

# Architecture PostgreSQL overview



## Copyright

© Postgres Professional, 2015–2022

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov

Translated by Alexander Meleshko

## Use of course materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

## Feedback

Please send your feedback, comments and suggestions to:

[edu@postgrespro.ru](mailto:edu@postgrespro.ru)

## Disclaimer

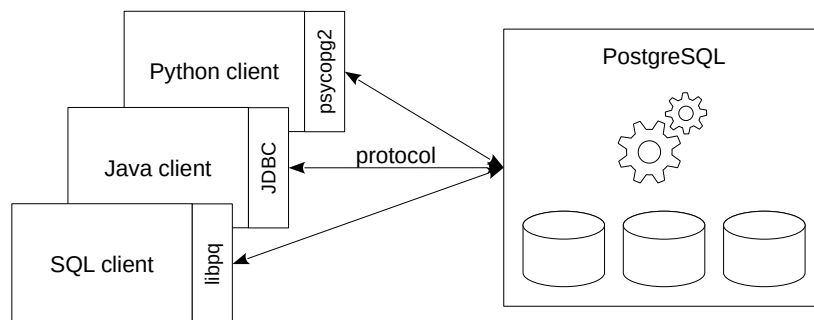
Postgres Professional assumes no responsibility for any damages and losses, including loss of income, caused by direct or indirect, intentional or accidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.



Client-server protocol  
Transactionality and its implementation  
Query processing and execution  
Processes and memory structures  
Storing data on disk and disk operations  
System extensibility



# Client and server



connection  
query generation  
transaction management

authentication  
query execution  
transactionality control

A client application, such as psql or any other program written in any programming language, connects to the server and “communicates” with it in one way or another. In order for the client and the server to understand each other, they must use the same *communication protocol*. Usually, the client uses a *driver* that implements the protocol and provides a set of functions to use in the program. Internally, the driver can use the standard protocol implementation (the libpq library), or can implement the protocol itself.

The language the client is written in is unimportant, as the functionality behind the syntax is defined by the protocol. As an example, we will use the SQL language and the psql client. Of course, no one really would program a client in SQL, but we will use it here purely for educational purposes. It should not be that difficult to substitute any of the SQL commands provided below with corresponding statements in your programming language of choice.

Generally speaking, a connection protocol allows the client to connect to a database in a cluster. The server performs *authentication*: decides if the client should be allowed to connect, i.e. by demanding a password.

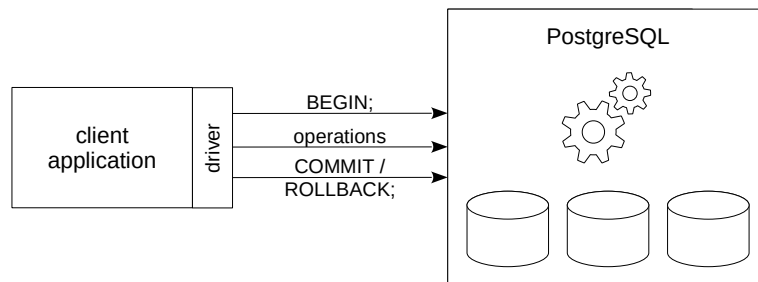
Then, the client sends the server queries in the SQL language, the server executes the queries and sends back the results. A powerful and convenient query language is one of the fundamentals of relational databases.

Another one is the ability to maintain consistency between transactions.

<https://postgrespro.com/docs/postgresql/13/protocol>



# Transactions



atomicity	— <i>everything or nothing</i>
consistency	— <i>integrity constraints and user constraints</i>
isolation	— <i>parallel processes impact</i>
durability	— <i>no data loss after a failure</i>

A *transaction* is a sequence of operations that preserves the consistency of data, provided that the operations are performed completely and without interference from other transactions.

