EPAM Systems, RD Dep., RD Dep.

POSTGRESQL DB FOR DWH AND ETL BUILDING

PostgreSQL Database Architecture

CONTENTS

1. INTRODUCTION TO POSTGRESQL

PostgreSQL is an enterprise-class relational database management system, on par with the very best proprietary database systems: Oracle, Microsoft SQL Server, and IBM DB2, just to name a few. PostgreSQL is special because it's not just a database: it's also an application platform, and an impressive one at that.

PostgreSQL invites you to write stored procedures and functions in numerous programming languages. This support for a wide variety of languages allows you to choose the language with constructs that can best solve the problem at hand.

Most database products limit you to a predefined set of data types: integers, texts, Booleans, etc. Not only does PostgreSQL come with a larger built-in set than most, but you can define additional data types to suit your needs.

Although PostgreSQL is fundamentally relational, you'll find plenty of facilities to handle nonrelational data. The ltree extension to PostgreSQL has been around since time immemorial and provides support for graphs. The hstore extensions let you store key-value pairs. JSON and JSONB types allow storage of documents like MongoDb. Many third-party extensions for PostgreSQL leverage custom types to achieve performance gains, provide domain-specific constructs for shorter and more maintainable code, and accomplish feats you can only fantasize about with other database products.

2. DATABASE CLUSTER

In PostgreSQL, if you have a working database, it means that you have a cluster. In PostgreSQL, a cluster refers to a set of databases, using the same configuration files, listening for requests at a common port. The databases belonging to the cluster use a common filesystem location. There is a common set of background processes and memory structures (such as shared buffers used by this set of databases).

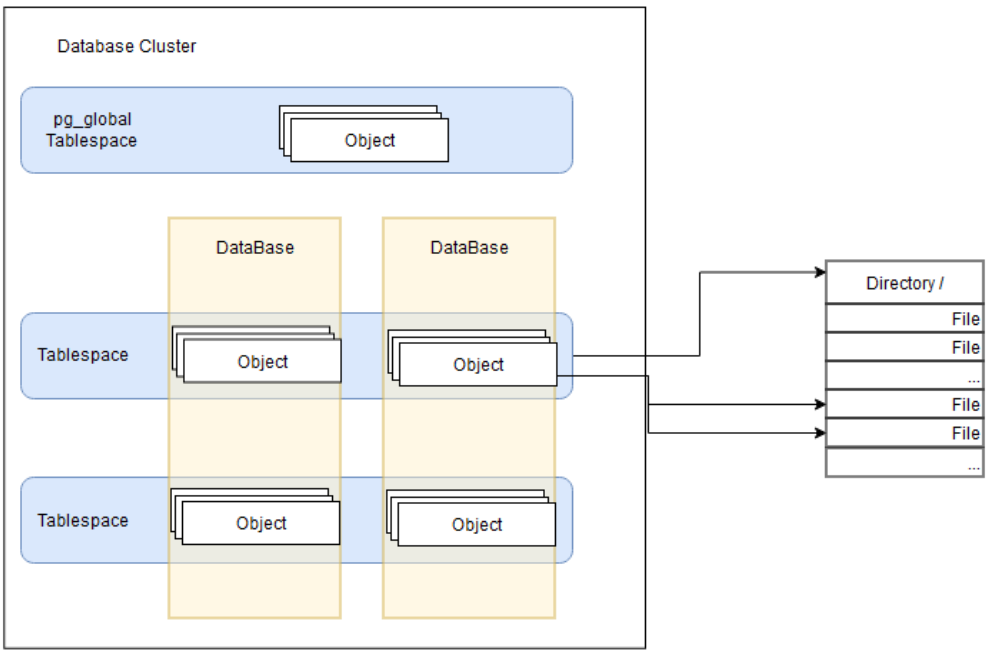
It's possible to run multiple PostgreSQL clusters on a server (although not a recommended practice in production) as long as they are listening on different ports and have separate storage areas defined.

2.1. LOGICAL STRUCTURE

Every instance of a running PostgreSQL server manages one or more databases. Databases are therefore the topmost hierarchical level for organizing SQL objects ("database objects").

All the database objects in PostgreSQL are internally managed by respective **object identifiers (OIDs)**, which are unsigned 4-byte integers. The relations between database objects and the respective OIDs are stored in appropriate system catalogs, depending on the type of objects.

