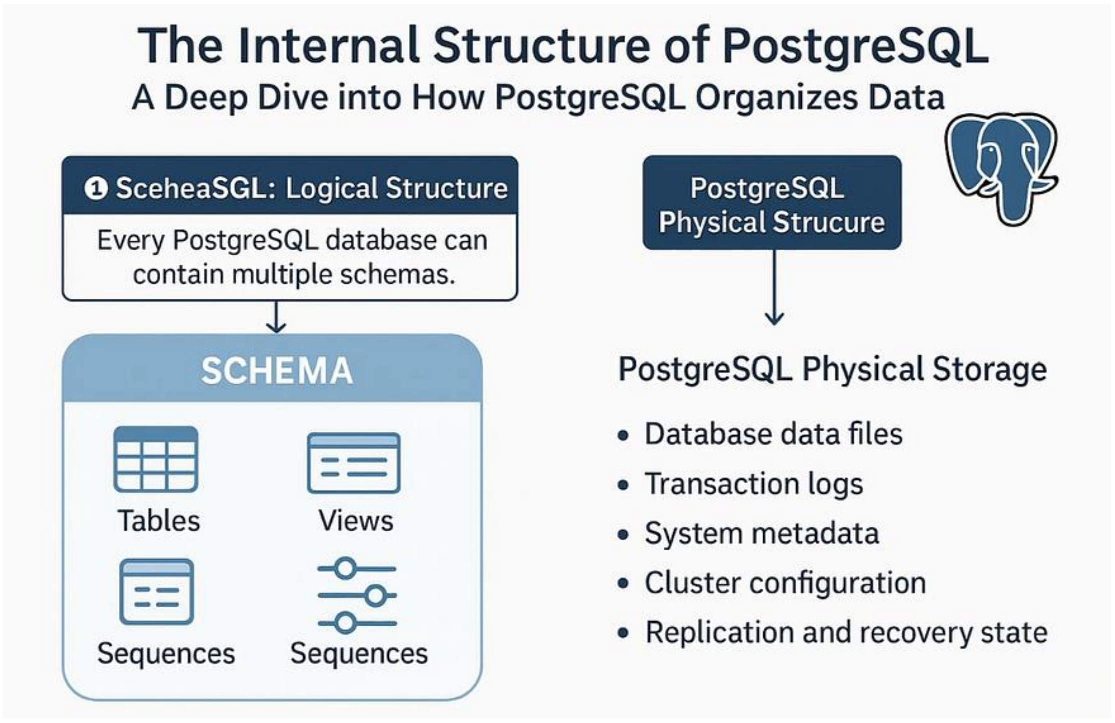
The Internal Structure of PostgreSQL


The Internal Structure of PostgreSQL
A Deep Dive into How PostgreSQL Organizes Data
❶ SceheaSGL: Logical Structure
Every PostgreSQL database can contain multiple schemas.
SCHEMA
Tables
Views
Sequences
Sequences
PostgreSQL Physical Strucure
PostgreSQL Physical Storage
• Database data files
• Transaction logs
• System metadata
• Cluster configuration
• Replication and recovery state

PostgreSQL is one of the most powerful and popular open-source relational database systems used in production today. But while most people interact with PostgreSQL through SQL queries, tables, and views, very few understand *how* PostgreSQL internally stores and manages data.

In this article, we will take a comprehensive look at PostgreSQL's internal architecture — both its **logical structure** (what users see) and its **physical structure** (how data is actually stored on disk).

Understanding this internal design not only helps you appreciate PostgreSQL's reliability but also equips you to better manage performance, backups, and troubleshooting as your databases scale.

Understanding PostgreSQL’s Logical Structure: A Deep Dive into Internal Data Organization

PostgreSQL is not just a database; it’s an incredibly sophisticated system that hides tremendous complexity behind its simple SQL interface. At the heart of this system lies its **logical structure** — the way data is modeled and organized for users, developers, and applications.

In this guide, we’ll explore PostgreSQL’s logical structure in great detail, including tables, views, indexes, sequences, functions, schemas, and tablespaces. Understanding these elements is essential for anyone designing, managing, or optimizing PostgreSQL databases.

What Is PostgreSQL Logical Structure?

In simple terms:

- **Logical structure** = how data is organized from the user’s point of view (schema objects).
- **Physical structure** = how data is actually stored on disk (files, directories, WAL, etc).

The logical structure includes all the objects you manipulate with SQL: tables, views, sequences, indexes, and more.

Logical structures exist **inside schemas** and can be further organized using tablespaces, but users are shielded from the complexities of how PostgreSQL physically stores this data.

