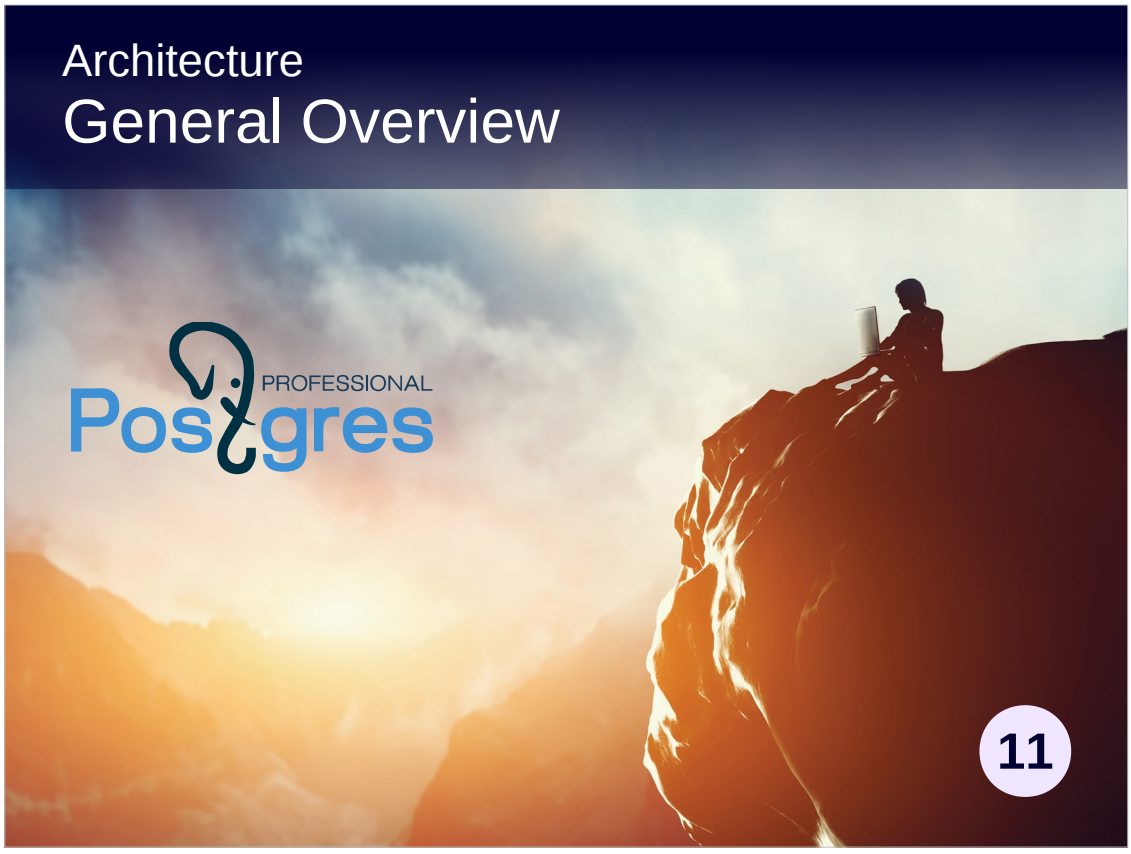


Authors: Egor Rogov, Pavel Luzanov

Feedback
Send feedback, comments and suggestions to:
edu@postgrespro.ru

Topics

Client/Server protocol

Transactions and implementation mechanisms

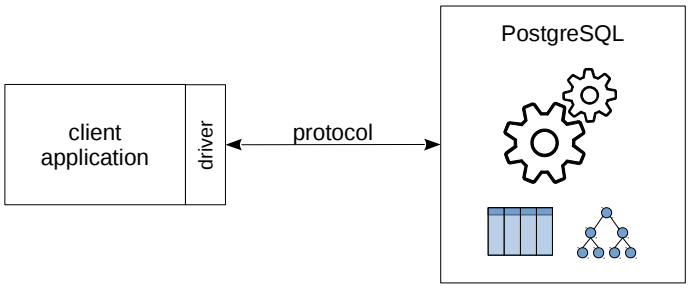
Parsing and executing queries

Processes and memory structures

On-disk data storage

System extensibility

Client and server



connecting
sending query
transaction management

authenticating
executing queries
transaction support

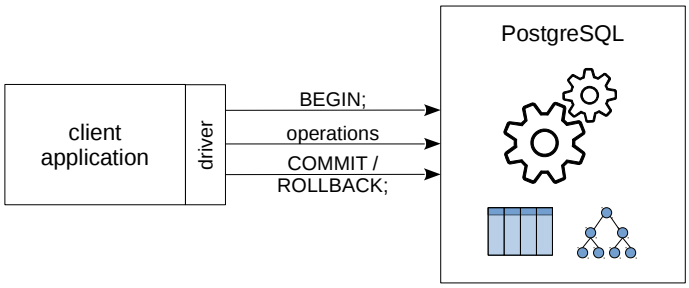
Let's start with a simple picture in which the server is represented by a «black box». A client application — for example, `psql`, or any other program written in any programming language (PL) — connects to the server and somehow «communicates» with it. In order for the client and server to understand each other, they must use the same communication protocol. The protocol is rather low-level and, of course, it is not necessary to implement it in each client. Typically, the client program uses a driver that provides a set of functions for use in the program. The driver can use existing PostgreSQL implementation of the protocol (`libpq` library), or it can implement the protocol itself.

Therefore, in fact, it is not so important in which PL the client is written — for different syntax there will be the same possibilities defined by the protocol.

Speaking in general terms, the protocol allows the client to connect; at the same time the server performs the so-called authentication — for example, it requests a password and decides whether the connection can be allowed. Next, the client sends requests to the server in SQL, and the server executes them and returns the result.

