**PostgreSQL DBA: Understanding the Architecture**

PostgreSQL is probably the most advanced database in the open source relational database market. It was first released in 1989, and since then, there have been a lot of enhancements. According to db-engines, it is the fourth most used database at the time of writing.

PostgreSQL is probably the most advanced database in the open source relational database market. It was first released in 1989, and since then, there have been a lot of enhancements. According to db-engines, it is the fourth most used database at the time of writing.

## PostgreSQL Architecture

The physical structure of PostgreSQL is very simple. It consists of shared memory and a few background processes and data files. (See Figure 1-1)

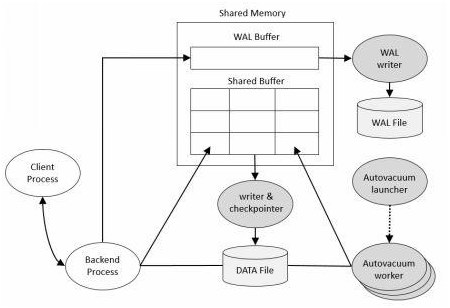


Figure 1-1. PostgreSQL structure

## Shared Memory

Shared Memory refers to the memory reserved for database caching and transaction log caching. The most important elements in shared memory are Shared Buffer and WAL buffers

### Shared Buffer

The purpose of Shared Buffer is to minimize DISK IO. For this purpose, the following principles must be met

* You need to access very large (tens, hundreds of gigabytes) buffers quickly.
* You should minimize contention when many users access it at the same time.
* Frequently used blocks must be in the buffer for as long as possible

### WAL Buffer

The WAL buffer is a buffer that temporarily stores changes to the database. The contents stored in the WAL buffer are written to the WAL file at a predetermined point in time. From a backup and recovery point of view, WAL buffers and WAL files are very important.

## PostgreSQL Process Types

PostgreSQL has four process types.

1. Postmaster (Daemon) Process
2. Background Process
3. Backend Process
4. Client Process

### Postmaster Process

The Postmaster process is the first process started when you start PostgreSQL. At startup, performs recovery, initialize shared memory, and run background processes. It also creates a backend process when there is a connection request from the client process. (See Figure 1-2)

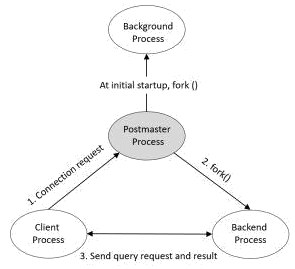
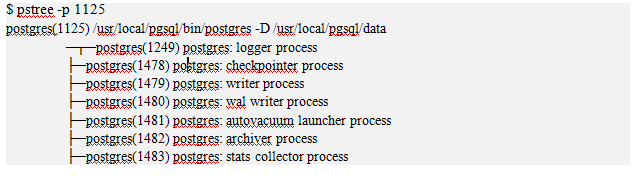


Figure 1-2. Process relationship diagram

If you check the relationships between processes with the pstree command, you can see that the Postmaster process is the parent process of all processes. (For clarity, I added the process name and argument after the process ID)



### Background Process

The list of background processes required for PostgreSQL operation are as follows. (See Table 1-1)

| Process | Role |
| --- | --- |
| logger | Write the error message to the log file. |
| checkpointer | When a checkpoint occurs, the dirty buffer is written to the file. |
| writer | Periodically writes the dirty buffer to a file. |
| wal writer | Write the WAL buffer to the WAL file. |
| Autovacuum launcher | Fork autovacuum worker when autovacuum is enabled.It is the responsibility of the autovacuum daemon to carry vacuum operations on bloated tables on demand |
| archiver | When in Archive.log mode, copy the WAL file to the specified directory. |
| stats collector | DBMS usage statistics such as session execution information ( pg\_stat\_activity ) and table usage statistical information ( pg\_stat\_all\_tables ) are collected. |

### Backend Process

The maximum number of backend processes is set by the max\_connections parameter, and the default value is 100. The backend process performs the query request of the user process and then transmits the result. Some memory structures are required for query execution, which is called local memory. The main parameters associated with local memory are:

1. work\_mem Space used for sorting, bitmap operations, hash joins, and merge joins. The default setting is 4 MB.
2. Maintenance\_work\_mem Space used for Vacuum and CREATE INDEX . The default setting is 64 MB.
3. Temp\_buffers Space used for temporary tables. The default setting is 8 MB.