□ Key Components of PostgreSQL Logical Structure

1□ Schemas: The Foundation Layer

In PostgreSQL, **schemas serve as the primary organizational layer** within a database. They function like folders or namespaces, allowing multiple database objects to coexist without name conflicts.

Instead of placing all tables, views, indexes, and functions into a single shared space, PostgreSQL uses schemas to group related objects together. This not only keeps databases well-organized but also helps with security, multi-tenancy, and ease of maintenance.

- Every PostgreSQL database can contain multiple schemas.
- Each schema can store its own:
- Tables
- Views
- Indexes
- Sequences
- Functions

By using schemas, you can:

- Isolate objects logically.
- Prevent naming conflicts (since the same object name can exist in different schemas).
- Simplify permission management by granting or revoking access at the schema level.

By default, when you create a new PostgreSQL database, it comes with a pre-created schema called **`public`**. If you do not explicitly specify a schema while creating database objects, PostgreSQL automatically places them into this public schema.

Example

For example, you can create a new schema called `finance` using:

```
CREATE SCHEMA finance;
```

After creating the schema, you can create tables or other objects inside it:

```
CREATE TABLE finance.transactions (...);
```

In this case, the `transactions` table is created inside the `finance` schema, separate from other schemas or the default `public` schema.

2□ Heap Tables (Base Tables)

Tables are the core structure in any relational database, and PostgreSQL heavily relies on them to store actual data. Specifically, PostgreSQL uses **heap tables** as its default storage format for rows. In heap tables, data is stored in the order it is inserted, but without any guaranteed physical ordering. Each row occupies a separate location in the data files.

Characteristics of Heap Tables

- Rows are inserted in any available space, with no guaranteed sequence.
- Due to PostgreSQL's **Multi-Version Concurrency Control (MVCC)** system, multiple versions of the same row (called *tuples*) may exist simultaneously.
- When rows are updated or deleted, the old versions are not immediately removed. These outdated versions are known as **dead tuples**.
- To prevent these dead tuples from accumulating and wasting space, PostgreSQL uses a background process called **autovacuum** that periodically cleans them up.

MVCC in Heap Tables

PostgreSQL’s MVCC mechanism allows multiple transactions to read and write to the same table at the same time without locking each other out.

- Readers and writers operate concurrently, ensuring high performance and reduced contention.
- Every transaction sees a consistent snapshot of the database at the moment the transaction started.
- Internal system columns like `xmin` (transaction ID when the row was created) and `xmax` (transaction ID when the row was deleted or updated) are used to control which row versions are visible to which transactions.

Example of Table Creation

Below is an example of creating a simple `employees` table using PostgreSQL's heap table structure:

```
CREATE TABLE employees (
    emp_id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    department TEXT,
    hire_date DATE
);
```

In this example:

- `emp_id` uses a serial type to auto-generate unique IDs.
- `name`, `department`, and `hire_date` store the employee’s information.
- The `PRIMARY KEY` constraint ensures that each `emp_id` is unique.

3□ Views (Virtual Tables) and Synonyms

In PostgreSQL, **views** are virtual tables created by saving SQL queries. Unlike regular tables, views do not store any actual data themselves. Instead, whenever a view is queried, PostgreSQL executes the underlying SQL query and returns the current data results dynamically.

Key Characteristics of Views

- Views are simply stored queries that act like tables.
- They provide a way to simplify access to data without duplicating storage.
- Every time you query a view, PostgreSQL fetches the latest data by running the stored query.

Benefits of Using Views

Views are useful for many reasons, including:

- **Simplifying complex joins:**
  You can hide complicated joins and calculations behind a simple view, making queries easier for end users.
- **Implementing security layers:**
  Views can restrict access to sensitive columns or rows, supporting row-level and column-level security.
- **Abstracting business logic:**
  Business rules can be embedded into views, allowing consistent query behavior across different applications and users.

Example of Creating a View

The following example creates a view called `active_employees`, which shows only the employees who have an "ACTIVE" status:

```
CREATE VIEW active_employees AS
SELECT emp_id, name, department
FROM employees
WHERE status = 'ACTIVE';
```

Now, instead of writing the full query each time, users can simply run:

```
SELECT * FROM active_employees;
```

PostgreSQL will automatically execute the stored query and return the latest results.

Synonyms in PostgreSQL

In some relational databases like Oracle, **synonyms** are special database objects that serve as alternative names (aliases) for other objects such as tables, views, or sequences. They allow users to access objects without needing to know the full name or schema location.

However, **PostgreSQL does not natively support synonyms** as a separate object type. Instead, similar functionality can be achieved in PostgreSQL using other features.

