
(some files may get deleted during backup)
- + no service interruption
- specialized tools required

Hot backup is performed on a running server, so it will probably contain inconsistent pages.

In addition to that, during backup, data is read directly from files rather than through the buffer cache. The contents of the files on disk obviously may change while the backup process is running, and there may not be a guarantee of the atomicity of reading/writing PostgreSQL pages (this depends on the file system and hardware. See "WAL Configuration" section of the DBA2 course for more). As a result, the backup will contain corrupted pages that can't be recovered even using WAL. This is the first challenge.

The second challenge is that copying data files can take a while. However, after completing a checkpoint, the server may delete some log files that are no longer needed for crash recovery but are still required for the backup.

Without proper safeguards, the resulting backup will be unusable for recovery. Hot backups therefore require specialized tools. PostgreSQL provides a low-level interface that enables reliable backup implementation. This interface is used by both the built-in `pg_basebackup` tool and various third-party backup solutions.

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

Replication Protocol



Purposes

backup and replication management
specifically, receiving a stream of WAL records

Handled by the wal_sender process

max_wal_senders = 10
wal_level = replica or logical, but not minimal

Connection

A role with the REPLICATION attribute or the SUPERUSER attribute
permission to connect granted in pg_hba.conf

7

PostgreSQL provides a specialized protocol for managing both backup and replication (covered in a dedicated module). Among other things, it facilitates receiving a stream of WAL records generated by the server.

On the server side, replication protocol connections are handled by the wal_sender process. It behaves like a regular backend process that starts when a client connects, but instead of SQL, it processes special replication commands. The number of concurrent processes (wal_sender) is limited by the *max_wal_senders* parameter (default is 10).

The WAL level must be set to replica or higher. On minimal WAL level, operations like CREATE TABLE AS SELECT, CREATE INDEX, and COPY FROM bypass WAL logging entirely: durability is ensured by writing data directly to disk rather than keeping it in RAM.

This is sufficient for crash recovery and recovery from a cold backup, but not for hot backup.

To use the replication protocol, the client must connect to the server using a role that has the REPLICATION attribute, or as a superuser. In pg_hba.conf, you need to allow this role to connect to the `replication` database (which is actually a keyword, not a database name).

Moreover, the permission for `all` is not enough, the connection to `replication` must be allowed explicitly.

<https://postgrespro.com/docs/postgresql/16/protocol-replication>

Replication slot

A server object that guarantees the retrieval of WAL records

- Tracks which record was last sent to the client

- The number of slots is limited by `max_replication_slots = 10`

Retains WAL segments until all records have been delivered to the client

- Allows the client to retrieve data at its own pace, including the ability to disconnect temporarily

- WAL files are accumulated on the server when the client is disconnected
- status monitoring: `pg_replication_slots`

You can use replication slots to prevent the server from deleting WAL files prematurely while a checkpoint is being created. When streaming WAL records through a replication slot, the slot keeps track of what records were delivered to the client, and will not let the server remove the WAL file until the client has received all records from it.

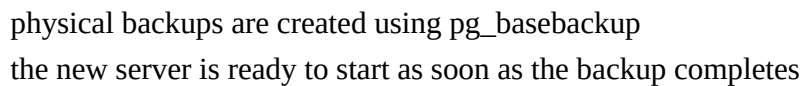
Using a slot means the client doesn't have to worry about the server deleting WAL files before they have been received properly. The client can even disconnect and then reconnect after some time to continue receiving WAL records from where it left off.

However, keep in mind that WAL files will accumulate on the server while the client is disconnected, so the server could run out of available disk space. Therefore, every created slot should be added to monitoring (the `pg_replication_slots` view) and slots that are no longer needed should be removed promptly.

The total number of slots that can be created is limited by the `max_replication_slots` configuration parameter (default is 10).

The use of slots is covered in more detail in the Replication module.