### Client Process

Client Process refers to the background process that is assigned for every backend user connection.Usually the postmaster process will fork a child process that is dedicated to serve a user connection.

# Process and Memory Architecture

In this chapter, the process architecture and memory architecture in PostgreSQL are summarized to help to read the subsequent chapters. If you are already familiar with them, you may skip over this chapter.

## 2.1. Process Architecture

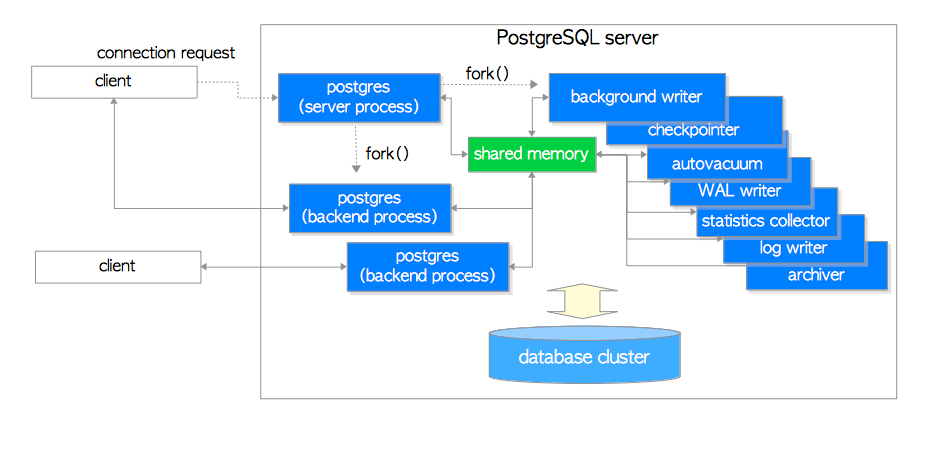
PostgreSQL is a client/server type relational database management system with the multi-process architecture and runs on a single host.

A collection of multiple processes cooperatively managing one database cluster is usually referred to as a *'PostgreSQL server'*, and it contains the following types of processes:

* A **postgres server process** is a parent of all processes related to a database cluster management.
* Each **backend process** handles all queries and statements issued by a connected client.
* Various **background processes** perform processes of each feature (e.g., VACUUM and CHECKPOINT processes) for database management.
* In the **replication associated processes**, they perform the streaming replication. The details are described in Chapter 11.
* In the **background worker process** supported from version 9.3, it can perform any processing implemented by users. As not going into detail here, refer to the official document.

In the following subsections, the details of the first three types of processes are described.

**Fig. 2.1. An example of the process architecture in PostgreSQL.**



This figure shows processes of a PostgreSQL server: a postgres server process, two backend processes, seven background processes, and two client processes. The database cluster, the shared memory, and two client processes are also illustrated.

### 2.1.1. Postgres Server Process

As already described above, a *postgres server process* is a parent of all in a PostgreSQL server. In the earlier versions, it was called ‘postmaster’.

By executing the pg\_ctl utility with *start* option, a postgres server process starts up. Then, it allocates a shared memory area in memory, starts various background processes, starts replication associated processes and background worker processes if necessary, and waits for connection requests from clients. Whenever receiving a connection request from a client, it starts a backend process. (And then, the started backend process handles all queries issued by the connected client.)

A postgres server process listens to one network port, the default port is 5432. Although more than one PostgreSQL server can be run on the same host, each server should be set to listen to different port number in each other, e.g., 5432, 5433, etc.

### 2.1.2. Backend Processes

A backend process, which is also called *postgres*, is started by the postgres server process and handles all queries issued by one connected client. It communicates with the client by a single TCP connection, and terminates when the client gets disconnected.

As it is allowed to operate only one database, you have to specify a database you want to use explicitly when connecting to a PostgreSQL server.