The presence of a sophisticated query language is one of the features that distinguish a DBMS from just working with data files directly. Another key feature is support for transactions to ensure consistency.

Transactions



atomicity — *all or nothing*
consistency — *data correctness (integrity and other constraints)*
isolation — *no influence of concurrent transactions*
durability — *committed data are permanent even in case of failure*

A transaction is a logically indivisible part of the work, which preserves the consistency of data in the database.

Four properties (ACID) are expected from transactions:

- Atomicity: the transaction is either succeeds completely or not at all. For this purpose the beginning of the transaction is marked with the BEGIN command, and the end is marked with either COMMIT (preserve changes) or ROLLBACK (abort transaction).

- Consistency: the transaction begins in a consistent state and, completing, also maintains consistency.

- Isolation: other concurrent transactions (running at the same time with this one) should not affect it.

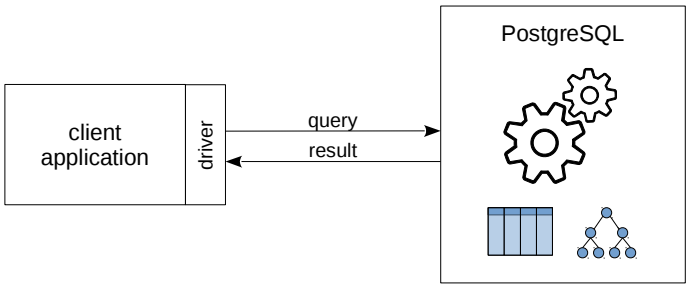
- Durability: after the transaction is committed, changes should not be lost even in case of failure.

https://postgrespro.com/docs/postgresql/11/tutorial-transactions

It is the client application which is responsible for managing transactions (start and end transaction commands) in PostgreSQL. However PostgreSQL 11 introduces so-called stored *procedures* (not to be confused with *functions*), which also can execute transaction control statements.

https://postgrespro.com/docs/postgresql/11/xproc

Query execution



parsing	← *system catalog*
transformation	← *rules*
planning	← *statistics*
execution	← *data*

Query execution is a challenge.