How Synonym-Like Behavior Can Be Achieved in PostgreSQL

- **Using Views:**
  You can create a view that references the target object. The view acts as an alias, allowing users to query it as if it were the original table or view.
- Example:

```
CREATE VIEW my_synonym AS SELECT * FROM actual_schema.actual_table;
```

- **Using Schema Search Path:**
  PostgreSQL allows you to modify the `search_path`, which determines which schemas PostgreSQL searches when you refer to objects without a schema prefix.
- By adjusting the search path, users can reference objects without needing to know the full schema-qualified name.
- Example:

```
SET search_path TO desired_schema, public;
```

Summary

- PostgreSQL does not have native synonym objects.
- Synonym-like functionality can still be implemented using:
- Views
- Schema search path configurations

4□ Indexes: The Key to Performance

Indexes are essential for improving query performance in PostgreSQL. They allow the database to quickly locate rows that match search conditions without scanning the entire table, which can save significant time for large datasets. Without indexes, PostgreSQL would need to perform a full table scan for every query, leading to slower response times as data grows.

Types of Indexes in PostgreSQL

PostgreSQL supports multiple types of indexes, each optimized for specific use cases:

B-tree (Default):

- Most common index type.
- Supports equality (`=`) and range queries (`<`, `>`, `BETWEEN`).
- Automatically used for primary keys and unique constraints.

Hash:

- Optimized for equality searches (`=` only).
- Rarely used because B-tree handles most equality cases efficiently.

GIN (Generalized Inverted Index):

- Ideal for full-text search, array columns, and JSONB fields.
- Enables efficient searching inside composite or nested data structures.

GiST (Generalized Search Tree):

- Used for spatial and geometric data types.
- Supports range types, geometric shapes, and complex search conditions.

SP-GiST (Space-partitioned GiST):

- Efficient for hierarchical data and non-balanced tree structures.
- Suitable for IP ranges, network subnets, and partitioned datasets.

BRIN (Block Range Index):

- Best for large, append-only tables with naturally ordered data.
- Uses less storage and is faster to maintain but less precise than B-tree.

Tradeoffs of Using Indexes

While indexes can dramatically speed up SELECT queries, they come with certain tradeoffs:

- **Additional Storage:**
  Indexes require extra disk space to store their structures alongside the table data.
- **Slower Writes:**
  Every INSERT, UPDATE, or DELETE operation must also update all relevant indexes, which can slow down write-heavy workloads.
- **Complexity in Design:**
  Poorly designed indexes can consume storage and hurt performance if not used effectively by queries.

Creating an Index Example

Here's a simple example of how to create an index on the `department` column of the `employees` table:

```
CREATE INDEX idx_department ON employees(department);
```

Once created, PostgreSQL can use this index to quickly find employees by department without scanning the entire table.

5 Sequences: Auto-Generating Unique Numbers

In PostgreSQL, **sequences** are special database objects used to generate unique numeric values, most commonly for primary keys. They act like high-performance counters that can increment automatically as new rows are inserted into a table. Sequences are very efficient and are designed to handle multiple concurrent transactions safely.

Features of Sequences

- **Non-blocking and highly concurrent:**
  Sequences can generate unique numbers for many simultaneous transactions without causing locks or conflicts.
- **Supports gaps in numbering:**
  If a transaction that generated a new sequence value is rolled back, PostgreSQL does not reuse the skipped value, resulting in gaps. This is normal behavior to ensure concurrency and avoid contention.
- **Fully customizable:**
  Sequences can be configured to:
- Set the starting value (`START`).
- Define the increment (`INCREMENT`).
- Specify minimum and maximum values.
- Determine whether the sequence should cycle when reaching the limit.
- Control caching to improve performance when generating numbers.

Create Sequence Example

You can create a standalone sequence using:

```
CREATE SEQUENCE emp_id_seq START 1 INCREMENT 1;
```

This creates a sequence named `emp_id_seq` that starts counting from 1 and increments by 1 for each new value generated.

Using Sequences with Tables

You can reference the sequence directly in your table definition to auto-generate primary keys:

```
CREATE TABLE employees (
    emp_id INTEGER DEFAULT nextval('emp_id_seq'),
    name TEXT
);
```

Here, `nextval('emp_id_seq')` automatically fetches the next available number from the sequence whenever a new row is inserted.

SERIAL Shortcut

PostgreSQL also provides a shorthand way to create auto-incrementing primary keys using the `SERIAL` keyword:

```
CREATE TABLE employees (
    emp_id SERIAL PRIMARY KEY,
    name TEXT
);
```

When you use `SERIAL`, PostgreSQL automatically:

- Creates a sequence behind the scenes.
- Sets the default value of the column to pull from the sequence.
- Ensures uniqueness if combined with `PRIMARY KEY`.

