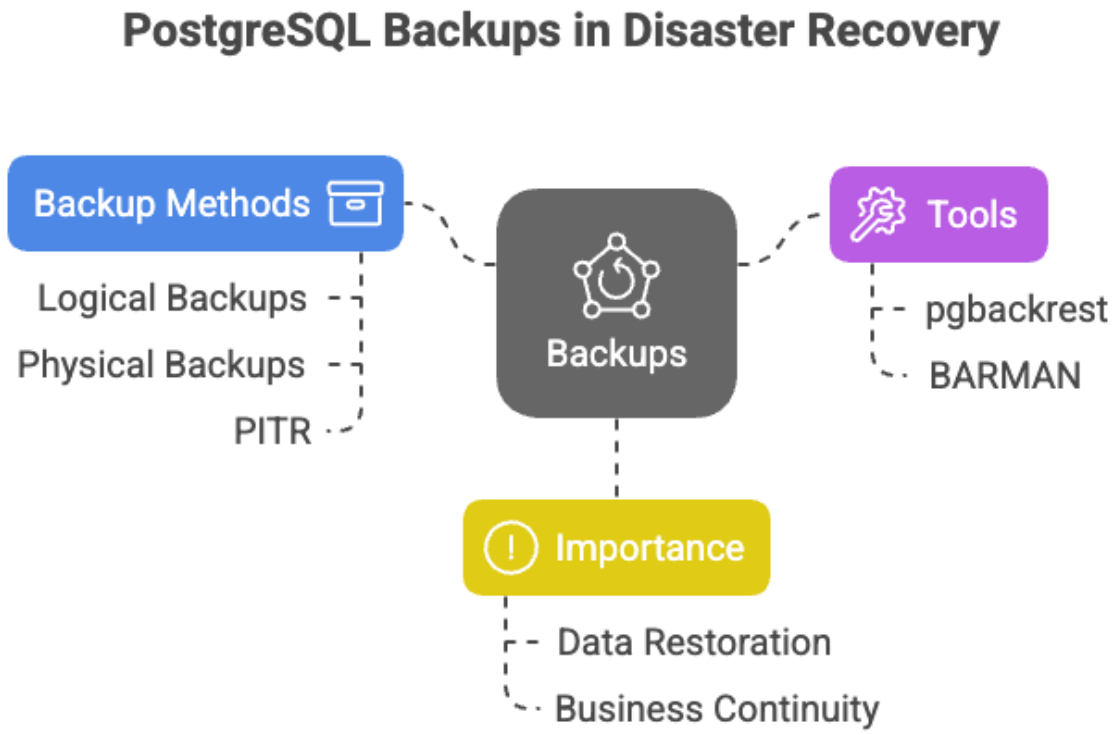
MS SQL DATABASE ADMINISTRATOR 30 DAYS COURSE

BY: Mukesh Chaurasia

DAY – 4

TOPIC: POSTGRES BACKUP, RESTORE AND DISASTER RECOVERY TECHNIQUES



MIND MAP

PostgreSQL Backup & Point-In-Time Recovery (PITR)

- **Why Backups are Needed**
  - Consistent Copy of Data
  - Avoid Data Loss
  - Restore Database
  - Avoid Catastrophic Losses
  - Compliance Reasons
  - Meet RPO/RTO
  - Protect Business Reputation
  - Causes of Data Loss
    - User Error
    - Hardware failure
    - Data Corruption
- **Concept Overview**
  - Definition

        - Restore to specific moment in time
    - Requirements
        - Backing up live database files
        - Archiving write ahead log (WAL)
    - WAL (write ahead log)
        - Log of all modification
        - Update, insert, deletes
        - Creating/deleting objects
    - Limitation/Scope
        - Can not upgrade database version
        - Only whole database cluster backup
        - Not individual databases/tables
    - Advantages
        - Can be done while database is online
- **Backup Steps for PITR**
    - Enable WAL Archiving
        - Modify postgresql.conf
        - Set 'wal_level' to 'replica' (default)
        - Set 'archive_mode' to 'bin'
        - Specify 'archive_command'
    - Manage WAL Files
        - Develop retention process
        - Periodic deletion
        - Copying to external storage (e.g 53)
    - Backup Live Database Files
        - Use 'pg_basebackup' (easiest way)
        - Other methods exist
    - Retain backup & WAL Files
        - Reply database to specific point in time
    - Manual Log Switch (optional)
        - Use pg_switch_wal
        - Use pg_switch_xlog
    - pg_basebackup options
        - Format: 'tar'
        - Stream WAL files (default in PG 10+)
        - Disable WAL streaming (-X non)
        - Compression (- | gzip)
- **Restore Steps for PITR**
    - Stop Database Cluster
        - Use systemctl stop postgresql

  - Destroy Existing Data
    - Remove all files from datadirectory (e.g /var/lib/postgresql/10/main)
  - Restore Cluster Files
    - Extract base.tar to new data directory
  - Restore WAL Files
    - Extract pg_wal.tar (pg 10+) or pg_xlog.tar to pg_wal directory
    - Copy archived WAL files to archive log directory
  - Create recovery.conf
    - Place in cluster directory
    - Specify restore_command
    - Optional: recovery_target_time
  - Start DB Cluster
    - Use syste,ctl start postgresql
    - Restores to specified point or latest WAL file
  - Verify Restore (Read-only mode)
    - Database ready for ready-only connections
    - Log indicates recovery has paused
  - Resume Recovery (writeable mode)
    - Execute Select pg_reply_resume()
    - Log indicates archive recovery complete
- **Practical Example (Ubuntu)**
  - PostgreSQL Version 10
    - Default install doesn't enable WAL archiving
  - Setup Archive Directory
    - Create '/var/lib/postgresql/pg_log_archive'
    - Ensure postgres user has write permissions
  - Configuration Files path
    - /etc/postgresql/10/main/postgresql.conf
  - Restart Service
    - Systemctl restart postgresql
  - Create/Populate database
    - Connect as postgres user
    - Create database and table, insert data
- **Backup Categories**
  - Logical Backups
    - Database is up / Running
    - Object level (tables, schemas)
    - Useful for Migration/Upgradation
    - Types

MS SQL DATABASE ADMINISTRATOR 30 DAYS COURSE

BY: Mukesh Chaurasia

- Database Dumps (pg_dump)
  - Individual database backup
  - Snapshot of DB at start time
  - Options
    - Data Only (-a)
    - Schema only (-s)
    - Specified schema (-n)
    - Exclude schema (-N)
    - Table backups (-t)
    - Output File (-f)
  - Formats
    - Plain text sql script (-Fp)
    - Tar format (-Ft)
    - Compressed (-fc)
    - Directory format (-fd)
    - Verbose (-v)
    - Compression (zip)
    - Split command
- Database Cluster Dumps (pg_dumpall)
  - Entire Instance backup
  - Useful for Global object loss
  - Options
    - Data only (-a)
    - Definitions only (-s)
    - Global objects only (-g)
    - Skip roles (-r)
    - Clean database (-c)
    - Skip object ownership (-O)
    - Do not grant/evoke privileges (-x)

- Restore Options
  - Psql utility
    - Plain text format
    - Clutser level (pg_dumpall)
    - Require DB creation (unless -C)
    - Ex: psql -f file.sql -d dbname -U user
  - Pg_restore utility
    - Compressed formate (-Ft, -Fd, -Fc)
    - Options
      - Backup file formate (-F)
      - Connect / Restore to DB (-d)

- Create DB if not exists (-C)
- Data Only (-a)
- Data definitions (-s)
- Specified Schema (-n)
- Exclude schema (-N)
- Table restore (-t)
- Verbose (-v)

- Physical Backups (File Level)
  - Data file, tablespaces
  - Data directory backup
  - Restoration with WAL files (point in time recovery)
  - Types
    - Offline backup (cold backup)
      - Database shut down
      - Consistent backup
      - Normal OS commands (e.g tar)
    - Online backup (hot backup)
      - Database up/running
      - 24*7 environments
      - Requires continuous archiving mode
      - Methods
        - Low level APIs (traditional)
          - Begin backup mode
          - Copy WAL files
          - End backup
        - Pg_basebackup (latest methods)
          - No manual begin / end backup mode
          - Prerequisite: Continuous archiving
          - Binary copy of cluster
          - Used for PITR / Streaming replication
          - Prerequisites
            - Wal_level = replics
            - Archive_mode = on
            - Max_wal_senders
            - Wal_keep_size
            - Pg_hba.conf changes (replication protocol)
          - Options
            - Destination directory (-D)