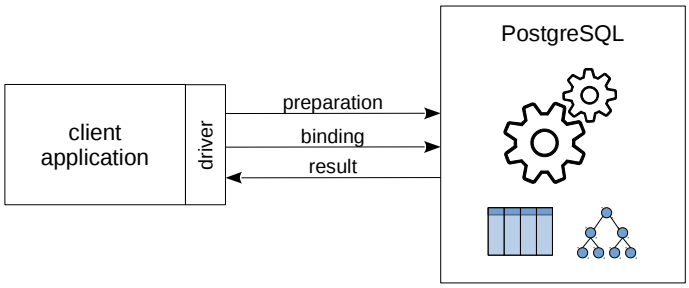
The request is transmitted from the client to the server as a text string. The text must be **parsed** — that is, a syntactic analysis (whether the letters are folded into words, and words into commands) and semantic analysis (match names with existing tables or other objects in the database, and also check access rights) must be performed. This requires information about what is contained in the database. Such meta-information is called the system catalog and is stored in the database itself in special tables.

The request can be **transformed** — for example, the text of the request is substituted for the name of the view. You can come up with your own transformations, for which there is a rules mechanism.

SQL is a declarative language: a query on it says *what* data you need, but does not say *how* to get it. Therefore, the request (already parsed and represented in the internal form of a tree) is transmitted to the **planner** (optimizer), which develops the execution plan. For example, the planner decides whether to use indexes or not. In order to come up with a decent plan, the planner needs information about the size of the tables, about the distribution of data — in other words, statistics.

Further, the request is **executed** in accordance with the plan and the result is returned to the client.

Prepared statements



parsing
transformation

---

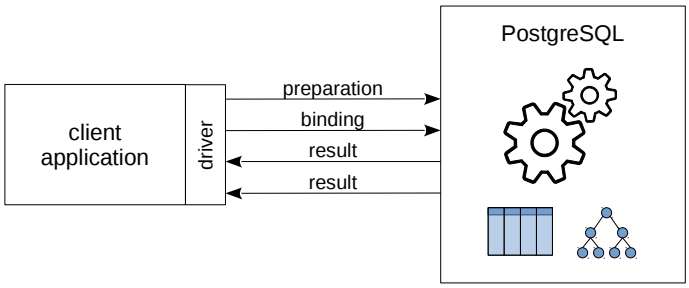
binding ← *parameters values*
planning
execution

Each request passes the steps listed above. But if the same query (up to parameters) is executed many times, there is no point in repeating the process every time.

PostgreSQL allows to **prepare** a SQL statement — perform parsing and transformation in advance and remember the parse tree.

When a query is executed, specific parameter values are bound. If necessary, planning is performed (in some cases, PostgreSQL remembers the query plan and does not re-plan the query). Then the query is executed.

Cursors



parsing
transformation

---

binding
planning
execution

← *parameters values*

---

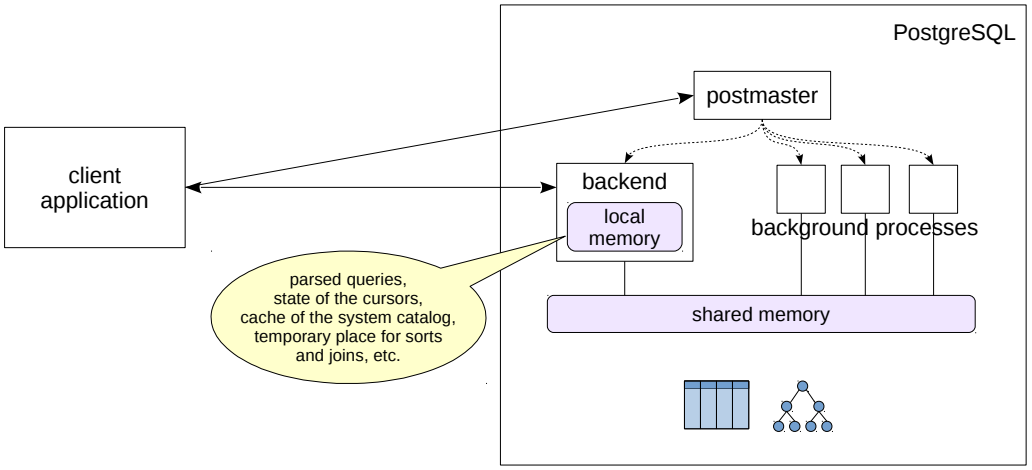
fetching results

It is not always convenient for the client to get all the results at once. There may be a lot of data, but not all of them may be needed.

