PostgreSQL Internals Part 1: Understanding database cluster, database and tables

PostgreSQL is one of the most popular and powerful relational database management systems, renowned for its robust features and flexibility. Going deeper into its internals uncovers a complex world though. Understanding the core concepts is much needed for developers, database administrators, and anyone involved in managing or interacting with PostgreSQL databases.

By understanding the core concepts users can optimize performance, troubleshoot issues effectively, and use the full potential of PostgreSQL for their projects.

In this blog series, We are going to discuss the core concepts of PostgreSQL internals starting with database clusters, tables, and tablespaces in PostgreSQL.

Let's dive into the details!

What is a Database Cluster?

A **database cluster** is a collection of multiple **databases** managed by a **PostgreSQL server** running on a single node or instance. It can be referred to as a **data/base** directory.

Base: It is the default directory for the **database cluster**, which contains **subdirectories** for each **database**. Each database's directory is represented by its respective **OID**. Within the **base/DATA_BASE_OID** directory, you'll find specific files for **tables**, **indexes**, and other **database objects**.

Tablespace: It is a distinct **physical** storage location on a **disk**, separate from the default **data** directory, used for storing database objects such as **tables** and **indexes**. **data/pg_tblspc** will contain the **symlink** to the tablespace inside the original data directory. **Tablespaces** allow you to manage storage by distributing data across multiple disk locations, **optimizing performance,** and **balancing load**. They also help segregate data by **application**, **user**, or **environment**, enhancing database structure and **security**.

What is a Database?

In PostgreSQL, a **database** is a **logical** collection of related data objects such as **tables**, **indexes**, **views**, **functions**, and other **database objects**. It is managed by a PostgreSQL server instance and provides a structured environment for **storing**, **organizing**, and **retrieving data**. Each database operates independently, with its own set of **objects**, **permissions**, and **configurations**. Although multiple databases can exist within a single PostgreSQL cluster, they do not share data directly with each other. When we install **PostgreSQL** first time it by default creates a database named **Postgres**.

What is a Database Object?

A **database object** in PostgreSQL refers to any distinct item or entity within a database that can be created, manipulated, or managed through **SQL** commands. These objects represent different aspects of **data storage**, **organization**, and **functionality**. for example

- **Tables**: Structures that store rows of data in a relational format, with columns defining the data types for each field.
- **Indexes**: Objects that improve the speed of data retrieval by creating a quick lookup structure for tables.

- **Views**: Virtual tables that represent a specific query's result set, allowing for reusable and abstracted queries.
- **Sequences**: Objects that generate unique numeric values, often used for auto-incrementing primary keys.
- **Schemas**: Logical containers for organizing and separating database objects, allowing for better data organization and security.
- **Functions**: Custom scripts or procedures written in SQL or other languages (like PL/pgSQL) that perform specific operations or calculations.
- **Triggers**: Actions automatically executed in response to specific events, such as insertions or updates on a table.
- **Constraints**: Rules or conditions applied to tables to ensure data integrity, like primary keys, foreign keys, and unique constraints.
  Each of these database objects plays a unique role in **defining**, **manipulating**, and **maintaining** the data within a PostgreSQL database.

What is an OID?

A short form of **Object identifier** it is a number represented in **unsigned 4-byte integers**. Which is used to uniquely identify the object that we create inside the database

An **unsigned 4-byte integer** has a maximum value of **(2^32 − 1)** which is **4,294,967,295**

When we initialize the data directory(PostgreSQL 16) it creates 3 databases by default **postgres**, **template0**, and **template1** we can check their respective OIDs with the following query

```
postgres=# SELECT datname, oid FROM pg_database WHERE datname = 'postgres';

 datname  | oid

----------+-----

 postgres |   5

(1 row)Copy to Clipboard
```

We can also see the same subdirectories inside the **data/base** directory containing the **OIDs** of the databases

```
postgres@9e7108914f4c:~/data/base$ ls -lhrt

total 12K

drwx------ 2 postgres postgres 4.0K Mar 14 14:08 4

drwx------ 2 postgres postgres 4.0K Mar 14 14:11 5

drwx------ 2 postgres postgres 4.0K Mar 14 14:12 1Copy to Clipboard
```

Similarly, if you have a table inside your database you can check its **OID** via

```
postgres=# SELECT oid, relname FROM pg_class WHERE relname = 'test';

  oid  | relname

-------+---------

16393 | test

(1 row)Copy to Clipboard
```

NOTE: Creating a table with **OID** was deprecated in **PostgreSQL-12** and it is not supported now

What is relfilenode?

