Transaction Management & Concurrency Control – PostgreSQL

Srinivas Maddali

There are multiple components for each transaction effecting each row.

1. Isolation levels of the transaction/s:
   a. Read committed.
   b. Read uncommitted.
   c. Repeatable read
   d. Phantom reads

ANSI/ISO SQL Isolation Levels

Isolation Level	Dirty Read	Non-Repeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

PostgreSQL offers Read Committed transactions.

Table Level Locks implemented by PostgreSQL

1. AccessShareLock – conflicts with AccessExclusiveLock – Read Lock Mode for SELECT queries.
2. RowShareLock - Acquired by LOCK TABLE for IN SHARE ROW EXCLUSIVE MODE statements. Conflicts with RowExclusiveLock, ShareLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock modes.
3. ExclusiveLock - Acquired by LOCK TABLE table for IN EXCLUSIVE MODE statements. Conflicts with RowShareLock, RowExclusiveLock, ShareLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock modes.
4. AccessExclusiveLock - Acquired by ALTER TABLE, DROP TABLE, VACUUM and LOCK TABLE statements. Conflicts with all modes (AccessShareLock, RowShareLock, RowExclusiveLock, ShareLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock).
5. Note: Only AccessExclusiveLock blocks SELECT (without FOR UPDATE) statement.

(source: https://www.postgresql.org/docs/7.1/locking-tables.html)

Row-level locks:

These locks are acquired when rows are being updated (or deleted or marked for update). Row-level locks do not affect data querying. They block writers to the same row only.

PostgreSQL does not remember any information about modified rows in memory and so has no limit to the number of rows locked at one time. However, locking a row may cause a disk write; thus, for example, SELECT FOR UPDATE will modify selected rows to mark them and so will result in disk writes.

In addition to table and row locks, short-term share/exclusive locks are used to control read/write access to table pages in the shared buffer pool. These locks are released immediately after a tuple is fetched or updated. Application writers normally need not be concerned with page-level locks, but we mention them for completeness.

Locking and Indices: (read https://www.postgresql.org/docs/7.1/locking-indices.html)

Though Postgres provides nonblocking read/write access to table data, nonblocking read/write access is not currently offered for every index access method implemented in Postgres.

The various index types are handled as follows:

GiST and R-Tree indices:

Share/exclusive index-level locks are used for read/write access. Locks are released after statement is done.

Hash indices:

Share/exclusive page-level locks are used for read/write access. Locks are released after page is processed.Page-level locks provide better concurrency than index-level ones but are subject to deadlocks.

Btree indices:

Short-term share/exclusive page-level locks are used for read/write access. Locks are released immediately after each index tuple is fetched/inserted. Btree indices provide the highest concurrency without deadlock conditions.

In short, btree indices are the recommended index type for concurrent applications.

DATA READ CONSISTENCY and PostgreSQL

Because readers in Postgres do not lock data, regardless of transaction isolation level, data read by one transaction can be overwritten by another concurrent transaction. In other words, if a row is returned by SELECT it does not mean that the row still exists at the time it is returned (i.e., sometime after the current transaction began); the row might have been modified or deleted by an already-committed transaction that committed after this one started. Even if the row is still valid "now", it could be changed or deleted before the current transaction does a commit or rollback.

Another way to think about it is that each transaction sees a snapshot of the database contents, and concurrently executing transactions may very well see different snapshots. So, the whole concept of "now" is somewhat suspect anyway. This is not normally a big problem if the client applications are isolated from each other, but if the clients can communicate via channels outside the database then serious confusion may ensue.

To ensure the current existence of a row and protect it against concurrent updates one must use SELECT FOR UPDATE or an appropriate LOCK TABLE statement. (SELECT FOR UPDATE locks just the returned rows against concurrent updates, while LOCK TABLE protects the whole table.) This should be considered when porting applications to Postgres from other environments.

Multi Version Concurrency Control (MVCC):

Transaction IDs (like SCN numbers in Oracle) maximum for each database in the cluster is 2**31 (2,147,483,648).

MaxTupleAttributeNumber limits the number of (user) columns in a tuple. The key limit on this value is that the size of the fixed overhead fora tuple, plus the size of the null-values bitmap (at 1 bit per column),plus MAXALIGN alignment, must fit into t_hoff which is uint8. On most machines the upper limit without making t_hoff wider would be a little over 1700. We use round numbers here and for MaxHeapAttributeNumber so that alterations in HeapTupleHeaderData layout will not change the supported max number of columns.

Transaction IDs in PostgreSQL in relation to space consumed are to fall under:

a. Visible (potentially for active transactions - compared to Inactive In Use (IIU) Transactions in Oracle RDBMS and Modified data is flushed only after the data is read by queried) and
b. Invisible

Visible are of the past and invisible are future. When a transaction is committed then the past transactions are dead transactions. But reader who started reading the data before the DML is committed need the dead transactions and hence the dead transactions are required to be stored in the database.

