

Architecture

PostgreSQL Fundamentals



Copyright

© Postgres Professional, 2017–2025

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov, Igor Gnatyuk

Translated by: Liudmila Mantrova, Alexander Meleshko, Elena Sharafutdinova

Photo: Oleg Bartunov (Phu Monastery and Bhrikuti Peak, Nepal)

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

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

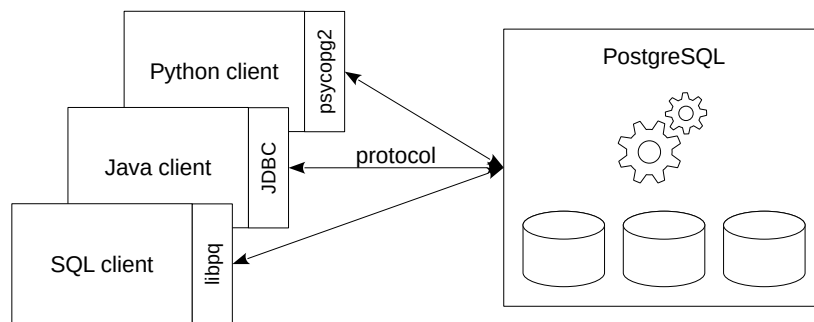
Query Processing and Execution

Processes and Memory Structures

Storing Data on Disk and Data Processing

System Extensibility

Client and Server



connection
query generation
transaction management

authentication
query execution
transactionality support

A client application, such as psql or any other program written in any programming language, connects to the server and “communicates” with it. 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.

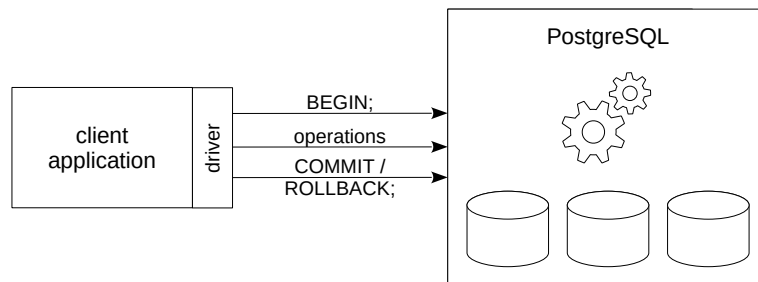
Generally speaking, a connection protocol allows the client to connect to a database in a cluster. The server performs *authentication*: decides if the client should be allowed to connect, i.e. by demanding a password.

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 transaction support.

<https://postgrespro.com/docs/postgresql/16/protocol>

Transactions



atomicity	— <i>everything or nothing</i>
consistency	— <i>integrity constraints and user restrictions</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 executed in full or not executed 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 the specific one.
- **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 be managed on the server side by stored procedures.

<https://postgrespro.com/docs/postgresql/16/sql-begin>

<https://postgrespro.com/docs/postgresql/16/sql-savepoint>

<https://postgrespro.com/docs/postgresql/16/transactions>

Transaction Management

By default, psql runs in autocommit mode:

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

on

This means that any single command sent without explicitly opening the transaction is committed immediately.

- Check if this mode is enabled by default in the PostgreSQL driver for your favourite programming language.

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 this table and row?

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

id	s
1	foo

(1 row)

Yes. Compare the result:

```
=> BEGIN; -- explicit start of 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 implicitly.

```
=> 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
```

Individual changes can be rolled back without interrupting the entire transaction (although this is rarely used).

```
=> 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.

