Authors: Egor Rogov, Pavel Luzanov
Translated by Liudmila Mantrova

Contact Us
Please send your feedback to: edu@postgrespro.ru

Agenda

Client-server protocol

Transactions and their implementation mechanisms

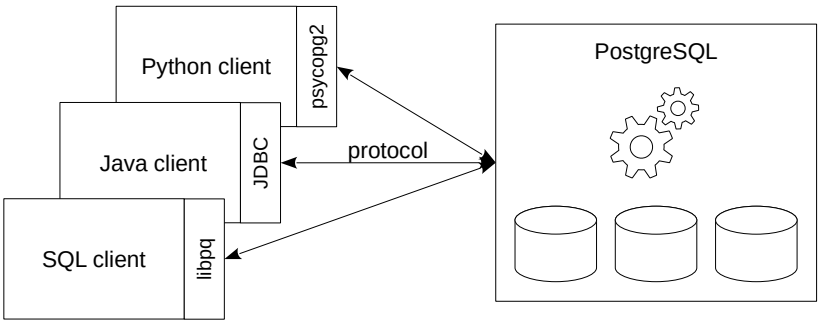
Query processing workflow and execution paths

Processes and memory structures

Storing and accessing data on disk

Extensibility

Client and Server



connection
generating queries
managing transactions

authentication
executing queries
supporting transactions

A client application, such as psql or any other application written in any programming language, connects to the server and communicates with it somehow. To understand each other, both the client and the server must use the same interfacing *protocol*. The client usually uses a *driver* that implements the protocol and provides a set of features to be used in the application. The driver can use the standard libpq library or provide a custom implementation of the protocol.

The programming language of the application is not that important—different syntactic structures implement the same features defined by the protocol. In our examples, we are going to use the SQL language and the psql client. It's clear that no one writes the frontend in SQL, but it is convenient for training purposes. We expect that it will not be hard for you to compare SQL commands with the capabilities of your favorite language.

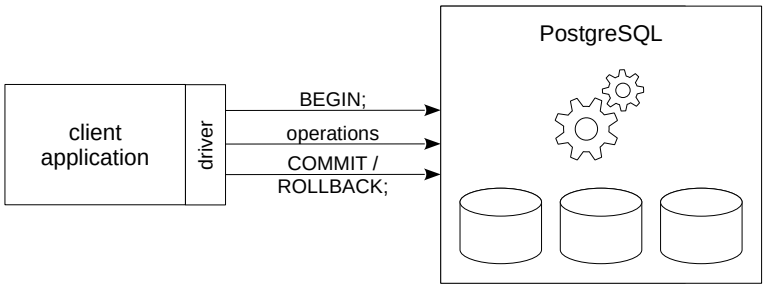
Speaking very roughly, the protocol enables the client to connect to one of the databases of a database cluster. At this point, the server performs *authentication*: it decides whether to allow this connection, for example, by asking for password.

Then the client sends SQL queries to the server, and the server executes these queries and returns the results. Having a powerful and user-friendly query language is one of distinctive features of relational database systems.

Another one is ensuring data consistency.

https://postgrespro.com/docs/postgresql/12/protocol

Transactions



atomicity — *all or nothing*
consistency — *integrity and user-defined constraints*
isolation — *handling concurrent processes*
durability — *data retention even after failures*

A transaction is a logically indivisible part of work that ensures data consistency in a database.

Transactions are expected to satisfy four criteria (ACID):

- Atomicity: a transaction is either executed completely, or is not executed at all. To achieve this, the beginning of a transaction is labeled with the `BEGIN` command, and the end is marked by either `COMMIT` (to keep the changes) or `ROLLBACK` (to cancel the changes).

- Consistency: each transaction begins in a consistent state and leaves the data consistent when it is complete.

- Isolation: concurrent transactions must not affect each other.

- Durability: once the data is committed, it must not be lost even in case of a failure.

It is usually the client application that manages transactions in PostgreSQL (i.e., defines the commands that constitute a transaction and commits or rolls back the transaction). On the server side, transactions can be managed by stored procedures (we will learn about them in the “SQL” and “PL/pgSQL” modules).

https://postgrespro.com/docs/postgresql/12/sql-begin

https://postgrespro.com/docs/postgresql/12/sql-savepoint

Managing Transactions

The default psql mode is autocommit:

```
=> \echo :AUTOCOMMIT
```

```
on
```

It means that if you enter a command without specifying the transaction start explicitly, it will be committed right away.