Free Space Mapping (FSM) – Exclusively implemented in PostgreSQL:

While inserting a heap or an index tuple, PostgreSQL uses the FSM of the corresponding table or index to SELECT the page which can be inserted it. All tables and indexes have respective FSMs. Each FSM stores the information about the free space capacity of each page within the corresponding table or index file (we know that separate files are created for each table and index created on the table).

INSERT commands can overwrite the pages or blocks that are marked FREE. This is comparable to free space list in Oracle.

DELETE commands can mark the pages/blocks with FSM. Dead tuples/rows should eventually be removed from pages. Cleaning dead tuples is referred to as VACUUM processing.

UPDATE = DELETE + INSERT.

Memory Structure management in oracle is based on Least Recently Used (LRU). Most Recently Used (MRU) is on the top of the page/block listing and LRU is at the bottom.

Vacuuming (cleaning of the database tables and releasing the space in PostgreSQL):

Current Vacuuming process includes:

1. Visibility Mapping
2. Freezing Processing of the transaction IDs
3. Removing unnecessary clog files

Vacuuming Process has daemon: source - https://www.postgresql.org/docs/current/routine-vacuuming.html

PostgreSQL databases require periodic maintenance known as vacuuming. For many installations, it is sufficient to let vacuuming be performed by the autovacuum daemon. You might need to adjust the autovacuuming parameters described there to obtain best results for your situation. Some database administrators will want to supplement or replace the daemon's activities with manually managed VACUUM commands, which typically are executed according to a schedule by cron or Task Scheduler scripts. To set up manually managed vacuuming properly, it is essential to understand the issues discussed in the next few subsections. Administrators who rely on autovacuuming may still wish to skim this material to help them understand and adjust autovacuuming.

Reasons for vacuuming:

1. To recover or reuse disk space occupied by updated or deleted rows.
2. To update data statistics used by the PostgreSQL query planner.
3. To update the visibility map, which speeds up index-only scans.
4. To protect against loss of very old data due to transaction ID wraparound or multixact ID wraparound.

Variants of Vacuuming:

There are two variants of VACUUM:
1. standard VACUUM and VACUUM FULL. VACUUM FULL can reclaim more disk space but runs much more slowly. Also, the standard form of VACUUM can run in parallel with production database operations. (Commands such as SELECT, INSERT, UPDATE, and DELETE will continue to function normally, though you will not be able to modify the definition of a table with commands such as ALTER TABLE while it is being vacuumed.)
2. VACUUM FULL requires exclusive lock on the table it is working on, and therefore cannot be done in parallel with other use of the table. Generally, therefore, administrators should strive to use standard VACUUM and avoid VACUUM FULL

VACUUM creates a substantial amount of I/O traffic, which can cause poor performance for other active sessions. There are configuration parameters that can be adjusted to reduce the performance impact of background vacuuming.

Recovering Disk Space:

In PostgreSQL, an UPDATE or DELETE of a row does not immediately remove the old version of the row. This approach is necessary to gain the benefits of multi version concurrency control (MVCC): the row version must not be deleted while it is still potentially visible to other transactions.

But eventually, an outdated or deleted row version is no longer of interest to any transaction. The space it occupies must then be reclaimed for reuse by new rows, to avoid unbounded growth of disk space requirements. This is done by running VACUUM.

The standard form of VACUUM removes dead row versions in tables and indexes and marks the space available for future reuse. However, it will not return the space to the operating system, except in the special case where one or more pages at the end of a table become entirely free and an exclusive table lock can be easily obtained. In contrast, VACUUM FULL actively compacts tables by writing a completely new version of the table file with no dead space. This minimizes the size of the table but can take a long time. It also requires extra disk space for the new copy of the table, until the operation completes.

Like Oracle RDBMS concept of Inactive In Use (IIU) transactions, in PostgreSQL used by potentially active transactions may use the dead transaction IDs for serving the user requests.

PostgreSQL's MVCC transaction semantics depend on being able to compare transaction ID (XID) numbers: a row version with an insertion XID greater than the current transaction's XID is "in the future" and should not be visible to the current transaction. But since transaction IDs have limited size (32 bits) a cluster that runs for a long time (more than 4 billion transactions) would suffer transaction ID wraparound: the XID counter wraps around to zero, and all sudden transactions that were in the past appear to be in the future — which means their output become invisible. In short, catastrophic data loss. (Actually, the data is still there, but that's cold comfort if you cannot get at it.) To avoid this, it is necessary to vacuum every table in every database at least once every two billion transactions.

