Database configuration

Enterprise applications

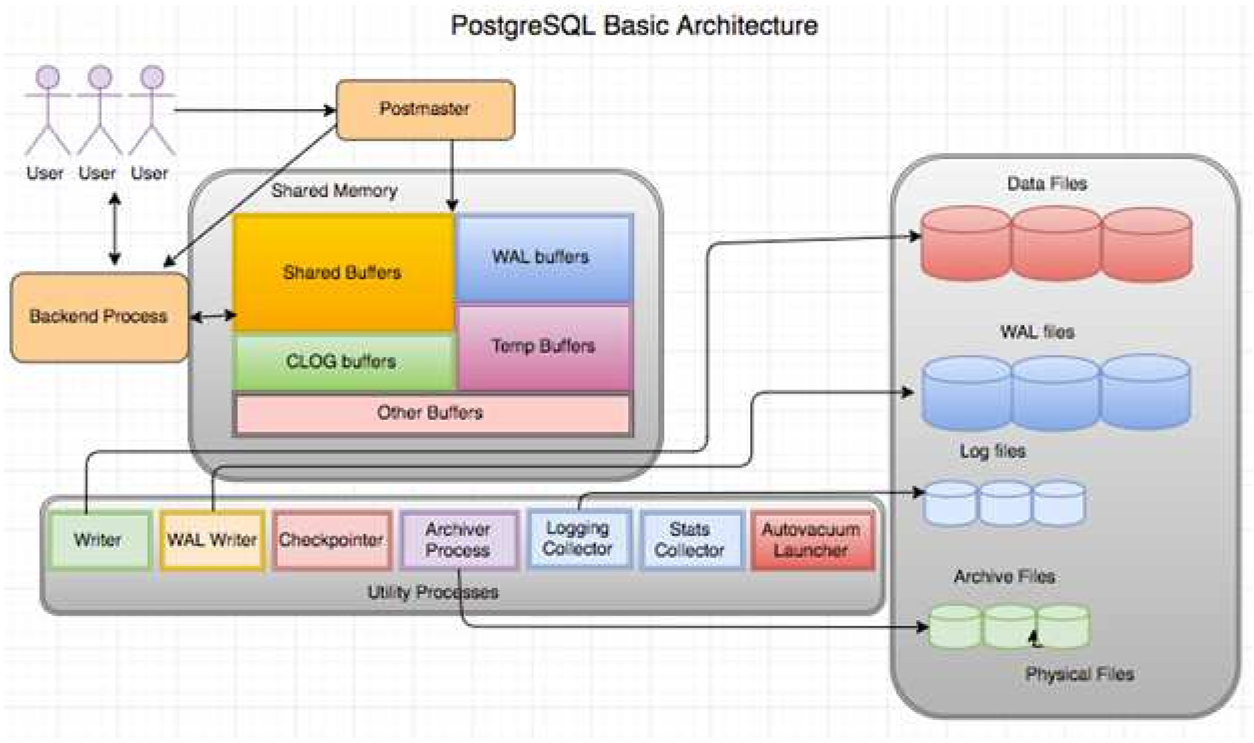
NetApp
February 10, 2026

Table of Contents

Database configuration

Architecture

PostgreSQL is an RDBMS based on client and server architecture. A PostgreSQL instance is known as a database cluster, which is a collection of databases as opposed to a collection of servers.



There are three major elements in a PostgreSQL database: the postmaster, the front end (client), and the back end. The client sends requests to the postmaster with information such as IP protocol and which database to connect to. The postmaster authenticates the connection and passes it to the back-end process for further communication. The back-end process executes the query and sends results directly to the front end (client).