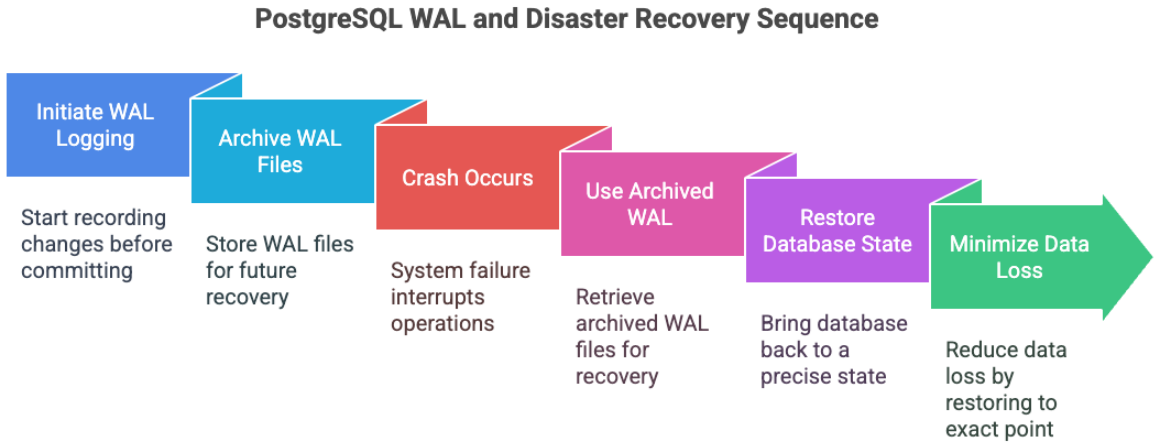
    - File formate (plain/tar) (-F)
    - Old/new tablespace dir (-d old -D new)
    - Compression level (-Z)
    - Progress (-P)

- **Continuous Archiving**
    - Prerequisite for online backups
    - Archives Wal files (transaction log)
    - Maintains ACID compliance
    - WAL file (16 mb each)
    - Prevents overwriting old WAL files
    - Configuration parameters
        - Wal_level = replica
        - Archive_mode = on
        - Archive_command (cp %p %f)
        - Restart database after changes
    - Methods
        - Archiver process (cp command)
        - Streaming Wal (pg_receivewal)
    - Pg_switch_wal (force log switch)
- **Point-In-Time Recovery**
    - Restore DB to specific time/transaction
    - Uses pg_basebackup + archived WAL files
    - Steps
        - Stop cluster
        - Take pg_basebackup
        - Restore base backup
        - Configure restore_command
        - Create recovery.signal file
        - Start database
    - Target settings
        - Target Time
        - Target Transaction ID
        - Target Action
    - Can perform PITR on production server
- **EBD Backup Tools**
    - Barman
        - Incremental backups (rsync)
        - Register multiple servers
        - Archive compresses backup

        - PITR support
        - Remote server mechanism
    - pgBackRest
        - Full, Incremental, Differential backups
        - Retention policies
        - Encryption support
        - Cloud storage (S3, Azure, GCP)
    - Comparison with pg_basebackup
        - Tools offer more feature (compression, encryption, parallel)
        - Native pg_basebackup is a basic feature
- **Backup Strategies & Best Practices**
    - Strategy
        - Determine DB size
        - Frequency of full/Incremental
        - Schedule backups
        - Separate WAL archive location
        - Meet RPO/RTO requirements
        - Adjust retention policies
        - 3-2-1 rule
            - 3 Copies of backup
            - 2 Local Copies
            - 1 Offsite copy (or cloud)
        - Encrypt backups
    - Best Practices
            - Document policies
            - Copy backup offsite / cloud (disaster prevention)
            - Regular test restore (test env)
            - Monitor backup process (PEM)
            - Logical backups are snapshots, not PITR
            - Restore to directory other than source
            - Clean DB before restore (good practice)

MS SQL DATABASE ADMINISTRATOR 30 DAYS COURSE

BY: Mukesh Chaurasia



🛡️ PostgreSQL Backup Strategies

PostgreSQL supports two main types of backups:

1. 🔍 Logical Backups

Logical backups extract database objects and data as SQL statements.

pg_dump – Backup a Single Database

- **Purpose**: Ideal for migrating databases or backing up individual databases.
- **Command**:

```
pg_dump -U postgres -d mydb -F c -f /backups/mydb.backup
```

  - -F c: Custom format
  - -f: Output file

- **Restore**:

```
pg_restore -U postgres -d mydb_restored /backups/mydb.backup
```

pg_dumpall – Backup Entire Cluster

- **Purpose**: Backs up all databases, roles, and tablespaces.
- **Command**:

```
pg_dumpall -U postgres > /backups/full_cluster.sql
```

- **Restore**:

```
psql -U postgres -f /backups/full_cluster.sql
```

2. 📦 Physical Backups

Physical backups copy the actual data files.

pg_basebackup – Base Backup for PITR

- **Purpose**: Creates a full copy of the data directory.
- **Command**:

```
pg_basebackup -U replication -D /backups/base -Ft -z -P --wal-method=stream
```

  - -Ft: Tar format
  - -z: Compress
  - --wal-method=stream: Include WAL files

WAL Archiving – Continuous Archiving

- **Purpose**: Enables Point-in-Time Recovery (PITR).
- **Setup in** postgresql.conf:

```
archive_mode = on
archive_command = 'cp %p /backups/wal_archive/%f'
archive_timeout = 60
```

🔄 Restore Strategies

1. 📜 Restoring Logical Backups

Using psql (Plain SQL Format)

```
psql -U postgres -d newdb -f /backups/mydb.sql
```

Using pg_restore (Custom/Tar Format)

```
pg_restore -U postgres -d newdb /backups/mydb.backup
```

- You can selectively restore:

```
pg_restore -U postgres -d newdb -t mytable /backups/mydb.backup
```

2. 📂 Restoring Physical Backups

- **Steps**:
  1. Stop PostgreSQL.
  2. Replace data directory with base backup.

3. Add recovery.signal file.
4. Configure postgresql.conf:

```
restore_command = 'cp /backups/wal_archive/%f %p'
```

5. Start PostgreSQL.

🔁 Recovery Strategies

1. ⏱️ Point-in-Time Recovery (PITR)

- **Purpose**: Restore to a specific timestamp.
- **Steps**:
    1. Restore base backup.
    2. Create recovery.signal file.
    3. Set recovery target in postgresql.conf:

```
restore_command = 'cp /backups/wal_archive/%f %p'
recovery_target_time = '2025-08-02 18:30:00'
```

    4. Start PostgreSQL.
- **Example**:

```
touch /var/lib/postgresql/data/recovery.signal
echo "restore_command = 'cp /backups/wal_archive/%f %p'" >> postgresql.conf
echo "recovery_target_time = '2025-08-02 18:30:00'" >> postgresql.conf
systemctl start postgresql
```

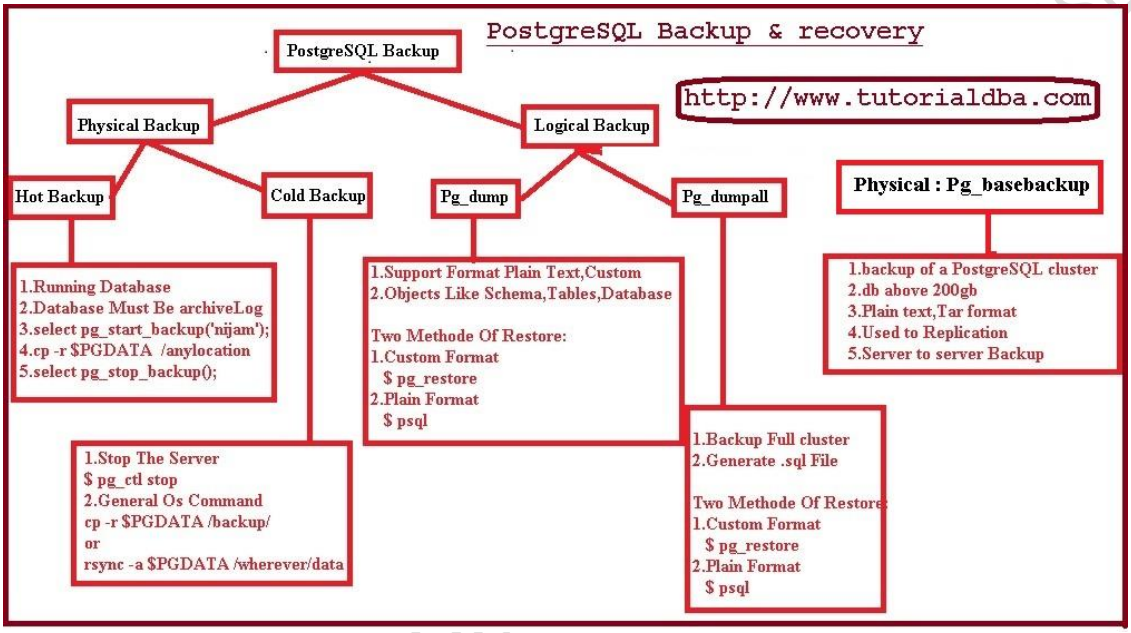
2. ⚠️ Crash Recovery

- **Automatic**: PostgreSQL uses WAL to recover after an unexpected shutdown.
- **No manual intervention needed** unless corruption occurs.