Transactions must satisfy four properties collectively known as ACID:

- **Atomicity.** A transaction is either completed in full or not completed at all. To that end, the beginning of a transaction is marked with the BEGIN command, and the end with either COMMIT (commit changes) or ROLLBACK (undo changes).
- **Consistency.** Transactions move the database from one consistent state to another consistent one (consistency here means that certain restrictions are fulfilled).
- **Isolation.** Transactions running simultaneously should not affect each other.
- **Durability.** Once data is committed, it should not be lost even after a server failure.

In PostgreSQL, the client application is the side usually responsible for transaction management (that is, for determining what commands make up a transaction, and for committing or canceling the transaction).

Transactions can also be managed on the backend by stored procedures.

<https://postgrespro.com/docs/postgresql/13/sql-begin>

<https://postgrespro.com/docs/postgresql/13/sql-savepoint>



## Transaction management

By default, psql runs in autocommit mode:

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

on

It means that any command is committed immediately unless the transaction is explicitly opened and not closed.

- Check if the PostgreSQL driver for your favorite programming language has this option on by default.

Create a table with one 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 the table and the row?

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

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

Yes, it will. Compare the result:

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

BEGIN

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

INSERT 0 1

What will the other transaction see now?

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

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

The changes are not yet committed so the other transaction does not see them.

```
=> COMMIT;
```

COMMIT

What about now?

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

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

When autocommit is off, a transaction is implicitly opened upon command input. The command then must be committed manually.

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

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

INSERT 0 1

What do we see now?



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

id	s
1	foo
2	bar

(2 rows)

The changes are not there, as the transaction was opened and never closed.

```
=> COMMIT;
```

COMMIT

Now, finally:

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

id	s
1	foo
2	bar
3	baz

(3 rows)

Turn the default autocommit mode back on.

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

You can roll changes back without interrupting a transaction (however, it is not often necessary).

```
=> BEGIN;
```

BEGIN

```
=> SAVEPOINT sp;
```

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 how the transaction sees its own changes, even the uncommitted ones.

Now, roll back to the save point.

Rollback is done without transferring control over the transaction (unlike GOTO). Only the changes done from the moment the save point was made and until the rollback command are rolled back.

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

ROLLBACK

Check the table:

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

id	s
1	foo
2	bar
3	baz

(3 rows)

The changes have been rolled back, but the transaction is still open:

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

INSERT 0 1

```
=> COMMIT;
```



COMMIT

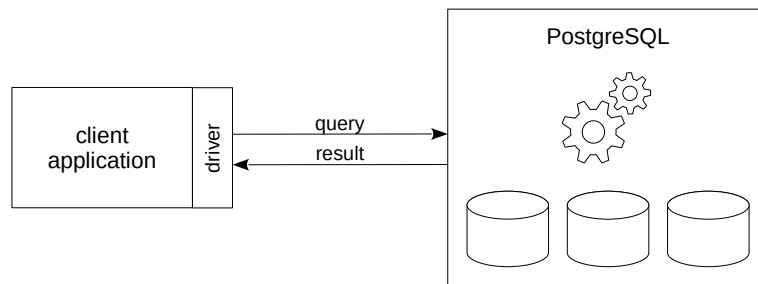
=> **SELECT** \* **FROM** t;

id	s
1	foo
2	bar
3	baz
4	xyz

(4 rows)



# Query execution



parsing	← system catalog
rewriting	← rules
planning	← statistics
execution	← data

Query execution is complicated. First, a query is sent from a client to the server as text. The server *parses* the text, analyzing its syntax (whether letters are formed into words, and words into commands) and semantics (whether there are tables and other objects in the database that the query refers to by name). To do that, the server needs data on what is actually stored in the database. This *meta-data* is called the *system catalog* and is stored in special tables in the same database.

A query can be *rewritten* (transformed). For example, a view name can be substituted with the query text. Users can implement their own transformations using the *rule system*.