- Is a similar mode enabled in the PostgreSQL driver for your favorite programming language?

Let's create a table with a single row:

```
=> CREATE TABLE t(
    id integer,
    s text
);
```

```
CREATE TABLE
```

```
=> INSERT INTO t(id, s) VALUES (1, 'foo');
```

```
INSERT 0 1
```

Will another transaction see this table and the row?

```
=> SELECT * FROM t;
```

```
 id |  s
----+-----
  1 | foo
(1 row)
```

Yes. Let's compare it with another scenario:

```
=> BEGIN; -- explicitly begins a transaction
```

```
BEGIN
```

```
=> INSERT INTO t(id, s) VALUES (2, 'bar');
```

```
INSERT 0 1
```

What will another transaction see this time?

```
=> SELECT * FROM t;
```

```
 id |  s
----+-----
  1 | foo
(1 row)
```

The changes are not yet committed, so they are not visible to another transaction.

```
=> COMMIT;
```

```
COMMIT
```

And now?

```
=> SELECT * FROM t;
```

```
 id |  s
----+-----
  1 | foo
  2 | bar
(2 rows)
```

If autocommit is turned off, transactions are started implicitly when the first command is entered, but all changes must be committed explicitly.

```
=> \set AUTOCOMMIT off
```

```
=> INSERT INTO t(id, s) VALUES (3, 'baz');
```

```
INSERT 0 1
```

And what about now?

```
=> SELECT * FROM t;
```

```
 id |  s
----+-----
  1 | foo
  2 | bar
(2 rows)
```

The changes are not visible yet; the transaction has been started implicitly.

```
=> COMMIT;
```

```
COMMIT
```

And finally:

```
=> SELECT * FROM t;
```

```
 id |  s
----+-----
  1 | foo
  2 | bar
  3 | baz
(3 rows)
```

Let's restore the default psql mode.

```
=> \set AUTOCOMMIT on
```

It is possible to roll back the changes without aborting the transaction (even though it is required quite rarely).

```
=> BEGIN;
```

```
BEGIN
```

```
=> SAVEPOINT sp; -- a savepoint
```

```
SAVEPOINT
```

```
=> INSERT INTO t(id, s) VALUES (4, 'qux');
```

```
INSERT 0 1
```

```
=> SELECT * FROM t;
```

```
 id |  s
----+-----
  1 | foo
  2 | bar
  3 | baz
  4 | qux
(4 rows)
```

Note: the transaction sees its own changes even if they are not committed yet.

Now let's roll back all changes to the savepoint.

Rollback to savepoint does not imply control transfer (i.e., it does not work like GOTO); it simply cancels all the database changes made between the savepoint and the current moment.

```
=> ROLLBACK TO sp;
```

```
ROLLBACK
```

What will we see?

```
=> SELECT * FROM t;
```

```
 id |  s
----+-----
  1 | foo
  2 | bar
  3 | baz
(3 rows)
```

Now the changes are rolled back, but the transaction is still running:

```
=> INSERT INTO t(id, s) VALUES (4, 'xyz');
```

```
INSERT 0 1
```

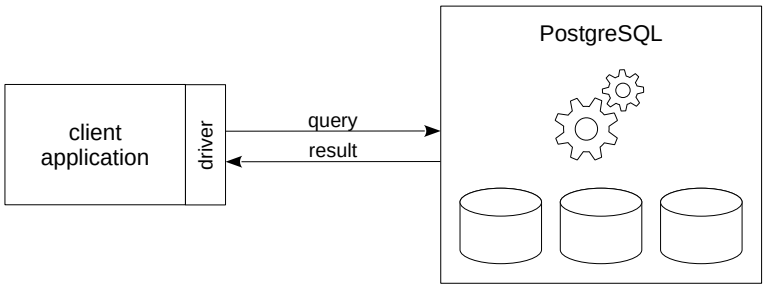
```
=> COMMIT;
```

```
COMMIT

=> SELECT * FROM t;

 id |  s
----+-----
  1 | foo
  2 | bar
  3 | baz
  4 | xyz
(4 rows)
```

Query Execution



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

Query execution is quite a complicated task. A query is transferred from a client to the server as plain text. This text must be *parsed*, i.e., it must undergo syntactic analysis (to check whether letters constitute words, and words constitute commands) and semantic analysis (to check whether the database contains tables and other objects that the query refers to by name). To achieve this, it is required to know what is located in the database. Such *metadata* is called a *system catalog*; it is stored in special tables in the database itself.

