04 Physical Backups File System Level

01 What Is a Physical Backup in PostgreSQL

In simple words

- A physical backup is a **byte-by-byte copy of PostgreSQL's data files**.
- It does not rebuild the database using SQL.
- It **clones the database exactly as it exists on disk**.
- This is why physical backups restore much faster than logical backups.

Why physical backups exist

Logical backups rebuild databases. That is slow for large systems.

Physical backups exist to:

- restore very fast
- preserve exact on-disk state
- support point-in-time recovery (PITR)
- handle large databases reliably

At scale, physical backups become mandatory.

What a physical backup actually includes

A physical backup copies:

- table and index files
- system catalogs
- visibility maps and FSM
- control files
- required WAL files

Everything under `$PGDATA` matters.

This is a **cluster-level backup**, not database-level.

What physical backups do NOT include

Physical backups do not include:

- OS packages
- PostgreSQL config outside PGDATA (sometimes)
- external scripts
- monitoring tools

DBAs must back these up separately if needed.

Offline vs online physical backups

Offline physical backup

- PostgreSQL is stopped
- files are copied
- consistency is guaranteed

This is simple but causes downtime.

Online physical backup

- PostgreSQL keeps running
- files are copied while users work
- WAL ensures consistency

This avoids downtime but needs careful planning.

The role of WAL in physical backups

When PostgreSQL runs:

- data pages may be half-written
- files can be inconsistent during copy

WAL solves this.

During restore:

- PostgreSQL replays WAL
- fixes partial writes
- reaches a consistent state

Without WAL, online physical backups are unusable.

Common tools for physical backups

- `pg_basebackup`
- filesystem snapshot tools (LVM, cloud snapshots)
- custom rsync-based scripts (carefully)

Each tool relies on WAL for safety.

Why physical backups are fast to restore

Restore steps:

- place files back into PGDATA
- start PostgreSQL
- replay WAL

No table rebuilds. No index recreation.

Speed is the biggest advantage.

Limitations of physical backups

Physical backups:

- must match PostgreSQL major version
- require similar architecture
- cannot restore single tables

They trade flexibility for speed.

When I use physical backups

I use physical backups when:

- database is large
- fast recovery is required
- PITR is needed
- downtime must be minimal

They are the backbone of production recovery.

Final mental model

- Physical backup = exact clone
- WAL = safety net
- Restore = file copy + WAL replay
- Speed beats flexibility

One-line explanation

A physical backup copies PostgreSQL's data files directly and restores them quickly using WAL replay for consistency.

02 PGDATA Directory Structure (What Actually Gets Backed Up)

In simple words

- $PGDATA is the **heart of PostgreSQL**.
- Every physical backup is basically a copy of this directory.
- If you don't understand what lives here, physical backups will always feel risky.

What PGDATA represents

PGDATA is the directory where PostgreSQL stores:

- all databases
- system catalogs
- transaction metadata
- WAL files (or links)

When PostgreSQL starts, it reads PGDATA first.

High-level layout of `PGDATA`

Inside `PGDATA`, you will usually see:

- `base/`
- `global/`
- `pg_wal/`
- `pg_multixact/`
- `pg_xact/`
- `pg_commit_ts/`
- `pg_tblspc/`
- `postgresql.conf`
- `pg_hba.conf`
- `pg_ident.conf`

Each directory has a very specific job.

`base/` directory (user databases)

This directory contains **actual table and index files**.

- each database has its own subdirectory
- filenames are numeric OIDs
- data is stored in 8KB pages

This is where most disk space is consumed.

`global/` directory (cluster metadata)

This stores cluster-wide information:

- roles
- databases list
- shared system catalogs

If `global/` is missing or corrupted:

- PostgreSQL will not start

`pg_wal/` directory (write-ahead log)

This is the **most critical directory for recovery**.