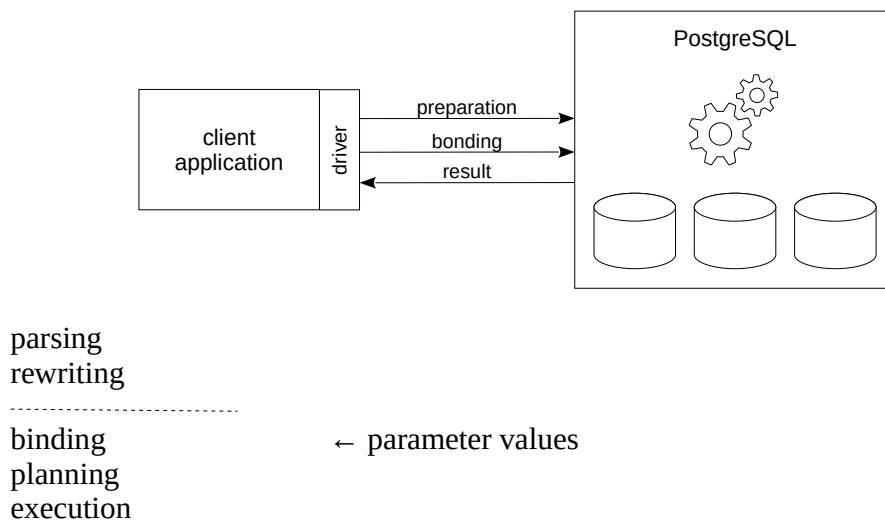
SQL is a declarative language: a query defines what data to get, but not how to get it. It is at this point when the query (already parsed and presented in the form of a tree) is passed on to the *planner*, which develops an *execution plan*. For example, the planner can decide whether or not to use indexes to find the data for the query. To plan the execution efficiently, the planner needs certain information about the tables it is going to work with, such as the size of the tables and the distribution of data within them. Together, this information is called *statistics*.

When a plan is selected, the query is executed in accordance with it, and the result is returned to the client in its entirety.

This is convenient and simple when we're talking about just a row or two, but it can quickly become problematic with large outputs.



# Prepared statements



7

Each query goes through the steps listed above: parsing, rewriting, planning and execution. But if the same query (possibly with different parameters) is executed over and over again, there is no point in parsing it anew every time.

Therefore, in addition to the usual query execution process, the PostgreSQL protocol provides an *extended mode* that can control statement execution more precisely.

One of its features is the ability to *prepare* a statement. When a statement is prepared, it is parsed and rewritten as usual and its parse tree is saved.

When the statement is executed, specific parameter values are bound to it. If necessary, planning is redone (in some cases, PostgreSQL remembers the query plan and does not repeat this step). Then, the statement is executed.

Another advantage of prepared statements is that they are protected from possible SQL injections.

<https://postgrespro.com/docs/postgresql/13/sql-prepare>

<https://postgrespro.com/docs/postgresql/13/sql-execute>



## Prepared statements

In SQL, you can prepare a statement by using the PREPARE command (it is a PostgreSQL extension not present in the SQL standard):

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

PREPARE

The statement is parsed and rewritten, and the parse tree is saved.

A prepared statement can be called by its name using arbitrary parameters:

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

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

For non-parametric statements, an execution plan is saved as well. If the statement does accept parameters, like in our case, the planner takes their actual values into account. If the planner decides that the generic plan it has built without considering any parameters is no worse than the parameterised one, it will stop trying to make new plans for the statement altogether.

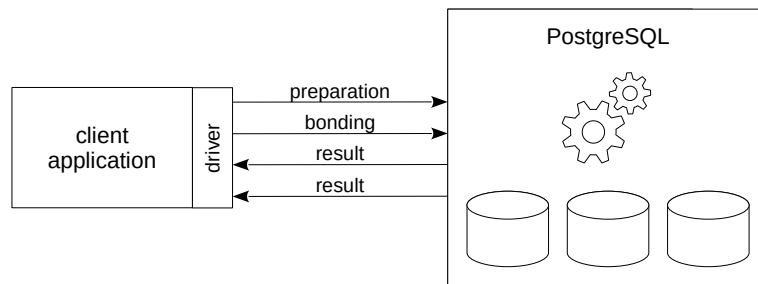
- How do you prepare a statement in your favorite programming language?
- Can you execute a statement WITHOUT preparing it?

