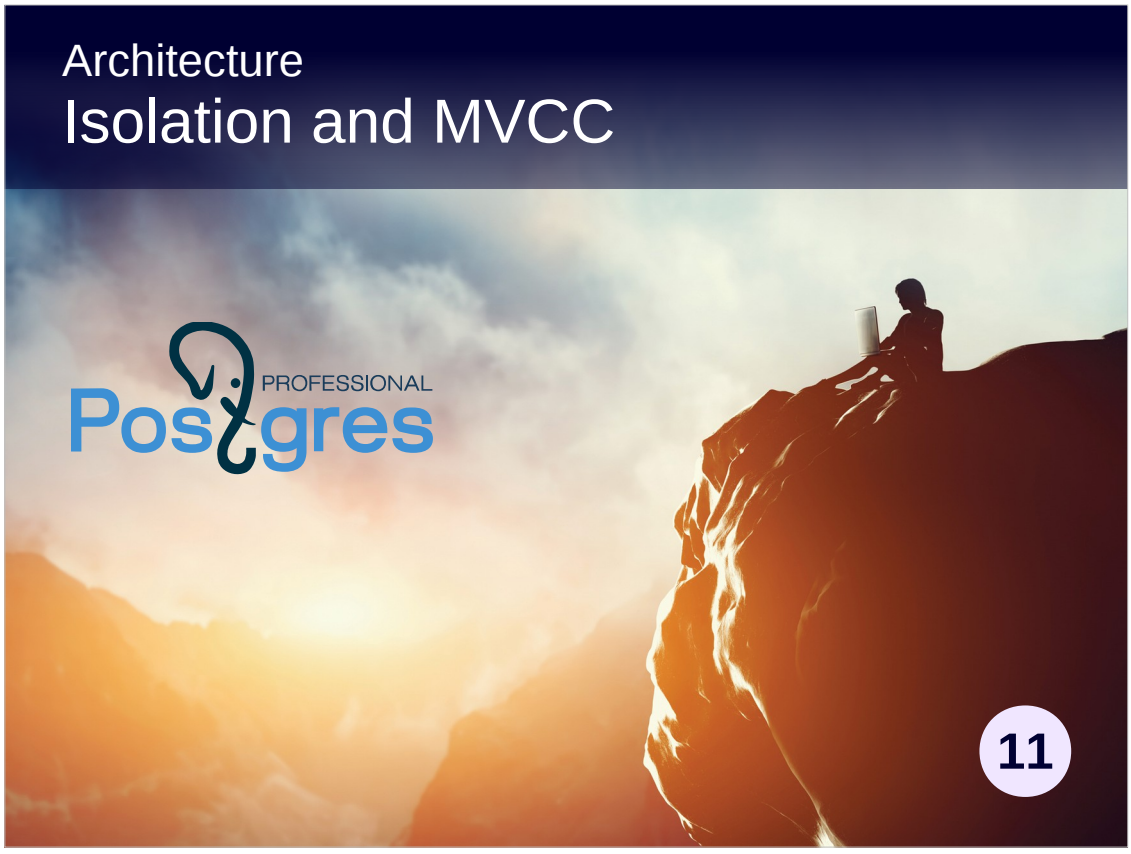


Authors: Egor Rogov, Pavel Luzanov

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

Topics

Multiversion Concurrency Control

Snapshots

Isolation levels

Locks

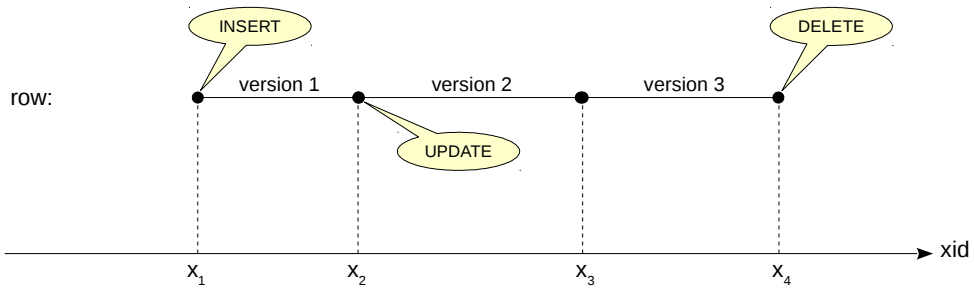
Vacuuming

MVCC

Multiple versions of the same row

versions differ in time of action

time = transaction id (issued in ascending order)