It stores WAL segments that:

- record every data change
- ensure crash safety
- enable PITR

If `pg_wal` fills up:

- database can stop accepting writes

`pg_xact/` (transaction status)

Tracks:

- committed transactions
- aborted transactions

PostgreSQL uses this to decide which rows are visible.

Missing or corrupted pg_xact leads to data inconsistency.

`pg_multixact/`

Used when:

- multiple transactions lock the same row

Common in systems with heavy concurrent updates.

This directory must be included in physical backups.

`pg_commit_ts/`

- Stores commit timestamps (if enabled).
- Not always active, but must be backed up if present.

`pg_tblspc/` (tablespaces)

Contains symbolic links to tablespaces located outside `PGDATA`.

Important rule:

- Backing up `PGDATA` alone is NOT enough when tablespaces exist.
- Tablespace directories must be backed up separately.

Configuration files inside `PGDATA`

Usually includes:

- `postgresql.conf`
- `pg_hba.conf`
- `pg_ident.conf`

Depending on setup, these may be outside `PGDATA`.

Do not assume configs are always included in backups.

Files you should never touch

Never manually edit:

- files inside `base/`
- WAL files
- transaction metadata

PostgreSQL expects full control over these.

Common DBA mistake

Copying only `base/` and ignoring:

- `global/`
- `pg_wal/`
- `pg_xact/`

This leads to broken restores.

Final mental model

- `PGDATA` = database brain
- `base` = user data
- `pg_wal` = recovery engine
- `global` = cluster identity
- tablespaces need extra care

One-line explanation

`PGDATA` contains all PostgreSQL data files, WAL, and metadata required for physical backup and recovery.

03 Offline Filesystem Backup – Step by Step (PostgreSQL)

In simple words

An offline filesystem backup means:

- PostgreSQL is **completely stopped**
- no users, no writes, no WAL activity
- data files are copied in a stable state

This is the **safest and simplest** form of physical backup.

Why offline backups still matter

Even though online backups exist, offline backups are still used when:

- database is small or medium
- maintenance window is available
- absolute safety is required
- environment is simple

With PostgreSQL stopped, there is zero consistency risk.

What makes offline backup safe

When PostgreSQL is stopped:

- no transactions are running
- no dirty buffers exist
- no WAL is being generated
- files are internally consistent

This removes the need for WAL replay during restore.

Step-by-step offline backup process

Step 1: Notify users and applications

Before stopping PostgreSQL:

- inform application teams
- stop background jobs
- ensure no active connections

Never stop PostgreSQL blindly in production.

Step 2: Stop PostgreSQL cleanly

```
sudo systemctl stop postgresql
```

Or:

```
pg_ctl stop -D $PGDATA
```

Verify:

```
ps aux | grep postgres
```

No postgres process should be running.

Step 3: Copy the data directory

```
cp -a $PGDATA /backup/pgdata_backup
```

Important:

- use recursive copy
- preserve ownership and permissions
- include hidden files

If tablespaces exist, back them up separately.

Step 4: Verify backup completeness

Check:

- directory size
- number of files
- backup logs

A half-copied backup is dangerous.

Step 5: Start PostgreSQL again

```
sudo systemctl start postgresql
```

Verify:

- database starts normally
- applications reconnect

Downtime ends here.

Restoring from an offline backup

Restore process:

- stop PostgreSQL
- replace PGDATA with backup copy
- ensure ownership and permissions
- start PostgreSQL

No WAL replay is needed.

Handling tablespaces

If tablespaces exist:

- copy tablespace directories separately
- restore them to the same paths
- ensure symlinks in pg_tblspc are intact

Missing tablespaces cause startup failure.

Common mistakes during offline backup

- copying data while PostgreSQL is still running
- forgetting tablespaces
- insufficient disk space
- wrong file permissions

Most failures are operational errors.