All prepared statements can be found 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  | 2024-03-07 13:48:44.424036+03
parameter_types | {integer}
from_sql      | t
```





parsing  
rewriting

binding  
planning  
execution

output

← parameter values

The client may not want to get all the output at once. There can be too much data, and not all of it may be needed.

This issue is solved by *cursors*, another feature of the extended mode. The protocol can open a cursor for any operator, and then receive the output row by row.

A cursor can be imagined as a sliding window that shows only a part of the output at a time. When an output row is received, the window shifts down. In other words, cursors allow you to work with relational data (that comes in sets) iteratively, row by row.

An open cursor is represented on the server by a so-called *portal*. This term is mentioned in the documentation, but in general, the words “cursor” and “portal” can be considered synonyms.

A statement used within a cursor is implicitly prepared (that is, its parsing tree and possibly execution plan are saved).

<https://postgrespro.com/docs/postgresql/13/sql-declare>

<https://postgrespro.com/docs/postgresql/13/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)
```

Cursors are used to output data in batches.

```
=> BEGIN;
```

```
BEGIN
```

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

```
DECLARE CURSOR
```

```
=> FETCH c;
```

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

You can set the size of the batch:

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

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

This is important for long outputs, as processing them row by row is inefficient.

What if we reach the end of the table?

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

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

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

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

FETCH will just stop returning rows. All regular programming languages have a way to check for this condition.

- How would you fetch data row by row with a cursor in your programming language?
- Can you get all the rows at once without using a cursor?
- How do you set the cursor batch size?

You can close your cursor when done, freeing up some resources:

```
=> CLOSE c;
```

```
CLOSE CURSOR
```

However, cursors close automatically on transaction completion, so you don't need to close them explicitly (except for cursors initiated with the WITH HOLD key.)

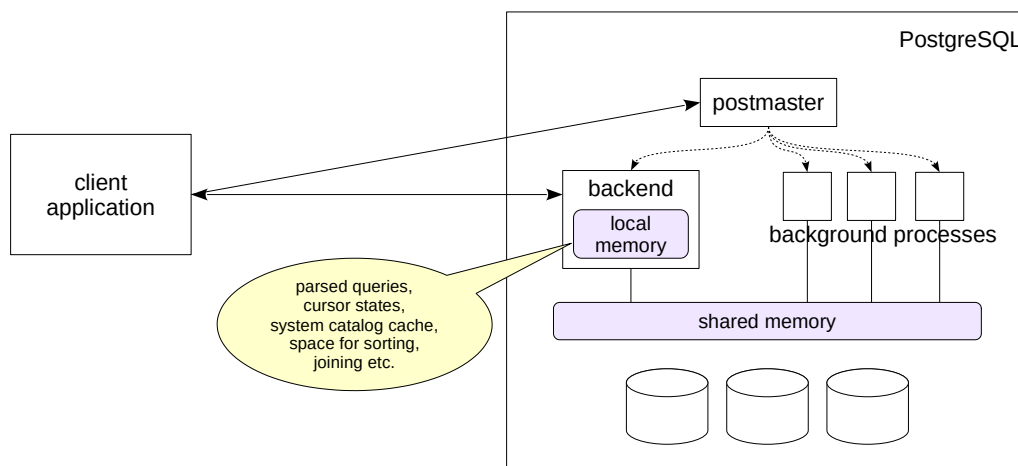
```
=> COMMIT;
```







# Processes and memory



11

Between processing queries from clients, the server must store technical information, such as parsed queries and their plans and the status of open cursors (portals). But where is it stored and how?

Under the hood, a PostgreSQL server consists of several interacting processes.

First of all, when the server starts, a process traditionally called **postmaster** is started. It starts all other processes (using the `fork` system call in Unix). It also “babysits” them: if any of the processes crashes, **postmaster** will restart it (or restart the entire server if it considers that the failed process could have damaged any of the shared data).