Internally, this is equivalent to manually creating the sequence and setting up the default value.

6 Functions and Stored Procedures: Reusable Business Logic

In PostgreSQL, **functions** allow you to store reusable business logic directly inside the database. By using functions, you bring computation closer to the data, which can improve performance, simplify application design, and ensure consistency across multiple systems.

Advantages of Functions

- **Centralized business logic:**
  You can define important rules, calculations, and data processing logic directly in the database, reducing duplication in application code.
- **Reusable across multiple applications:**
  Once created, functions can be called by any client or application that connects to the database, ensuring consistent behavior everywhere.
- **Supports procedural logic, conditionals, and loops:**
  PostgreSQL functions can include complex control structures like `IF`, `LOOP`, and error handling, allowing for advanced operations beyond simple SQL statements.
- **Supports transaction control inside procedures:**
  Certain types of functions (procedures) allow you to control transactions within the function, giving more flexibility for complex workflows.

Supported Languages

PostgreSQL supports multiple languages for writing functions:

- **PL/pgSQL:**
  The native procedural language for PostgreSQL; supports full procedural logic and tight SQL integration.
- **SQL:**
  Allows you to define simple, inline SQL functions.

- **PL/Python, PL/Perl, PL/Tcl, and others:**
  Supported via extensions for advanced or specialized processing needs.

Example of a PL/pgSQL Function

Here’s a simple function that calculates a 10% bonus based on a salary input:

```
CREATE FUNCTION calculate_bonus(salary NUMERIC) RETURNS NUMERIC AS $$
BEGIN
    RETURN salary * 0.10;
END;
$$ LANGUAGE plpgsql;
```

Invoking the Function

Once created, you can call the function like this:

```
SELECT calculate_bonus(50000);
```

This query returns the calculated bonus for a salary of 50,000.

7 Triggers (Closely Related to Functions)

In PostgreSQL, **triggers** are special database objects that automatically execute functions when certain events occur on a table or view. Triggers help automate tasks inside the database, ensuring that certain actions happen automatically whenever data changes.

When Triggers Are Used

Triggers are commonly used for:

- **Audit logging:**
  Automatically record changes made to tables, such as who updated a row and when, for tracking and auditing purposes.
- **Enforcing business rules:**
  Automatically check or modify data to ensure that it meets business or validation requirements before being saved.
- **Cascading updates:**
  Ensure that related data in other tables is updated or deleted when changes occur, helping maintain data consistency across related tables.

Example of Creating a Trigger

Here's an example of a trigger that runs a function called `audit_changes` whenever an update occurs on the `employees` table:

```
CREATE TRIGGER log_update AFTER UPDATE ON employees
FOR EACH ROW EXECUTE FUNCTION audit_changes();
```

- **AFTER UPDATE:**
  The trigger fires after an update operation has completed.
- **FOR EACH ROW:**
  The trigger executes for every row that is updated, not just once per statement.
- **EXECUTE FUNCTION:**
  Specifies which function to run when the trigger fires.

□ Tablespaces: Logical Objects Mapped to Physical Storage

While logical objects (like tables, indexes, and sequences) define **what** data the database stores, **tablespaces** control **where** that data physically resides on disk. Tablespaces give database administrators more control over storage management, allowing them to optimize how and where data is stored, especially in systems with multiple storage devices.

Why Tablespaces Matter

Using tablespaces provides several important benefits:

- **Control storage layout:**
  Administrators can choose specific directories or disks for different data objects.
- **Spread large datasets across multiple storage volumes:**
  Helps balance I/O load across multiple disks and prevents any single disk from becoming a bottleneck.
- **Tune performance for critical tables or indexes:**
  High-traffic tables or indexes can be placed on faster storage (e.g., SSDs) to improve query performance.

Built-In Tablespaces

PostgreSQL comes with two default tablespaces:

pg_default:

- This is the default location where most user-created objects (tables, indexes, sequences) are stored.
- Physically located at:
  `$PGDATA/base/`

pg_global:

- This stores cluster-wide objects that apply across all databases, including:
- Roles (users and permissions)

- System catalogs
- Configuration settings
- Physically located at:
  `$PGDATA/global/`

Custom Tablespaces

In addition to the built-in tablespaces, PostgreSQL allows you to create custom tablespaces to better manage storage needs.

- Custom tablespaces are useful for:
- Isolating high-performance data on faster disks.
- Organizing storage for large tables.
- Improving overall system performance.