Pros and cons of offline backups

Pros:

- simplest method
- highest safety
- easiest restore

Cons:

- requires downtime
- not suitable for 24x7 systems

When I choose offline backup

I choose offline backup when:

- database is non-critical
- downtime is acceptable
- environment is small
- simplicity matters more than availability

Final mental model

- Offline = stop DB, copy files
- Zero consistency risk
- Downtime is the cost
- Restore is simple

One-line explanation

An offline filesystem backup copies PostgreSQL data files after stopping the server, ensuring maximum consistency at the cost of downtime.

04 Snapshot-Based Backups in PostgreSQL (LVM / Cloud Snapshots)

In simple words

Snapshot-based backup means:

- I take a **filesystem or storage snapshot**
- instead of manually copying files

The snapshot freezes disk state instantly.

PostgreSQL keeps running.

This gives **fast backups with very low downtime** — if done correctly.

Why snapshot-based backups exist

Copying large data directories takes time.

Stopping PostgreSQL is often not acceptable.

Snapshots exist to:

- freeze disk state in seconds
- reduce downtime to near-zero
- back up very large databases

This is common in enterprise and cloud setups.

Types of snapshots used

Snapshot backups are usually taken at:

- LVM level (on Linux)
- Cloud storage level (AWS EBS, Azure Disk, GCP PD)
- Enterprise storage arrays

PostgreSQL does not create these snapshots itself.

It cooperates with them.

The BIG misconception (very important)

A filesystem snapshot alone is **NOT** enough.

If PostgreSQL is writing data while snapshot is taken:

- files can be inconsistent
- restore may fail

Snapshots must be coordinated with PostgreSQL.

Safe snapshot workflow (correct way)

Step 1: Force WAL consistency

Before snapshot:

```
SELECT pg_start_backup('snapshot_backup');
```

This tells PostgreSQL:

- I am about to take a filesystem snapshot
- make sure WAL protects all in-flight changes

Step 2: Take filesystem snapshot

At OS or cloud level:

- create snapshot of all data volumes
- include tablespaces if present

This operation is usually instant.

Step 3: End backup mode

After snapshot:

```
SELECT pg_stop_backup();
```

This releases WAL pressure and marks snapshot complete.

What pg_start_backup / pg_stop_backup actually do

They do NOT stop writes.

They ensure:

- full-page writes are enabled
- WAL contains enough data to fix inconsistencies
- restore can recover safely

This is why WAL size often increases during snapshot backups.

Restore from snapshot backup

Restore process:

- attach snapshot volume to server
- mount filesystem
- place data directory back
- start PostgreSQL
- WAL replay fixes partial pages

Restoring is usually fast.

Tablespaces and snapshots

If tablespaces exist:

- snapshot **every volume**
- snapshot them **at the same time**

Missing or mismatched snapshots cause restore failure.

Common snapshot mistakes

- taking snapshot without `pg_start_backup`
- forgetting tablespace volumes
- restoring without required WAL
- assuming crash recovery is enough

These mistakes lead to silent corruption.

Snapshot vs `pg_basebackup`

Snapshots:

- extremely fast
- storage dependent
- more operational risk

`pg_basebackup`:

- slower
- PostgreSQL-managed
- safer and simpler

Senior DBAs choose based on environment maturity.

When I use snapshot-based backups

I use them when:

- database is very large
- downtime must be minimal
- storage supports snapshots well
- WAL archiving is reliable

Snapshots demand discipline.

Final mental model

- Snapshot freezes disk, not PostgreSQL
- WAL ensures consistency
- Coordination is mandatory
- Speed comes with responsibility

One-line explanation

Snapshot-based backups use filesystem or storage snapshots coordinated with PostgreSQL WAL to enable fast, low-downtime physical backups.

05 WAL Requirement and Consistency in Physical Backups

In simple words

- WAL (Write-Ahead Log) is the **safety net** of PostgreSQL.
- Without WAL, physical backups are unreliable.
- With WAL, PostgreSQL can repair half-written pages and reach a consistent state after restore.

If you remember one thing:

> **Physical backup without WAL is incomplete.**

Why WAL exists

PGSQL never writes data pages directly and blindly.

