Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov
Translated by Alexander Meleshko
Cover photo by Oleg Bartunov (Phu monastery and Bhrikuti peak, Nepal)

Feedback
Please send your feedback, comments and suggestions to:
edu@postgrespro.ru

Agenda

Multiversion concurrency control (MVCC)

Data snapshot

Isolation levels

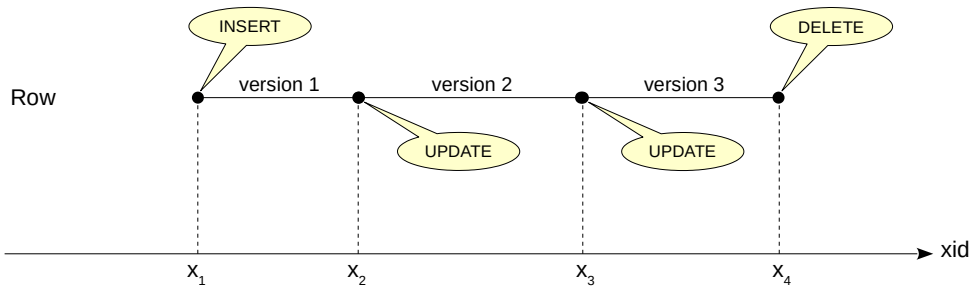
Locking

MVCC

Storing multiple versions of the same row

versions have different time frames

timestamp = transaction ID (IDs are given in ascending order)



3

When multiple sessions are running at the same time, two transactions may access the same row at the same time. If both transactions are just reading the row, there is no problem. If both transactions try to write, no problem either (in this case, they line up and make the changes one after the other). The tricky part is when one transaction wants to read a row and another one wants to change it at the same time.

There are two simple ways about it. You can let such transactions block each other, but then performance suffers. Otherwise, you can let the reading transaction immediately see the changes made by the writing transaction, even if they are not committed (this is called a "dirty read"). This is dangerous, because the changes can be rolled back.

PostgreSQL goes the hard way and utilizes what is known as *Multiversion concurrency control*. In essence, the system stores multiple versions of each row. This allows transactions to see different versions of the same row, while only one transaction can modify a version at a time.

To distinguish between the versions, PostgreSQL marks each one with two timestamps, which together specify a version's "time frame".

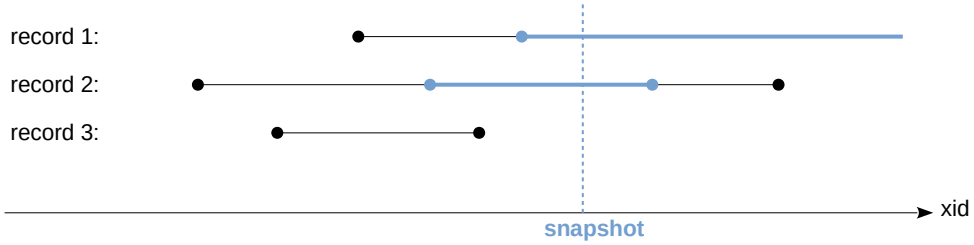
The timestamps are essentially just transaction IDs, which always come in ascending order. (In reality, the whole thing is a bit more complicated, but not worth getting into right now.) Upon creation, a row version is marked with the ID if the transaction that executed the INSERT command. When deleted, the version is marked with the the ID if the transaction that did the DELETE command (but is not physically deleted). An UPDATE command is a DELETE and an INSERT executed back to back.

https://postgrespro.com/docs/postgresql/15/mvcc-intro

Data snapshot

A representation of database data in a consistent state at a specific point in time

- a transaction ID defines a point in time
- a list of active transactions helps the system to exclude changes that have not been committed



4

PostgreSQL uses *snapshot-based transaction isolation*.

A transaction accessing a table should see only one of the versions of each record (or none at all). To achieve this, PostgreSQL presents the transaction with a data *snapshot* created at a certain point in time. The snapshot includes the most recent versions of all committed records but does not include any non-committed changes from active transactions. In other words, the snapshot takes the version of each record that corresponds to the moment when the snapshot was created.