3

When several transactions are working simultaneously, a problem arises: what to do if two of them concurrentry access the same row? If both transactions are read-only, there is no difficulty. Also there is no problem with two transactions trying to change the row (in this case, they are entering the queue and making changes one after another). The most interesting case is how the writing and reading transactions interact.

There are two simple ways. Such transactions can block each other — but then performance suffers. Or, the reading transaction immediately sees the changes made by the writing transaction, even if they are not committed — but this strongly undesirable, because the changes can be rolled back afterwards (this is called «dirty reading»).

PostgreSQL goes the hard way and uses MVCC: Multiversioning Concurrency Control. It stores several versions of the same row (these versions are often called *tuples* in PostgreSQL jargon). In this case, the writing transaction works with its version, and the reader sees its own.

To distinguish versions from each other, they are marked with two numbers defining the «time» of the action of this version. Time is represented not by timestamp, but by always-increasing transaction ids (xid). In fact, everything is a little more complicated, but its not relevant for now.

When a tuple is created, it is marked with the id of the transaction that executed the INSERT command. When deleted, the tuple is marked with the id of the transaction that performed DELETE (but not physically deleted). UPDATE consists of two operations DELETE and INSERT.

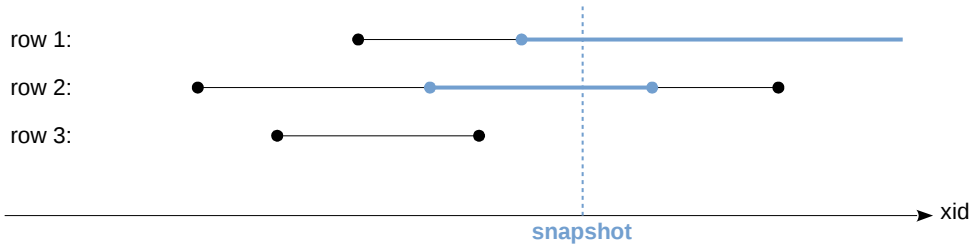
https://postgrespro.com/docs/postgresql/11/mvcc

Snapshot

Consistent slice of data at a certain point

transaction id — determines the point in time

list of active transactions — not to look on not-yet-committed changes



4

Transaction must see at most one version of each row, so that the entire data set is consistent. For this, the transaction works with a snapshot of data created at a specific point in time. A snapshot is not a physical copy of the data, but only a few numbers:

- the current transaction id at the time of the snapshot creation (it determines that very point in time),

- the list of active transactions at this moment.

The list is needed in order to consider the changes of only those transactions that were committed before the creation of the snapshot. We are not interested in transactions that began before the creation of the snapshot, but have not yet been committed, as well as those that began after the creation of the snapshot.

Knowing the snapshot, we can always tell which row version will be visible in it. Sometimes this will be the actual (most recent) version, as for row 1 in the illustration. Sometimes not the latest: row 2 is deleted (and the change is already committed), but the transaction still continues to see this row while working with its snapshot. This is the correct behavior: it gives a consistent data snapshot at the selected point in time.

Some rows do not fall into the snapshot at all: row 3 is deleted before the snapshot was built, so it is not in the snapshot.

Locks

Row locks

- reading never blocks rows
- changing a row blocks it for changes, but not for reads

Table locks

- forbid changing or deleting a table while working on it
- forbid reading the table when rebuilding or moving
- and so on

Locks lifetime

- set automatically as needed or manually
- removed automatically upon completion of the transaction

Why does MVCC worth the effort? It allows PostgreSQL to do only the most necessary minimum of locks, thereby increasing system performance.