Create Tablespace Example

You can create a new tablespace on a specific storage location like this:

```
CREATE TABLESPACE fast_ssd LOCATION '/mnt/fast_ssd';
```

Use Tablespace Example

After creating a tablespace, you can store new tables or indexes in it by specifying the tablespace name during creation:

```
CREATE TABLE large_table (
    id SERIAL PRIMARY KEY,
    data TEXT
) TABLESPACE fast_ssd;
```

This places the `large_table` data files in the `fast_ssd` tablespace location rather than the default storage directory.

PostgreSQL Physical Structure: How PostgreSQL Organizes Data on Disk

PostgreSQL is admired for its robust design, data integrity, and high availability. Much of this strength comes from its sophisticated internal architecture, which not only manages logical database objects but also carefully controls **how data is stored physically on disk**.

In this article, we will take an in-depth look at PostgreSQL's physical storage structure — the part most users never see, but every serious DBA and architect should understand.

□ PostgreSQL Physical Storage Starts at `$PGDATA`

When you install PostgreSQL, you specify a **data directory**, commonly referred to as `$PGDATA`. This directory serves as the root location for everything PostgreSQL needs to run. It includes:

- Database data files
- Transaction logs
- System metadata
- Cluster configuration
- Replication and recovery state

PostgreSQL manages every object, transaction, and piece of internal state by reading from and writing to files within this directory.

□ Core Directories Inside `$PGDATA`

Let's break down the most important subdirectories in PostgreSQL's data directory. Understanding each part helps you manage backups, replication, disaster recovery, and performance optimization.

1⃣ `base/` — The Main Data Storage

The `base/` directory is one of the most important components of PostgreSQL’s physical storage system. This is where the **actual user data** resides on disk. Each PostgreSQL database within the cluster is assigned its own subdirectory inside `base/`, which contains the files that store all the tables, indexes, sequences, and other related data objects for that particular database.

Key Characteristics

- **Each database has its own subdirectory:**
  Every PostgreSQL database is uniquely identified by an internal number called an **Object ID (OID)**. This OID is used to name the database’s subdirectory inside `base/`.

- **Naming convention:**
  Subdirectories inside `base/` are named after their respective database OIDs. These OIDs are assigned automatically by PostgreSQL when the database is created.

Inside Each Database Directory

- **Tables, indexes, and sequences are stored as individual files:**
  Each object (such as a table or index) is stored in a separate file. These files are also named using OIDs assigned to each object.

- **Large tables are divided into multiple segments:**
  PostgreSQL limits the size of any single data file (usually to 1GB per file). If a table grows larger than this limit, it is automatically split into multiple files (segments) to handle the size.

Example Directory Structure

For example, consider the following path inside the `base/` directory:

```
$PGDATA/base/16384/24576
```

- `16384` is the database’s OID (i.e., which database this belongs to).
- `24576` is the OID of a specific table or index within that database.

By reading these file paths, experienced DBAs can directly map files on disk to specific database objects.

> ☐ *Why This Matters*

- When performing **physical backups**, it’s important to copy these files accurately to ensure data consistency.
- During **corruption recovery** or advanced troubleshooting, you may need to analyze these files to identify specific tables or indexes that are affected.
- In **forensic analysis**, understanding the file structure helps to map disk files back to logical database objects.

2☐ `global/` — Cluster-Wide Objects

The `global/` directory inside PostgreSQL’s data directory (`$PGDATA`) stores **cluster-wide information** — data that applies to the entire PostgreSQL instance, not just a specific database. These objects are shared across all databases managed by the PostgreSQL server.

What `global/` Contains

Since these objects affect the full PostgreSQL cluster, they are stored globally to ensure consistency across all databases within the same instance. The `global/` directory contains:

- **System catalogs:**
  These are internal PostgreSQL tables that store metadata about the

entire cluster, such as information about all databases, tablespaces, and other system-level objects.

- **Roles and permissions:**
  PostgreSQL stores all user accounts (roles), their associated permissions, and privileges at the cluster level.
- **Cluster-wide settings:**
  Certain configuration values that apply to the whole server, rather than to an individual database, are maintained here.
- **Cluster-level configuration state:**
  Other system files related to transaction control, system state, and PostgreSQL internal bookkeeping are also stored here.

Why These Objects Are Global

- Objects stored here are **not tied to any single database**.
- This design allows PostgreSQL to manage roles, permissions, and configuration uniformly across all databases in the instance.
- Changes made to roles or cluster-wide settings automatically affect every database managed by the server.

Physical Location

The `global/` directory is physically located at:

```
$PGDATA/global/
```

Understanding this directory is essential for tasks like full cluster backups, role management, and recovering cluster-wide configuration if issues arise.