It's a value stored in the **pg_class** system table for relation objects (tables, indexes, etc.) It acts as a **numeric identifier** that points to the **actual physical disk file** where the **data** for the relation is stored. For instance, for the table **testing**, we can see its **OID** and **relfilenode** value with the following query

```
postgres=# select oid,relname,relfilenode from pg_class where relname = 'testing';

  oid  | relname | relfilenode

-------+---------+-------------

 16385 | testing |       16385

(1 row)Copy to Clipboard
```

Initially, **oid** and **relfilenode** can be the same and point to the same location on the disk

```
postgres@9e7108914f4c:~/data/base/5$ ls -lhrt 16385

-rw------- 1 postgres postgres 0 Mar 14 20:59 16385Copy to Clipboard
```

But if we issue **REINDEX** or **TRUNCATE** commands value of **relfilenode** can be changed

```
postgres=# truncate testing;

TRUNCATE TABLE

postgres=# select oid,relname,relfilenode from pg_class where relname = 'testing';

  oid  | relname | relfilenode

-------+---------+-------------

 16385 | testing |       16397

(1 row)Copy to Clipboard
```

As we can see **TRUNCATE** changed the value of **relfilenode** from **16385** to **16397** and we can see a new file in our filesystem

```
ls -lhrt 16397

-rw------- 1 postgres postgres 0 Mar 14 21:04 16397Copy to Clipboard
```

NOTE: The value of **relfilenode** can be **0** for system catalog objects

---

Recommended Reading: Optimizing PostgreSQL Cluster Performance, Part 1 – Load Balancing

---

16389.1, and 16389.2 files

In PostgreSQL, database objects like **tables** and **indexes** are stored in files called **segments**. Each segment has a predetermined size, and a table may span multiple segments as its data

volume increases. By default, if the **–with-segsize** option isn't specified during PostgreSQL's source code compilation, the segment size is usually set to **1GB**. When the size of tables and indexes surpasses this limit, PostgreSQL generates additional files named after their respective **relfilenode.1, relfilenode.2**, and so forth, to accommodate the excess data. This behavior can be seen in the example below

```
postgres=# CREATE TABLE my_table (

    id SERIAL PRIMARY KEY,

    number INT

);

CREATE TABLE

postgres=# INSERT INTO my_table (number)

SELECT generate_series(1, 50000000);

INSERT 0 50000000

postgres=# dt+ my_table

                                    List of relations

 Schema |   Name   | Type  |  Owner   | Persistence | Access method |  Size   | Description

--------+----------+-------+----------+-------------+---------------+---------+-------------

 public | my_table | table | postgres | permanent   | heap          | 1729 MB |

(1 row)

postgres=# select oid,relname,relfilenode from pg_class where relname = 'my_table';

  oid  | relname  | relfilenode

-------+----------+-------------

 16385 | my_table |       16385

(1 row)

postgres@9e7108914f4c:~$ ls -lhrt data/base/5/16385 (Tab for auto-complete)

16385      16385.1Copy to Clipboard
```

Free Space Map(FSM) files

Each **table** and **index** relation, except for **hash indexes**, has a **Free Space Map (FSM)** to keep track of available space in the relation's **pages**. It stores all free space-related information alongside **primary** relations. **VACUUM** and **Autovacuum** can change or update **FSM** value when they execute their operations

Visibility Map(VM) files

In PostgreSQL, the system includes a mechanism to monitor pages within **tables** and **indexes**, determining which ones contain only tuples that are visible to all ongoing **transactions**. Each table has its own **visibility map**, a structure that indicates whether individual pages within the table's file are entirely **clean** or contain any **dead tuples**. This visibility map serves as a quick reference for **vacuuming** operations, allowing PostgreSQL to skip over pages that don't require cleanup,

thereby improving vacuuming efficiency. With the visibility map, PostgreSQL can optimize maintenance tasks, reducing the time and resources needed to manage the database's health and performance.