📌 Key Considerations

Aspect	Description
Backup Frequency	Daily logical + weekly physical recommended

Aspect	Description
Storage Location	Use off-server or cloud storage for redundancy
RPO/RTO	Define acceptable data loss and downtime
Testing	Regularly test restore and PITR procedures



PostgreSQL databases, like all systems holding valuable information, necessitate regular backups. While the process itself is straightforward, a solid grasp of the underlying techniques and assumptions is crucial. There are three fundamentally different approaches to backing up PostgreSQL data: SQL dump, file system level backup, and continuous archiving. Each method possesses distinct strengths and weaknesses, which will be discussed in detail below.

1. SQL Dump

An SQL dump involves exporting the database's schema and data into a plain-text file (or a custom format) using PostgreSQL's pg_dump utility. This utility essentially generates a series of SQL commands that, when executed, can recreate the database.

How it works:

- pg_dump connects to a PostgreSQL database and extracts the definitions of tables, indexes, views, functions, etc., along with the data contained within them.

- The output can be directed to a file.
- For restoration, the generated SQL file is fed into psql (the PostgreSQL interactive terminal) or a similar client, which executes the commands to rebuild the database.

Key Features and Variations:

- **Plain Text Format:** The default output is a plain-text SQL script, making it human-readable and easily transportable.
- **Custom Format (-Fc or --format=c):** This is a compressed, custom format that pg_dump can produce. It's often preferred for larger databases as it's more compact and allows for more flexible restoration (e.g., restoring specific tables). It's also suitable for pg_restore.
- **Directory Format (-Fd or --format=d):** This creates a directory containing multiple files, including a table of contents, which can be useful for very large databases and parallel restoration.
- **Individual Database Backup:** pg_dump backs up a single database at a time. To back up an entire PostgreSQL cluster (all databases, roles, tablespaces), you'd typically use pg_dumpall.
- **Logical Backup:** This is considered a "logical" backup because it reconstructs the data by re-executing SQL statements. It's independent of the underlying file system structure.

Strengths:

- **Portability:** SQL dumps are highly portable. You can restore them to a different PostgreSQL version (within reason), a different operating system, or even a different hardware architecture.
- **Flexibility:** You can easily inspect, modify, or even selectively restore parts of the backup (especially with custom or directory formats).
- **Human-Readable:** Plain-text dumps are easy to understand and troubleshoot.
- **Simplicity:** The pg_dump utility is relatively simple to use and understand.
- **Schema-Only or Data-Only:** pg_dump can be used to export only the schema (-s) or only the data (-a), which is useful for development or data migration.

Weaknesses:

- **Performance for Large Databases:** For very large databases, generating and restoring SQL dumps can be slow, as it involves processing all data logically.

- **Point-in-Time Recovery Limitations:** SQL dumps provide a "snapshot" of the database at the time of the dump. They do not natively support fine-grained point-in-time recovery (PITR).
- **Resource Intensive:** During the dump process, pg_dump can consume significant CPU and I/O resources on the database server.
- **Replication Slots/Publications Not Backed Up:** pg_dump does not back up information related to replication slots or logical publications, which need to be recreated manually if using logical replication.

Best Use Cases:

- Small to medium-sized databases.
- Regular full backups where PITR is not a strict requirement.
- Database migrations between different PostgreSQL versions or platforms.
- Creating development or testing environments from production data.

2. File System Level Backup

A file system level backup involves directly copying the data files (the PGDATA directory) of a PostgreSQL cluster. This is a "physical" backup, meaning it captures the exact state of the database files as they are on disk.

How it works:

- **Offline Backup (Simplest):** The simplest approach is to stop the PostgreSQL server completely and then copy the entire PGDATA directory to a backup location using standard file system utilities (e.g., cp, rsync, tar).
- **Online Backup (with WAL archiving):** To perform a file system backup while the database is running, it's crucial to put the database in a "backup mode" and archive Write-Ahead Log (WAL) files.
    - pg_start_backup(): This function prepares the database for a file system backup. It flushes dirty buffers to disk and records a "backup label file" in the PGDATA directory, indicating the starting point of the backup.
    - Copy PGDATA: While the database is in backup mode, the entire PGDATA directory is copied.
    - WAL Archiving: Crucially, during an online file system backup, all WAL files generated during the backup process *must* be archived. These WAL files are essential for consistency during restoration and for any subsequent point-in-time recovery.

- pg_stop_backup(): This function signals the end of the backup, removes the backup label file, and ensures all necessary WAL files for the backup period have been archived.

Strengths:

- **Fast for Large Databases:** Copying files directly is generally much faster than logical dumps for large databases, as it avoids the overhead of SQL parsing and execution.
- **Efficient Restoration:** Restoration is typically quick; you just copy the PGDATA directory back and start the server.
- **Point-in-Time Recovery (with WAL archiving):** When combined with continuous WAL archiving, file system backups form the basis for powerful point-in-time recovery (PITR). You can restore to any transaction commit point after the base backup.
- **Includes Everything:** A file system backup includes all database objects, tablespaces, configuration files (postgresql.conf, pg_hba.conf), and even any custom extensions.

Weaknesses:

- **Lack of Portability:** File system backups are generally not portable between different major PostgreSQL versions or operating systems. They are tied to the specific architecture and version from which they were taken.
- **Requires Downtime (for offline backup):** A full offline backup requires stopping the database, leading to service interruption.
- **Complexity (for online backup):** Online file system backups are more complex to set up and manage due to the necessity of pg_start_backup(), pg_stop_backup(), and robust WAL archiving.
- **Disk Space:** Backups consume significant disk space as they are byte-for-byte copies of the data directory.
- **Cannot Selectively Restore:** You cannot easily restore individual tables or databases from a file system backup; it's an all-or-nothing restoration of the entire cluster.

Best Use Cases:

- Very large databases where pg_dump is too slow.

- Environments requiring aggressive RTO (Recovery Time Objective) and RPO (Recovery Point Objective) with PITR.
- As the base backup for continuous archiving setups.
- Disaster recovery scenarios where rapid full restoration is paramount.

3. Continuous Archiving (WAL Archiving)

Continuous archiving, also known as Write-Ahead Log (WAL) archiving, is not a backup *method* in itself, but rather a crucial component that, when combined with a base file system backup, enables robust Point-in-Time Recovery (PITR). WAL archiving involves continuously shipping the transaction log (WAL) files to a separate, safe storage location as they are generated by the PostgreSQL server.

How it works:

- **WAL Files:** PostgreSQL writes all changes to the database to WAL files before they are written to the main data files. These files are a sequential log of all transactions.
- **archive_mode:** The archive_mode parameter in postgresql.conf must be enabled.
- **archive_command:** A custom command (specified in postgresql.conf) is executed by PostgreSQL whenever a WAL file is ready to be archived. This command typically copies the WAL file to an archive location (e.g., a network share, cloud storage).
- **Base Backup:** A complete file system backup (as described above) serves as the starting point. This is called the "base backup."
- **Recovery:** To recover, you start with the base backup and then replay the archived WAL files on top of it, bringing the database forward to a desired point in time.

Strengths:

- **Point-in-Time Recovery (PITR):** This is the primary strength. You can restore your database to *any* specific transaction commit point, down to the second, after your last base backup. This is invaluable for recovering from accidental data deletions, corruption, or logical errors.
- **Low RPO:** By continuously archiving WAL files, your potential data loss (RPO) can be incredibly low, often measured in seconds or minutes.

