Authors: Egor Rogov, Pavel Luzanov
Translated by Liudmila Mantrova

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

Agenda

Multi-version concurrency control (MVCC)

Data snapshots

Isolation levels

Locks

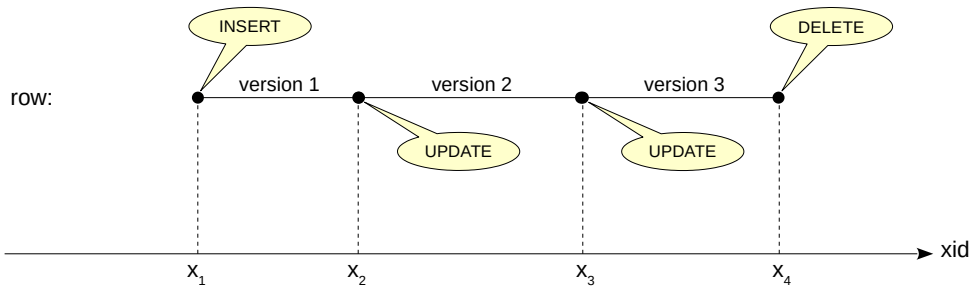
Vacuum

MVCC

There can be multiple versions of one and the same row

different versions belong to different time frames

time = transaction ID (xid), assigned in the ascending order



3

Running several concurrent sessions poses an interesting question: what should be done if two transactions access one and the same row simultaneously? The answer is obvious if both of them read data. Handling two writing transactions is also easy (they are simply queued to update the data sequentially). The most intricate case is managing reads and writes.

There are two straightforward ways to do it. Transactions can lock each other, but then performance will suffer. Alternatively, a reading transaction could immediately see the changes made by a writing transaction, even if they are not committed yet (it is called “dirty read”); but it is a very bad scenario because these changes can be rolled back.

PostgreSQL does it the hard way: it keeps several versions of one and the same row using *multi-version concurrency control (MVCC)*. Thus, while a reading transaction sees one version, a writing transaction modifies another version.

To tell one version from another, PostgreSQL marks them with two tags that define the “lifetime” of each version. Instead of timestamps, these tags use transaction IDs, which are always increasing (it’s a bit more complex than that, but we won’t get into details.) When a row is inserted, it is tagged with ID of the transaction that has performed the `INSERT` command. When it is deleted, it is tagged with ID of the transaction that has run the `DELETE` command (but is not physically deleted). `UPDATE` consists of two operations: `DELETE` and `INSERT`.

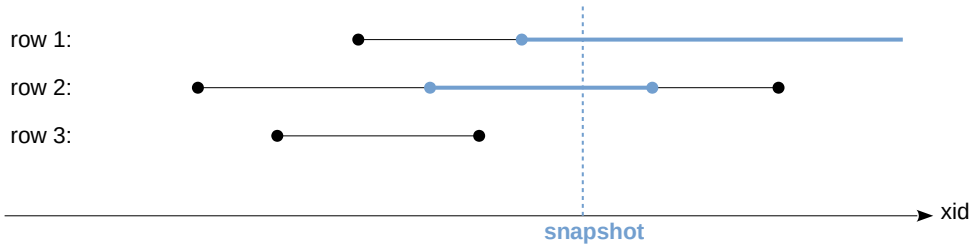
https://postgrespro.com/docs/postgresql/12/mvcc-intro

Data Snapshot

A consistent slice of data at a particular moment

transaction ID (xid) defines the snapshot creation moment

all changes that are not yet committed at this moment are filtered out using the list of active transactions



4

PostgreSQL uses *snapshot isolation*.

When accessing a table, a transaction must see only one version of each row (or no versions at all). To achieve this, transactions use data snapshots taken at a particular moment. Each snapshot contains only the latest versions of committed data; if the data was not committed at that moment, it won't be visible in the snapshot. In other words, each row is represented by its version that was current at the time of snapshot creation.