For that purpose there is a **cursor** mechanism: by opening a cursor for a request, the client can fetch data from it line by line as needed.

It is clear that the server has to store auxiliary information: parsed requests and their plans, the state of open cursors. Where and how does it do it? To answer this question we need to look inside the server and get familiar with its structure.

Processes and memory



From the inside, the PostgreSQL server can be represented as several interacting processes.

First of all, during the server startup the *postmaster* process starts first. It starts all other processes (using the Unix `fork` system call) and «looks after» them: if any process terminates abnormally, postmaster restarts it (or restarts the entire server if it considers that the process could damage the shared data).

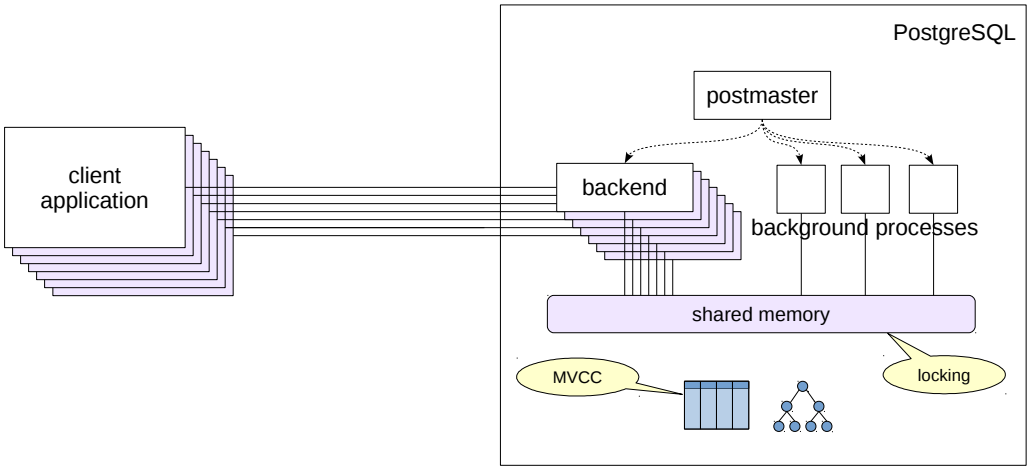
The server also run a number of background processes. Later we will discuss the main ones.

To allow processes to exchange information, postmaster allocates a *shared memory*, access to which can be obtained by all processes. In addition to shared memory, each process has its own *local memory*, accessible only to itself.

So that the client can connect to the server, postmaster listens to incoming connections. When a client appears, the postmaster spawns a *backend process* for it, so that each client communicates with its dedicated server process.

The space required to execute the query (parsed queries and their plans, the state of the cursors, the cache of the system catalog, the place for sorts and joins, etc.) is allocated in the local memory of the backend process.

Many clients



When many clients connect to the server, for each of them a separate backend process is forked. This is not a problem up to a certain number (say, several hundreds) of processes if the server has enough memory and connections do not occur too often.

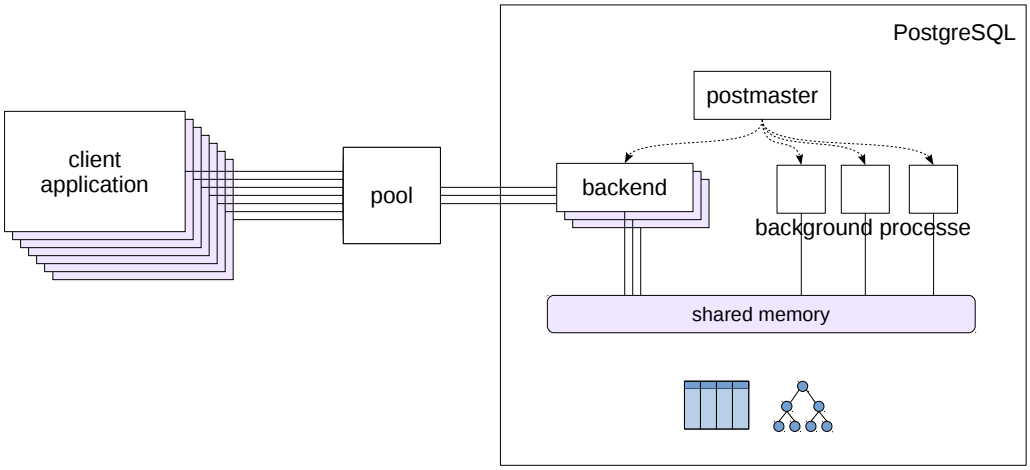
However, concurrent access to objects requires special handling so that one process does not change any data while another process is working with it.