3 `pg_commit_ts/` — Transaction Commit Timestamps

The `pg_commit_ts/` directory in PostgreSQL is responsible for storing **commit timestamps** for completed transactions. These timestamps record the exact time when each transaction was committed in the system.

What It Stores

- This directory maintains a record of **when each transaction was committed**.
- The information allows PostgreSQL to track the commit time of transactions in the system, which can be helpful for various administrative and replication tasks.

Key Characteristics

- **Optional feature:**
  Commit timestamp tracking is not always enabled by default in PostgreSQL.
  It can be controlled using the configuration setting:

```
track_commit_timestamp = on
```

- When enabled, PostgreSQL begins recording the exact commit time of each transaction, adding a small overhead for the extra tracking.

> ☐ *Why `pg_commit_ts/` Is Useful*

- **Conflict resolution in logical replication:**
  In logical replication scenarios, commit timestamps can help determine the correct order of transactions across multiple nodes.
- **Auditing transaction timelines:**
  Administrators can use commit timestamps for compliance, forensic

analysis, or to track the exact timing of specific transactions during audits.

Summary

- `pg_commit_ts/` helps PostgreSQL capture fine-grained transaction timing information.
- It is mostly useful for advanced replication, conflict management, and audit logging.
- Because it's optional, many systems may not have this directory unless commit timestamp tracking is explicitly enabled.

4️⃣ `pg_clog/` (renamed to `pg_xact` in PostgreSQL 10+) — Transaction State

The `pg_clog/` (now called `pg_xact` in PostgreSQL 10 and newer versions) directory plays a critical role in managing **transaction status information**. This directory allows PostgreSQL to keep track of the state of every transaction that occurs in the system.

What It Manages

- This directory stores the **commit and abort status** of transactions.
- It records whether each transaction is:
- **In-progress:** The transaction has started but not yet completed.
- **Committed:** The transaction successfully finished and its changes are permanent.
- **Rolled back (aborted):** The transaction was canceled, and its changes were discarded.

Importance for MVCC

- PostgreSQL's **MVCC (Multi-Version Concurrency Control)** system relies heavily on this transaction state information.

- MVCC allows multiple transactions to read and write data simultaneously while maintaining data consistency.
- PostgreSQL uses this transaction status to decide which version of a row should be visible to each transaction.

PostgreSQL 10+ Change

- Starting from PostgreSQL version 10, the directory name `pg_clog/` was renamed to `pg_xact/` to better reflect its purpose (tracking transaction states).
- The physical location is now:

```
$PGDATA/pg_xact/
```

Why This Information Is Critical

- Without the data in `pg_xact/`, PostgreSQL would not be able to determine:
- Which transactions have completed.
- Which data versions are visible or hidden.
- Which rows are active, deleted, or obsolete.
- This makes `pg_xact/` essential for ensuring **data integrity, isolation, and consistency** in every PostgreSQL operation.

5 `pg_logical/` — Logical Replication State

The `pg_logical/` directory in PostgreSQL is responsible for managing the internal state of **logical replication**. Logical replication allows PostgreSQL to replicate data at a more granular level, such as individual tables or subsets of data, instead of replicating entire databases like physical replication.

What It Manages

- The `pg_logical/` directory stores **metadata** required for logical replication to function correctly.
- This metadata includes:
- **Replication slots:**
  These are objects that keep track of how much WAL (Write-Ahead Log) data needs to be retained for each logical replication subscriber.
- **Logical decoding state:**
  Tracks how far each subscriber has read and applied changes from the WAL stream.

Key Benefits of Logical Replication

- Allows replication of specific tables instead of the whole database.
- Supports more flexible replication use cases, such as:
- Partial data replication
- Real-time data feeds
- Cross-database or cross-version replication
- Zero-downtime upgrades

□ Why `pg_logical/` Is Important

- PostgreSQL uses this directory to track the **current position** of each replication consumer (subscriber).
- This ensures that replication clients always receive the correct stream of data changes without missing or duplicating transactions.
- Without this tracking information, logical replication could not resume correctly after restarts or failures.

6□ `pg_stats/` — Query Planner Statistics

The `pg_stats/` directory in PostgreSQL is responsible for storing **optimizer statistics** that are essential for PostgreSQL's query planner. These statistics help PostgreSQL make smart decisions about how to execute queries efficiently.

What Statistics Are Stored

PostgreSQL collects and stores various types of data to help estimate the cost and performance of query plans:

- **Row counts:**
  The estimated number of rows in a table.
- **Data distributions:**
  Information about how data values are spread across columns.
- **Histograms:**
  Representations of the frequency distribution of column values.
- **Correlation estimates:**
  Measurements of how values in one column relate to values in another, which help optimize index usage and join conditions.