PostgreSQL allows multiple clients to connect simultaneously; the configuration parameter *max\_connections* controls the maximum number of the clients (default is 100).

If many clients such as WEB applications frequently repeat the connection and disconnection with a PostgreSQL server, it increases both costs of establishing connections and of creating backend processes because PostgreSQL has not implemented a native connection pooling feature. Such circumstance has a negative effect on the performance of database server. To deal with such a case, a pooling middleware (either pgbouncer or pgpool-II) is usually used.

### 2.1.3. Background Processes

Table 2.1 shows a list of background processes. In contrast to the postgres server and the backend process, it is impossible to explain each of the functions simply, because these functions depend on the individual specific features and PostgreSQL internals. Thus, in this chapter, only introductions are made. Details will be described in the following chapters.

|  |  |  |
| --- | --- | --- |
| **Table 2.1: background processes.** | | |
| **process** | **Description** | **reference** |
| background writer | In this process, dirty pages on the shared buffer pool are written to a persistent storage (e.g., HDD, SSD) on a regular basis gradually. (In version 9.1 or earlier, it was also responsible for checkpoint process.) | Section 8.6 |
| checkpointer | In this process in version 9.2 or later, checkpoint process is performed. | Section 8.6, Section 9.7 |
| autovacuum launcher | The autovacuum-worker processes are invoked for vacuum process periodically. (More precisely, it requests to create the autovacuum workers to the postgres server.) | Section 6.5 |
| WAL writer | This process writes and flushes periodically the WAL data on the WAL buffer to persistent storage. | Section 9.9 |
| statistics collector | In this process, statistics information such as for pg\_stat\_activity and for pg\_stat\_database, etc. is collected. |  |
| logging collector (logger) | This process writes error messages into log files. |  |
| archiver | In this process, archiving logging is executed. | Section 9.10 |

The actual processes of a PostgreSQL server is shown here. In the following example, one postgres server process (pid is 9687), two backend processes (pids are 9697 and 9717) and the several background processes listed in Table 2.1 are running. See also Fig. 2.1.

postgres> pstree -p 9687

-+= 00001 root /sbin/launchd

\-+- 09687 postgres /usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data

|--= 09688 postgres postgres: logger process

|--= 09690 postgres postgres: checkpointer process

|--= 09691 postgres postgres: writer process

|--= 09692 postgres postgres: wal writer process

|--= 09693 postgres postgres: autovacuum launcher process

|--= 09694 postgres postgres: archiver process

|--= 09695 postgres postgres: stats collector process

|--= 09697 postgres postgres: postgres sampledb 192.168.1.100(54924) idle

\--= 09717 postgres postgres: postgres sampledb 192.168.1.100(54964) idle in transaction

## 2.2. Memory Architecture

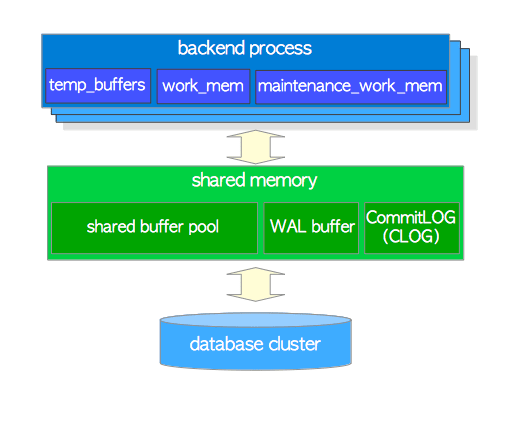
Memory architecture in PostgreSQL can be classified into two broad categories:

* Local memory area – allocated by each backend process for its own use.

Shared memory area – used by all processes of a PostgreSQL server.

In the following subsections, those are briefly descibed.