Operations of the server are maintained by a number of background processes. The main ones will be discussed in later topics.

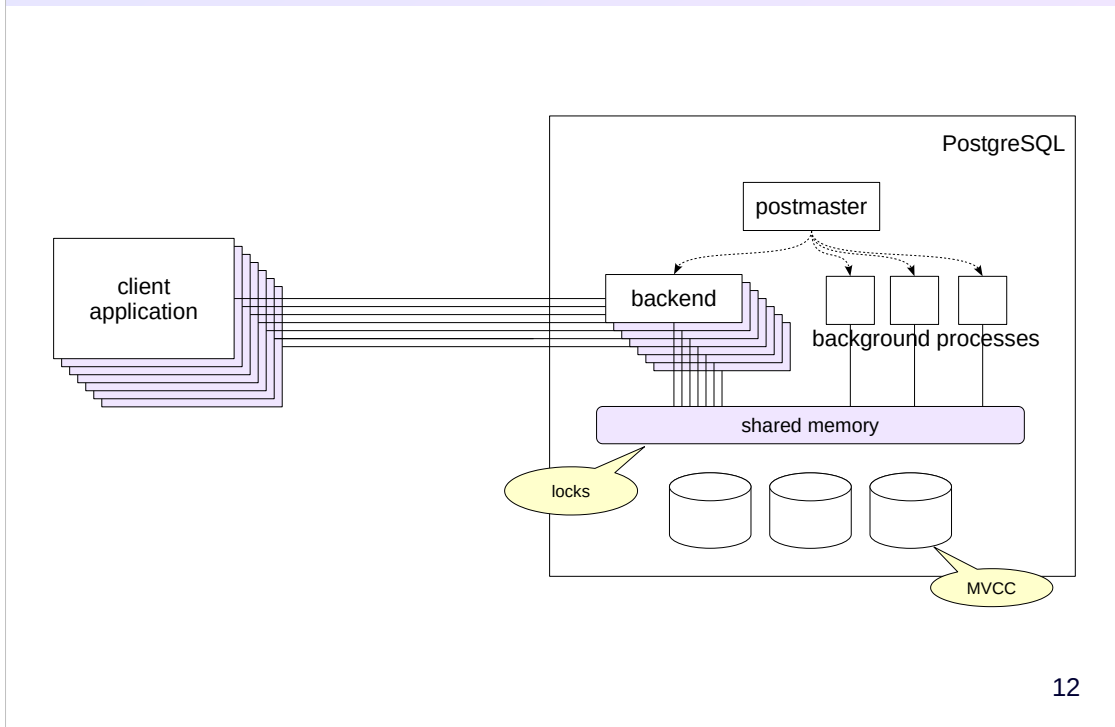
In order for the processes to exchange information between them, **postmaster** allocates *shared memory* that all the processes can access. In addition to shared memory, each process has its own *local memory*, accessible only to itself.

**Postmaster** also listens for incoming connections. For each connecting client, **postmaster** generates a designated backend process for the client to communicate with on the server side, and each client gets its own process.

The space required to execute a client’s query (parsed queries and their plans, cursor states, system catalog cache, a place to sort data, etc.) is allocated in the *local memory* of the backend process of this client.



# Multiple clients



When multiple clients connect to a server, each one gets a backend process created for it. As long as there are not too many clients, RAM is sufficient, and connections do not occur too often, sustaining many connections at once isn't a problem in itself.

However, when multiple processes try to access the same database object, things must be done to ensure that one process will not change the data while another is in the process of reading it.