Flow:

- change is written to WAL first
- WAL is flushed to disk
- data pages are written later

This guarantees crash safety and recovery.

Why WAL is critical for physical backups

During online physical backups:

- files are copied while PGSQL is running
- some pages may be copied mid-write
- data directory snapshot may look inconsistent

WAL allows PGSQL to:

- replay changes
- fix partial writes
- make the backup usable

Without WAL, restore may fail or corrupt data silently.

What “consistency” really means

Consistency means:

- all committed transactions are present
- no partial transactions exist
- database behaves like a real point in time

WAL is what enforces this during restore.

WAL and offline physical backups

When PostgreSQL is stopped:

- no writes happen
- data files are consistent

In this case:

- WAL replay is minimal
- offline backup is naturally consistent

Still, WAL files are usually included for safety.

WAL and online physical backups

For online backups:

- WAL is mandatory
- PostgreSQL increases WAL generation
- full-page writes protect torn pages

Restoring without required WAL segments will fail.

Required WAL for restore

To restore a physical backup, PGSQL needs:

- WAL up to the backup end
- WAL required for crash recovery

Missing WAL leads to:

- startup failure
- recovery abort

WAL retention during backup

During backup:

- PGSQL prevents WAL removal
- WAL accumulates until backup completes

If disk space is insufficient:

- WAL directory can fill
- database may stop

Monitoring WAL size is mandatory.

WAL archiving and physical backups

In production:

- WAL archiving is usually enabled
- archived WALs support PITR

Physical backups + archived WAL = full recovery chain.

Common WAL-related mistakes

- deleting WAL files manually
- underestimating WAL growth during backup
- assuming snapshots don't need WAL
- restoring backup without matching WAL timeline

These cause real outages.

DBA best practices

- never touch `pg_wal` manually
- monitor WAL growth during backups
- ensure archive destination has space
- test restore with WAL replay

Final mental model

- WAL = change history
- Physical backup = file snapshot
- Restore = file copy + WAL replay
- Consistency comes from WAL

One-line explanation

WAL ensures physical backups can be restored to a consistent state by replaying changes and fixing incomplete writes.

06 Tablespaces and Multi-Filesystem Risks in PostgreSQL Backups

In simple words

- Tablespaces allow PGSQL to store data **outside the main `PGDATA` directory**.
- This improves flexibility and performance, but it **greatly increases backup risk** if not handled carefully.
- Most broken physical backups involve tablespaces.

What a tablespace really is

A tablespace is:

- a directory on disk
- located outside `PGDATA`
- linked internally via `pg_tblspc`

PGDQL uses tablespaces to:

- spread I/O across disks
- store large tables separately
- manage storage growth

Why tablespaces complicate backups

When tablespaces exist:

- data is spread across multiple filesystems
- `PGDATA` alone is incomplete
- restoring only `PGDATA` breaks the database

Every tablespace directory is part of the physical backup.

How PostgreSQL tracks tablespaces

Inside PGDATA:

- `pg_tblspc` contains symbolic links
- links point to external directories

If these directories are missing during restore:

- PostgreSQL fails to start

Offline backup with tablespaces

For offline backups:

- stop PGSQL
- copy `PGDATA`
- copy **all tablespace directories**

Restore requires:

- same directory paths
- same ownership and permissions

Missing any tablespace is fatal.

Online backup and tablespaces

For online backups:

- WAL must cover all tablespace writes
- snapshot or base backup must include every filesystem

Partial snapshots across filesystems cause corruption.

Snapshot backups and ordering risk

Taking snapshots one volume at a time is dangerous.

If:

- `PGDATA` is snapshotted first
- tablespace disks snapshot later

Then:

- internal references mismatch
- restore may fail or corrupt

Snapshots must be:

- coordinated
- taken at the same moment

A **snapshot backup** is a quick capture of data exactly as it looks at one moment in time. Instead of copying files, the storage system freezes the state and takes an instant point-in-time copy.