**Fig. 2.2. Memory architecture in PostgreSQL.**



### 2.2.1. Local Memory Area

Each backend process allocates a local memory area for query processing; each area is divided into several sub-areas – whose sizes are either fixed or variable. Table 2.2 shows a list of the major sub-areas. The details will be described in the following chapters.

|  |  |  |
| --- | --- | --- |
| **Table 2.2: Local memory area** | | |
| **sub-area** | **Description** | **reference** |
| work\_mem | Executor uses this area for sorting tuples by ORDER BY and DISTINCT operations, and for joining tables by merge-join and hash-join operations. | Chapter 3 |
| maintenance\_work\_mem | Some kinds of maintenance operations (e.g., VACUUM, REINDEX) use this area. | Section 6.1 |
| temp\_buffers | Executor uses this area for storing temporary tables. |  |

### 2.2.2. Shared Memory Area

A shared memory area is allocated by a PostgreSQL server when it starts up. This area is also divided into several fix sized sub-areas. Table 2.3 shows a list of the major sub-areas. The details will be described in the following chapters.

|  |  |  |
| --- | --- | --- |
| **Table 2.3: Shared memory area** | | |
| **sub-area** | **Description** | **reference** |
| shared buffer pool | PostgreSQL loads pages within tables and indexes from a persistent storage to here, and operates them directly. | Chapter 8 |
| WAL buffer | To ensure that no data has been lost by server failures, PostgreSQL supports the WAL mechanism. WAL data (also referred to as XLOG records) are transaction log in PostgreSQL; and WAL buffer is a buffering area of the WAL data before writing to a persistent storage. | Chapter 9 |
| commit log | Commit Log(CLOG) keeps the states of all transactions (e.g., in\_progress,committed,aborted) for Concurrency Control (CC) mechanism. | Section 5.4 |

In addition to them, PostgreSQL allocates several areas as shown below:

* Sub-areas for the various access control mechanisms. (e.g., semaphores, lightweight locks, shared and exclusive locks, etc)
* Sub-areas for the various background processes, such as checkpointer and autovacuum.
* Sub-areas for transaction processing such as save-point and two-phase-commit.

and others.

## Database Structure

Here are some things that are important to know when attempting to understand the database structure of PostgreSQL.

Items related to the database

1. PostgreSQL consists of several databases. This is called a database cluster.
2. When initdb () is executed, template0 , template1 , and postgres databases are created.
3. The template0 and template1 databases are template databases for user database creation and contain the system catalog tables.
4. The list of tables in the template0 and template1 databases is the same immediately after initdb (). However, the template1 database can create objects that the user needs.
5. The user database is created by cloning the template1 database.

Items related to the tablespace

1. The pg\_default and pg\_global tablespaces are created immediately after initdb().
2. If you do not specify a tablespace at the time of table creation, it is stored in the pg\_dafault tablespace.
3. Tables managed at the database cluster level are stored in the pg\_global tablespace.
4. The physical location of the pg\_default tablespace is $PGDATA\base.
5. The physical location of the pg\_global tablespace is $PGDATA\global.
6. One tablespace can be used by multiple databases. At this time, a database-specific subdirectory is created in the table space directory.
7. Creating a user tablespace creates a symbolic link to the user tablespace in the $PGDATA\tblspc directory.

Items related to the table

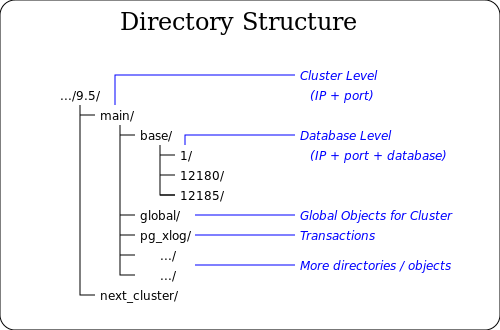
1. There are three files per table.
2. One is a file for storing table data. The file name is the OID of the table.
3. One is a file to manage table free space. The file name is OID\_fsm .
4. One is a file for managing the visibility of the table block. The file name is OID\_vm .
5. The index does not have a \_vm file. That is, OID and OID\_fsm are composed of two files.