Most of locks are acquired at the row level. Reading never blocks neither reading nor writing transactions. Changing the row does not block its reading. The only case where the transaction will wait for the lock to be released is if it tries to change a row that has already been changed by another, not yet committed, transaction.

Locks are also acquired at a higher level, in particular on tables. They are needed so that no one can delete a table while other transactions read data from it, or to deny access to a table during rebuilt. As a rule, such locks do not cause problems, since deleting or rebuilding tables is a quite rare operation.

All necessary locks are acquired automatically and automatically released at the end of the transaction. You can also use additional user locks; the need for this may arises when using isolation levels less strict than Serializable.

https://postgrespro.com/docs/postgresql/11/explicit-locking

Transactions status (xact)

Postgres PROFESSIONAL

Transactions status

- service information; two bits per transaction
- special files on disk
- shared memory buffers

Commit

- set the «transaction committed» bit

Abort

- set the «transaction aborted» bit
- runs as fast as commit (no rollback required)

MVCC requires to know the status of transactions. Transaction can be in-progress, or it can be completed either with commit or rollback. Thus, the status of each transaction requires two bits. The statuses are stored in special service files, and most recent data is cached in the shared memory for efficiency.

Previously, transaction status files were located in the `PGDATA/pg_clog` directory; since version 10 this directory has been renamed to `pg_xact`.

At any completion of the transaction (both successful and unsuccessful), PostgreSQL only needs to set the corresponding status bits. Both commit and rollback occur equally quickly.

If the aborted transaction managed to create new row versions, these versions remain in data files (there is no «physical rollback» of data). Thanks to the status information, other transactions will understand that the transaction that created or deleted the row versions is in fact aborted and will not take its changes into account.

Vacuum

Old row versions are stored along with the current one

- over time, the size of tables and indexes increases

Cleaning process (vacuum)

- deletes row versions that are no longer needed
  (i. e. not visible in any data snapshot)
- works in parallel with other processes
- deleted versions leaves «holes» in data files,
  which can be used for new row versions

Full vacuum

- completely rebuilds the data files, making them compact
- exclusively locks the table at run time

In PostgreSQL, all row versions of a table, both current and historical, are stored together in the same data file. It is clear that over time old row versions accumulate and this leads to bloat in tables (and in indexes) and therefore to performance degradation.

Meanwhile, there is no need to store tuples that are no longer visible in any snapshot. Such tuples are called *dead* and removed by a special cleaning process named *vacuum*. Vacuum physically removes unnecessary tuples from files, leaving «holes» that are then used for new tuples.

Vacuuming does not block other processes and works in parallel with them.

It is possible to completely rebuild the table and all its indexes by performing a full vacuum. In this case, the data files become compact, but this process completely blocks the work with the table during its work.