A snapshot is not a physical copy of all data; it's just several numbers:

- the ID of the last transaction committed by the moment of snapshot creation (which defines that moment)

- the list of transactions that were active at that time

The list is required to ensure that the snapshot does not contain any changes of those transactions that had started before the snapshot creation, but had not been committed by that time.

Knowing these numbers, we can always say which row version is visible in the snapshot. Sometimes it is the current (the most recent) version, like in the case of row 1 on this slide. Sometimes it is an earlier version: row 2 is deleted (and this change is already committed), but the transaction still sees this row while using the snapshot. Such behavior is correct: it ensures that the data is consistent at each point in time.

Some rows will not make it into the snapshot at all: row 3 had been deleted before the snapshot was built, so it was not included into the snapshot.

Isolation Levels

Read Uncommitted

is not supported by PostgreSQL: works as Read Committed

Read Committed (*default*)

the snapshot is taken at the beginning of an operator
identical queries can return different data

Repeatable Read

the snapshot is taken at the beginning of the first operator in the transaction
transactions can end with a serialization error

Serializable

full isolation, but additional overhead
transactions can end with a serialization error

The SQL standard defines four isolation levels: the stricter the level, the less the interference between concurrent transactions. At the time when the standard was adopted, it was assumed that stricter levels are harder to implement and have higher negative impact on performance (since then these views have somewhat changed).

The most relaxed level is **Read Uncommitted**, which allows dirty reads. This level is not supported by PostgreSQL since it is of no practical use and provides no performance gains.

The **Read Committed** level is the default isolation level in PostgreSQL. At this level, snapshots are built at the beginning of each SQL operator. Thus, each operator works with constant and consistent data, but two identical queries can return different results when run one after another.

At the **Repeatable Read** level, a snapshot is built at the beginning of each transaction (while executing the first operator), so all queries within one transaction see the same data. This level is convenient for cases like creating reports using series of queries.

The **Serializable** level provides full isolation: you can write operators as if there is only one transaction. The price you pay for convenience is that some transactions complete with an error; your application should be able to retry such transactions.

https://postgrespro.com/docs/postgresql/12/transaction-iso

Visibility of Row Versions

How can we check that one and the same row can have several versions simultaneously?

Let's create a table:

```
=> CREATE TABLE t(s text);
```

```
CREATE TABLE
```

Insert a row. As we know, if the BEGIN command is not explicitly specified at the beginning of a transaction, psql executes the command and commits the result immediately:

```
=> INSERT INTO t VALUES ('Version one');
```

```
INSERT 0 1
```

Start a transaction and display its ID:

```
=> BEGIN;
```

```
BEGIN
```

```
=> SELECT txid_current();
```

```
 txid_current
--------------
          502
(1 row)
```

This transaction sees the first version of the row (which is currently the only one available):

```
=> SELECT *, xmin, xmax FROM t;
```

```
      s      | xmin | xmax
-------------+------+------
 Version one |  501 |    0
(1 row)
```

Here we also display IDs of transactions that define the visibility limits of this row version. The row is inserted by the previous transaction, and xmax=0 means that it is the current version of the row.

Now let's start another transaction in another session:

```
=> BEGIN;
```

```
BEGIN
```

```
=> SELECT txid_current();
```

```
 txid_current
--------------
          503
(1 row)
```

The transaction sees the first version of the row:

```
=> SELECT *, xmin, xmax FROM t;
```

```
      s      | xmin | xmax
-------------+------+------
 Version one |  501 |    0
(1 row)
```

Now let's update the row in the second transaction.

```
=> UPDATE t SET s = 'Version two';
```

```
UPDATE 1
```

Here is what we've got:

```
=> SELECT *, xmin, xmax FROM t;
```

```
       s      | xmin | xmax
-------------+------+------
 Version two |  503 |    0
(1 row)
```

What will the first transaction see?

```
=> SELECT *, xmin, xmax FROM t;
```

```
      s      | xmin | xmax
-------------+------+------
 Version one |  501 |  503
(1 row)
```

Since this update is not committed, the first transaction still sees the first version of the row.