Rolling back to a save point does not imply a transfer of control (that is, it does not work as GOTO); only those changes to the database state that were made from the moment the point was set and up to the current moment are canceled.

```
=> 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

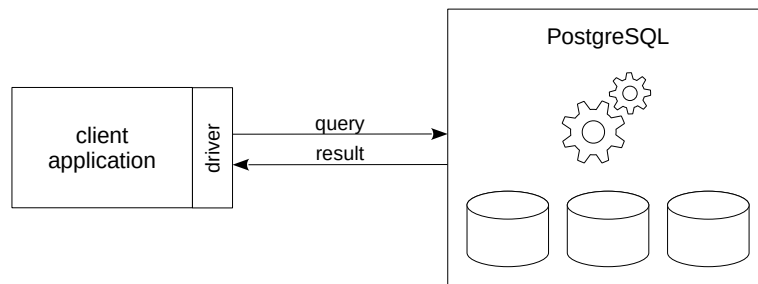
```
=> COMMIT;
```

```
COMMIT
```

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

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

Query Processing



parsing	← <i>system catalog</i>
rewriting	← <i>rules</i>
planning	← <i>statistics</i>
execution	← <i>data</i>

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.

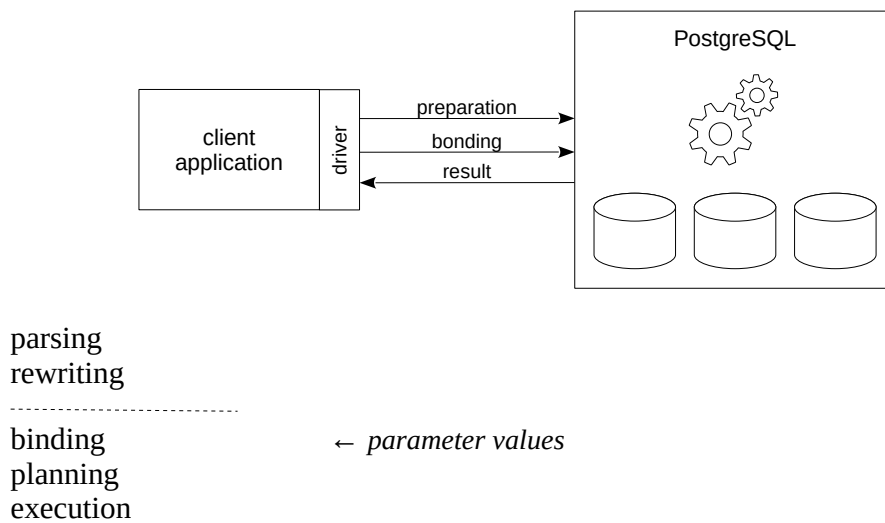
SQL is a declarative language: a query defines what data to get, but not how to get it. It is at this point when the query (already parsed and presented in the form of a tree) is passed on to the *planner*, which develops an *execution* plan. For example, the planner can decide whether or not to use 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.

<https://postgrespro.com/docs/postgresql/16/query-path>

This is convenient and simple when we are talking about just a row or two, provided by the *simple mode* of the protocol.

Prepared Statements



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/16/sql-prepare>

<https://postgrespro.com/docs/postgresql/16/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)
```

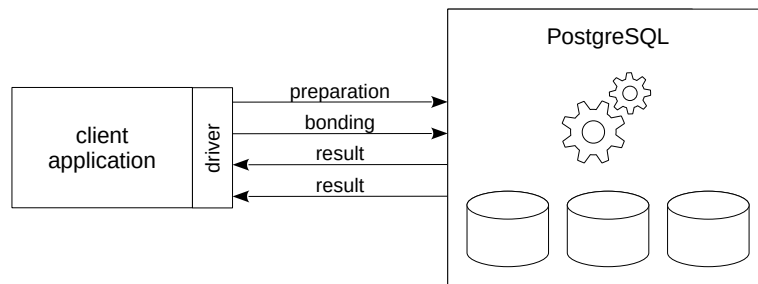
For non-parametric statements, an execution plan is saved as well. For queries with parameters, as in the example, the parameter values are taken into account at the planning stage. The planner may decide that the plan built without considering the parameters is good enough. In this case, it will not try to plan the query again.

- How do you prepare and execute a statement using your favourite programming language?
- Can you execute a statement WITHOUT preparing it?