<https://postgrespro.com/docs/postgresql/16/warm-standby#STREAMING-REPLICATION-SLOTS>



9

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

Cluster Cloning

Now let's create a base backup of the running server alpha and use it to restore server beta.

```
student$ sudo pg_ctlcluster 16 alpha start
```

```
student$ psql backup_base
```

Permission for local connection via the replication protocol is already present in pg_hba.conf:

```
student$ psql -U postgres << EOF
SELECT type, database, user_name, address, auth_method
FROM pg_hba_file_rules() WHERE 'replication' = ANY(database);
EOF
```

type	database	user_name	address	auth_method
local	{replication}	{all}		trust
host	{replication}	{all}	127.0.0.1	scram-sha-256
host	{replication}	{all}	:::1	scram-sha-256

(3 rows)

For the pg_basebackup tool to connect to the server as the student role, this role must have the REPLICATION attribute:

```
α=> \du student
```

```

              List of roles
Role name | Attributes
-----+-----
student  | Create role, Create DB, Replication

```

Stop beta and delete its data directory.

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

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

Create a backup with a quick checkpoint. Data will be compressed on the server side and then unpacked with the tool. This speeds up network transfer, but has no effect if the connection is purely local. The option `--pgdata` points to the directory to save the backup to. The directory is created automatically. If it already exists, it must be empty.

```
postgres$ pg_basebackup \
--checkpoint=fast \
--compress=server-zstd \
--pgdata=/var/lib/postgresql/16/beta
```

The command runs with default values:

- `--format=plain`
- `--wal-method=stream`

Write-ahead log files have been copied to the pg_wal subdirectory:

```
postgres$ ls -l /var/lib/postgresql/16/beta/pg_wal
```

```
total 16388
-rw----- 1 postgres postgres 16777216 Sep 16 23:55 000000010000000000000002
drwx----- 2 postgres postgres   4096 Sep 16 23:54 archive_status
```

Data file verification

objective: to ensure that the backup receives consistent data

pg_basebackup and pg_checksums utilities

check relation file page checksums

Backup verification

objective: verify that the integrity of the data in the backup has not deteriorated over time

manifest file backup_manifest

pg_verifybackup utility checks files from the manifest list

Data can be corrupted both when writing to disk and during storage

It is imperative to verify the integrity of data as it is written, but also that this integrity is not compromised over time.

Relation file checksum calculation is enabled by default when a cluster is initialized (or when running pg_checksums utility). The checksum is stored in the page header and updates whenever the the page changes. The old checksum and the new one are compared:

- each time the page is read into the buffer cache;
- by pg_basebackup when creating a backup;
- by pg_checksums in verification mode.

pg_basebackup creates a file backup_manifest and puts it into the backup file

with information about files (names, sizes, calculated checksums) and log records (WAL start and end positions, time line), as well as the checksum of the manifest itself.

<https://postgrespro.com/docs/postgresql/16/backup-manifest-format>

The utility pg_verifybackup checks the backup against the manifest: the presence of files, the correctness of checksums, the ability to read and parse WAL records necessary for recovery.

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

You should still deploy the backup to make sure it works and check the health of the server, because the coincidence of checksums does not guarantee that the data was not damaged.

Integrity Verification

The backup includes a manifest file – here is its beginning and end:

```
postgres$ head -5 /var/lib/postgresql/16/beta/backup_manifest
```

```
{ "PostgreSQL-Backup-Manifest-Version": 1,
  "Files": [
    { "Path": "backup_label", "Size": 225, "Last-Modified": "2026-09-16 20:54:59 GMT",
      "Checksum-Algorithm": "CRC32C", "Checksum": "9a280c44" },
    { "Path": "postgresql.auto.conf", "Size": 88, "Last-Modified": "2026-07-15 20:46:12 GMT",
      "Checksum-Algorithm": "CRC32C", "Checksum": "536f950b" },
    { "Path": "pg_logical/replorigin_checkpoint", "Size": 8, "Last-Modified": "2026-09-16
20:54:59 GMT", "Checksum-Algorithm": "CRC32C", "Checksum": "c74b6748" },
```

```
postgres$ tail -6 /var/lib/postgresql/16/beta/backup_manifest
```

```
{ "Path": "global/pg_control", "Size": 8192, "Last-Modified": "2026-09-16 20:54:59 GMT",
  "Checksum-Algorithm": "CRC32C", "Checksum": "43872087" }
],
"WAL-Ranges": [
{ "Timeline": 1, "Start-LSN": "0/2000028", "End-LSN": "0/2000100" }
],
"Manifest-Checksum": "41199043083bdc941bdc51a4610953d3c0b903911aef0bd0d043f5dfe5981320"}
```