The system catalogs are the place where a relational database management system stores schema metadata, such as information about tables and columns, and internal bookkeeping information. PostgreSQL's system catalogs are regular tables. You can drop and recreate the tables, add columns, insert and update values, and severely mess up your system that way. Normally, one should not change the system catalogs by hand, there are normally SQL commands to do that. (For example, CREATE DATABASE inserts a row into the pg_database catalog — and actually creates the database on disk.) There are some exceptions for particularly esoteric operations, but many of those have been made available as SQL commands over time, and so the need for direct manipulation of the system catalogs is ever decreasing.



A small number of objects, like role, database, and tablespace names, are defined at the cluster level and stored in the pg_global tablespace. Inside the cluster are multiple databases, which are isolated from each other but can access cluster-level objects. Inside each database are multiple schemas, which contain objects like tables and functions. So the full hierarchy is: cluster, database, schema, table (or some other kind of object, such as a function).

2.1.1. Databases

When connecting to the database server, a client must specify the database name in its connection request. It is not possible to access more than one database per connection. However, clients can open multiple connections to the same database, or different databases.

It's not possible to access objects in one database from another database directly. We can access other databases using a database link (or foreign data wrappers).

Since you need to be connected to the database server in order to create new database, the question remains how the first database at any given site can be created. The first database is always created by the initdb command when the data storage area is initialized. This database is called postgres. So, to create the first “ordinary” database you can connect to postgres.

A second database, template1, is also created during database cluster initialization. Whenever a new database is created within the cluster, template1 is essentially cloned. This means that any changes you make in template1 are propagated to all subsequently created databases. Because of this, avoid creating objects in template1 unless you want them propagated to every newly created database.

2.1.2. Tablespaces

In PostgreSQL, tablespace is a link to a location in the filesystem, that is, a directory. It's a container to hold all other objects, such as tables, indexes, and so on.

When we initialized a cluster, two default tablespaces got created, one was a tablespace called pg_default. All objects which are created by users without specifying a tablespace will be created in the pg_default tablespace. The location for the default tablespace: pg_default is the base directory under PGDATA. The other one: pg_global holds the system tables shared by all the databases in the cluster.

By using tablespaces, an administrator can control the disk layout of a PostgreSQL installation. This is useful in at least two ways. First, if the partition or volume on which the cluster was initialized runs out of space and cannot be extended, a tablespace can be created on a different partition and used until the system can be reconfigured.

Second, tablespaces allow an administrator to use knowledge of the usage pattern of database objects to optimize performance. For example, an index which is very heavily used can be placed on a very fast, highly available disk, such as an expensive solid state device. At the same time a table storing archived data which is rarely used or not performance critical could be stored on a less expensive, slower disk system.

Even though located outside the main PostgreSQL data directory, tablespaces are an integral part of the database cluster and cannot be treated as an autonomous collection of data files. They are dependent on metadata contained in the main data directory, and therefore cannot be attached to a different database cluster or backed up individually. Similarly, if you lose a tablespace (file deletion, disk failure, etc.), the database cluster might become unreadable or unable to start. Placing a tablespace on a temporary file system like a RAM disk risks the reliability of the entire cluster.

2.1.3. Schemas

Another important concept is the schema, which is a container or a namespace within a database. Any object that we create in a database (such as a table, an index, a view, and so on) gets created under a schema. We can use schemas to group together related objects within the same database. To some extent, these can be associated to the concept of databases in MySQL. We can access objects in different schemas from the same connection. When we create objects without specifying the schema, they get created under a default schema called public.