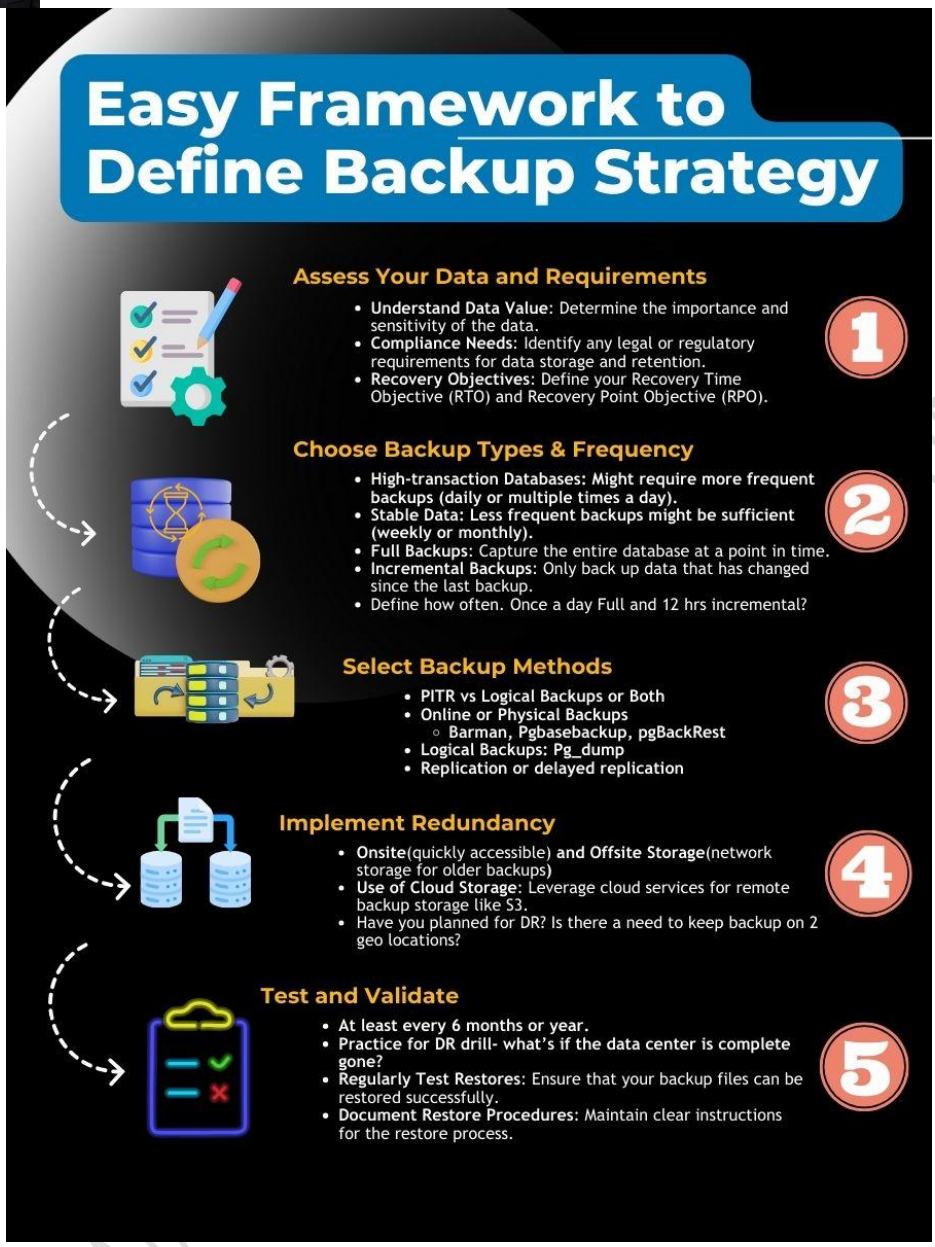
- **Minimizes Downtime for Backups:** Once configured, WAL archiving runs continuously in the background with minimal impact on database performance.
- **Hot Standby Support:** WAL archiving is fundamental for setting up and maintaining hot standby servers (read replicas) for high availability and disaster recovery.
- **Scalability:** Can handle very large and busy databases effectively.

Weaknesses:

- **Complexity to Set Up:** Setting up and managing continuous archiving requires a deeper understanding of PostgreSQL internals and careful configuration.
- **Increased Storage Requirements:** You need significant storage for the archived WAL files, which can accumulate rapidly on busy systems.
- **Monitoring Crucial:** Requires vigilant monitoring to ensure WAL archiving is functioning correctly and that WAL files are being safely transferred. Failures in archiving can compromise your ability to recover.
- **Network/I/O Dependency:** The performance and reliability of WAL archiving depend heavily on the network and I/O capabilities of the archive destination.
- **Recovery Can Be Slow for Long Periods:** If you need to replay a very long sequence of WAL files from an old base backup, the recovery process can take a considerable amount of time.

Best Use Cases:

- Mission-critical databases with high RPO and RTO requirements.
- Environments where data loss must be absolutely minimized.
- When advanced disaster recovery capabilities like hot standbys are needed.
- Large, highly transactional databases.



Choosing the Right Strategy:

The best backup strategy often involves a combination of these approaches:

- **Regular File System Base Backups with Continuous Archiving:** This is generally the most robust and recommended approach for production environments, providing excellent PITR capabilities and fast full restorations.
- **Periodic SQL Dumps:** Can be used in conjunction with file system backups for specific purposes, such as:
    - Testing logical restoration processes.
    - Providing a highly portable backup for development or staging environments.

- Backing up specific schemas or tables for quick, selective restoration.
- Auditing or data analysis by providing a human-readable snapshot.

SQL Dump with pg_dump and pg_dumpall

The SQL dump method is a logical backup approach that generates a file containing SQL commands. When these commands are executed against a PostgreSQL server, they recreate the database in the exact state it was at the time of the dump. PostgreSQL provides the pg_dump utility for single database backups and pg_dumpall for backing up an entire database cluster (including global objects like roles and tablespaces).

1. pg_dump: Backing Up a Single Database

pg_dump is a powerful client application that connects to a PostgreSQL database and extracts its schema and data. It can be run from any host with network access to the database server, provided it has the necessary permissions.

Basic Usage:

The most basic way to use pg_dump is to direct its output to standard output, which you then redirect to a file:

```
pg_dump dbname > dumpfile.sql
```

- dbname: The name of the database you want to back up.
- dumpfile.sql: The name of the file where the SQL commands will be saved.

Example: Backing up a database named my_production_db to my_production_db_backup.sql:

```
pg_dump my_production_db > my_production_db_backup.sql
```

Understanding pg_dump Permissions and Connectivity:

pg_dump operates as a regular PostgreSQL client. This means:

- **Read Access:** It must have read access to all tables you intend to back up. For a full database backup, you almost always need to run pg_dump as a database superuser.
- **Selective Backups:** If you lack superuser privileges, you can still back up specific portions of the database using options like -n schema (for a specific schema) or -t table (for a specific table).

Example: Backing up only the public schema:

pg_dump -n public my_production_db > public_schema_backup.sql

Example: Backing up only the users table from the public schema:

pg_dump -t public.users my_production_db > users_table_backup.sql

- **Server Connection Options:**
    - -h host: Specifies the hostname or IP address of the database server.
    - -p port: Specifies the port number the database server is listening on.
    - **Defaults:** If not specified, pg_dump will use the local host or the PGHOST environment variable, and the PGPORT environment variable or the compiled-in default port (usually 5432).

Example: Backing up a database on a remote server at db.example.com on port 5433:

pg_dump -h db.example.com -p 5433 my_production_db > remote_db_backup.sql

- **Database User:**
    - -U username: Specifies the database user name to connect as.
    - **Default:** By default, it uses the current operating system user name. You can also set the PGUSER environment variable.
    - **Authentication:** pg_dump connections are subject to normal client authentication mechanisms (e.g., pg_hba.conf rules). You might be prompted for a password.

Example: Backing up as the postgres superuser:

pg_dump -U postgres my_production_db > my_production_db_as_superuser.sql

Consistency and Non-Blocking Nature:

Dumps created by pg_dump are internally consistent. This means the dump represents a snapshot of the database at the time pg_dump began running. Importantly, pg_dump **does not block other operations** on the database while it is working, with exceptions for operations requiring exclusive locks (like most ALTER TABLE commands).

Portability Advantage:

A significant advantage of pg_dump over file-level backups and continuous archiving is its portability. pg_dump's output can generally be reloaded into:

- **Newer PostgreSQL Versions:** This is invaluable for upgrades.

- **Different Machine Architectures:** Such as moving from a 32-bit to a 64-bit server.
- **Different Operating Systems:** E.g., Linux to Windows.

1.1. Restoring the Dump

Text files created by pg_dump are designed to be read by the psql program.

Basic Restoration:

```
psql -X dbname < dumpfile.sql
```

- dbname: The name of the database where the data will be restored. This database **must be created beforehand** from template0.
- dumpfile.sql: The SQL dump file generated by pg_dump.
- -X (--no-psqlrc): Ensures psql runs with its default settings, preventing issues from custom ~/.psqlrc files.

Example: Restoring my_production_db_backup.sql to a new database named new_restored_db:

1. **Create the target database:**

```
createdb -T template0 new_restored_db -U postgres
```

2. **Restore the dump:**

```
psql -X new_restored_db -U postgres < my_production_db_backup.sql
```

Important Considerations for Restoration:

- **User Existence:** Before restoring, all database users who owned objects or were granted permissions in the original dumped database **must already exist** on the target server. Otherwise, object ownership and permissions will not be recreated correctly.
- **Error Handling in psql:**
    - By default, psql will continue executing even if an SQL error occurs, potentially leading to a partially restored database.
    - To make psql exit on the first SQL error:

```
psql -X --set ON_ERROR_STOP=on dbname < dumpfile.sql
```

    - To restore the entire dump as a single transaction (all or nothing):

```
psql -X -1 dbname < dumpfile.sql
```

Or using the long option:

psql -X --single-transaction dbname < dumpfile.sql

Caution: Even a minor error in single-transaction mode can roll back a restore that has run for many hours.