A snapshot is not a physical copy of the data, but just a few numbers:

- the number of the last recorded transaction at the time of snapshot (determines an exact moment in time),

- list of active transactions at that point in time.

The list is needed in order to exclude from the snapshot any changes those transactions may have made but not yet committed.

With just these numbers, we can always tell which record version will be visible in the snapshot. Sometimes it will be the current (most recent) version, as with record 1 in the diagram. Sometimes not: record 2 has been deleted after the snapshot has been created (and the change has already been committed), but the transaction still continues to see it while working with the snapshot. This is the correct behavior, the snapshot gives a consistent representation of data at the selected point in time.

Some records will not get into the snapshot at all: record 3 was deleted before the snapshot was made, so it is not included.

Isolation levels

Read Uncommitted

not supported by PostgreSQL: works as Read Committed

Read Committed *(default)*

the snapshot is created as of the time a statement starts
the same queries may receive different data each time

Repeatable Read

the snapshot is created as of the time the first transaction statement starts
a transaction may fail with a serialization error

Serializable

total isolation, but additional overhead
a transaction may fail with a serialization error

The SQL standard defines four isolation levels: the stricter the level, the less concurrent transactions affect 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 views have changed somewhat).

The most lax level of **Read Uncommitted** allows dirty reads. It is not supported by PostgreSQL, because it is of no practical value and does not give a performance gain.

The **Read Committed** level is the default isolation level in PostgreSQL. At this level, data snapshots are created at the beginning of each SQL statement execution. Thus, the statement works with an unchanged and consistent data snapshot, but two identical queries following one after the other may show different data.

At the **Repeatable Read** level, the snapshot is built at the beginning of a transaction (when executing the first statement). This makes all queries inside the same transaction see the same data. This level is convenient, for example, for generating reports from several queries.

The **Serializable** level guarantees total isolation. At this level, you can safely issue any statements as if the transaction is running alone. The drawback is that some transactions will fail, and your application must be able to repeat such transactions.

https://postgrespro.com/docs/postgresql/15/transaction-iso

Row version visibility

Let's see how the same row can have multiple versions of itself.

Create a table:

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

```
CREATE TABLE
```

Insert a row. Note that if you don't preface your command with the keyword BEGIN, then psql will execute the command immediately and return the output:

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

```
INSERT 0 1
```

Let's begin a transaction and select its ID:

```
=> BEGIN;
```

```
BEGIN
```

```
=> SELECT pg_current_xact_id();
```

```
 pg_current_xact_id
--------------------
                741
(1 row)
```

The transaction sees the first (the only, for now) row version:

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

```
      s      | xmin | xmax
-------------+------+------
 Version one |  740 |    0
(1 row)
```

Here we display the transactions which determine the row version's visibility. The row was created by the last transaction, and xmax=0 means that this is the up-to-date version.

Let's open another session and begin a new transaction:

```
=> BEGIN;
```

```
BEGIN
```

```
=> SELECT pg_current_xact_id();
```

```
 pg_current_xact_id
--------------------
                742
(1 row)
```

The transaction sees the only existing version:

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

```
      s      | xmin | xmax
-------------+------+------
 Version one |  740 |    0
(1 row)
```

But when the second transaction modifies the row...

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

```
UPDATE 1
```

We get this:

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

```
       s      | xmin | xmax
-------------+------+------
 Version two |  742 |    0
(1 row)
```

And this is what the first transaction sees:

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

```
      s      | xmin | xmax
-------------+------+------
 Version one |  740 |  742
(1 row)
```

Because the change hasn't been committed yet, the first transaction still sees the original version.

Note the xmax value: it shows that another transaction is currently modifying the row. Strictly speaking, this "peeking" breaks isolation. That's why the xmin and xmax fields are hidden and should not be used in production.

Now, let's commit the changes.

```
=> COMMIT;
```

```
COMMIT
```

This changes what the first transaction sees:

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

```
      s      | xmin | xmax
-------------+------+------
 Version two |  742 |    0
(1 row)
```

Both transactions see the same row version, i.e. version two.