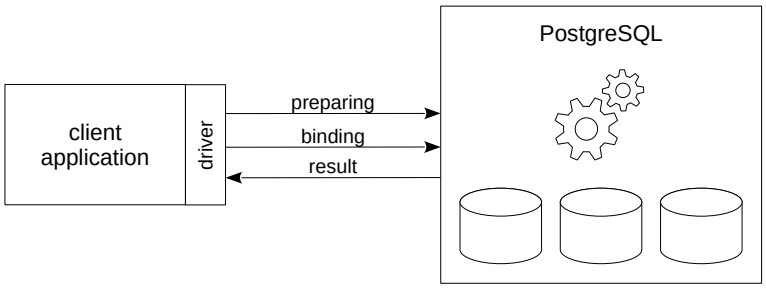
A query can get transformed: for example, the name of the view is replaced with the query text. You can configure your own transformations using the provided *rule* system.

SQL is a declarative language: an SQL query specifies which data we need to get, but says nothing about how to get it. That's why the query (already parsed and transformed into a tree) is passed to the *planner* that builds a *query plan*. For example, the planner decides whether it is required to use indexes. To produce a good plan, the planner needs to know the size of the tables and data distribution, i.e., *statistics*.

Then the query is executed according to the plan, and the whole result is returned to the client at once.

It is a simple and convenient way to perform small data selections, but it can be less suitable for large data volumes.

Prepared Statements



parsing
transformation

---

binding
planning
execution

← *parameter values*

Each query has to go through the stages mentioned above: parsing, transformation, planning, and execution. But if one and the same query (down to the specified parameters) is executed multiple times, there is no reason to parse it again and again.

That's why apart from simple query execution, the PostgreSQL protocol also provides an *extended mode* that allows managing query execution more precisely.

Among other features, the extended mode enables you to *prepare* a query—that is, to parse and transform the query in advance and keep the parse tree.

Binding of actual parameter values takes place during query execution. If required, planning is performed (in some cases, PostgreSQL keeps the query plan and skips this step). Then the query is executed.

Another advantage of using prepared statements is protection against SQL injection (we'll discuss it in detail in the "PL/pgSQL. Dynamic Commands" topic).

https://postgrespro.com/docs/postgresql/12/sql-prepare

https://postgrespro.com/docs/postgresql/12/sql-execute

Prepared Statements

An SQL statement can be prepared using the PREPARE command (it's a PostgreSQL extension, this command is not defined in the SQL standard):

```
=> PREPARE q(integer) AS
    SELECT * FROM t WHERE id = $1;
```

```
PREPARE
```

The statement is parsed and transformed, and the resulting query tree is kept in memory.

Once the statement is prepared, you can call it by name, passing the actual parameters:

```
=> EXECUTE q(1);
```

```
 id |  s
----+-----
  1 | foo
(1 row)
```

If the statement has no parameters, the query plan built during preparation is also kept in memory. If there are some parameters, like in this example, their actual values are taken into account at the planning stage. But the planner can consider a plan that ignores parameters to be just as good; then it will stop repeating the planning stage.

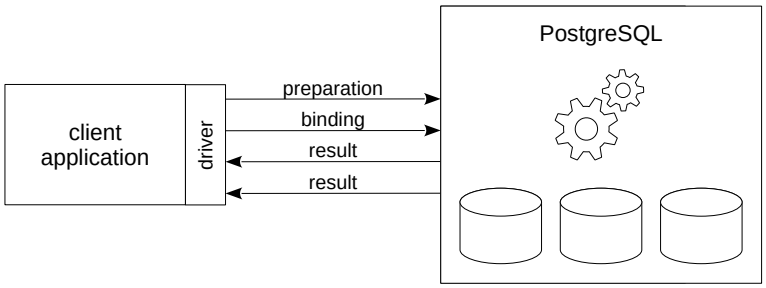
- How can you prepare and execute a statement in your favorite programming language?
- Is it possible to execute a statement without preparing it first?