Verify the backup integrity using the pg_verifybackup tool:

```
postgres$ /usr/lib/postgresql/16/bin/pg_verifybackup /var/lib/postgresql/16/beta
```

backup successfully verified

The tool uses the manifest to verify that all files are present and their checksums are correct, and that all WAL records required for recovery exist and are readable.

Start the second server.

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

Now both servers run concurrently and independently. Check:

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

```
β=> INSERT INTO t VALUES ('...and this one on β.');
```

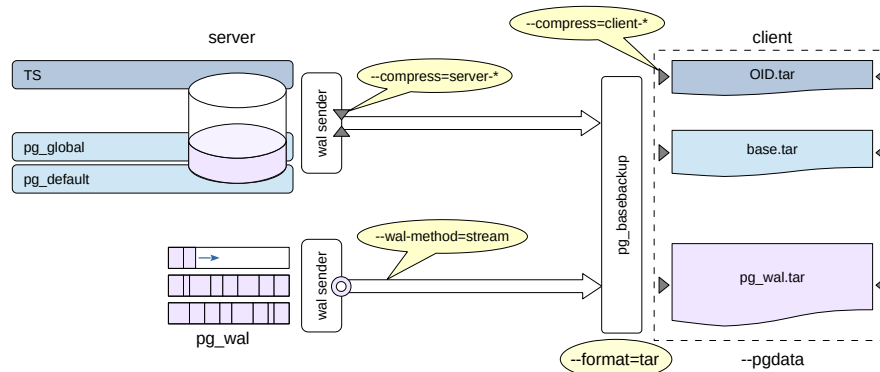
```
INSERT 0 1
```

```
β=> SELECT * FROM t;
```

```
      s
```

```
-----
This row was inserted on α...
...and this one on β.
(2 rows)
```

WAL Streaming Delivery



creating a backup for future use

The tar format is used for archive backups. WAL streaming delivery is used by default (wal-method=stream).

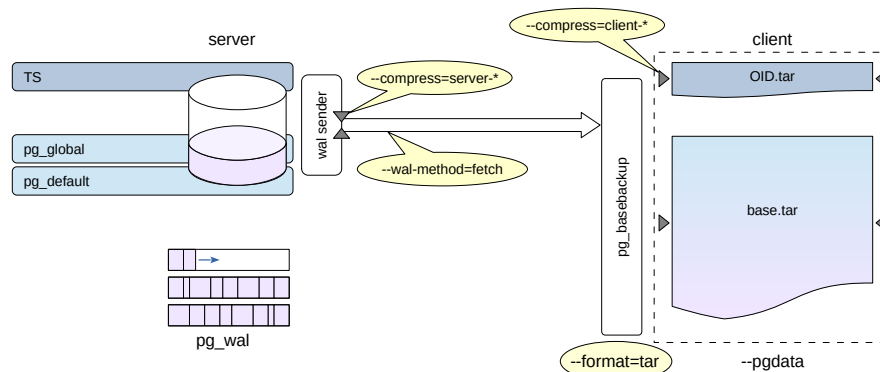
The cluster files will be saved as a base.tar file. Each additional tablespace is copied to a separate tar file with the same name as the tablespace OID. WAL segments are placed in a separate pg_wal.tar file.

The utility can perform compression both on the client side and on the server side (in the latter case, the pg_wal.tar file is not compressed).

A suffix corresponding to the compression method will be added to the names of compressed tar archives. For example, when compressed using the gzip method, the main file will be called base.tar.gz.

To restore from such a backup, you'll first need to extract the tar files to their proper locations. You can change the paths of tablespaces by editing the tablespace_map file before starting the server or using the --tablespace-mapping command line parameter.

WAL File Delivery to Client



creating a backup for future use

When creating a backup, you can use another mode – WAL file delivery (`--wal-method = fetch`).