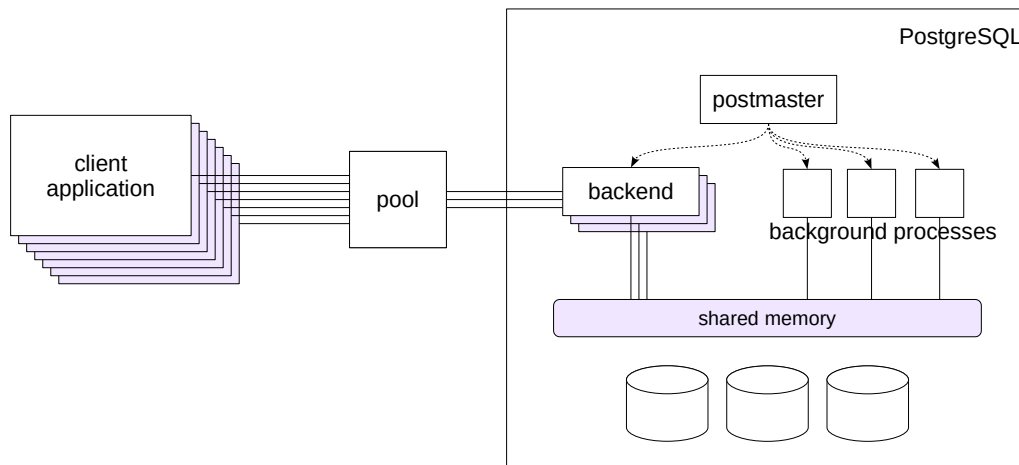
For objects in shared memory, this is ensured by short-term locks. PostgreSQL does this carefully enough so that the system scales well with an increase in the number of processors (cores).

Tables are more complicated. Locks will have to be held until the end of transactions (that is, potentially for a long time), so scalability may suffer. To avoid that, PostgreSQL uses a *multiversion concurrency control mechanism* (MVCC) and *snapshot isolation*: multiple versions of the same data can exist simultaneously, and each process sees only its own (but always consistent) data snapshot. Now, only those processes that are trying to change data that has already been changed, but not yet committed by other processes, will be locked.

MVCC is the main mechanism that enables the first three properties of transactions (atomicity, consistency, and isolation). We will talk about it more in its dedicated topic.



# Connection pool



13

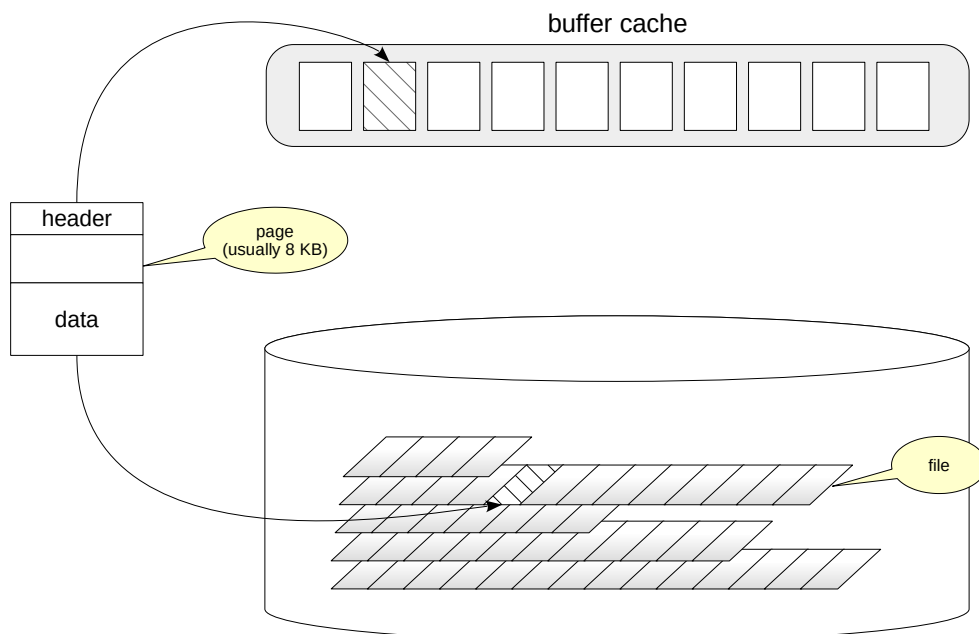
If there are too many clients, or connections are established and broken too often, using a connection pool may help. This functionality is usually provided by the application server or third-party pool managers (the most popular of which is PgBouncer).