- **Restoring Non-Text Dumps:** Dumps created with pg_dump's custom or directory formats (-Fc, -Fd) *must* be restored using the pg_restore utility, not psql.

Example: Restoring a custom-format dump (-Fc):

pg_restore -d new_restored_db custom_format_dumpfile

- **template0 Dependency:** Dumps produced by pg_dump are relative to template0. If your template1 database has custom objects (languages, procedures, etc.), these will be included in the dump. When restoring, always create the empty database from template0 to avoid conflicts.
- **Post-Restoration Optimization:** After restoring, it's highly recommended to run ANALYZE on each database. This updates statistics for the query optimizer, ensuring efficient query plans.

psql -c "ANALYZE VERBOSE;" new_restored_db

Piping Dumps Between Servers:

pg_dump and psql can read from and write to standard input/output, enabling direct database transfers between servers without intermediate files:

pg_dump -h host1 dbname | psql -X -h host2 dbname

Example: Copying my_db from serverA to serverB:

1. **On serverB (or a client machine with access to both):**

First, ensure the target database exists on serverB, created from template0

createdb -T template0 my_db -h serverB -U postgres

Then, pipe the dump directly

pg_dump -h serverA my_db -U postgres | psql -X -h serverB my_db -U postgres

2. pg_dumpall: Backing Up the Entire Cluster

pg_dump backs up only a single database. It does not include cluster-wide data such as:

- **Roles (users)**
- **Tablespaces**
- **Global configuration (though often manually managed)**

The pg_dumpall utility is provided to back up the entire contents of a PostgreSQL cluster, including all databases and global objects.

Basic Usage:

```
pg_dumpall > all_cluster_backup.sql
```

Restoring a pg_dumpall Dump:

A pg_dumpall dump is also a plain-text SQL file and is restored using psql. You must connect to an *existing* database (e.g., postgres or template1) to run the restoration.

```
psql -X -f all_cluster_backup.sql postgres
```

- -f dumpfile: Specifies the input file to read from.

Important Notes for pg_dumpall Restoration:

- **Superuser Access:** You **must** have database superuser access when restoring a pg_dumpall dump because it needs to recreate roles and tablespace information.
- **Tablespace Paths:** If you use tablespaces, ensure that the tablespace paths specified in the dump (and potentially created by pg_dumpall) are appropriate and accessible on the new installation.
- **Consistency:** pg_dumpall works by dumping global objects first, then invoking pg_dump for each individual database. While each *individual database* dump is internally consistent, the snapshots of *different databases* are **not synchronized**. This means if transactions span multiple databases, pg_dumpall cannot guarantee a globally consistent snapshot across the entire cluster.
- **Global-Only Dumps:** You can dump only the cluster-wide data (roles, tablespaces) using the --globals-only option. This is useful if you are performing individual pg_dump backups for each database separately but still need to preserve global definitions.

```
pg_dumpall --globals-only > global_objects_backup.sql
```

3. Handling Large Databases with pg_dump

For very large databases, plain-text SQL dumps can become problematic due to file size limits or the desire for faster, more flexible backups. pg_dump offers several solutions:

3.1. Compressed Dumps (using external tools)

You can pipe pg_dump's output through a compression program like gzip.

Dumping with gzip:

```
pg_dump dbname | gzip > filename.gz
```

Reloading with gzip:

```
gunzip -c filename.gz | psql dbname

# OR

cat filename.gz | gunzip | psql dbname
```

3.2. Splitting Dumps (using split)

The split command allows you to break a large dump into smaller, manageable files.

Dumping and Splitting (e.g., into 2GB chunks):

```
pg_dump dbname | split -b 2G - filename_part_
```

This will create files like filename_part_aa, filename_part_ab, etc.

Reloading Split Dumps:

```
cat filename_part_* | psql dbname
```

Combining split with gzip (GNU split feature):

If you have GNU split, you can combine splitting and compression:

```
pg_dump dbname | split -b 2G --filter='gzip > $FILE.gz' filename_part_
```

This will create files like filename_part_aa.gz, filename_part_ab.gz, etc.

Restoring such a dump:

```
zcat filename_part_*.gz | psql dbname
```

3.3. pg_dump's Custom Dump Format (-Fc)

pg_dump's custom format is a compressed binary format (if zlib is installed on the system where PostgreSQL was built). It offers several advantages over plain-text dumps:

- **Compression:** Similar to gzip, but built-in.
- **Selective Restoration:** Allows restoring specific tables or objects using pg_restore.
- **Parallel Restoration:** Can be restored in parallel by pg_restore.

Dumping in Custom Format:

pg_dump -Fc dbname > custom_format_dumpfile.dump

Restoring Custom Format Dumps (using pg_restore):

pg_restore -d dbname custom_format_dumpfile.dump

Example: Restoring only the products table from a custom dump:

pg_restore -d dbname -t products custom_format_dumpfile.dump

3.4. pg_dump's Parallel Dump Feature (-j)

For very large databases, pg_dump can dump multiple tables concurrently using its parallel mode. This significantly speeds up the dump process. This feature is only supported for the "directory" archive format (-Fd).

Dumping in Parallel (e.g., with 4 parallel jobs):

pg_dump -j 4 -F d -f out.dir dbname

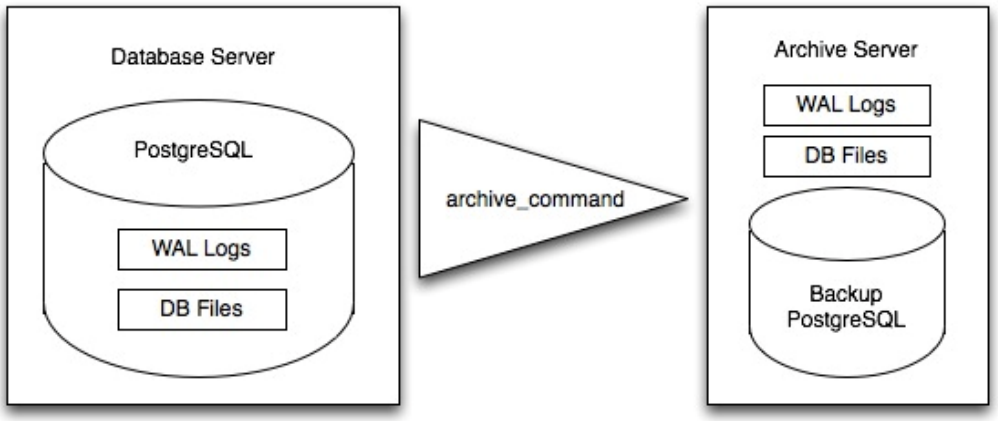
- -j num: Specifies the number of parallel jobs.
- -F d: Specifies the "directory" archive format.
- -f out.dir: Specifies the output directory (which pg_dump will create).

Restoring Parallel Dumps (using pg_restore):

pg_restore can also restore dumps in parallel, regardless of whether the original dump was created with parallel jobs, as long as it's in "custom" or "directory" format.

pg_restore -j 4 -d dbname out.dir

By understanding and leveraging these pg_dump and pg_dumpall features, you can create robust, flexible, and efficient SQL-based backup strategies for your PostgreSQL databases, from small development environments to large production systems. Remember to always **test your backup and restore procedures regularly** to ensure their integrity and your ability to recover data when needed.



File System Level Backup

An alternative and fundamentally different approach to backing up PostgreSQL data involves directly copying the files that PostgreSQL uses to store its data. This method is a "physical" backup, meaning it captures the raw binary data files as they exist on the file system, rather than the logical SQL commands to recreate the database.

The primary location for PostgreSQL data files is typically within the PGDATA directory. The exact location is determined during installation but is commonly /var/lib/postgresql/data, /usr/local/pgsql/data, or a user-defined path. You can use any preferred file system backup method, such as tar, rsync, or specialized snapshot tools.

Example (using tar for an offline backup):

 # First, stop the PostgreSQL server

sudo systemctl stop postgresql # Or appropriate command for your OS/installation

 # Then, create the tar archive of the data directory

tar -cf /path/to/backup/backup.tar /usr/local/pgsql/data

 # Restart the PostgreSQL server

sudo systemctl start postgresql

Key Restrictions and Considerations

While seemingly straightforward, file system level backups have critical restrictions that make them more complex than a simple file copy, especially for active databases:

1. **Server Must Be Shut Down (for Simple Copy):**

- **The absolute most crucial restriction** for a simple file copy (e.g., using tar or cp) is that the PostgreSQL database server *must be completely shut down* to obtain a usable and consistent backup.
- **Why?** PostgreSQL constantly writes to its data files and uses internal buffers. File system utilities like tar do not take an atomic snapshot of the file system. If the server is running, even with all connections disallowed, the files on disk might be in an inconsistent state (e.g., a transaction partially written, indexes out of sync with table data, or WAL files not reflecting the latest on-disk state of tables). Copying these inconsistent files will result in a corrupted or unusable backup.
- **Restoration:** Similarly, when restoring such a backup, the server must be shut down before copying the data back, and then restarted.