In this case, no separate connection is created, and the WAL segments are copied and sent over the main connection. This happens after all cluster files have been backed up. During checkpoint creation, the server can delete WAL segments that are no longer needed for recovery. To prevent losing necessary files, set a sufficiently large value for the `wal_keep_size` parameter.

WAL segments are written to the `base.tar` file together with the rest of the cluster files.

The fetch mode is almost never used on the client. It remains as a legacy, and the only viable use case for it now is sending WAL records compressed – which is impossible in streaming mode.

Client-Side Backup

We are going to create a client-side backup. Compression will also be performed on the client.

```
student$ pg_basebackup \
--checkpoint=fast \
--compress=client-gzip:level=9 \
--format=tar \
--pgdata=/home/student/tmp/alpha-stream.backup
```

This command used streaming WAL delivery by default (--wal-method=stream).

Let's take a look at the resulting backup:

```
student$ ls -lh /home/student/tmp/alpha-stream.backup

total 5.2M
-rw----- 1 student student 220K Sep 16 23:55 backup_manifest
-rw----- 1 student student 5.0M Sep 16 23:55 base.tar.gz
-rw----- 1 student student 17K Sep 16 23:55 pg_wal.tar.gz
```

When streaming WAL records in tar format, a separate file is created for them:

```
student$ tar tf /home/student/tmp/alpha-stream.backup/pg_wal.tar.gz

00000001000000000000000004
```

Now let's create another client-side backup, but this time using the fetch delivery mode. In this mode, WAL segments are transferred over the same connection as the data. Let the server handle the compression.

```
student$ pg_basebackup \
--checkpoint=fast \
--wal-method=fetch \
--compress=server-gzip:level=9 \
--format=tar \
--pgdata=/home/student/tmp/alpha-fetch.backup
```

In the resulting backup, WAL segments are included in the base.tar.gz file.

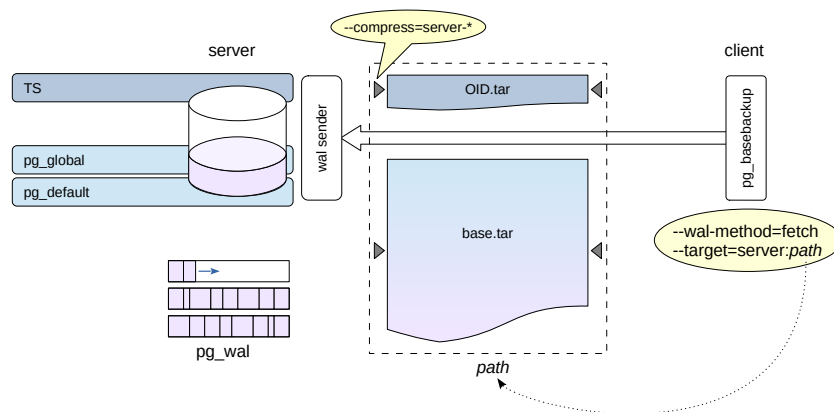
```
student$ ls -lh /home/student/tmp/alpha-fetch.backup

total 5.2M
-rw----- 1 student student 220K Sep 16 23:55 backup_manifest
-rw----- 1 student student 5.0M Sep 16 23:55 base.tar.gz

student$ tar tf /home/student/tmp/alpha-fetch.backup/base.tar.gz pg_wal

pg_wal/
pg_wal/00000001000000000000000005
pg_wal/archive_status/000000010000000000000005.done
```

WAL File Delivery to Server



creating a backup for future use

the backup is created on the server, nothing is sent over the network

You can create a backup on the server by running `pg_basebackup` on the client. No data is transmitted over the network.

To create a backup on the server from a client, use the command line parameter `pg_basebackup --target = server: path`, where *path* is the directory path

in the server file system (default is `--target=client`).

This mode supports compression, but only on the server side, naturally.

When creating a backup on the server, you cannot use WAL streaming delivery because no WAL records are sent to the client.

Therefore, the fetch delivery mode is used, and the accumulated WAL segments are placed in the `base.tar` file along with the rest of the cluster files.