All prepared statements are displayed in the following view:

```
=> SELECT * FROM pg_prepared_statements \gx
```

```
-[ RECORD 1 ]---+----------------------------------
name            | q
statement       | PREPARE q(integer) AS             +
                |     SELECT * FROM t WHERE id = $1;
prepare_time    | 2021-10-19 17:00:28.044289+03
parameter_types | {integer}
from_sql        | t
```

Cursors



parsing
transformation

---

binding
planning
execution

← *parameter values*

---

fetching the result

It is not always convenient to receive the whole query result in a single batch. There can be a lot of data, but the client may need only a small part of it.

To address such cases, the extended mode provides *cursors*. The protocol allows you to open a cursor for some operator and then fetch query results row by row, as required.

We can compare a cursor to a window that shows only some data that satisfies the query. Once a row is retrieved, the window moves on. In other words, cursors allow iterating through relational data (sets), row by row.

An open cursor is represented on the server by a so-called *portal*. This word appears in documentation; in simplistic terms, “cursor” and “portal” can be called synonyms.

The query used in a cursor is implicitly prepared (that is, the server keeps the query’s parse tree and sometimes also the query plan).

https://postgrespro.com/docs/postgresql/12/sql-declare

https://postgrespro.com/docs/postgresql/12/sql-fetch

Cursors

A regular SELECT command returns all rows at once:

```
=> SELECT * FROM t ORDER BY id;
```

```
 id |  s
----+-----
  1 | foo
  2 | bar
  3 | baz
  4 | xyz
(4 rows)
```

A cursor allows fetching data row by row.

```
=> BEGIN;
```

```
BEGIN
```

```
=> DECLARE c CURSOR FOR
    SELECT * FROM t ORDER BY id;
```

```
DECLARE CURSOR
```

```
=> FETCH c;
```

```
 id |  s
----+-----
  1 | foo
(1 row)
```

You can specify the number of rows to fetch at a time:

```
=> FETCH 2 c;
```

```
 id |  s
----+-----
  2 | bar
  3 | baz
(2 rows)
```

Fetch size plays an important role if there are a lot of rows: processing a large volume of data row by row is very inefficient.

What if we reach the end of the table while reading data?

```
=> FETCH 2 c;
```

```
 id |  s
----+-----
  4 | xyz
(1 row)
```

```
=> FETCH 2 c;
```

```
 id | s
----+---
(0 rows)
```

FETCH will simply stop returning rows. In common programming languages it is always possible to check this condition.

- How can you retrieve data row by row using a cursor in your programming language?
- Is it possible NOT to use a cursor and get all the rows at once?
- How can you configure the cursor's fetch size?

Once data retrieval is complete, you can close the cursor, releasing the resources:

```
=> CLOSE c;
```