All prepared statements in the current session can be displayed with:

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

```
-[ RECORD 1 ]---+-----
name          | q
statement     | PREPARE q(integer) AS
               |     SELECT * FROM t WHERE id = $1;
prepare_time  | 2025-11-27 14:18:05.968349+03
parameter_types | {integer}
result_types  | {integer,text}
from_sql      | t
generic_plans | 0
custom_plans  | 1
```



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/16/sql-declare>

<https://postgrespro.com/docs/postgresql/16/sql-fetch>

Cursors

When the SELECT command is executed, the server sends and the client receives all the rows at once:

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

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

Cursors let the client to retrieve data in batches of rows.

```
=> 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)
```

The batch size is paramount for huge outputs, when processing them one row at a time 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 do you get data row by row using the cursor in your favorite programming language?
- Is it possible NOT to use the cursor and get all the rows at once?
- How is the cursor batch size set?

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

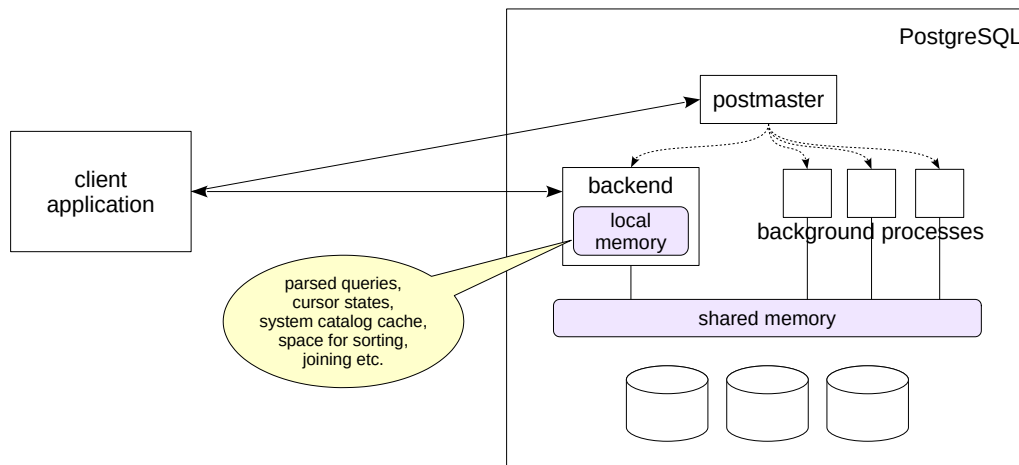
```
=> CLOSE c;
```

```
CLOSE CURSOR
```