Server-Side Backup

Run `pg_basebackup` on the client while saving the backup on the server. Here, we must use the fetch delivery mode, so set the `wal_keep_size` parameter (and hope the value will be sufficient):

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

```
ALTER SYSTEM
```

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

Run `pg_basebackup`, specifying the directory on the server where the backup will be stored, and set compression mode to `zstd`.

```
student$ pg_basebackup \
--checkpoint=fast \
--wal-method=fetch \
--target=server:/var/lib/postgresql/alpha.backup \
--compress=server-zstd:level=12
```

We get a compressed backup in tar format (which is the default).

```
postgres$ ls -lh /var/lib/postgresql/alpha.backup
```

```
total 4.2M
-rw----- 1 postgres postgres 220K Sep 16 23:55 backup_manifest
-rw----- 1 postgres postgres 4.0M Sep 16 23:55 base.tar.zst
```

With the fetch delivery mode, the utility stores WAL segments in the main archive:

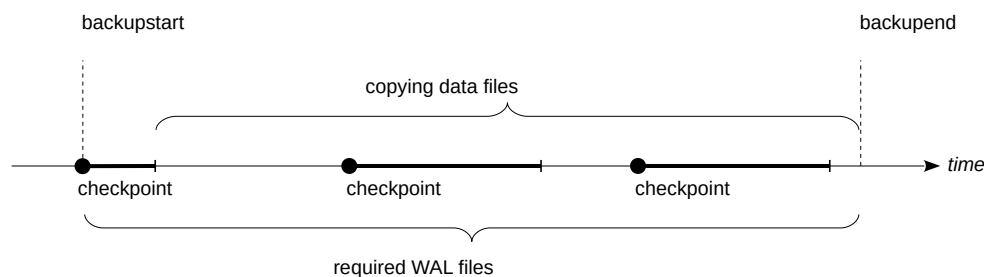
```
postgres$ tar tf /var/lib/postgresql/alpha.backup/base.tar.zst pg_wal
```

```
pg_wal/
pg_wal/00000001000000000000000000000007
pg_wal/archive_status/00000001000000000000000000000007.done
```

Delete the backup.

```
postgres$ rm -rf /var/lib/postgresql/alpha.backup
```

Low-Level Perspective



full_page_writes is set to on during the backup process

A checkpoint is performed at the very beginning (either fast or normal)

The low-level backup algorithm is the same whether you're using `pg_basebackup` or any third-party tools (which may be necessary since `pg_basebackup` only provides the most basic functionality).

1. You need to notify the server that a backup is starting.

During the backup process, the system first enables the *full_page_writes* parameter: whenever a page is modified for the first time after a checkpoint, its complete image is written to the WAL.

During recovery, WAL records will be applied not to pages in the file (due to non-atomic I/O, parts of the page may be mismatched), but to the page images from WAL.

Second, a checkpoint is performed. There are two checkpoint modes: fast (which can cause the I/O load to peak) and spread (controlled by the `checkpoint_completion_target` parameter).

2. Once the checkpoint completes, you can copy the data files using any method that works for you.

3. Once you've copied everything, you need to let the server know that the backup is complete.

4. Additionally, you need to ensure that all WAL records generated from the start of the backup until its completion are included in the backup.

Backup start: `pg_backup_start`

performs a checkpoint and sets *full_page_writes*
returns the WAL starting position

Backup end: `pg_backup_stop`

returns the WAL ending position

Copying files

either the replication protocol or any OS tools

Copying the WAL from the starting to ending position

replication protocol + slot or OS tools + *wal_keep_size*

You can use this backup algorithm to build your own custom backup utility.

The backup starts by calling the `pg_backup_start` function. It sets the *full_page_writes* parameter and performs a checkpoint, as discussed earlier. The function returns a specific position in the WAL. Records starting from that position must be included into the backup.

At the end of the backup, the `pg_backup_stop` function is called. It returns the final position in the WAL.