A PostgreSQL instance is based on a multiprocess model instead of a multithreaded model. It spawns multiple processes for different jobs, and each process has its own functionality. The major processes include the client process, the WAL writer process, the background writer process, and the checkpointer process:

- When a client (foreground) process sends read or write requests to the PostgreSQL instance, it doesn't read or write data directly to the disk. It first buffers the data in shared buffers and write-ahead logging (WAL) buffers.
- A WAL writer process manipulates the content of the shared buffers and WAL buffers to write into the WAL logs. WAL logs are typically transaction logs of PostgreSQL and are sequentially written. Therefore, to improve the response time from the database, PostgreSQL first writes into the transaction logs and acknowledges the client.
- To put the database in a consistent state, the background writer process checks the shared buffer periodically for dirty pages. It then flushes the data onto the data files that are stored on NetApp volumes or LUNs.
- The checkpointer process also runs periodically (less frequently than the background process) and prevents any modification to the buffers. It signals to the WAL writer process to write and flush the

checkpoint record to the end of WAL logs that are stored on the NetApp disk. It also signals the background writer process to write and flush all dirty pages to the disk.

Initialization parameters

You create a new database cluster by using the `initdb` program. An `initdb` script creates the data files, system tables, and template databases (template0 and template1) that define the cluster.

The template database represents a stock database. It contains definitions for system tables, standard views, functions, and data types. `pgdata` acts as an argument to the `initdb` script that specifies the location of the database cluster.

All the database objects in PostgreSQL are internally managed by respective OIDs. Tables and indexes are also managed by individual OIDs. The relationships between database objects and their respective OIDs are stored in appropriate system catalog tables, depending on the type of object. For example, OIDs of databases and heap tables are stored in `pg_database` and `pg_class, respectively. You can determine the OIDs by issuing queries on the PostgreSQL client.

Every database has its own individual tables and index files that are restricted to 1GB. Each table has two associated files, suffixed respectively with `_fsm` and `_vm`. They are referred to as the free space map and the visibility map. These files store the information about free space capacity and have visibility on each page in the table file. Indexes only have individual free space maps and don't have visibility maps.

The `pg_xlog/pg_wal` directory contains the write-ahead logs. Write-ahead logs are used to improve database reliability and performance. Whenever you update a row in a table, PostgreSQL first writes the change to the write-ahead log, and later writes the modifications to the actual data pages to a disk. The `pg_xlog` directory usually contains several files, but initdb creates only the first one. Extra files are added as needed. Each xlog file is 16MB long.

Settings

There are several PostgreSQL tuning configurations that can improve performance.

The most commonly used parameters are as follows:

- `max_connections = <num>`: The maximum number of database connections to have at one time. Use this parameter to restrict swapping to disk and killing the performance. Depending on your application requirement, you can also tune this parameter for the connection pool settings.
- `shared_buffers = <num>`: The simplest method for improving the performance of your database server. The default is low for most modern hardware. It is set during deployment to approximately 25% of available RAM on the system. This parameter setting varies depending on how it works with particular database instances; you might have to increase and decrease the values by trial and error. However, setting it high is likely to degrade performance.
- `effective_cache_size = <num>`: This value tells PostgreSQL's optimizer how much memory PostgreSQL has available for caching data and helps in determining whether to use an index. A larger value increases the likelihood of using an index. This parameter should be set to the amount of memory allocated to `shared_buffers` plus the amount of OS cache available. Often this value is more than 50% of the total system memory.
- `work_mem = <num>`: This parameter controls the amount of memory to be used in sort operations and hash tables. If you do heavy sorting in your application, you might need to increase the amount of memory,

but be cautious. It isn't a system wide parameter, but a per-operation one. If a complex query has several sort operations in it, it uses multiple work_mem units of memory, and multiple back ends could be doing this simultaneously. This query can often lead your database server to swap if the value is too large. This option was previously called sort_mem in older versions of PostgreSQL.