However, cursors are automatically closed when the transaction ends, so explicit closing is optional (except WITH HOLD cursors.)

```
=> 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). 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).

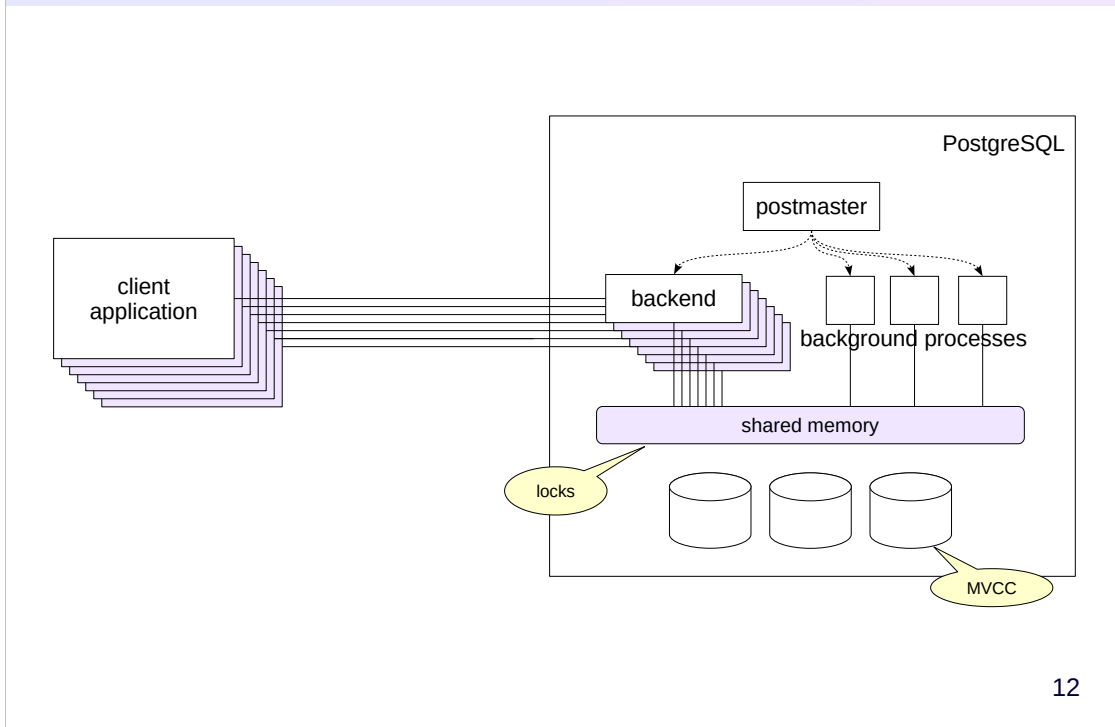
Operations of the server are maintained by a number of background processes. The main ones will be discussed in the following lessons.

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 backend process performs authentication, among other things.

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 is not a problem in itself.

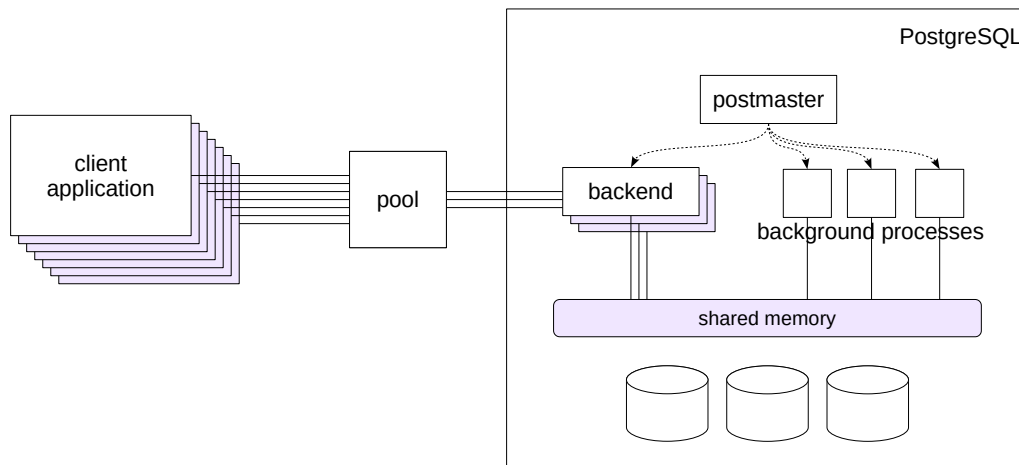
However, when multiple processes try to access the same database object, things must be done to ensure that one process will not change the data while another is in the process of reading it.

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 lesson.

Connection Pool



13

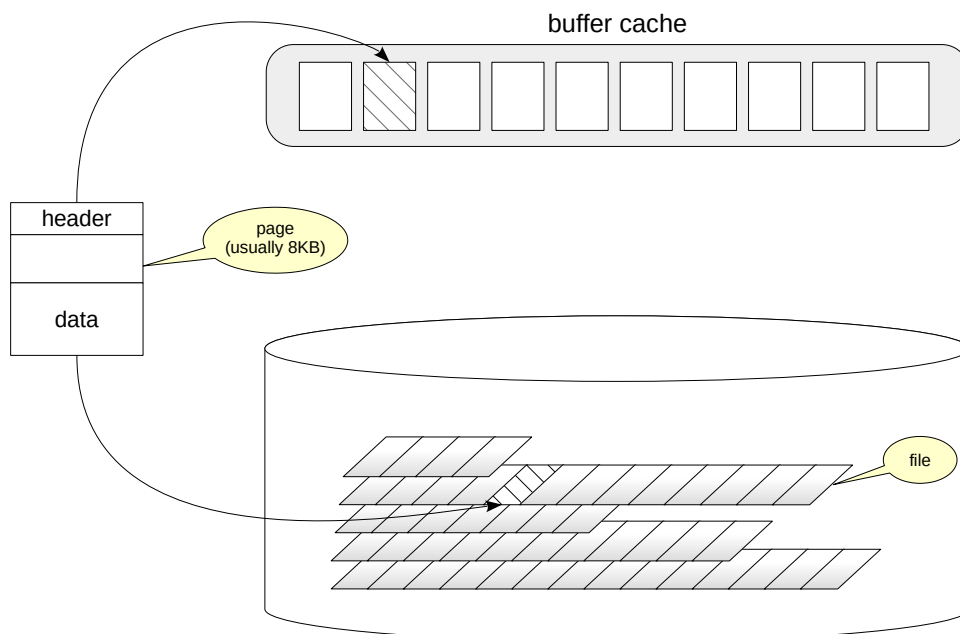