All WAL records between the starting and ending positions need to be included into the backup. Since the server may delete WAL files that aren't needed for recovery during operation, you should either use the replication protocol with a replication slot to receive records during the backup process (like `pg_basebackup` does in `--wal-method=stream` mode), or alternatively — though less convenient — copy the files using OS tools after setting the *wal_keep_size* parameter.

You can copy files in parallel, use `rsync`, and so on.

<https://postgrespro.com/docs/postgresql/16/continuous-archiving#BACKUP-LOWLEVEL-BASE-BACKUP>

A base backup is a complete copy of the database cluster, including tablespaces, along with the WAL needed to restore consistency

Hot backup allows the server to remain running during the backup process

Tools

- replication protocol
- replication slot
- low-level backup API

1. In the first cluster, create a tablespace and a database containing a table within that tablespace.
2. Create a compressed tar-format backup of the cluster using `pg_basebackup`.
3. Deploy a second cluster from this backup. Place the tablespace in a different directory by modifying the `tablespace_map` file.
4. Start the second server and verify that it's working properly.

After the practice is complete, delete the database and tablespace in both clusters.

1. Tablespace and Database

```
student$ sudo mkdir /var/lib/postgresql/ts_dir
```

```
student$ sudo chown postgres /var/lib/postgresql/ts_dir
```

Only a superuser can create tablespaces:

```
α=> \c - postgres
```

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

```
α=> CREATE TABLESPACE ts OWNER student LOCATION '/var/lib/postgresql/ts_dir';
```

```
CREATE TABLESPACE
```

```
α=> \c - student
```

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

```
α=> CREATE DATABASE backup_base;
```

```
CREATE DATABASE
```

```
α=> \c backup_base
```

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

```
α=> CREATE TABLE t(s text) TABLESPACE ts;
```

```
CREATE TABLE
```

```
α=> INSERT INTO t VALUES ('Hello, world!');
```

```
INSERT 0 1
```

2. Base Backup

Run pg_basebackup as the student with tar compression:

```
student$ pg_basebackup \
--checkpoint=fast \
--format=tar \
--compress=client-gzip \
--pgdata /home/student/tmp/backup
```

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

```
total 5352
-rw----- 1 student student    330 Sep 17 00:07 16409.tar.gz
-rw----- 1 student student 224520 Sep 17 00:07 backup_manifest
-rw----- 1 student student 5229317 Sep 17 00:07 base.tar.gz
-rw----- 1 student student   17101 Sep 17 00:07 pg_wal.tar.gz
```

3. Recovery from a Base Backup

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

Cluster is not running.

Clear the main data directory on the beta cluster. The directory must be accessible only by its owner.

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

```
student$ sudo mkdir /var/lib/postgresql/16/beta
```

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

```
student$ sudo chmod 700 /var/lib/postgresql/16/beta
```

The directory for the ts tablespace on the server that will be restored from the backup:

```
student$ sudo mkdir /var/lib/postgresql/ts_beta_dir
```

```
student$ sudo chown postgres /var/lib/postgresql/ts_beta_dir
```

Deploy the backup:

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

```
student$ sudo tar xf /home/student/tmp/backup/16409.tar.gz -C /var/lib/postgresql/ts_beta_dir
student$ sudo tar xf /home/student/tmp/backup/pg_wal.tar.gz -C /var/lib/postgresql/16/beta/pg_wal
```

The pg_tblspc directory is currently empty:

```
student$ sudo ls /var/lib/postgresql/16/beta/pg_tblspc/
```

The symbolic link will be created when the server starts, based on the tablespace_map file (which was located inside base.tar):

```
student$ sudo cat /var/lib/postgresql/16/beta/tablespace_map
```

```
16409 /var/lib/postgresql/ts_dir
```

Update the tablespace path in this file:

```
student$ sudo sed -i 's/ts_dir/ts_beta_dir/' /var/lib/postgresql/16/beta/tablespace_map
```

4. Startup and Verification

Start the server.

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

```
student$ sudo ls -l /var/lib/postgresql/16/beta/pg_tblspc/
```

```
total 0
lrwxrwxrwx 1 postgres root 31 Sep 17 00:07 16409 -> /var/lib/postgresql/ts_beta_dir
```

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

```
| β=> \c backup_base
|
| You are now connected to database "backup_base" as user "student".
|
| β=> SELECT * FROM t;
|
|          s
|-----
| Hello, world!
| (1 row)
```

Cleanup

Delete the backup.