In the below diagram, we can see a table has 2 pages and **Page 0** contains four tuples. **Tuple-2** is determined to be a **dead tuple**, PostgreSQL will remove it and rearrange the remaining tuples to address **fragmentation** or **bloat**. Subsequently, it updates both the **FSM** and **Visibility Map** associated with this page. PostgreSQL repeats this process until it reaches the final page.

Note: Indexes only have individual free space maps and do not have visibility maps.

Dead Tuples in PostgreSQL

In PostgreSQL, a **dead tuple** refers to a row in a table that is no longer needed but has not yet been physically removed from the database. Dead tuples are created as a natural byproduct of PostgreSQL's **Multiversion Concurrency Control (MVCC)** system, which allows concurrent transactions to **read** and **write** without interfering with each other.

Dead Tuples can be generated by

- **Updates**: When a row is updated, PostgreSQL creates a new version of the row (a new tuple) while keeping the old version. The old version becomes a dead tuple because it's no longer part of the active dataset, but it's retained temporarily to ensure transaction consistency and visibility for other transactions that might still reference it.
- **Deletes**: When a row is deleted, it's not immediately removed from the table. Instead, it becomes a dead tuple, allowing other transactions that might be reading the row to complete without error.
NOTE: Every action we do in PostgreSQL is **append-only** so we **DO NOT** perform in-place updates

What are the Tablespaces? and their Benefits

In PostgreSQL, it is a **physical storage area** separate from the default **data directory**. It allows organizing database objects on different storage devices
for **performance** or **administrative** purposes. We can create a new **tablespace** with the following query

```
CREATE TABLESPACE my_tablespace

  OWNER postgres

  LOCATION '/var/lib/postgresql/tbs';

CREATE TABLESPACECopy to Clipboard
```

Upon creating a new tablespace on a specified directory, two key actions occur

Firstly, A new directory gets created by the name of **PG_SERVER_CATALOG** on desired location i.e, **/var/lib/postgresql/PG_16_202307071**

```
ls -lhrt /var/lib/postgresql/tbs/PG_16_202307071/

total 0Copy to Clipboard
```

The **catalog version number** in PostgreSQL is an internal mechanism for maintaining data directory compatibility and preventing inconsistencies. Its format is **YYYYMMDDN** representing the date the number was changed, with **N** representing a simple counter to accommodate for more than one change on the same date.

We can check the catalog version by

```
postgres=# SELECT catalog_version_no FROM pg_control_system();

 catalog_version_no

--------------------

      202307071

(1 row)Copy to Clipboard
```

Secondly, A symlink to the tablespace directory gets created inside the original **data/pg_tblspc** directory

```
postgres@9e7108914f4c:~$ ls -lhrt data/pg_tblspc/

total 0

lrwxrwxrwx 1 postgres postgres 23 Mar 15 01:53 16386 -> /var/lib/postgresql/tbsCopy to Clipboard
```

Creating a **table** within a tablespace associated with the PostgreSQL database will result in the creation of a new directory inside the **PG_16_202307071** directory, bearing the same **OID** as the PostgreSQL database, within the **data/base** directory.

```
postgres=# CREATE TABLE my_table (

    id SERIAL PRIMARY KEY,

    name VARCHAR(50)

) TABLESPACE my_tablespace;

CREATE TABLECopy to Clipboard
```

The **OID** of the PostgreSQL database is **regenerated** and remains unchanged.

```
postgres@9e7108914f4c:~$ ls -lhrt tbs/PG_16_202307071/5/16392

-rw------- 1 postgres postgres 0 Mar 15 01:57 tbs/PG_16_202307071/5/16392Copy to Clipboard
```

Benefits of Tablespaces

- We can assign different storage properties (storage type, filesystem options) to individual tablespaces based on their access patterns and performance requirements.
- Tablespaces can provide flexibility for horizontal scaling. You can add additional storage capacity by adding new tablespaces without affecting existing ones
- Tablespaces can potentially contribute to a layered security approach. By placing sensitive tables in dedicated tablespaces with stricter access controls, you might add an extra layer of protection
In this first part of our series, we've explored fundamental concepts of **database clusters, tables,** and **tablespaces**. Read the next part in the PostgreSQL Internals series here.