With a connection pool in place, clients connect not to the PostgreSQL server directly, but to the pool manager. The manager keeps several connections to the database server open and uses free ones to fulfill client queries. From the server's point of view, the number of clients remains constant regardless of how many clients access the pool manager.

The drawback is that multiple clients end up sharing the same backend process, which, as we remember, stores client-specific state in its local memory (such as parsed queries for prepared statements). Therefore, care should be taken when developing applications for such deployments.

The use of connection pools is discussed in greater detail in the DEV2 course.





Data is stored as regular OS files on disks. How exactly the data is distributed among the files is discussed in the topic “Low level”.

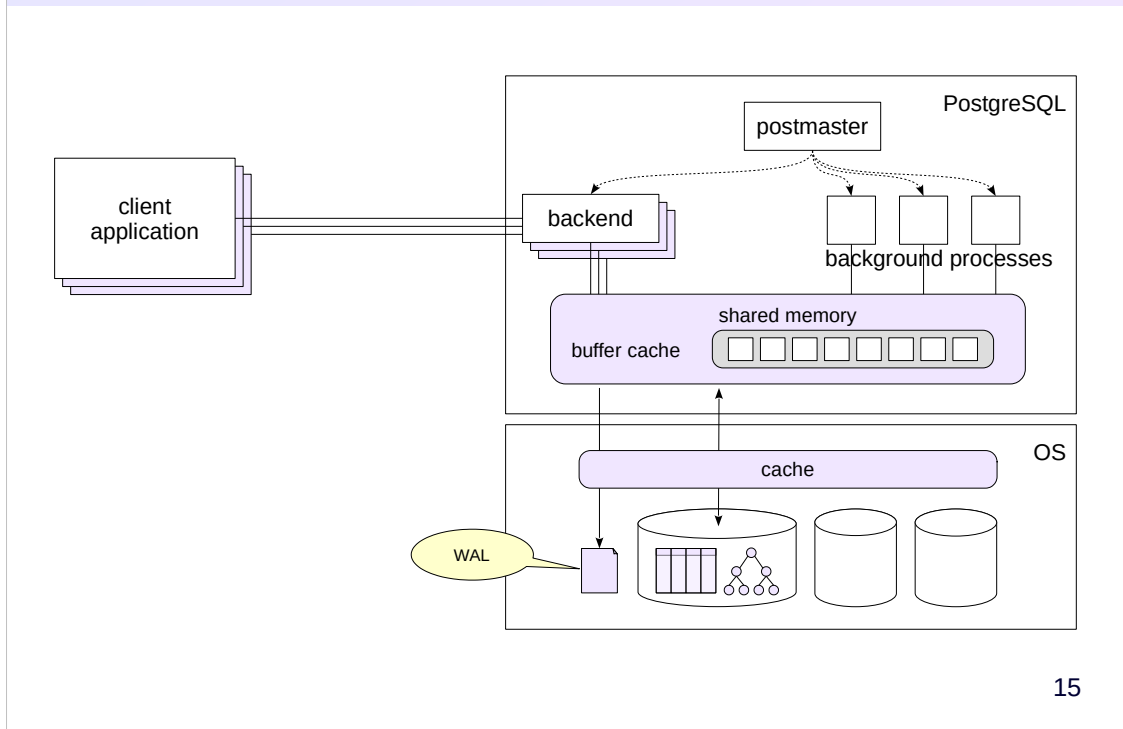
Logically, the files are divided into *pages* (sometimes the term *block* is used). A page is usually 8 KB in size. It can be changed within some limits (16 KB or 32 KB), but only during server compilation. A cluster that has been compiled and started up can work with pages of only one size.

Each page has a certain internal structure. It contains a header and actual data. There may be free space between them if the page is not fully occupied.