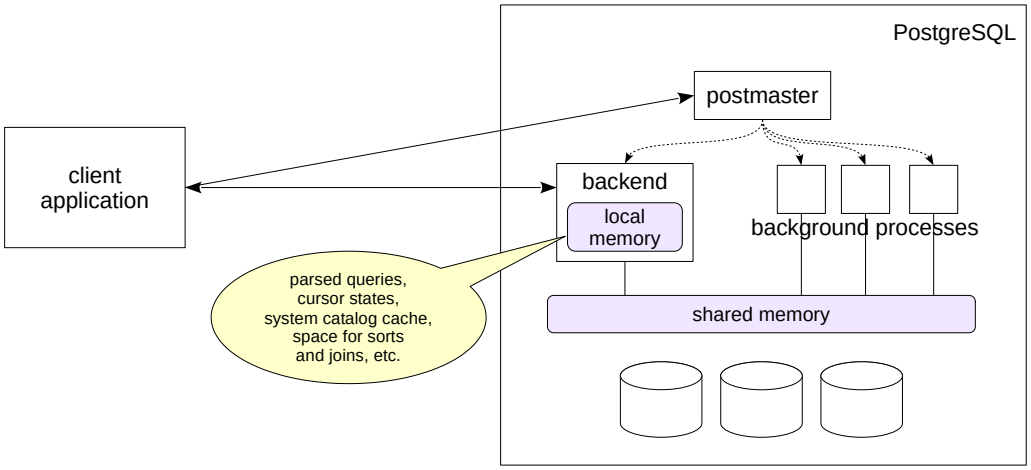
```
CLOSE CURSOR
```

Note that cursors are usually closed automatically at the end of transactions, so there is no need to do it explicitly (except for the cursors open with the WITH HOLD clause).

```
=> COMMIT;
```

COMMIT

Processes and Memory



While the client is connected, the server must keep all the related information, such as parsed queries and their plans or open cursor states (portals). Where and how is it done?

The PostgreSQL server consists of several interacting processes.

The first process launched at the server start is traditionally called postmaster. It spawns all other processes (using the fork system call on Unix) and “supervises” them: if any process fails, postmaster restarts this process (or the whole server if there is a chance that the shared data has been damaged).

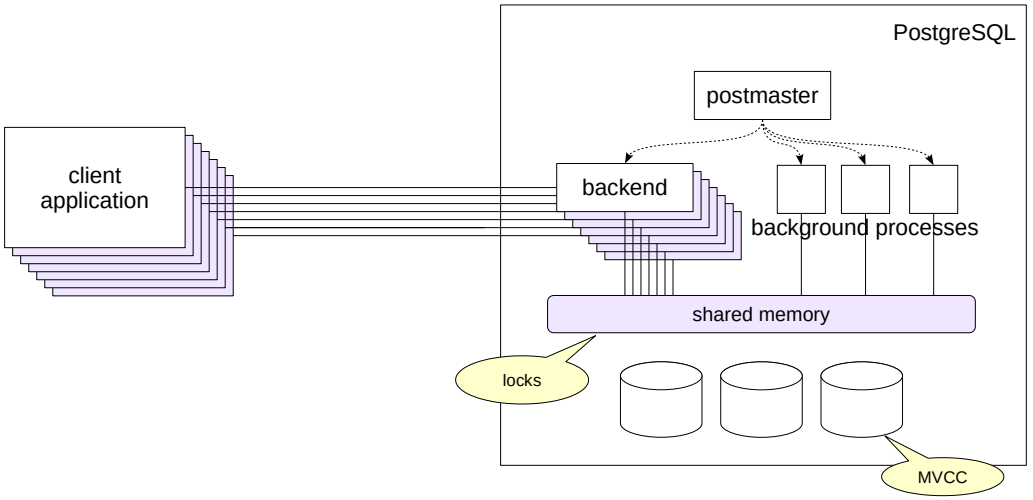
Server operation is maintained by a number of background processes. We are going to discuss the main ones in the next topics of this module.

To enable information exchange between processes, postmaster allocates *shared memory*, which is accessible to all processes. Apart from the shared memory, each process has its own *local memory*, available exclusively to this process.

Postmaster listens for incoming connections. When a client appears, postmaster forks a separate backend for it, and from this point on the client communicates with its own backend.

The space required for query execution (to store parsed queries and their plans, cursor states, system catalog cache, space for data sorting, etc.) is allocated in *local memory* of each backend.

Multiple Clients



When multiple clients connect to the server, a separate backend is spawned for each client. It is not a problem while there are not too many clients, there is enough RAM for everyone, and connections are established not too often.