A **volume** is a storage unit, like a disk or partition, where data lives. PGSQL can use multiple volumes, for example one for `PGDATA` and others for tablespaces.

The key idea is this: snapshots work at the volume level, not at the database level. If PGSQL data is spread across multiple volumes, all of them must be snapshotted at the same time, or the backup becomes unsafe.

Snapshot backups are risky when volumes are snapshotted one by one instead of together. If `PGDATA` is captured first and tablespace disks are snapshotted later, internal references can go out of sync, which can cause restore failures or silent corruption. To be safe, snapshots must always be coordinated and taken at the exact same moment.

Cloud snapshot pitfalls

In cloud environments:

- volumes are snapshotted independently
- snapshot timing differences matter

DBAs must ensure:

- all volumes are frozen together
- PostgreSQL backup mode is active

Cloud convenience hides real risk.

- In cloud environments, snapshots are taken per volume, and each volume is snapshotted independently. Even small timing differences between these snapshots can break database consistency. That's why a DBA must make sure all volumes are frozen together and PostgreSQL backup mode is active during the snapshot. Cloud platforms make snapshots look easy, but that convenience hides real and serious risk if not handled properly.

Common DBA mistakes

- forgetting to back up tablespaces
- assuming `pg_basebackup` includes external mounts automatically
- restoring tablespaces to wrong paths
- mixing snapshots from different times

These mistakes surface only during restore.

Best practices for tablespace safety

- document all tablespaces
- standardize mount points
- include tablespaces in backup scripts
- test restore with tablespaces present

Tablespaces require discipline.

Final mental model

- Tablespaces live outside `PGDATA`
- Backups must include every filesystem
- Snapshots must be coordinated
- Restore paths must match

One-line explanation

Tablespaces store PostgreSQL data outside `PGDATA`, requiring coordinated backups across multiple filesystems to avoid restore failures.

07 Physical Backup vs Logical Backup (DBA Perspective)

In simple words

- Physical and logical backups solve **different problems**.
- Logical backups rebuild the database using SQL.
- Physical backups clone the database using files.
- A good DBA does not choose one.
- They design a strategy using **both**.

Core idea difference

Logical backup:

- rebuilds database objects
- works at SQL level
- portable and flexible

Physical backup:

- copies data files
- works at filesystem level
- fast and exact

This difference drives every decision.

Backup speed comparison

Logical backup:

- reads data row by row
- generates SQL statements
- slower for large databases

Physical backup:

- copies files sequentially
- much faster on large data

Backup time matters, but restore time matters more.

Restore speed comparison

Logical restore:

- executes SQL statements
- rebuilds indexes
- may take hours on large DBs

Physical restore:

- places files back
- replays WAL
- completes much faster

This is why production recovery prefers physical backups.

Flexibility vs rigidity

Logical backup:

- restore single table or schema
- migrate across versions
- usable across platforms

Physical backup:

- restore full cluster only
- same PostgreSQL version required
- same architecture expected

Flexibility costs time.

Speed costs flexibility.

Impact on production systems

Logical backups:

- generate heavy read I/O
- can slow queries
- usually safe but slow

Physical backups:

- also I/O heavy
- faster completion
- WAL growth must be monitored

Both must be scheduled carefully.

Use cases in real life

I use logical backups when:

- migrating databases
- upgrading PostgreSQL versions
- restoring specific objects

I use physical backups when:

- database is large
- fast recovery is required
- point-in-time recovery is needed

Real systems use both simultaneously.

Disaster recovery strategy

Logical backup alone:

- slow recovery
- long downtime

Physical backup alone:

- less flexible
- not suitable for migrations

Best strategy:

- physical backup + WAL for recovery
- logical backup for flexibility and audits

Common wrong thinking

Asking:

> "Which backup is better?"

Correct question:

> "Which backup fits this recovery scenario?"

DBA decisions are scenario-driven.