Since disks work much slower than RAM (especially HDD, but SSD too), data heading to disk and back is *cached* first. A *buffer cache*, a certain amount of space in RAM, is allocated for recently read pages. The idea is that the system may want to read the same pages multiple times, and keeping them on hand may save time compared to repeated disk scans. Any recently changed data is also cached for some time before being written on disk.



# Data storage



15

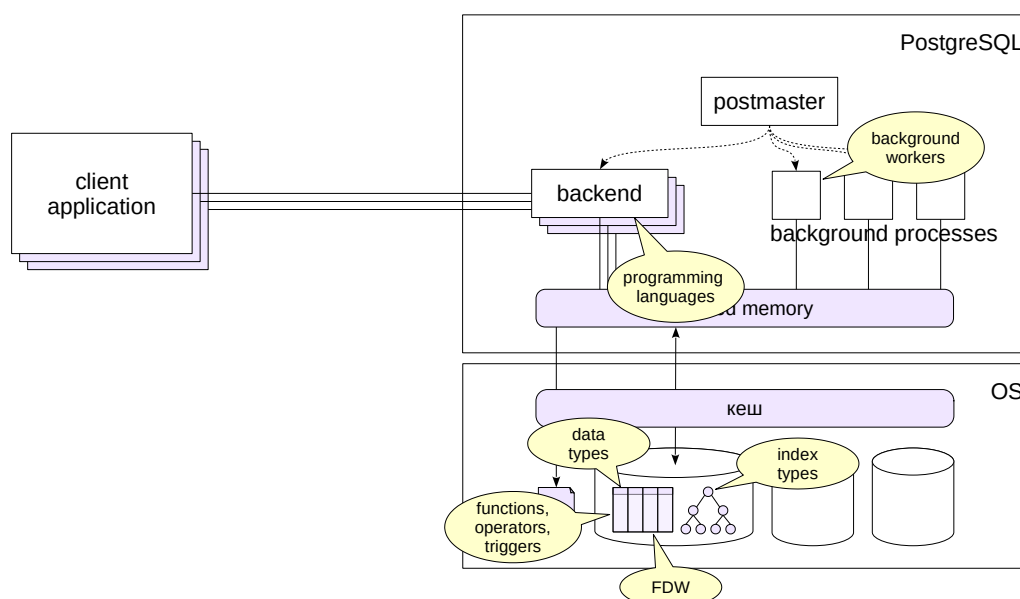
The PostgreSQL buffer cache is located in shared memory so that all processes have access to it.

PostgreSQL doesn't directly access disks storing its data. Instead, it relies on the operating system. The operating system also has its own data cache. Therefore, if a page is not found in the buffer cache, there is a chance that it is in the OS cache and access to the disk will be avoided.

In case of a failure (for example, power supply dies), the contents of the RAM are lost and the data changed but not yet written to disk will be lost. This is unacceptable and breaks the durability property of transactions. To avoid that, PostgreSQL keeps a log that allows it to redo lost operations and restore data to a consistent state. We will talk about the buffer cache and the write-ahead log in the dedicated topic later on.



# Extensibility



16

PostgreSQL is designed with extensibility in mind.

An application developer can create their own data types based on existing ones (composite types, ranges, arrays, enumerations) and stored functions for data processing (including triggers for specific events).

And with the C programming language, extensions can be developed to add arbitrary functionality to the system. Most extensions can be installed “hot”, without stopping the server. Thanks to this architecture, there are plenty of existing extensions doing things such as:

- adding support for programming languages (in addition to standard SQL, PL/pgSQL PL/Perl, PL/Python and PL/Tcl),
- introducing new data types and operators to work with them,
- creating new index types that work more efficiently with specific data types (in addition to standard B-trees, GiST, SP-GiST, GIN, BRIN, Bloom),
- interfacing with external systems using foreign data wrappers (FDW),
- starting background workers to perform periodic tasks.

Extensibility is discussed in more detail the DEV2 course.



A server manages a database cluster

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

Each client is served by a dedicated backend process

Data is stored in files and accessed via the operating system

Data is cached both in local memory (system catalog, parsed queries) and in shared memory (buffer cache)