Why These Statistics Are Important

- The **query planner's accuracy** depends heavily on having up-to-date and accurate statistics.
- Without reliable statistics, PostgreSQL might:
- Choose suboptimal execution plans.
- Perform unnecessary full table scans.
- Use inefficient indexes.
- Experience slower query performance.
- Regularly running the `ANALYZE` or `VACUUM ANALYZE` command helps keep these statistics current.

Physical Path

- The statistics are physically stored in:

```
$PGDATA/pg_stats/
```

Note for Newer PostgreSQL Versions

- In more recent PostgreSQL versions, much of the statistics system has been redesigned:
- Many statistics now reside in **shared memory** and **system catalogs** rather than directly on disk.
- However, `pg_stats/` still plays a role in certain internal processes and retains backward compatibility for some functions.

7️⃣ `pg_wal/` (formerly `pg_xlog`) — Write-Ahead Log (WAL)

The `pg_wal/` directory is one of the most critical parts of PostgreSQL's physical storage system. It stores the **Write-Ahead Log (WAL)** files, which play a central role in ensuring the database's durability, consistency, and crash recovery capabilities.

What WAL Does

- Before any changes are written to the main data files on disk, PostgreSQL first records those changes in WAL files.
- This approach ensures that even if the database crashes unexpectedly, all committed transactions can be safely recovered from the WAL files.

WAL Enables Several Key Features

- **Crash recovery:**
  After a crash, PostgreSQL replays the WAL files to bring the database back to a consistent state, applying all committed changes that may not have been flushed to disk.

- **PITR (Point-In-Time Recovery):**
  Allows administrators to restore the database to any point in time by replaying WAL files up to a specific point.
- **Streaming replication:**
  Standby servers continuously receive WAL data from the primary server to stay synchronized, enabling high availability and failover setups.

Why WAL Is Essential

- WAL is the foundation of PostgreSQL's **ACID compliance**, specifically:
- **Atomicity:** All or nothing transactions.
- **Durability:** Once a transaction commits, its changes survive crashes.
- WAL guarantees that committed data is never lost, even during system failures.

Physical Path

- WAL files are physically stored at:

```
$PGDATA/pg_wal/
```

WAL File Management

- WAL files grow continuously as the database processes transactions.
- To prevent disk space from being exhausted, WAL files must be:
- **Archived:** Moved to long-term storage for backups and PITR.
- **Recycled:** Reused for new WAL entries when old segments are no longer needed.

Proper WAL management is a crucial responsibility for every PostgreSQL DBA.

8⃣ `pg_tblspc/` — Tablespace Symlinks

The `pg_tblspc/` directory in PostgreSQL is used to manage **tablespaces**, which provide flexible control over where data is physically stored on disk. This directory does not contain the actual data files but instead holds **symbolic links** that point to external storage locations defined as user-created tablespaces.

What Tablespaces Do

- **Tablespaces** allow PostgreSQL administrators to place specific tables, indexes, or entire databases on different physical storage devices.
- By distributing data across multiple disks, tablespaces offer:
- **Storage flexibility:** Spread data across multiple storage volumes or devices.
- **Performance tuning:** Place high-read or high-write tables on faster disks like SSDs.
- **Storage cost optimization:** Store rarely used (cold) data on cheaper, slower storage.

How `pg_tblspc/` Works

- The directory contains **symlinks** (shortcuts) that point to the actual physical locations where custom tablespace data is stored.
- These symbolic links allow PostgreSQL to reference and manage external directories as part of the database cluster.
- The physical data files for objects stored in these tablespaces reside **outside of `$PGDATA`**.

Physical Path

- The symbolic links for tablespaces are located at:

```
$PGDATA/pg_tblspc/
```

Why This Is Useful

- With tablespaces, DBAs can optimize storage layouts based on workload characteristics.
- They help avoid storage limitations inside `$PGDATA` and provide more control over disk space usage.

□ Base Directory Substructures — The Templates

Inside PostgreSQL's `base/` directory, along with the directories for individual user-created databases, you'll also find a few **special subdirectories** known as **templates**. These templates play a key role when creating new databases.

1□ `template0` — Pristine System Template

In PostgreSQL, `template0` is one of the two default database templates present in every cluster. It serves as a **pristine system template** that remains unmodified after the PostgreSQL installation.

Key Characteristics of `template0`:

- **Pristine system template:**
  `template0` contains only the original system catalogs and structures that PostgreSQL creates during initialization. It does not include any user-defined objects, extensions, or custom changes.
- **Read-only:**
  This template is protected from modification to ensure that it always remains in its original, clean state. You cannot add or remove objects from `template0`.