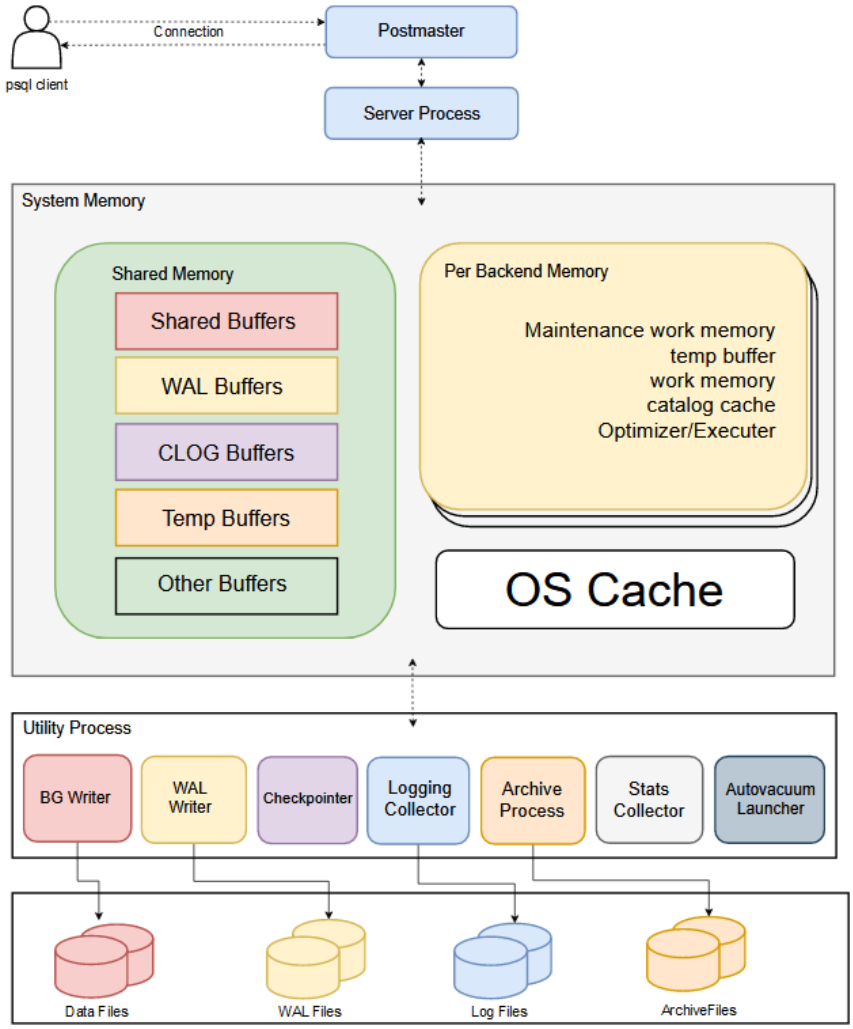
PostgreSQL uses a setting called search_path to figure out where it should search for the object the user is trying to access.

By default, users cannot access any object in schemas they do not own. To allow users to make use of the objects in other schemas, additional privileges (for example, SELECT and UPDATE on tables) must be granted, as appropriate for the object.

Unlike databases, schemas are not rigidly separated: a user can access objects in any of the schemas in the database, assuming that the user has privileges to do so.

2.2 PHYSICAL STRUCTURE

PostgreSQL uses the client-server model, where the client and server programs are usually on different hosts. The communication between the client and server is normally done via TCP/IP protocols or via Linux sockets.



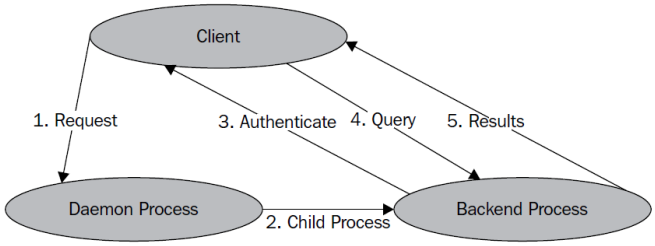
The database frontend application can perform any database action. The frontend can be a web server that wants to display a web page or a command-line tool that does maintenance tasks. PostgreSQL provides frontend tools such as psql, createdb, dropdb, and createuser and so on.

The server process manages database files, accepts connections from client applications, and performs actions on behalf of the client. The server process name is “postgres”. The main server process forks a new process for each new connection. Therefore, client and server processes communicate with each other without the intervention of the main server process. In addition, these new processes have a certain lifetime, which is determined by accepting and terminating a client connection.

2.2.1 Process Manager

The first process that is launched when we start PostgreSQL is /usr/local/pgsql/bin/postgres. This process has quite a few responsibilities such as performing recovery, initializing shared data structures/memory space, and kicking off the mandatory and optional processes. These processes are also referred to as utility processes and include bgwriter, checkpointer, autovacuum launcher, log writer, stats collector process, and so on. Furthermore, the daemon process also listens for connection requests, receives requests for connections from clients, and spawns server processes for the client. It’s obvious that the daemon itself is a mandatory process that should be running in order for a user to connect to the database.

The following diagram walks you through the process of how the daemon process receives a connection request and starts (forks) a backend process.



The backend process will, on successful authentication, start handling requests from that client. Similarly, the process is repeated for all connection requests (unless we hit the max_connections settings, in which case we get an error). As a result, an active server, after a period of time, will have the processes that were there when the server started, plus quite a few processes to serve client connections.

Once a user is connected to a database, the user typically wants to read (SELECT) data or write (UPDATE/DELETE/INSERT) data, not to mention making changes to table structure, adding indexes, and so on.