### Other Things to Remember...

The file name at the time of table and index creation is OID, and OID and pg\_class.relfilenode are the same at this point. However, when a rewrite operation ( Truncate , CLUSTER , Vacuum Full , REINDEX , etc.) is performed, the relfilenode value of the affected object is changed, and the file name is also changed to the relfilenode value. You can easily check the file location and name by using pg\_relation\_filepath ('< object name >'). template0, template1, postgres database

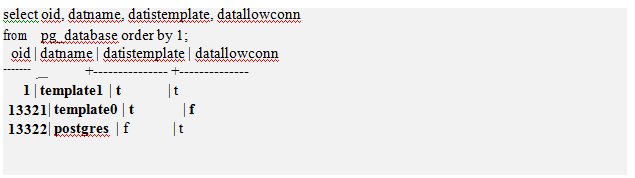
## The Directory Structure[edit]

Within a cluster there is a fix structure of subdirectories and files. At last all information is stored within these files. Some information contains to the cluster at all, and some belongs to single databases - especially tables and indexes.

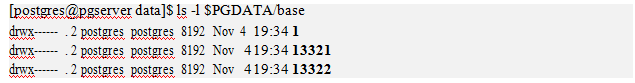
[](https://commons.wikimedia.org/wiki/File:PostgreSQL_cluster_2.svg)

## Running Tests

If you query the pg\_database view after initdb() , you can see that the template0 , template1 , and postgres databases have been created.

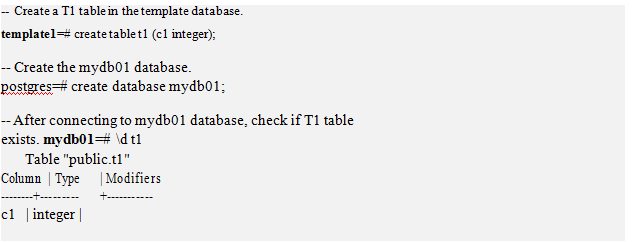


* Through the datistemplate column, you can see that the template0 and template1 databases are database for template for user database creation.
* The datlowconn column indicates whether the database can be accessed. Since the template0 database can’t be accessed, the contents of the database can’t be changed either.
* The reason for providing two databases for the templateis that the template0 database is the initial state template and the template1 database is the template added by the user.
* The postgres database is the default database created using the template1 database. If you do not specify a database at connection time, you will be connected to the postgres database.
* The database is located under the $PGDATA/base directory. The directory name is the database OID number.



## Create User Database

The user database is created by cloningthe template1 database. To verify this, create a user table T1 in the template1 database. After creating the mydb01 database, check that the T1 table exists. (See Figure 1-3.)



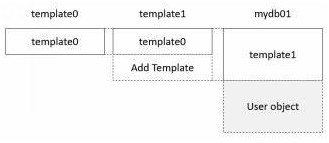
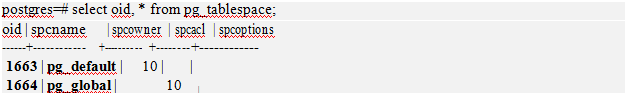


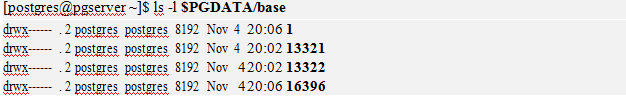
Figure 1-3. Relationship between Template Database and User Database

### pg\_default tablespace

If you query pg\_tablespace after initdb (), you can see that the pg\_default and pg\_global tablespaces have been created.



The location of the pg\_default tablespace is $PGDATA\base. There is a subdirectory by database OID in this directory. (See Figure 1-4)



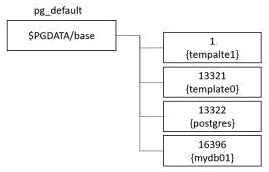


Figure 1-4. Pg\_default tablespace and database relationships from a physical configuration perspective

### pg\_global tablespace

The pg\_global tablespace is a tablespace for storing data to be managed at the 'database cluster' level.

* For example, tables of the same type as the pg\_database table provide the same information whether they are accessed from any database. (See Figure 1-5)
* The location of the pg\_global tablespace is $PGDATA\global.