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

To delete the database, you must first disconnect from it.

```
α=> \c student
```

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

```
α=> DROP DATABASE backup_base;
```

```
DROP DATABASE
```

```
α=> DROP TABLESPACE ts;
```

```
DROP TABLESPACE
```

```
student$ sudo rm -rf /var/lib/postgresql/ts_dir
```

And on the second server:

```
| β=> \c student
|
| You are now connected to database "student" as user "student".
|
| β=> DROP DATABASE backup_base;
|
| DROP DATABASE
|
| β=> DROP TABLESPACE ts;
|
| DROP TABLESPACE
```

```
student$ sudo rm -rf /var/lib/postgresql/ts_beta_dir
```

1. Load the `basebackup_to_shell` module to create a base backup on the server using a user-defined command. This command should compress the tar backup using the `bzip2` utility.
2. Back up the server in tar format. Make sure that the archive is compressed by `bzip2`.
3. Restore the beta cluster from the resulting compressed archive and verify that it works.

1. To load the module, add the library of the same name to the `shared_preload_libraries` parameter. With the `basebackup_to_shell.command` parameter, specify the command to run when backing up:

```
bzip2 -c> archive.tar.bz2
```

<https://postgrespro.com/docs/postgresql/16/basebackup-to-shell>

2. To load the module for backup, specify

```
pg_basebackup --target=shell
```

3. To extract files from the tar archive compressed by the `bzip2` utility, use the command:

```
tar xjf archive.tar.bz2 -C destination directory
```

After completing the practice, delete the created databases and the backup.

1. Loading the Module

Create a configuration for the basebackup_to_shell module in the configuration file.

```
postgres$ cat << EOF > /etc/postgresql/16/alpha/conf.d/basebackup_to_shell.conf
shared_preload_libraries = basebackup_to_shell
basebackup_to_shell.command = 'bzip2 -c > /var/lib/postgresql/archive.backup/%f.bz2'
EOF
```

Restart the server.

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

Check the configuration.

```
student$ psql
```

```
α=> SHOW shared_preload_libraries;
```

```
shared_preload_libraries
-----
basebackup_to_shell
(1 row)
```

```
α=> SHOW basebackup_to_shell.command;
```

```
basebackup_to_shell.command
-----
bzip2 -c > /var/lib/postgresql/archive.backup/%f.bz2
(1 row)
```

Create a database and a table in it.

```
α=> CREATE DATABASE backup_base;
```

```
CREATE DATABASE
```

```
α=> \c backup_base
```

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

```
α=> CREATE TABLE t AS SELECT now() AS createtime;
```

```
SELECT 1
```

2. Backup

Create a directory for the backup.

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

Perform the backup. Streaming mode is not available, so use the WAL fetch delivery mode.

```
student$ pg_basebackup \
--checkpoint=fast \
--wal-method=fetch \
--target=shell
```

```
postgres$ ls -l /var/lib/postgresql/archive.backup
```

```
total 4440
-rw----- 1 postgres postgres 9927 Sep 17 00:07 backup_manifest.bz2
-rw----- 1 postgres postgres 4531299 Sep 17 00:07 base.tar.bz2
```

3. Recovery from Backup

Delete the beta cluster files.

```
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
```

Extract the files from the archive.

```
postgres$ tar xjf /var/lib/postgresql/archive.backup/base.tar.bz2 -C /var/lib/postgresql/16/beta
```

Check whether the data has been restored.

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

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

```
β=> SELECT * FROM t;
```

```
      creatime
-----
2026-09-17 00:07:40.876136+03
(1 row)
```

Cleanup

```
α=> \c student
```

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

```
α=> DROP DATABASE backup_base;
```

DROP DATABASE

```
β=> \c student
```

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

```
β=> DROP DATABASE backup_base;
```

DROP DATABASE

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

```
postgres$ rm -rf /var/lib/postgresql/archive.backup
```