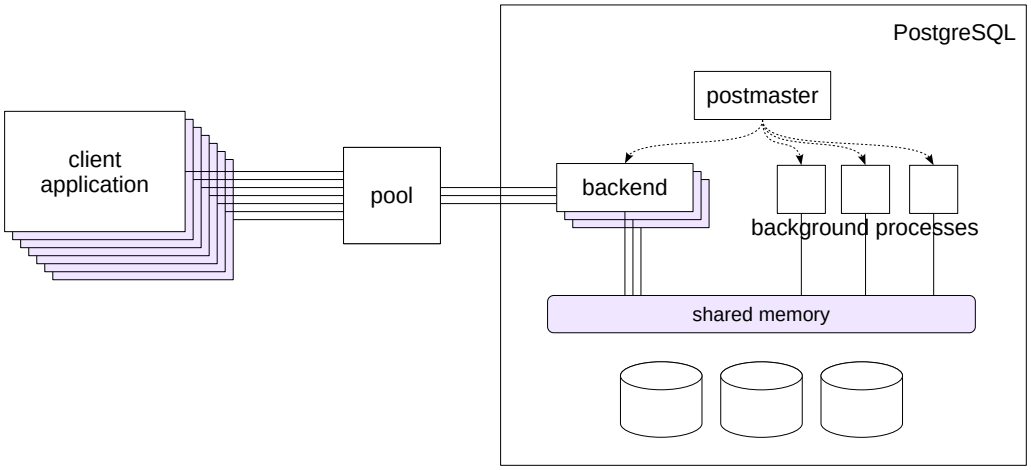
Nevertheless, when some objects are processed concurrently, certain precautions have to be taken not to change the data that is already in use.

For objects in shared memory, short-lived locks are used. PostgreSQL does it quite smart, so that the system could scale well if the number of cores is increased.

Handling tables is a bit harder as locks should be held till the end of transactions (that is, potentially for quite a long time), which could negatively affect scalability. That's why PostgreSQL uses *multi-version concurrency control (MVCC)* and *snapshot isolation*: different versions of the same data can exist simultaneously, and each process sees its own (but always consistent) data snapshot. It allows the server to lock only those processes that attempt to change the data that is already modified, but not yet committed by another process.

It is MVCC that guarantees the first three properties of transactions (atomicity, consistency, isolation). We'll discuss it separately in the "Isolation and MVCC" topic.

Connection Pooling



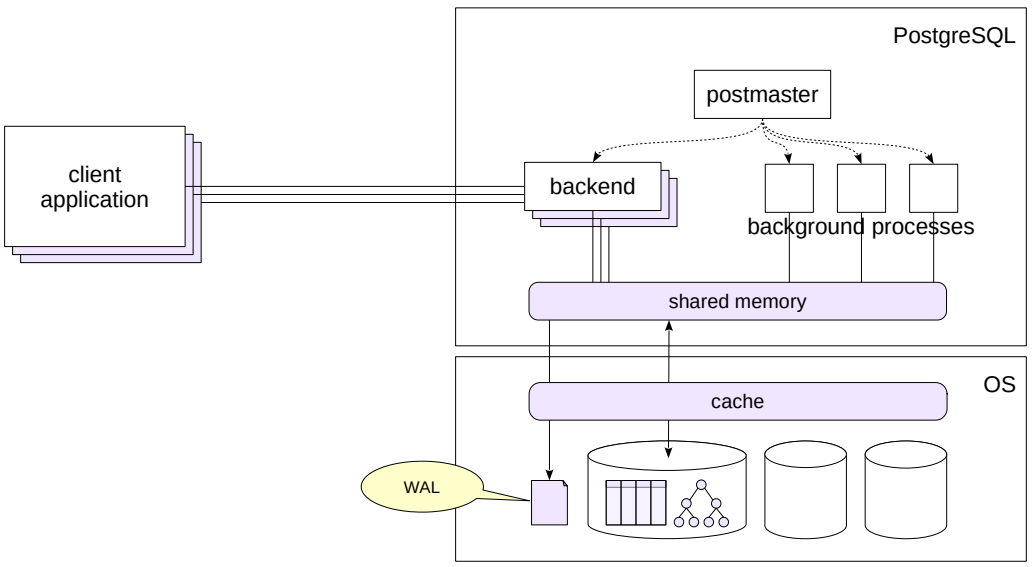
If there are too many clients, or connections are established and terminated too often, it makes sense to employ connection pooling. This functionality is usually provided by the application server; alternatively, you can use third-party pool managers (the most common one is PgBouncer).

Instead of connecting to the PostgreSQL server, clients get connected to the pool manager. The manager holds several open connections with the database server and uses one of them to execute queries of a particular client. Thus, the number of clients remains constant from the server's point of view, regardless of how many clients actually access the pool manager.