The reason that periodic vacuuming solves the problem is that VACUUM will mark rows as frozen, indicating that they were inserted by a transaction that committed sufficiently far in the past that the effects of the inserting transaction are certain to be visible to all current and future transactions. Normal XIDs are compared using modulo-2**32 arithmetic. This means that for every normal XID, there are two billion XIDs that are "older" and two billion that are "newer"; another way to say it is that the normal XID space is circular with no endpoint. Therefore, once a row version has been created with a particular normal XID, the row version will appear to be "in the past" for the next two billion transactions, no matter which normal XID we are talking about. If the row version still exists after more than two billion transactions, it will suddenly appear to be in the future. To prevent this, PostgreSQL reserves a special XID, FrozenTransactionId, which does not follow the normal XID comparison rules and is always considered older than every normal XID. Frozen row versions are treated as if the inserting XID were FrozenTransactionId, so that they will appear to be "in the past" to all normal transactions regardless of wraparound issues, and so such row versions will be valid until deleted, no matter how long that is.

Note:

In PostgreSQL versions before 9.4, freezing was implemented by actually replacing a row's insertion XID with FrozenTransactionId, which was visible in the row's xmin system column. Newer versions just set a flag bit, preserving the row's original xmin for possible forensic use. However, rows with xmin equal to FrozenTransactionId (2) may still be found in databases pg_upgrade'd from pre-9.4 versions.

Also, system catalogs may contain rows with xmin equal to BootstrapTransactionId (1), indicating that they were inserted during the first phase of initdb. Like FrozenTransactionId, this special XID is treated as older than every normal XID.

Collection of statistics for the query execution planner:

The PostgreSQL query planner relies on statistical information about the contents of tables to generate good plans for queries. These statistics are gathered by the ANALYZE command, which can be invoked by itself or as an optional step in VACUUM. It is important to have reasonably accurate statistics, otherwise poor choices of plans might degrade database performance.

The autovacuum daemon, if enabled, will automatically issue ANALYZE commands whenever the content of a table has changed sufficiently. However, administrators might prefer to rely on manually scheduled ANALYZE operations, particularly if it is known that update activity on a table will not affect the statistics of "interesting" columns. The daemon schedules ANALYZE strictly as a function of the number of rows inserted or updated; it has no knowledge of whether that will lead to meaningful statistical changes.

Vacuuming is for recovering space. As with vacuuming for space recovery, frequent updates of statistics are more useful for heavily updated tables than for seldom-updated ones. But even for a heavily updated table, there might be no need for statistics updates if the statistical distribution of the data is not changing much. A simple rule of thumb is to think about how much the minimum and maximum values of the columns in the table change.

For example:

A timestamp column that contains the time of row update will have a constantly increasing maximum value as rows are added and updated; such a column will probably need more frequent statistics updates than, say, a column containing URLs for pages accessed on a website. The URL column might receive changes just as often, but the statistical distribution of its values probably changes relatively slowly.

It is possible to run ANALYZE on specific tables and even just specific columns of a table, so the flexibility exists to update some statistics more frequently than others if your application requires it. In practice, however, it is usually best to just analyze the entire database, because it is a fast operation. ANALYZE uses a statistically random sampling of the rows of a table rather than reading every single row.

Good to Know:

The autovacuum daemon does not issue ANALYZE commands for foreign tables, since it has no means of determining how often that might be useful. If your queries require statistics on foreign tables for proper planning, it's a good idea to run manually managed ANALYZE commands on those tables on a suitable schedule.

Updating the Visibility Map:

Vacuum maintains a visibility map for each table to keep track of which pages contain only tuples that are known to be visible to all active transactions (and all future transactions, until the page is again modified). This has two purposes.

1. First, vacuum itself can skip such pages on the next run, since there is nothing to clean up.
2. Second, it allows PostgreSQL to answer some queries using only the index, without reference to the underlying table.

Since PostgreSQL indexes do not contain tuple visibility information, a normal index scan fetches the heap tuple for each matching index entry, to check whether it should be seen by the current transaction. An index-only scan, on the other hand, checks the visibility map first. If it is known that all tuples on the page are visible, the heap fetch can be skipped. This is most useful on large data sets where the visibility map can prevent disk accesses. The visibility map is vastly smaller than the heap, so it can easily be cached even when the heap is exceptionally large.

Oracle IIU Transactions or ITL is determined by marking a row “in use” by a transaction, the transaction fills its data into an unused ITL slot and sets the row’s lock byte to that entry’s index. A lock byte value of zero indicates that no ITL entry applies to this row. Like wise in PostgreSQL the visibility map concepts work and determine which transaction is dead and do not need to be in the visibility map.

Preventing Transaction ID Wraparound Failures:

PostgreSQL's MVCC transaction semantics depend on being able to compare transaction ID (XID) numbers: a row version with an insertion XID greater than the current transaction's XID is “in the future” and should not be visible to the current transaction. But since transaction IDs have limited size 2**32 = 4,294,967,296 (32 bits) a cluster that runs for a long time (more than 4 billion transactions) would suffer transaction ID wraparound: the XID counter wraps around to zero, and all of a sudden transactions that were in the past appear to be in the future — which means their output become invisible. In short, catastrophic data loss. (Actually the data is still there, but that's cold comfort if you cannot get at it.) To avoid this, it is necessary to vacuum every table in every database at least once every two billion transactions.