https://postgrespro.com/docs/postgresql/11/routine-vacuuming

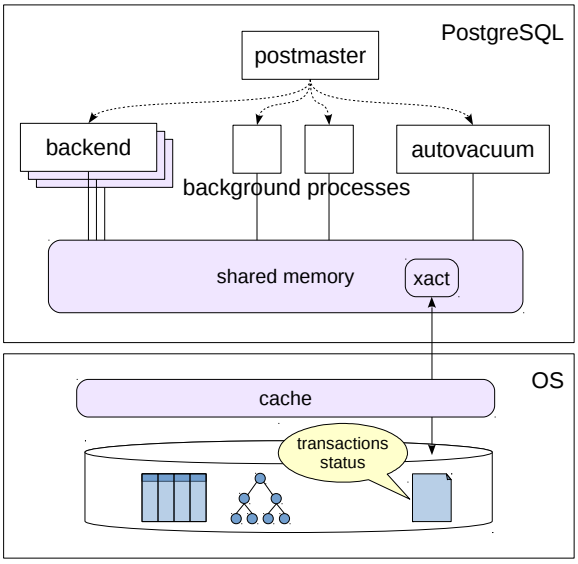
Autovacuum

Autovacuum launcher

- background process
- reacts to data change activity

Autovacuum worker

- started by the launcher as needed
- performs actual cleaning



Vacuum usually works automatically and is configured by the administrator to clear the data on time, avoiding a large increase in the size of the files. To do this, autovacuum responds to the activity of changing data in the tables, and not just run on a schedule: the more frequently changes data in a table, the more frequently the table is vacuumed.

Autovacuum consists of several processes. Autovacuum launcher background process schedules the work. If necessary, it starts autovacuum worker processes, which are doing actual cleaning.

Isolation levels

Read uncommitted — *not supported by PostgreSQL*

- allows to read uncommitted data

Read committed — *used by default*

- snapshot is built at the beginning of each statement
- subsequent identical request can get different data

Repeatable read

- snapshot is built at the beginning of the first statement in the transaction
- transaction may end in serialization error

Serializable

- absolute isolation, but additional overhead
- transaction may end in serialization error

The SQL standard defines four levels of isolation: the stricter the level, the less influence concurrent transactions have on each other. At the time when the standard was adopted, it was believed that the stricter the level, the more difficult it is to implement and the stronger its impact on performance (since then, these ideas have changed somewhat).

The less strict **Read Uncommitted** level allows dirty reads and is not supported by PostgreSQL: it does not have any practical value and does not give a gain in performance.

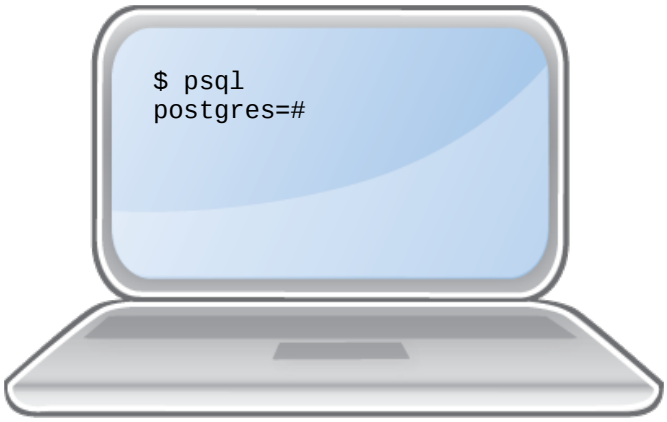
The **Read Committed** level is the default isolation level in PostgreSQL. At this level, a snapshot of the data is built at the beginning of each SQL statement. Thus, the statement works with constant and consistent data, but two identical requests, following one after the other, can show different results.

At **Repeatable Read** level, a snapshot is built at the beginning of a transaction (when the first statement is executed). Therefore, all requests in one transaction see the same data. This level is useful, for example, for reports consisting of several queries.

The **Serializable** level guarantees complete isolation: you can write statements as if the transaction is running alone. Cost of convenience is failure of a certain percentage of transactions (with «cannot serialize access» error message). The application must be able to re-run such transactions.

https://postgrespro.com/docs/postgresql/11/transaction-iso

Demonstration



Summary

Multiple versions of each row can be stored in data files

Transactions operate on a consistent data snapshot

Writers do not block readers, readers do not block anyone

Snapshot creation time determines isolation level

Row versions accumulate, so periodic vacuuming is needed

Practice

1. Create a table with one row.
2. Start the first transaction and query the table.
3. In the second session, delete the row and commit the changes.
4. How many rows will the first transaction see by running the same query again? Check it out.
5. Complete the first transaction.
6. Repeat the same thing, but now let the transaction work at the Repeatable Read isolation level:

   ```
   BEGIN ISOLATION LEVEL REPEATABLE READ;
   ```

   Explain the differences.