2. **Entire Cluster Backup Only:**
   - You **cannot** back up or restore individual tables, databases, or even specific directories within the PGDATA structure.
   - **Why?** PostgreSQL's data files are highly interconnected and rely on cluster-wide information, particularly the **commit log files (pg_xact/*)**. These files contain the transaction commit status for *all* transactions across the entire cluster. A table's data file is meaningless without its corresponding commit status in pg_xact. Restoring only a part of the data directory would inevitably render the entire database cluster inconsistent and unusable.
   - Therefore, file system backups *only work for complete backup and restoration of an entire database cluster*.

Alternative File System Backup Approaches (Online Methods)

To circumvent the downtime requirement, more sophisticated file system backup techniques exist. These generally involve methods that can take a "consistent snapshot" of the data directory while the server is running.

1. **Consistent Snapshot (Volume Snapshots):**
   - **Concept:** If your underlying file system (e.g., LVM, ZFS, Btrfs, or storage array snapshots) supports "frozen snapshots," you can leverage this feature. A snapshot creates a point-in-time, read-only copy of a volume, which can then be safely backed up.
   - **Procedure:**

1. Instruct the file system or storage system to create a "frozen snapshot" of the volume containing the PGDATA directory.
2. Copy the *entire* PGDATA directory (and WAL files) from this snapshot to your backup destination.
3. Release the frozen snapshot.

- **Online Operation:** This method can be performed while the database server is running, significantly reducing downtime.
- **Post-Snapshot State:** A backup created this way will save the database files in a state as if the database server was *not cleanly shut down*. When you start PostgreSQL using this backed-up data, it will perform crash recovery (replay the WAL log) as if the previous server instance crashed. This is normal and expected behavior and not a problem, provided **you include the WAL files in your backup**.
- **Reducing Recovery Time:** You can issue a CHECKPOINT command just before taking the snapshot (CHECKPOINT; in psql) to force all dirty buffers to disk. This can reduce the amount of WAL replay needed during recovery, speeding up startup time.
- **Distributed File Systems Challenge:** If your database is spread across multiple file systems (e.g., data files on one volume, WAL on another, or tablespaces on different mounts), obtaining *simultaneously* frozen snapshots of all volumes can be challenging or impossible. If snapshots are not perfectly simultaneous, the backup will be inconsistent. Read your file system documentation very carefully before trusting this technique in such distributed scenarios.
- **Workarounds for Non-Simultaneous Snapshots:**
  - **Brief Downtime:** Shut down the database server long enough to establish all frozen snapshots sequentially.
  - **Continuous Archiving Base Backup:** The most robust solution is to use a continuous archiving base backup (as described in the "Continuous Archiving" section). This method is inherently designed to be immune to file system changes during the backup because it relies on the WAL stream for consistency. This would involve temporarily enabling continuous archiving if it's not already on, performing the base backup, and then potentially disabling it (though keeping it on is generally recommended for PITR). Restoration would then use continuous archive recovery.

2. **rsync with Minimal Downtime:**
    - **Concept:** This method leverages rsync's ability to efficiently synchronize files, combined with a very brief period of server downtime.
    - **Procedure:**
        1. **First rsync (Online):** Run rsync to copy the PGDATA directory while the database server is running. This creates a nearly complete copy.

```
rsync -a /usr/local/pgsql/data/ /path/to/backup/temp_data_copy/
```

        2. **Shut Down Server (Brief Downtime):** Stop the PostgreSQL server for a minimal period.

```
sudo systemctl stop postgresql
```

        3. **Second rsync (Offline, with Checksum):** Run rsync again with the --checksum (-c) option. This will only transfer files that have changed since the first rsync and uses checksums to ensure file integrity, which is crucial because rsync's default modification-time granularity is one second, which might miss rapid PostgreSQL changes. The server is down during this final, quick sync, ensuring consistency.

```
rsync -ac /usr/local/pgsql/data/ /path/to/backup/final_data_copy/
```

        4. **Restart Server:** Restart PostgreSQL.

```
sudo systemctl start postgresql
```

    - **Benefit:** This method allows a file system backup to be performed with significantly reduced downtime compared to a full offline copy.

Comparison to SQL Dump

- **Size:** A file system backup will typically be **larger** than an SQL dump. pg_dump does not need to dump the contents of indexes (only the commands to recreate them), temporary files, or other internal structures that are part of the physical data directory.
- **Speed:** Taking a file system backup (especially with rsync or snapshots) can often be **faster** than generating an SQL dump for very large databases, as it involves less logical processing.

- **Portability:** File system backups are generally **not portable** between different major PostgreSQL versions or different machine architectures/operating systems. They are tied to the specific environment from which they were taken. SQL dumps, in contrast, are highly portable.
- **Granularity:** File system backups are "all or nothing" for the entire cluster. SQL dumps allow for granular restoration of individual databases, schemas, or even tables.
- **PITR:** When combined with continuous archiving, file system backups form the necessary base for Point-in-Time Recovery (PITR), which SQL dumps do not natively support.

Conclusion on File System Backups

While a simple file system copy requires full server downtime and lacks portability, advanced techniques like volume snapshots or rsync with minimal downtime offer viable options for online physical backups. For robust disaster recovery and Point-in-Time Recovery (PITR), file system backups serve as the **base backup** that is then complemented by continuous WAL archiving. This combination is generally the preferred strategy for mission-critical PostgreSQL deployments.

Continuous Archiving and Point-in-Time Recovery (PITR)

At the heart of robust PostgreSQL backup and disaster recovery lies the **Write-Ahead Log (WAL)**. PostgreSQL constantly maintains a WAL in the pg_wal/ subdirectory of the cluster's data directory. This log records every change made to the database's data files. While its primary purpose is crash-safety (ensuring data consistency after an unexpected shutdown by replaying log entries since the last checkpoint), the existence of the WAL makes a powerful third backup strategy possible: **continuous archiving combined with a file-system-level base backup.**

This approach is more complex to administer than simple SQL dumps or offline file system copies, but its benefits are significant:

- **Relaxed Base Backup Consistency:** Unlike simple file system backups that require a perfectly consistent (shut down) state, an online base backup for continuous archiving doesn't need to be perfectly "clean." Any internal inconsistencies in the base backup will be automatically corrected by WAL replay during recovery. This means you don't *strictly* need file system snapshot capabilities; basic archiving tools like tar are sufficient for the base backup.

- **Continuous Backup for Large Databases:** By continuously archiving WAL files, you achieve a continuous backup without needing frequent full base backups. This is incredibly valuable for very large databases where a full backup might be inconvenient or take too long.
- **Point-in-Time Recovery (PITR):** This is the crowning jewel of continuous archiving. Since you can replay an indefinitely long sequence of WAL files, you can stop the replay at *any arbitrary point in time* and have a consistent snapshot of the database as it was at that exact moment. This allows recovery from logical errors, accidental deletions, or corruption that occurred hours or days ago.
- **Warm Standby Systems:** Continuously feeding the WAL stream to another machine (a "standby" server) that has the same base backup creates a "warm standby" or "hot standby" (if configured for read-only queries). This provides high availability: the second machine can be brought online almost instantly with a nearly current copy of the database if the primary fails.

Important Note: pg_dump and pg_dumpall are **logical backups** and do not produce file-system-level backups. They **cannot** be used as part of a continuous archiving solution because their output does not contain the necessary information for WAL replay.

Like plain file-system-level backups, this method only supports the restoration of an **entire database cluster**, not a subset. It also requires substantial archival storage for both the base backups and the continuous stream of WAL files, which can accumulate rapidly on busy systems. Despite these demands, it is the preferred backup technique for environments requiring high reliability and low Recovery Point Objectives (RPOs).

To recover successfully, you need a continuous sequence of archived WAL files extending back at least to the start time of your base backup. Therefore, the first step is always to set up and test your WAL archiving procedure *before* taking your initial base backup.

---

1. Setting Up WAL Archiving

A running PostgreSQL system generates an unending sequence of WAL records, physically divided into **WAL segment files**, typically 16MB each (configurable during initdb). These segments are named numerically, reflecting their position in the WAL sequence. Normally, PostgreSQL recycles old segments once their contents precede the last checkpoint. For archiving, we need to capture and save each filled segment file before it's recycled.

PostgreSQL offers flexibility in *how* these files are saved, relying on the administrator to specify an external command or library.