When there are thousands of users trying to read/write data to many different tables, reading from the directories/files (which we saw getting created when we installed PostgreSQL and created a database with a couple of tables) will result in a miserably non-scalable system. To make this a lot more scalable and faster, the concept of shared buffers (memory area) is introduced.

2.2.2 Shared buffers

Now, the backend processes are no longer reading from the files and writing to the files, but dealing with buffers or RAM, with significant improvement in performance.

It's not this memory chunk alone that is responsible for improving the response times, but the OS cache also helps quite a bit by keeping a lot of data ready-to-serve. Together, these two caches result in a significant reduction in the actual number and volume of physical reads and writes. In addition to these two levels of caching, there could be a disk controller cache, disk drive cache, and so on. The bottom line is that these caches improve performance by reducing the physical I/O necessary.

The first thing the process will check is whether the data it wants is available in the database buffer cache. If it is not available in the database buffer cache, a request goes to the OS to fetch the specific file/block(s). There is a chance that the OS cache already has the file/block(s) and passes it to the database cache. In both these cases, a physical I/O is avoided. It's only when the data is not present in either of these caches (or other caches), that a user initialized read/write will really result in a physical I/O.

It's evident that most of the user-driven data fetches and writes will happen via buffers. Even in a scenario where a user makes a lot of changes to table data and issues a commit, it might not immediately result in writes to the underlying data files. This might just result in ensuring that Write Ahead Log (WAL) files are synchronized with the WAL buffer.

2.2.3 WAL

WAL forms a critical component in ensuring the Durability (D) and, to some extent, the Atomicity (A) of ACID properties of transactions.

Changes are made to the blocks in the buffer and records of these changes are written to the WAL buffer (as soon as changes to data are made). The changes are flushed to the WAL segments when the changes are committed.

Recovery is the primary purpose of the WAL concept. Here, we are referring to recovering transactions that have been committed, but have not found their way to the data files. All the changes made to the database will find their way into the WAL segments, irrespective of whether the changes have been reflected into the data files or not. In fact, it's mandatory that changes have been written to WAL files before they are written to the data files themselves. Loss of WAL files almost certainly means lost transactions.

We can take a snapshot of the PostgreSQL filesystem and then set up a WAL archival process. The snapshot taken need not be a consistent one. The WAL segments generated will keep getting archived and we can use the snapshot and the archived WAL segment to perform a point-in-time recovery. In this process, we restore the file snapshot, and then replay the WAL segments until a specific point in time or until a transaction.

WAL segments are used for ***replication*** also. The rationale is simple. All the changes happening in the server are being recorded in the WAL segments anyway.

WAL also reduces the number of disk writes necessary to guarantee that a transaction is committed, thus improving performance. This is achieved because WAL writes are sequential. If a number of small transactions are committed, they will appear sequentially in the log. If the database were to work without WAL, a transaction commit will immediately result in writing data out to all the data files that were affected by the transaction.

2.2.5 Checkpoint

Checkpoint is a mandatory process. Once a user makes changes to the data (which has been made available in the buffer), that buffer is dirty. The fact that a user has committed a change does not mean that the change has been written to the data file. It's the job of the checkpointer process to write the change to the data file. The checkpointer writes all dirty (modified) pages to the table and index files. The process also marks the pages as clean. It also marks the write-ahead log as applied up to this point.

Let's consider a server with around 16 GB shared buffer. If a significant proportion of this load consists of writes, then, most of this 16 GB buffer can become dirty in a few minutes. A low setting for checkpoint_segments will result in the available segments getting filled quickly and frequent checkpoints. Similarly, a low setting for checkpoint_timeout will also result in frequent checkpoints. This parameter tells PostgreSQL how quickly it must try and finish the checkpointing process in each iteration. This results in excessive disk throughput. On the other hand, if we keep these values very high, this will result in infrequent checkpoints. In a write-heavy system, this can result in significant I/O spikes during checkpoints, which affects the performance of other queries.

2.2.6 stats collector

This process, as the name indicates, collects statistics about the database. It's an optional process with the default value as on. The process keeps track of access to tables and indexes in both disk-block and individual row-terms. It also keeps track of record counts for tables, and tracks the vacuum and analyze actions. It's important to note that individual processes transmit new statistical counts to the collector just before going idle. As a result, many of the counters will not reflect activities of in-flight transactions.

3. TRANSACTIONS

PostgreSQL provides you with highly advanced transaction machinery that offers countless features to developers and administrators alike. The first important thing to know is that, in PostgreSQL, everything is a transaction. If you send a simple query to the server, it is already a transaction. If more than one statement has to be a part of the same transaction, the BEGIN statement must be used.