Note the xmax value: it shows that another transaction is changing this row at the moment. In fact, such “peeking” violates isolation, so xmin and xmax fields are hidden and should not be used in actual work.

Now let’s commit the changes.

```
=> COMMIT;
```

```
COMMIT
```

What will the first transaction see this time?

```
=> SELECT *, xmin, xmax FROM t;
```

```
      s      | xmin | xmax
-------------+------+------
 Version two |  503 |    0
(1 row)
```

Now the first transaction sees the second version of the row.

Once the update is committed, the first version of the row is not visible to any transaction anymore.

```
=> COMMIT;
```

```
COMMIT
```

Locks

Row-level locks

- reading never blocks writing
- an update locks the affected rows for other updates, but not for reading

Table-level locks

- forbid changing and deleting the table while it is worked with
- forbid reading the table while it is being rebuilt or moved
- address other similar needs

Lifetime of locks

- locks are acquired either automatically as needed or manually
- locks are released automatically at the end of transactions

What does MVCC give us? It minimizes the number of required locks, thus increasing system performance.

Most of the locks are set at the row level. That being said, reading never blocks neither reading, nor writing transactions. A row update does not block reading of this row. The only case when a transaction has to wait for the lock release is when it tries to modify the row that is already updated by another transaction, but this change is not committed yet.

Locks can also be acquired at a higher level, e.g., on tables. They are needed to ensure that no one can delete the table while other transactions are reading data from it, or to forbid access to the table that is being rebuilt. As a rule, such locks cause no issues because deleting and rebuilding tables are very rare operations.

All the required locks are acquired automatically and are released automatically once the transaction completes. It is also possible to set custom locks; such need arises not too often.

https://postgrespro.com/docs/postgresql/12/explicit-locking

Locks

Let’s retry our experiment, but have both transactions try to change one and the same row.

```
=> BEGIN;
```

```
BEGIN
```

```
=> UPDATE t SET s = 'Version three';
```

```
UPDATE 1
```

Run the second transaction:

```
=> BEGIN;
```

```
BEGIN
```

```
=> UPDATE t SET s = 'Version four';
```

The second transaction hangs: it cannot update the row until the first transaction releases the lock.

```
=> COMMIT;
```

```
COMMIT
```

Now the execution of the second transaction can continue:

```
UPDATE 1
```

```
=> COMMIT;
```

```
COMMIT
```

Transaction Status

Transaction status (xact)

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

Commit

- sets the bit “transaction is committed”

Abort

- sets the bit “transaction is aborted”
- completes as fast as a commit (no data rollback is required)

To maintain MVCC, it is required to know the status of transactions.
A transaction can be either active or completed. A completed transaction is either committed or aborted. Thus, keeping the state of each transaction requires two bits. Status information is stored in special service files; for recent transactions, this data is usually kept in shared memory to avoid frequent disk access.

By any transaction outcome (both successful and unsuccessful), it’s enough to simply set the corresponding status bits. Both commits and aborts happen equally fast.

If an aborted transaction has already created new row versions, these versions are not deleted (there is no “physical” data rollback). Thanks to the status information, other transactions will see that the transaction that has created or deleted row versions is actually aborted, and will ignore these changes.

Vacuum

Both previous and current row versions are stored together

- the size of tables and indexes grows over time

Vacuum

- deletes row versions that are no longer required (i.e., they are not visible in any data snapshot anymore)
- works in parallel with other processes
- when deleting row versions, leaves “holes” in data files, to be later filled with new row versions

Vacuum full

- fully rebuilds data files, making them compacted
- locks the table while running

In PostgreSQL, all row versions (both current and outdated) are stored together, in the same data files. It’s clear that outdated versions pile up over time, which leads to table (and index) bloating and causes performance degradation.

However, there is no need to store outdated row versions that are not visible in any data snapshot anymore. Such versions are cleaned up by the *vacuum* process. It physically removes deprecated row versions from data files, leaving “holes” that will be later filled with new row versions.