- **Primary purpose:**
  PostgreSQL uses `template0` when you want to create a brand-new database that inherits no customizations or extensions. This is especially useful for:
- Creating fully isolated databases.
- Avoiding inherited objects from other templates.
- Ensuring system integrity for certain specialized use cases.

2⃣ `template1` — Default Template for New Databases

In PostgreSQL, `template1` is the **default template** that the system uses when creating most new databases. Unlike `template0`, this template can be customized to include additional objects and configurations.

Key Characteristics of `template1`:

- **Default template:**
  When you create a new database without specifying a template, PostgreSQL automatically uses `template1` as the source.

Customizable:
You can modify `template1` by adding:

- Extensions (e.g., `pgcrypto`, `postgis`)
- Schemas
- Functions
- Tables
- Other database objects

Purpose:
Customizing `template1` allows you to ensure that all newly created databases automatically include your desired default setup. This can save

time when creating multiple databases with common structures or pre-installed features.

- **Flexibility:**
  Since `template1` is editable, administrators often use it to establish standard configurations for development, testing, or production environments.

3 `postgres` — The Default Administrative Database

In PostgreSQL, `postgres` is the **default administrative database** that is automatically created during the installation of the database cluster. It serves as a simple and convenient workspace for administrators to perform system-level tasks.

Key Characteristics of `postgres`:

- **Default administrative database:**
  This database exists primarily for system management, maintenance tasks, and quick administrative operations.
- **Always present after installation:**
  PostgreSQL automatically creates the `postgres` database when initializing the cluster, ensuring that there is always at least one database available for administrative access.

Purpose:

- Run configuration checks.
- Perform quick queries.
- Manage users and roles.
- Test new commands without affecting production data.
- Install and verify extensions.

Safe workspace:
Since it contains no application data by default, it is often used as a neutral space for routine database administration.

□ Why Understanding PostgreSQL's Physical Structure Matters

For most application developers, PostgreSQL's physical storage system remains invisible — they simply interact with SQL queries, tables, and views. However, for **database administrators (DBAs)**, architects, and anyone managing PostgreSQL at the system level, understanding the physical structure is critically important. This knowledge enables better management, maintenance, and optimization of the database system.

Key Reasons Why Physical Structure Knowledge Is Essential:

- □ **Backup & Restore:**
- Knowing exactly which files store your data allows you to create consistent backups.
- Understanding the file layout ensures that backups include all necessary components to fully restore a database if needed.
- ☼ **High Availability & Replication:**
- The **Write-Ahead Log (WAL)** plays a key role in replication.
- Understanding how WAL files work allows you to set up and manage streaming replication efficiently, ensuring minimal downtime and data loss.
- □ **Disaster Recovery:**
- In the event of data corruption or hardware failure, knowing where specific files are located helps you isolate and recover affected parts of the system, reducing the risk of complete data loss.
- □ **Performance Optimization:**

- Proper use of **tablespaces** and an understanding of how PostgreSQL distributes data can help you optimize disk I/O.
- This allows you to balance workloads across multiple storage devices, improve performance, and control storage costs.
- ☐ **Troubleshooting:**
- When issues arise, such as missing data or performance bottlenecks, knowing where tables, indexes, and transaction logs physically reside allows faster diagnosis and resolution of problems.

☐ Conclusion

PostgreSQL's physical storage architecture is designed to be **modular, efficient, and highly reliable**. It carefully separates the logical organization of data (tables, indexes, views, etc.) from the actual physical files and directories that store the data on disk. This separation provides flexibility and stability, making PostgreSQL suitable for a wide range of applications — from small projects to large-scale enterprise systems.

Key Benefits of PostgreSQL's Physical Storage Design:

- **Maximum data integrity:**
  The combination of Write-Ahead Logging (WAL), transaction tracking, and MVCC ensures that data remains safe and consistent even during system failures or crashes.
- **High concurrency:**
  PostgreSQL allows multiple users and applications to read and write data simultaneously without blocking each other, thanks to its efficient concurrency control mechanisms.
- **Efficient crash recovery:**
  WAL files and transaction logs enable PostgreSQL to quickly recover from unexpected shutdowns and maintain data consistency.
- **Fine-grained storage management:**
  With features like tablespaces and separate directories for different

storage components, administrators can optimize disk usage, balance performance, and manage storage costs effectively.

The Value of Understanding the Physical Layer

- While most users only interact with PostgreSQL at the SQL level, understanding how data is physically stored empowers DBAs and system architects to:
- Set up replication and high availability systems confidently.
- Design effective backup and disaster recovery plans.
- Perform advanced troubleshooting and performance tuning.