The important point here is that both timestamps will be identical. As we mentioned earlier, we are talking about transaction time. To end the transaction, COMMIT can be used.

ROLLBACK is the counterpart of COMMIT. Instead of successfully ending a transaction, it will simply stop the transaction without ever making things visible to other transactions.

In professional applications, it can be pretty hard to write reasonably long transactions without ever encountering a single error. To solve this problem, users can utilize something called SAVEPOINT. As the name indicates, a savepoint is a safe place inside a transaction that the application can return to if things go terribly wrong.

In PostgreSQL, it is possible to run DDLs (commands that change the data's structure) inside a transaction block. Apart from some minor exceptions (DROP DATABASE, CREATE TABLESPACE, DROP TABLESPACE, and so on), all DDLs in PostgreSQL are transactional (do not implicitly commit the current transaction).

3.1 MULTI-VERSION CONCURRENCY CONTROL (MVCC)

MVCC is one of the main techniques Postgres uses to implement transactions. MVCC lets Postgres run many queries that touch the same rows simultaneously, while keeping those queries isolated from each other.

Postgres handles transaction isolation by using MVCC to create a concept called “snapshots”. Whenever a query starts, it takes a snapshot of the current state of the database. Only the effects of transactions that committed before the snapshot was created are visible to the query. Rows inserted by transactions that didn’t commit before the query started are invisible to the query, and rows deleted by a transaction that didn’t commit before the query started are still visible.

MVCC works by assigning every transaction a serially incremented transaction id (commonly abbreviated txid), with earlier transactions having smaller txids. Whenever a query starts, it records the next txid to be issued, and the txid of every transaction currently running. Collectively this information allows Postgres to determine if a transaction had completed before the query started. A transaction occurred before the query started if the txid of the transaction is both smaller than the next txid to be issued when the query started and is not in the list of txids of transactions that were running when the query started.

As part of MVCC, every row records both the txid of the transaction that inserted it, and if the row was deleted, the txid of the transaction that deleted it. With this information and the ability to determine whether a transaction completed before a query started, Postgres has enough information to implement snapshots. A row should be included in the snapshot of the query if that row was inserted by a transaction that completed before the query started and not deleted by a transaction that completed before the query.

Under MVCC, an insert is handled by inserting a new row with the txid of the current transaction as the insert id, deletes are handled by setting the delete txid of the row being deleted to the txid of the current transaction, and an update is just a delete followed by an insert of the new row1. Since a row isn’t physically deleted when a query deletes it, the physical disk space will need to freed sometime later. This is typically handled by a background job called the vacuum.

3.2 VACUUMING

Assume that we delete a few records from a table. PostgreSQL does not immediately remove the deleted tuples from the data files. These are marked as deleted. Similarly, when a record is updated, it's roughly equivalent to one delete and one insert. The previous version of the record continues to be in the data file. Each update of a database row generates a new version of the row. The reason is simple: there can be active transactions, which want to see the data as it was before. As a result of this activity, there will be a lot of unusable space in the data files. After some time, these dead records become irrelevant as there are no transactions still around to see the old data. However, as the space is not marked as reusable, inserts and updates (which result in inserts) happen in other pages in the data file.

VACUUM will visit all of the pages that potentially contain modifications and find all the dead space. The free space that's found is then tracked by the free space map (FSM) of the relation. Note that VACUUM will, in most cases, not shrink the size of a table. Instead, it will track and find free space inside existing storage files. Vacuum does not lock the table.

VACUUM FULL, in addition to marking the space as reusable, removes the deleted or updated records and reorders the table data. This requires an exclusive lock on the table.

Executing vacuum with the "analyze" option reads the records in the tables and generates statistics used by the query planner.

Autovacuum automates the vacuum process. It's recommended to have the autovacuum process do the cleanup of the data files unless there are specific reasons not to. In cases where the database is under heavy load for most part of the day.

Vacuum can be scheduled during off-peak hours. Although, there are few or no deletes/updates in the cluster, it's useful to have routine vacuuming as vacuum updates the data statistics used by the planner.

4. SOURCE BOOKS AND ARTICLES

1. PostgreSQL 13.4 Documentation, The PostgreSQL Global Development Group, 1996–2021
2. Regina O. Obe and Leo S. Hsu, PostgreSQL: Up and Running THIRD EDITION, O'Reilly Media, 2018
3. Jayadevan Maymala, PostgreSQL for Data Architects, Packt Publishing, 2015
4. Hans-Jürgen Schönig, Mastering PostgreSQL 12 Third Edition, Packt Publishing, 2019