After committing, version one is no longer visible to either transaction.

```
=> COMMIT;
```

```
COMMIT
```

Locking

Row locks

- reading never locks rows
- changing rows locks them for changes, but not for reads

Table locks

- prohibit changing or deleting a table while it is being worked on
- prohibit reading the table when rebuilding or moving
- etc.

Lock lifetime

- acquired as needed or manually
- released automatically upon transaction completion

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

The main locks are set at the row level. Reading never blocks either reading or writing transactions. Changing a row does not lock it for reading. The only case when a transaction will wait for the lock to be released is if it tries to change a row that has already been changed by another transaction that has not been committed yet.

Locks can also be set at a higher level, particularly on tables. They are needed so that no one can delete the table while other transactions are reading data from it, or to prohibit access to the table being rebuilt. Such locks generally do not cause problems, since deleting or rebuilding tables is only done once in a while.

All necessary locks are set automatically and automatically removed at the end of the transaction. You can also set additional custom locks, but this is rarely necessary.

https://postgrespro.com/docs/postgresql/15/explicit-locking

Locks

Let’s try this again, but now have both transactions attempt to change the value.

```
=> BEGIN;
```

```
BEGIN
```

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

```
UPDATE 1
```

And in the other transaction:

```
=> BEGIN;
```

```
BEGIN
```

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

The second transaction hangs up. It can’t change the row value until the first transaction releases the lock.

```
=> COMMIT;
```

```
COMMIT
```

Now, the second transaction can do its thing:

```
UPDATE 1
```

```
=> COMMIT;
```

```
COMMIT
```

Transaction status

Transaction status (clog)

- service information; two bits per transaction
- stored in files on disk
- cached in shared memory

Commit

- the “transaction committed” bit is set

Termination

- the “transaction aborted” bit is set
- performed as fast as commit (no data rollback needed)

For multiversion concurrency control to work, the server needs to understand the status of transactions. A transaction can be active or finished. A transaction can end either in a commit or in an abort. Therefore, only two bits are required to represent the state of each transaction.

Transaction statuses (commit log, clog) are stored in special service files in the PGDATA/pg_xact directory and worked upon in the server's shared memory, as to avoid constantly accessing the disk.

At any transaction completion (either successful or not), it is enough to set the appropriate status bits. Both transaction commit and abort occur equally quickly.

If an aborted transaction managed to create new row versions, these versions are not destroyed (there is no "physical" rollback of data). Thanks to the status information, other transactions will see that the transaction that created or deleted the row versions was actually aborted, and will not take changes made by it into account.

Takeaways

Multiple versions of each row can be stored in data files

Transactions work with a data snapshot, a representation of database data in a consistent state at a specific point in time

Writers don’t block readers, readers don’t block anyone

Isolation levels differ in snapshot creation times

Practice

1. Create a table with one row.

   Begin a transaction at the Read Committed isolation level and query the table.
   In another session, delete the row and commit the changes.
   How many rows will the first transaction see after executing the same query again? Try and see.

   Complete the first transaction.

2. Repeat the same thing, but now let the transaction work at the isolation level Repeatable Read:

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

   Explain the differences.

Task 1. Read Committed isolation level

Create a table:

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

```
CREATE TABLE
```

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

```
INSERT 0 1
```

Make a query from one transaction (Read Committed is the isolation level by default):

```
=> BEGIN;
```

```
BEGIN
```

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

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

Remove the row in another transaction and commit:

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

```
DELETE 1
```

Retry the query:

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

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

The first transaction sees the changes.

```
=> COMMIT;
```

```
COMMIT
```

Task 2. Repeatable Read isolation level

Put the row back in:

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

```
INSERT 0 1
```

Query from the first transaction:

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

```
BEGIN
```

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

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

Remove the row in the second transaction and commit:

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

```
DELETE 1
```

Retry the query:

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

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

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

```
=> COMMIT;
```

```
COMMIT
```