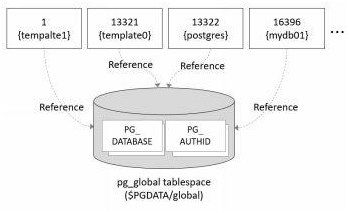
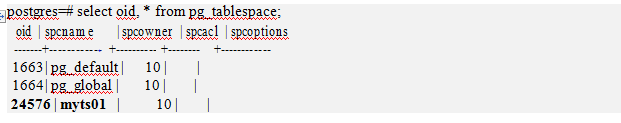


Figure 1-5. Relationship between pg\_global tablespace and database

## Create User Tablespace

|  |  |
| --- | --- |
| 1 | postgres=# create tablespace myts01 location '/data01'; |

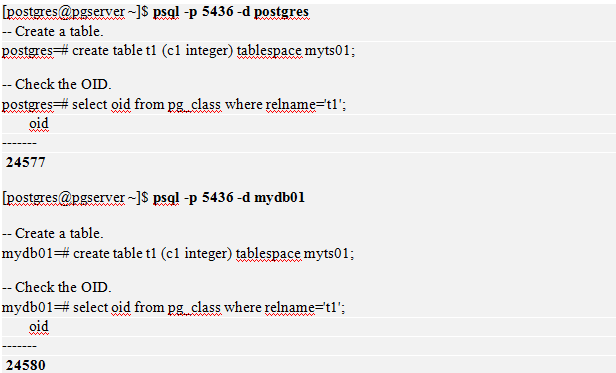
The pg\_tablespace shows that the myts01 tablespace has been created.



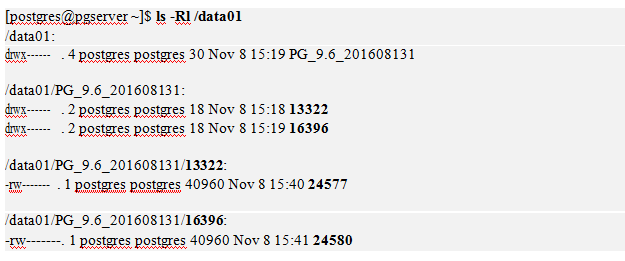
Symbolic links in the $PGDATA/pg\_tblspc directory point to tablespace directories.

https://severalnines.com/sites/default/files/blog/node_5122/image11.png

Connect to the postgres and mydb01 databases and create the table.

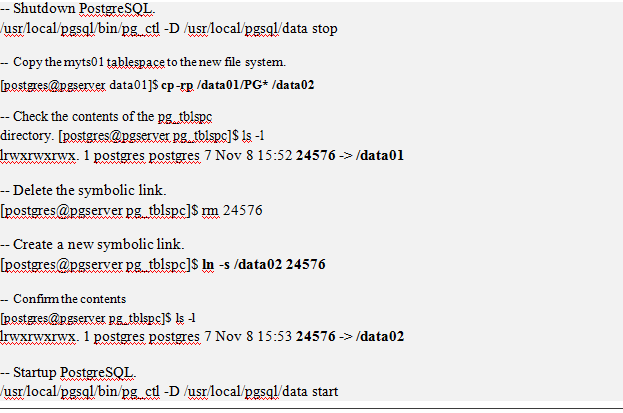


If you look up the /data01 directory after creating the table, you will see that the OID directory for the postgres and mydb01 databases has been created and that there is a file in each directory that has the same OID as the T1 table.



### How to Change Tablespace Location

PostgreSQL specifies a directory when creating tablespace. Therefore, if the file system where the directory is located is full, the data can no longer be stored. To solve this problem, you can use the volume manager. However, if you can’t use the volume manager, you can consider changing the tablespace location. The order of operation is as follows.



Note: Tablespaces are also very useful in environments that use partition tables. Because you can use different tablespaces for each partition table, you can more flexibly cope with file system capacity problems.

## What is Vacuum?

Vacuum does the following:

1. Gathering table and index statistics
2. Reorganize the table
3. Clean up tables and index dead blocks
4. Frozen by record XID to prevent XID Wraparound

#1 and #2 are generally required for DBMS management. But #3 and #4 are necessary because of the PostgreSQL MVCC feature