Configuration Parameters (in postgresql.conf):

To enable WAL archiving, you need to modify these parameters and restart the PostgreSQL server (some can be reloaded, but wal_level requires a restart):

- **wal_level = replica or higher:** This setting dictates the amount of information written to the WAL. minimal level optimizes some SQL commands by avoiding WAL logging, which would prevent successful archive recovery. replica (or logical) ensures enough information is present.
- **archive_mode = on:** Enables WAL archiving.
- **archive_command or archive_library:**
    - **archive_command = 'shell_command':** This is a shell command executed by PostgreSQL for each completed WAL segment.
        - %p is replaced by the full path of the file to archive (relative to the data directory).
        - %f is replaced by only the file name.
        - %% is used to embed a literal % character.
    - **archive_library = 'module_name':** (Introduced in PostgreSQL 15) This specifies a C-language archive module. While more complex to write, it can offer better performance and access to server resources.

Example archive_command (Unix/Linux):

A common and robust example for Unix-like systems, copying to an NFS-mounted directory:

Ini, TOML

archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/server/archivedir/%f'

- test ! -f /mnt/server/archivedir/%f: This crucial part checks if a file with the same name already exists in the archive directory.
- && cp %p /mnt/server/archivedir/%f: If the file doesn't exist, the cp command copies the WAL segment.

This test prevents overwriting existing archive files, which is an important safety feature against accidental double-archiving or issues with different servers trying to write to the

same archive location. The command **must return a zero exit status for success** and non-zero for failure; PostgreSQL will retry archiving failed segments until it succeeds.

Security and Permissions:

- The archive_command is executed under the ownership of the same user that the PostgreSQL server runs as (e.g., postgres).
- Ensure the archive directory (/mnt/server/archivedir in the example) has appropriate permissions to protect sensitive database data. It should typically restrict access to the PostgreSQL user only.

Monitoring and Alerting:

- **Speed:** The speed of the archive command is less critical than its reliability. It just needs to keep up with the *average* WAL generation rate.
- **Falling Behind:** If archiving falls significantly behind (e.g., due to a full archive disk, network issues, or operator intervention being required), the pg_wal/ directory will fill up. If the file system containing pg_wal/ becomes full, PostgreSQL will initiate a **PANIC shutdown** to prevent data corruption. No committed transactions will be lost, but the database will remain offline until space is freed and the archive process resumes.
- **Monitoring is vital:** Implement monitoring for the archiving process to ensure it's working as intended.

WAL File Naming:

- Archived WAL filenames will be up to 64 characters long and consist of ASCII letters, digits, and dots (e.g., 000000010000000A900000065).
- It's essential to preserve the original filename (%f) during archiving, though the original relative path (%p) isn't strictly necessary for restoration.

archive_timeout:

- By default, WAL segments are only archived once they are completely filled (16MB). For systems with low WAL traffic or during slack periods, this could mean a long delay between a transaction committing and its WAL record being safely archived.
- The archive_timeout parameter (e.g., archive_timeout = 60s for 1 minute) forces PostgreSQL to switch to a new WAL segment at least that often, even if the current one isn't full. This ensures more frequent archiving.

- **Caution:** Setting a very short archive_timeout can bloat your archive storage, as partially filled segments are still archived as 16MB files. A few minutes is usually a reasonable setting.
- You can also manually force a segment switch using SELECT pg_switch_wal(); in psql to archive recent transactions immediately.

Configuration Files and Backups:

- WAL archiving only backs up database data, not manual changes to configuration files (postgresql.conf, pg_hba.conf, pg_ident.conf).
- Always keep configuration files in a location that is included in your regular file system backups. You can relocate them if needed (see PostgreSQL documentation for data_directory and config_file options).

---

2. Making a Base Backup

A **base backup** is a consistent snapshot of the data directory at a specific point in time, serving as the starting point for recovery. While you can use a low-level API, the easiest and recommended method is pg_basebackup.

2.1. Using pg_basebackup

pg_basebackup is a utility designed to take consistent base backups of a running PostgreSQL cluster. It can output the backup as regular files or as a tar archive.

Example: Creating a base backup to a directory:

Bash

```
pg_basebackup -h localhost -D /path/to/backup/base_backup_$(date +%Y%m%d_%H%M%S) -U postgres -Ft -X fetch -P -v
```

- -h localhost: Connect to the local server.
- -D /path/to/backup/base_backup_...: Specifies the destination directory for the backup. $(date +%Y%m%d_%H%M%S) creates a timestamped directory.
- -U postgres: Connect as the postgres superuser.
- -Ft: Specifies tar format output (creates a single tar file per tablespace). For a directory copy, use -Fd.
- -X fetch: Ensures all necessary WAL files generated during the backup are included or fetched. This is crucial for consistent recovery.

- -P: Shows progress.
- -v: Verbose output.

What pg_basebackup does:

1. It automatically puts the server into backup mode (similar to pg_backup_start()).
2. Copies all data files and tablespaces.
3. Ensures necessary WAL files are preserved or fetched.
4. Creates a **backup history file** in the WAL archive area. This file (e.g., 000000010000123400055CD.007C9330.backup) identifies the first WAL segment needed for recovery from this base backup.
5. Takes the server out of backup mode (similar to pg_backup_stop()).

Key points about base backups:

- **WAL Retention:** You **must keep all WAL segment files** generated *during and after* the base backup. The backup history file tells you the starting WAL segment needed.
- **Storage vs. Recovery Time:** The interval between base backups should balance archived WAL storage consumption with desired recovery time. Replaying many months of WAL can be time-consuming.
- **full_page_writes:** If you normally run with full_page_writes disabled (which is rare and not recommended for crash safety), it will be effectively forced on during the pg_basebackup process to ensure data consistency, which might temporarily impact performance.

2.2. Making an Incremental Backup (--incremental with pg_basebackup)

PostgreSQL 14 introduced incremental backups using pg_basebackup with the --incremental option. This aims to reduce backup size and time by only copying changed blocks.

How it works:

- You provide the **backup manifest** (a file output by a previous pg_basebackup) of an earlier backup from the *same server*.
- Non-relation files are copied entirely.
- Relation files (tables, indexes) may be replaced by smaller "incremental files" containing only blocks changed since the earlier backup, plus metadata to reconstruct the full file.

- The server uses **WAL summaries** (stored in pg_wal/summaries) to identify changed blocks. These summaries must cover the WAL range between the previous and current backup start LSNs. If missing or not caught up, the incremental backup will fail.

Example: Creating an incremental backup:

Assume you have a previous full backup at
/path/to/backup/full_backup_20250701_100000

And its manifest is located at
/path/to/backup/full_backup_20250701_100000/backup_manifest

pg_basebackup -h localhost -D /path/to/backup/incremental_backup_$(date +%Y%m%d_%H%M%S) -U postgres -Ft -X fetch -P -v --
incremental=/path/to/backup/full_backup_20250701_100000/backup_manifest

Restoring Incremental Backups (requires pg_combinebackup):

Restoring an incremental backup is more complex:

1. You need **all earlier backups** (full and subsequent incrementals) that directly or indirectly supply omitted blocks for the incremental backup you want to restore.
2. These backups must be in separate directories, not the final target directory.
3. Use the pg_combinebackup utility to combine these backups and create a synthetic full backup in your target directory.

pg_combinebackup -D /path/to/target/data_directory
/path/to/latest_incremental_backup /path/to/previous_full_or_incremental_backup ...

4. After combining, you still need all the WAL segments generated *during and after* the latest incremental backup for PITR.
5. pg_combinebackup has limitations, e.g., if the cluster's checksum status has changed.

Considerations for Incremental Backups:

- **Complexity:** Managing incremental backups and their dependencies is inherently more complex. You are responsible for tracking the relationships between backups.
- **Suitability:** Best for very large databases where a significant portion of data is static or changes slowly. For smaller databases, full backups are simpler. For

databases with high change rates across most data, incremental backups might not offer significant size savings.

- **Standby Server Incremental Backups:** Incremental backups on a standby require new restart points to have been created. If a standby has very little activity, an incremental backup might fail due to insufficient WAL summarization data.

2.3. Making a Base Backup Using the Low-Level API

This method gives you finer control but requires more manual steps than pg_basebackup. It involves using SQL functions to signal the start and end of a backup. Multiple backups can run concurrently.