For objects in shared memory, short-lived locks are used. PostgreSQL does this quite carefully so that the system scales well with an increase in the number of CPU cores.

With tables it is more difficult, since locks will have to be held until the end of transactions (that is, potentially for a long time). That is why scalability may suffer. Therefore PostgreSQL uses the multiversion concurrency control (MVCC). The idea is that the same row can simultaneously exist in different versions. Each process sees its own (but always consistent) snapshot of the data. This allows PostgreSQL to block only those processes that are trying to change data that has already been changed (but not yet committed) by other processes.

MVCC is the main mechanism that provides the first three properties of transactions (atomicity, consistency, isolation). We will talk about it in more detail later.

Connection pool

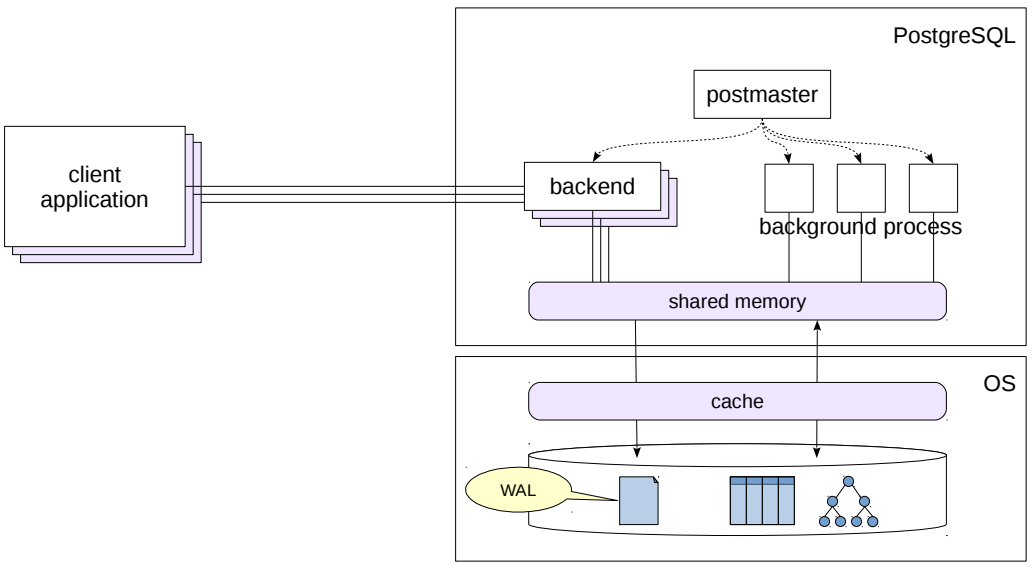


If there are too many clients, or connections are established and dropped too often, you should consider using a *connection pool*. This function is usually provided by an application server or you can use third-party pool managers (the most famous is PgBouncer https://pgbouncer.github.io/).

Clients connect not to the database server directly, but instead to the pool manager. The manager keeps several connections to the database server open and uses one of the free ones to fulfill client requests. Thus, from the point of view of the database server, the number of clients remains constant regardless of how many clients access the pool manager.

But with this mode of operation, several clients share the same backend process, which (as was mentioned) stores a certain state in its local memory (in particular, parsed requests for prepared statements). This must be considered when developing an application.