- `fsync = <boolean> (on or off)`: This parameter determines whether all your WAL pages should be synchronized to disk by using fsync() before a transaction is committed. Turning it off can sometimes improve write performance and turning it on increases protection from the risk of corruption when the system crashes.
- `checkpoint_timeout`: The checkpoint process flushes committed data to disk. This involves a lot of read/write operations on disk. The value is set in seconds and lower values decrease crash recovery time and increasing values can reduce the load on system resources by reducing the checkpoint calls. Depending on application criticality, usage, database availability, set the value of checkpoint_timeout.
- `commit_delay = <num>` and `commit_siblings = <num>`: These options are used together to help improve performance by writing out multiple transactions that are committing at once. If there are several commit_siblings objects active at the instant your transaction is committing, the server waits for commit_delay microseconds to try to commit multiple transactions at once.
- `max_worker_processes / max_parallel_workers`: Configure the optimal number of workers for processes. Max_parallel_workers correspond to the number of CPUs available. Depending on application design, queries might require a lesser number of workers for parallel operations. It is better to keep the value for both parameters the same but adjust the value after testing.
- `random_page_cost = <num>`: This value controls the way PostgreSQL views non-sequential disk reads. A higher value means PostgreSQL is more likely to use a sequential scan instead of an index scan, indicating that your server has fast disks Modify this setting after evaluating other options like plan-based optimization, vacuuming, indexing to altering queries or schema.
- `effective_io_concurrency = <num>`: This parameter sets the number of concurrent disk I/O operations that PostgreSQL attempts to execute simultaneously. Raising this value increases the number of I/O operations that any individual PostgreSQL session attempts to initiate in parallel. The allowed range is 1 to 1,000, or zero to disable issuance of asynchronous I/O requests. Currently, this setting only affects bitmap heap scans. Solid-state drives (SSDs) and other memory-based storage (NVMe) can often process many concurrent requests, so the best value can be in the hundreds.

See the PostgreSQL documentation for a complete list of PostgreSQL configuration parameters.

TOAST

TOAST stands for The Oversized-Attribute Storage Technique. PostgreSQL uses a fixed page size (commonly 8KB) and does not allow tuples to span multiple pages. Therefore, it is not possible to store large field values directly. When you attempt to store a row that exceeds this size, TOAST breaks up the data of large columns into smaller "pieces" and stores them in a TOAST table.

The large values of toasted attributes are pulled out (if selected at all) only at the time the result set is sent to the client. The table itself is much smaller and can fit more rows into the shared buffer cache than it could without any out-of-line storage (TOAST).

VACUUM

In normal PostgreSQL operation, tuples that are deleted or made obsolete by an update are not physically removed from their table; they remain present until VACUUM is run. Therefore, you must run VACUUM periodically, especially on frequently updated tables. The space it occupies must then be reclaimed for reuse by new rows, to avoid disk space outage. However, it does not return the space to the operating system.

The free space inside a page is not fragmented. VACUUM rewrites the entire block, efficiently packing the remaining rows and leaving a single contiguous block of free space in a page.

In contrast, VACUUM FULL actively compacts tables by writing a completely new version of the table file with no dead space. This action minimizes the size of the table but can take a long time. It also requires extra disk space for the new copy of the table until the operation completes. The goal of routine VACUUM is to avoid VACUUM FULL activity. This process not only keeps tables at their minimum size, but also maintains steady-state usage of disk space.

Tablespaces

Two tablespaces are automatically created when the database cluster is initialized.

The `pg_global` tablespace is used for shared system catalogs. The `pg_default` tablespace is the default tablespace of the template1 and template0 databases. 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.

An index that is heavily used can be placed on a fast, highly available disk, like a solid-state device. Also, a table storing archived data that is rarely used or not performance critical can be stored on a less expensive, slower disk system like SAS or SATA drives.

Tablespaces are a part of the database cluster and cannot be treated as an autonomous collection of data files. They depend 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 (through file deletion, disk failure, and so on), 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.

After it is created, a tablespace can be used from any database if the requesting user has sufficient privileges. PostgreSQL uses symbolic links to simplify the implementation of tablespaces. PostgreSQL adds a row to the `pg_tablespace` table (a clusterwide table) and assigns a new object identifier (OID) to that row. Finally, the server uses the OID to create a symbolic link between your cluster and the given directory. The directory `$PGDATA/pg_tblspc` contains symbolic links that point to each of the non-built-in tablespaces defined in the cluster.