Vacuum does not lock other processes and can be run concurrently.

You can fully rebuild the table and its indexes by running the `VACUUM FULL` command. This process makes files more compacted, but the table is fully locked while it is running.

https://postgrespro.com/docs/postgresql/12/routine-vacuuming

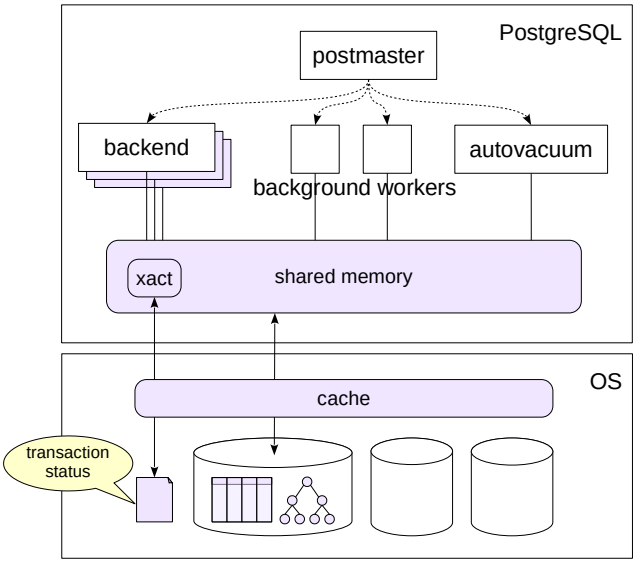
Autovacuum

Autovacuum launcher

a background process launches background workers from time to time

Autovacuum worker

performs cleanup of a particular database, for those tables that actually need it



Autovacuum is set up by DBA to perform timely data cleanup, so that considerable file bloating is avoided. Instead of cleaning up all tables on schedule, it is triggered automatically by update activity in tables.

However, abnormally long transactions can interfere with timely vacuuming. The DEV2 course provides more information on how to avoid impeding system operation.

Autovacuum is performed as follows. The background process called autovacuum launcher regularly starts autovacuum workers in various databases. These workers, in their turn, create the list of tables that require vacuuming, and then perform the cleanup.

Summary

Files can store several versions of each row

Each transaction accesses its own data snapshot, which is a consistent slice of data at a particular point in time

Writes do not block reads, reads do not block any operations

Isolation levels differ in snapshot creation times

Row versions pile up, so they require periodic cleanup

Practice

1. Create a table with a single row.

   Start the first transaction at the Read Committed isolation level and run a query on the table.
   In another session, delete the row and commit this change.
   How many rows will the first transaction see if it re-runs the query? Check the result.

   Commit the first transaction.

2. Repeat the experiment at the Repeatable Read isolation level:

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

   Explain the difference.

Task 1. Read Committed Isolation Level

Let's create a table:

```
=> CREATE TABLE t(n integer);
```

```
CREATE TABLE
```

```
=> INSERT INTO t VALUES (42);
```

```
INSERT 0 1
```

Run a query in the first transaction:

```
=> BEGIN;
```

```
BEGIN
```

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

```
 n
----
 42
(1 row)
```

Delete the row in the second transaction and commit the result:

```
=> DELETE FROM t;
```

```
DELETE 1
```

Repeat the query:

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

```
 n
---
(0 rows)
```

The first transaction sees the applied changes.

```
=> COMMIT;
```

```
COMMIT
```

Task 2. Repeatable Read Isolation Level

Let's insert a row again:

```
=> INSERT INTO t VALUES (42);
```

```
INSERT 0 1
```

Run a query in the first transaction:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN
```

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

```
 n
----
 42
(1 row)
```

Delete the row in the second transaction and commit the result:

```
=> DELETE FROM t;
```

```
DELETE 1
```

Repeat the query:

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

```
 n
----
 42
(1 row)
```

At this isolation level, the first transaction does not see the changes.

```
=> COMMIT;
```

```
COMMIT
```