Final mental model

- Logical = rebuild
- Physical = clone
- Flexibility vs speed trade-off
- Strategy > tool

Physical vs Logical Backup - Difference Table

Aspect	Logical Backup	Physical Backup
Core idea	Rebuilds the database using SQL commands	Clones the database using data files
Backup level	SQL / object level	Filesystem / block level
What it copies	Tables, data, indexes as SQL	Actual data files, WAL, catalogs
Backup speed	Slow on large databases	Very fast on large databases
Restore method	Executes SQL again	Copies files back + WAL replay
Restore speed	Slow, can take hours	Much faster
Flexibility	Can restore single table or schema	Full cluster restore only
Portability	Works across versions and platforms	Same version & architecture needed
Impact on production	Heavy read load for long time	Heavy I/O but finishes faster
Best use cases	Migrations, upgrades, object-level restore	Large DBs, fast recovery, PITR
Disaster recovery	Slow recovery, long downtime	Fast recovery, less flexible
Mental model	Rebuild	Clone

One-line explanation

Logical backups rebuild PostgreSQL databases using SQL for flexibility, while physical backups clone data files for fast and reliable recovery.

08 Common Physical Backup Mistakes in PostgreSQL (Real-World Failures)

In simple words

- Physical backups are powerful, but also dangerous when done casually.
- Most PostgreSQL disaster stories happen not because physical backups are bad, but because **DBAs misunderstood or skipped critical steps**.
- This file lists the mistakes that actually cause data loss.

Mistake 1: Copying PGDATA while PostgreSQL is running

Some DBAs assume:

> "`cp -r $PGDATA` is enough"

If PGSQL is running:

- files are changing
- pages may be half-written
- backup becomes inconsistent

This backup may restore but corrupt data silently.

Correct approach:

- stop PostgreSQL
- or use WAL-aware methods

Mistake 2: Ignoring WAL during online backups

Online physical backups **require WAL**.

Common wrong assumptions:

- snapshot alone is enough
- crash recovery will fix everything

Without required WAL:

- restore fails
- or data corruption occurs

Never take online physical backups without WAL planning.

Mistake 3: Running out of disk due to WAL growth

During backup:

- WAL retention increases
- archived WAL piles up

If disk fills:

- database may stop
- writes fail
- replication breaks

Monitoring WAL size during backups is mandatory.

Mistake 4: Forgetting tablespaces

Backing up only `PGDATA` while tablespaces exist:

- misses real data
- breaks restore

This mistake is extremely common.

Correct approach:

- always inventory tablespaces
- back up all tablespace paths

Mistake 5: Inconsistent snapshots across filesystems

Snapshots taken at different times:

- PGDATA snapshot now
- tablespace snapshot later

This creates mismatched states.

Restore may succeed but data will be wrong.

Snapshots must be coordinated.

Mistake 6: Restoring to different paths or permissions

Physical restore expects:

- same directory layout
- correct ownership
- correct permissions

Wrong paths or ownership:

- PostgreSQL refuses to start
- recovery fails

Always match the original environment.

Mistake 7: Mixing PostgreSQL versions

Physical backups are version-specific.

Restoring:

- PG 14 backup into PG 15
- will fail

Upgrades require logical backups or pg_upgrade.

Mistake 8: No restore testing

Many teams:

- take physical backups daily
- never test restore

Until the day recovery is needed.

Physical backups must be tested, especially with WAL replay.

Mistake 9: Deleting WAL files manually

Deleting WAL to free space:

- breaks recovery chain
- makes backups unusable

WAL cleanup must be automated and controlled.

Mistake 10: Overconfidence in automation

Automation hides failures.

If:

- scripts fail silently
- snapshot does not complete

You think backups exist, but they don't.

Automation must include verification.

Final mental model

- Physical backups are unforgiving
- WAL and tablespaces are critical
- Snapshots need coordination
- Testing is non-negotiable

One-line explanation

Most physical backup failures in PostgreSQL occur due to WAL mishandling, missing tablespaces, inconsistent snapshots, or untested restore processes.