Data storage



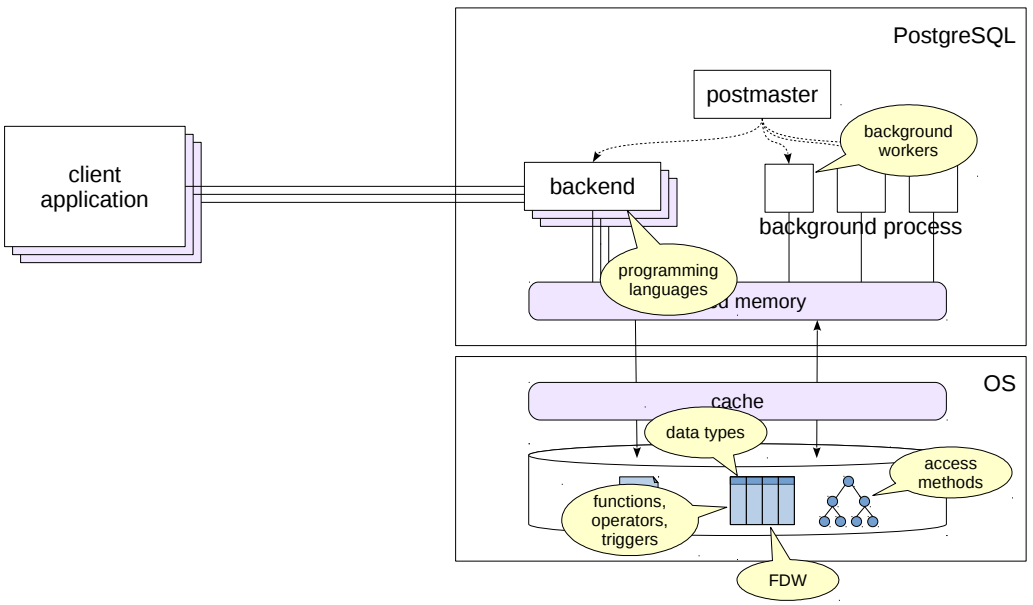
PostgreSQL works with disks on which data is located through the operating system. It almost does not use direct I/O. Data is stored in regular files and read or written using the appropriate system calls.

Due to the fact that the disks are much slower than RAM (especially HDD, but also SSD), caching is used: space for recently read pages is allocated in RAM in the hope that there will be several hits and you can save on re-access to the disk. Changed data is also written to disk not immediately, but after a while in background.

Important point: the data is cached both in the operating system level and in the PostgreSQL level. The PostgreSQL data cache (called *buffer cache*) is located in shared memory so that all processes can access it.

In the case of a failure (for example, power outage), the contents of the RAM disappear and some data may be lost — which is unacceptable as it violates the durability property. Therefore, during its operation PostgreSQL constantly writes the so-called Write-Ahead Log (WAL) to the disk. This allows to re-perform lost operations and restore data in a consistent state. We will discuss the buffer cache and the WAL later on.

Extensibility



PostgreSQL is designed for extensibility.

An application programmer has the ability to create his own data types based on existing ones (composite types, ranges, arrays, enumerations), write stored functions for data processing (including triggers that fire upon the occurrence of certain events).

If you are familiar with the C programming language, you can write an extension that adds the necessary functionality and usually can be installed on the fly without restarting the server. Thanks to this architecture, a large number of extensions is available that:

- add support for programming languages (in addition to standard SQL, PL/pgSQL, PL/Perl, PL/Python and PL/Tcl);

- introduce new data types and operators to work with them;

- create new types of indices (access methods) for efficient work with various data types (in addition to standard B-trees, there is also a hash index, GiST, SP-GiST, GIN, BRIN, Bloom);

- connect to external systems with the help of foreign data wrappers (FDW);

- run background processes to perform periodic tasks.

Summary

The protocol allows clients to connect to the server, perform queries and manage transactions

Each client is served by its own backend process

Data is stored in files; access occurs through the operating system calls

Caching both in local memory (system catalog, parsed queries) and in shared memory (buffer cache)