If there are too many clients, or connections are established and broken too often, using a connection pool may help. This functionality is usually provided by the application server or third-party pool managers (the most popular of which is pgBouncer).

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.

One of pgBouncer's features is the ability to temporarily pause client connections without disconnecting them. This pause can be used for server software updates or other operations that require a server restart.

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



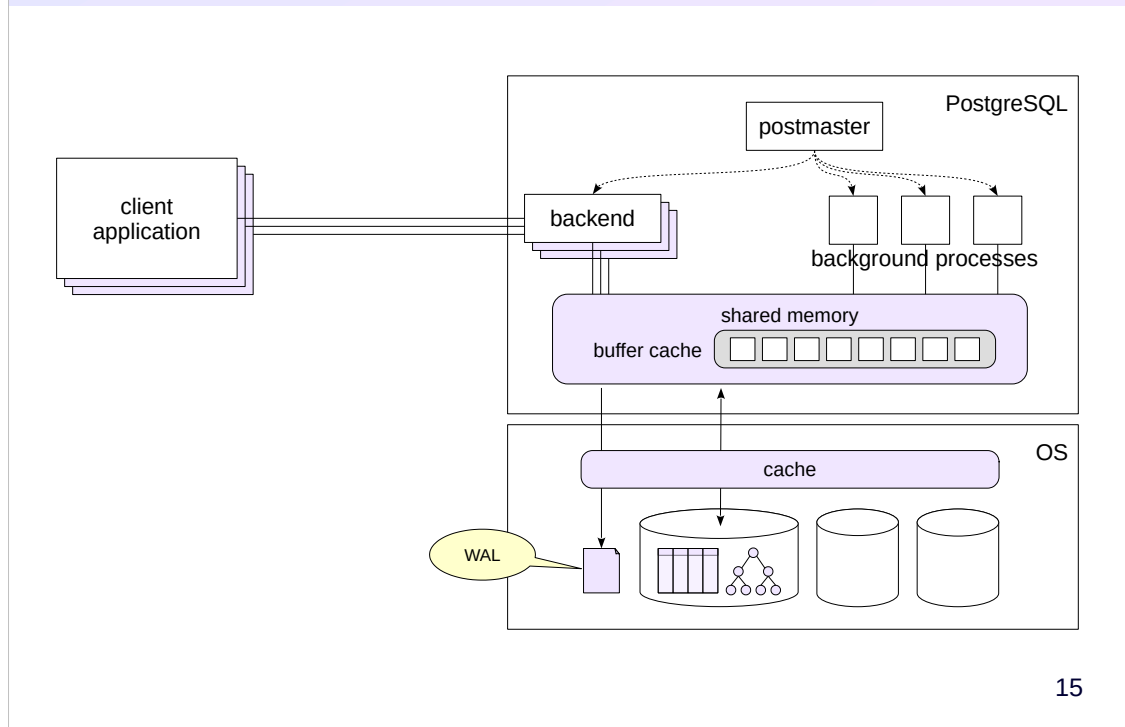
Data is stored as regular OS files on disks. How exactly the data is distributed among the files is discussed in one of following lessons.

Logically, the files are divided into *pages* (sometimes the term *block* is used). A page is usually 8KB 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 this way 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 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