Steps:

1. **Ensure WAL archiving is enabled and working.**
2. **Start Backup Mode:** Connect to the server as a superuser (or user with EXECUTE on pg_backup_start) and execute:

```
SELECT pg_backup_start(label => 'my_daily_backup_20250802', fast => false);
```

   - label: A unique string to identify this backup.
   - fast => false (default): Waits for the next scheduled checkpoint, minimizing impact.
   - fast => true: Requests an immediate checkpoint, speeding up the backup start but potentially increasing I/O load.
   - **Crucial:** The SQL connection running pg_backup_start **must be maintained** until pg_backup_stop is called, otherwise the backup will be aborted.
3. **Perform the File System Backup:** While the database is in backup mode (and running normally), use your preferred file system tool (tar, rsync, cpio) to copy the entire PGDATA directory and any tablespaces.
   - **Exclude pg_wal/:** It's generally advisable to *omit* the pg_wal/ subdirectory from this copy. WAL files are handled by the archiving process. This reduces risk during restoration.
   - **Exclude other volatile directories:** pg_replslot/, pg_dynshmem/, pg_notify/, pg_serial/, pg_snapshots/, pg_stat_tmp/, pg_subtrans/, and any pgsql_tmp* files/directories can also be safely omitted as they are recreated or volatile.

- **Tablespaces:** Ensure symbolic links for tablespaces in pg_tblspc/ are correctly copied as links, not followed, to avoid corruption upon restoration.
- **Checksum/Vanished File Warnings:** Be aware that file system backup tools might report warnings or errors about files changing during the copy. For active databases, this is normal. Configure your tool or script to accept these specific warnings as non-errors (e.g., rsync's "vanished source files" exit code, or GNU tar's --warning=no-file-changed).

4. **End Backup Mode:** In the *same SQL connection* used for pg_backup_start, issue:

```
SELECT * FROM pg_backup_stop(wait_for_archive => true);
```

- This terminates backup mode.
- On a primary, it triggers an automatic switch to a new WAL segment, ensuring the last WAL generated during the backup is ready for archiving.
- wait_for_archive => true (default): pg_backup_stop will wait until all WAL segments generated during the backup interval have been successfully archived. This is highly recommended to ensure backup integrity. If you set it to false, you *must* have robust external monitoring of your archive process.
- This function returns three values. The second field contains the content for the backup_label file, and the third for tablespace_map. You **must** save these exactly (byte-for-byte, in binary mode) into backup_label and tablespace_map files within the *root directory of your backup*. These files are critical for recovery.

5. **Verify WAL Archiving:** Ensure that the WAL segments active during the backup (ending with the segment identified by pg_backup_stop) are successfully archived.

---

3. Recovering Using a Continuous Archive Backup

When disaster strikes, here's how to recover your database using a continuous archive:

1. **Stop the Server:** If the PostgreSQL server is running on the target machine, shut it down gracefully.

```
sudo systemctl stop postgresql
```

2. **Preserve Current Data (Optional but Recommended):** If space permits, copy the entire existing cluster data directory (PGDATA) and any tablespaces to a temporary location. This acts as a safety net if something goes wrong. At a minimum, save the contents of the pg_wal/ subdirectory, as it might contain critical unarchived WAL files.

```
cp -rp /var/lib/postgresql/data /var/lib/postgresql/data_pre_recovery_backup
```

3. **Clean Target Directories:** Remove all existing files and subdirectories from the target PGDATA directory and the root directories of any tablespaces.

```
rm -rf /var/lib/postgresql/data/*
```

4. **Restore Base Backup:**
    - **Full Backup:** Restore your chosen base backup (the one you took with pg_basebackup or the low-level API) into the now-empty target PGDATA directory and tablespace locations. Ensure correct ownership (the PostgreSQL system user) and permissions are set. Verify symbolic links for tablespaces.

```
# If using pg_basebackup -Ft (tar format)
tar -xf /path/to/backup/base_backup_20250802_100000.tar -C /var/lib/postgresql/data/
# If using pg_basebackup -Fd (directory format)
cp -rp /path/to/backup/base_backup_20250802_100000/* /var/lib/postgresql/data/
chown -R postgres:postgres /var/lib/postgresql/data
chmod -R 0700 /var/lib/postgresql/data
```

    - **Incremental Backup:** If restoring an incremental backup, first restore the latest incremental backup and *all previous full and incremental backups it depends on* into separate temporary directories. Then use pg_combinebackup to merge them into a synthetic full backup in the target PGDATA directory.

```
# Assuming incremental_backup_latest and full_backup_earlier are your backups
pg_combinebackup -D /var/lib/postgresql/data /path/to/incremental_backup_latest /path/to/full_backup_earlier
chown -R postgres:postgres /var/lib/postgresql/data
chmod -R 0700 /var/lib/postgresql/data
```

5. **Handle pg_wal/ and Unarchived WALs:**

- Remove any files that might have been restored into pg_wal/ from the base backup (they're likely obsolete).
- If pg_wal/ was a symbolic link and wasn't recreated during the base backup restore, recreate it with proper permissions.
- Copy any unarchived WAL segments saved in step 2 into the (now empty) pg_wal/ directory. Copy, don't move, in case you need to restart the recovery process.

rm -rf /var/lib/postgresql/data/pg_wal/* # Clean any old WALs from base backup

cp -rp /var/lib/postgresql/data_pre_recovery_backup/pg_wal/* /var/lib/postgresql/data/pg_wal/ # Copy any unarchived WALs

6. **Configure Recovery:**
   - Create a file named recovery.signal directly in the root of the cluster data directory (PGDATA). This file tells PostgreSQL to enter recovery mode.
   - Edit postgresql.conf to set the restore_command. This command tells PostgreSQL how to retrieve archived WAL files.

Ini, TOML

In postgresql.conf

Set primary_conninfo if this will be a standby

primary_conninfo = 'host=primary_host port=5432 user=replication_user application_name=my_standby'

restore_command = 'cp /mnt/server/archivedir/%f %p' # Unix example

restore_command = 'copy "C:\\server\\archivedir\\%f" "%p"' # Windows example

For S3: restore_command = 's3cmd get s3://your-bucket/path/%f %p'

Optional: Specify target for PITR

recovery_target_time = '2025-08-01 14:30:00 AEST'

recovery_target_xid = '123456789'

recovery_target_name = 'my_recovery_point'

recovery_target_lsn = '0/16B00000'

recovery_target_action = 'pause' # or 'promote', 'shutdown'

- restore_command:
  - %f: Placeholder for the WAL file name (e.g., 000000010000000A900000065).
  - %p: Placeholder for the path where the WAL file should be copied (e.g., /var/lib/postgresql/data/pg_wal/000000010000000A900000065).
  - **Crucial:** This command must return a zero exit status for success.

- **Temporary pg_hba.conf:** Optionally, modify pg_hba.conf to temporarily restrict connections to prevent ordinary users from accessing the database during recovery.

7. **Start the Server:**

sudo systemctl start postgresql

- PostgreSQL will enter recovery mode, read the recovery.signal file, and begin replaying WAL files from the base backup's starting point up to the specified recovery_target (or the end of the archived WALs if no target is set).
- If recovery is interrupted, simply restart the server, and it will resume from where it left off.

8. **Verify and Finalize:**

- Upon successful completion of recovery (reaching the recovery_target or the end of available WALs), the server will automatically remove recovery.signal and begin normal database operations.
- Inspect the database contents to ensure it's in the desired state.
- If satisfied, restore pg_hba.conf to its normal settings to allow users to connect. If not, return to step 1 and try again with different recovery parameters.

---

4. Timelines

When performing PITR, PostgreSQL uses the concept of **timelines**. Each time recovery completes, a new timeline is created. This is crucial if you need to recover to a point *before* a previous recovery and then explore a different branch of history. WAL files from different timelines are distinctly named. This ensures that you don't accidentally mix WAL files from different recovery paths.

5. Tips and Examples

- **Test Your Backups:** Regularly test your entire backup and recovery procedure. The only good backup is a tested one.
- **Automate:** Automate your base backups and WAL archiving as much as possible using cron jobs or a dedicated backup solution.
- **Monitor:** Set up robust monitoring for WAL archiving status, disk space for pg_wal/, and archive storage.
- **pg_receivewal:** For streaming WAL from a primary, pg_receivewal (part of pg_basebackup) is often preferred over archive_command for its robustness and efficiency, especially for standbys.

6. Caveats

- **Storage Requirements:** Continuous archiving demands significant storage for both base backups and the continuous stream of WAL files. Plan your storage capacity carefully.
- **Management Complexity:** This is the most complex backup strategy to set up and manage, requiring a deeper understanding of PostgreSQL internals.
- **Not for Configuration Files:** WAL archiving only covers database data. Configuration files (postgresql.conf, pg_hba.conf, etc.) need to be backed up separately.
- **No Subset Restoration:** You cannot restore individual tables or databases from a continuous archive; it's always an entire cluster restoration.
- **Monitoring archive_command Return Status:** The archive_command *must* return a zero exit status for success. Incorrectly configured commands (e.g., cp -i on GNU/Linux returning 0 on non-overwrite) can lead to silent backup failures.

Continuous archiving provides the highest level of data protection and recovery flexibility for PostgreSQL, making it the go-to solution for production environments where data loss must be minimized and rapid, granular recovery is essential.