In this mode, several clients share a single backend, which, as we have already mentioned, stores a particular state in its local memory (for example, parsed queries for prepared statements). It has to be taken into account in application development.

Connection pooling is discussed in more detail in the DEV2 course.

Data Storage



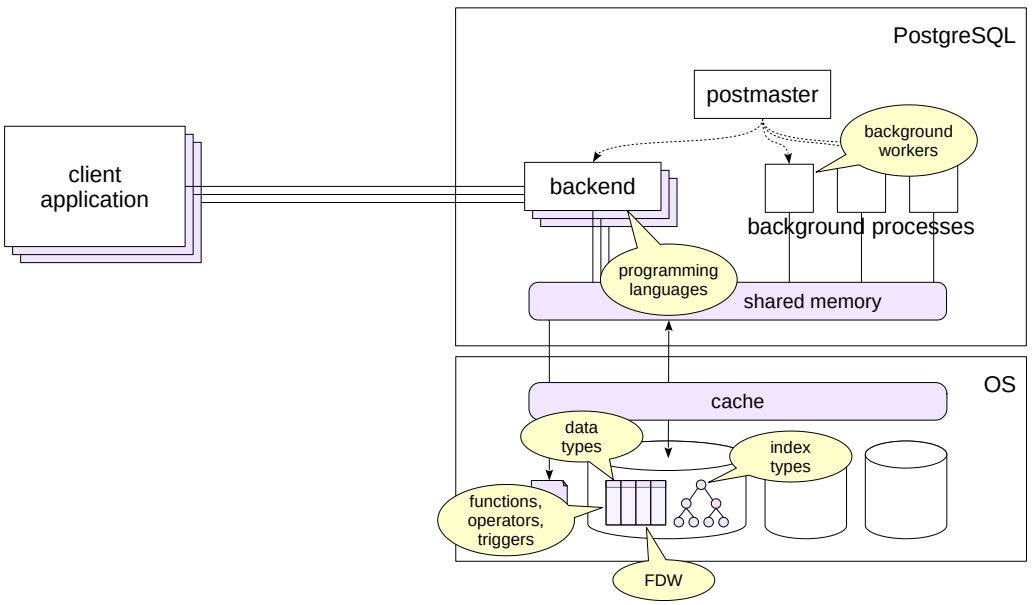
PostgreSQL has no direct access to disks that store data: it uses operating system mechanisms to reach them. Data is stored in regular files; it is read and written by the corresponding system calls.

Since disks operate much slower than RAM (especially HDDs, but it is still true for SSDs as well), *caching* is used: some RAM is allocated for recently used pages in hope that they will be accessed more than once, so we could save some time on disk access. For the same reason, the server waits for a while to flush data changes; they are not written to disk immediately.

It's important to note that both the operating system and PostgreSQL have their own cache. PostgreSQL data cache (buffer cache) is located in shared memory, for all processes to have access to it.

In case of a failure (for example, power loss) the contents of RAM is cleared, and some data can get lost, which is unacceptable (as required by the durability property). That's why PostgreSQL writes all the executed commands into the write-ahead log (WAL); it ensures that any lost operations can be repeated to restore data consistency. We'll talk about buffer cache and WAL separately in the corresponding topic.

Extensibility



PostgreSQL was designed to be extensible.

An application developer can create custom data types based on the already available ones (composite types, ranges, sets, enumerations), write stored procedures for data processing (including triggers that get activated by a particular event).

If you know C, you can develop an extension to implement the features that you need. Most extensions can be installed “on the fly,” without a server restart. Thanks to such architecture, there is a large number of extensions that

- provide support for programming languages (in addition to SQL, PL/pgSQL, PL/Perl, PL/Python, and PL/Tcl, which are available out of the box);

- implement new data types and the corresponding operators;

- create new index types for efficient work with various data types (in addition to the built-in ones: B-tree, GiST, SP-GiST, GIN, BRIN, Bloom);

- plug in external systems using foreign data wrappers (FDW);

- start background processes to handle recurrent tasks.

Extensibility is discussed at length in the DEV2 course.

Summary

A database cluster is managed by the server

The protocol enables clients to connect to the server, execute queries, and manage transactions

Each client is served by a separate process

Data is stored in files, which are accessed via OS calls

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