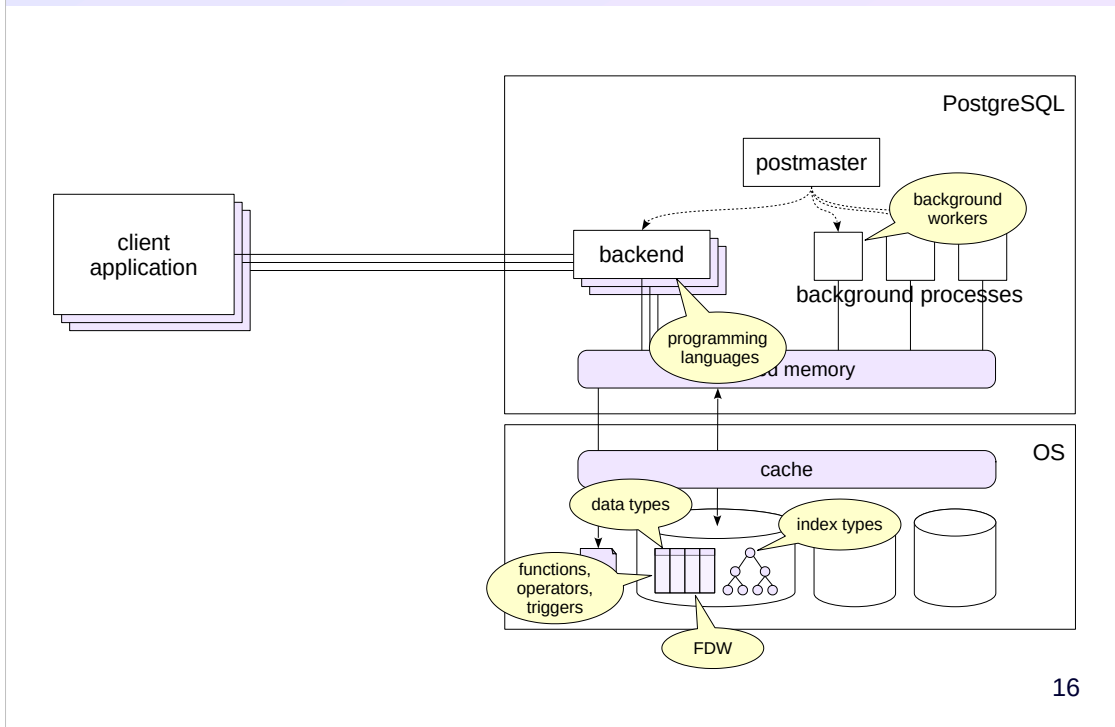
15

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

PostgreSQL does not 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 lesson later on.

Extensibility



PostgreSQL is designed with extensibility in mind. It allows to create new data types based on existing ones, write stored procedures for data processing, and offers a convenient toolkit for administration, monitoring, and performance tuning.

You can always write an extension that adds the necessary functionality that you need. 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, hash indexes, GiST, SP-GiST, GIN, BRIN);
- starting background workers to perform additional tasks;
- connecting to external data sources;
- collecting system load information, performing monitoring, and generating reports;
- exploring system data structures.

Extensibility is discussed in more detail in the DBA2 and DEV2 courses.

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)