Lecture #05: Multi-Version Concurrency Control (MVCC) – Part 1

15-721 Advanced Database Systems (Spring 2018)
http://15721.courses.cs.cmu.edu/spring2018/
Carnegie Mellon University
Prof. Andy Pavlo

Compare-and-Swap

- Atomic instruction that compares contents of a memory location M to a given value V. If the values are equal, installs new given value V' in M. Otherwise, operation fails.
- Can be done with an **intrinsic** function call, a function that a compiler knows to convert to an instruction without having to write the machine code.

Isolation Levels

- Serializability is useful because it allows programmers to ignore concurrency issues but enforcing it may allow too little parallelism and limit performance.
- We may want to use a weaker level of consistency to improve scalability.
- Isolation levels control the extent that a transaction is exposed to the actions of other concurrent transactions.
- Anomalies:
  - **Dirty Read:** Reading uncommitted data.
  - **Unrepeatable Reads:** Redoing a read results in a different result.
  - **Phantom Reads:** Insertion or deletions result in different results for the same range scan queries.
- Isolation levels (strongest to weakest)
  1. `SERIALIZABLE`: No Phantoms, all reads repeatable, and no dirty reads.
  2. `REPEATABLE READS`: Phantoms may happen.
  3. `READ-COMMITTED`: Phantoms and unrepeatable reads may happen.
  4. `READ-UNCOMMITTED`: All anomalies may happen.
- The isolation levels defined as part of SQL-92 standard only focused on anomalies that can occur in a 2PL-based DBMS [1]. There are two additional isolation levels:
  1. `CURSOR STABILITY`
     - Between repeatable reads and read committed
     - Prevents Lost Update Anomaly.
     - Default isolation level in **IBM DB2**.
  2. `SNAPSHOT ISOLATION`
     - Guarantees that all reads made in a transaction see a consistent snapshot of the database that existed at the time the transaction started.
     - A transaction will commit only if its writes do not conflict with any concurrent updates made since that snapshot.
     - Susceptible to write skew anomaly.

Multi-version Concurrency Control (MVCC)

Originally proposed in 1978 MIT dissertation [2].

MVCC is currently the best approach for supporting transactions in mixed workloads. The DBMS maintains multiple **physical** versions of an object of a single **logical** object in the database. When a transaction writes to an object, the DBMS, creates a new version of that object. When a transaction reads an object, it reads the newest version that existed when a transaction started.

Main Benefits

- Writes do not block readers.
- Read-only transactions can read a consistent snapshot without acquiring locks.
- Easily support time-travel queries.

MVCC is more than just a "concurrency control protocol". It completely affects how the DBMS manages transactions and the database. There are four key design decisions: [3]

- Concurrency Control Protocol
- Version Storage
- Garbage Collection
- Index Management
- Transaction Id Wraparound

Concurrency Control Protocol

The DBMS is able to use all of the same concurrency control protocols under MVCC.

Timestamp Ordering (MV-TO):

- Use a `read-ts` field in the header to keep track of the timestamp of the last transaction that read it.
- Transaction is allowed to read version if the lock is unset and its T_id is between `begin-ts` and `end-ts`.
- For writes, transaction creates a new version if no other transaction holds lock and T_id is greater than `read-ts`.

Version Storage

The DBMS uses the tuple's pointer field to create a latch-free **version chain** per logical tuple. This allows the DBMS to find the version that is visible to a particular transaction at runtime. Indexes always point to "head" of the chain.

Thread store versions in "local" memory regions to avoid contention on centralized data structures. Different storage schemes determine where/what to store for each version.

For non-inline attributes, the DBMS can reuse pointers to variable-length pool for values that do not change between versions. This requires reference counters to know when it is safe to free memory. This optimization also makes it more difficult to relocate memory from the variable-length pool.

Append-Only Storage

- All of the physical versions of a logical tuple are stored in the same table space (table heap).
- On update, append new tuple to same table heap in an empty slot.

- **Oldest-to-Newest (O2N):** Append new version to end of chain, traverse entire chain on lookup.
- **Newest-to-Oldest (N2O):** Have to update index pointers for every new version, but do not have to traverse chain on look ups.

Time-Travel Storage

- On every update, copy current version to the time-travel table, and update pointer.
- Overwrite master version in main table, and update pointer.

Delta Storage

- On every update, copy only the values that were modified into the delta storage and overwrite the master version.
- Transaction can recreate old versions by applying the delta in reverse order.

Garbage Collection

The DBMS needs to remove **reclaimable** physical versions from the database over time. A version is reclaimable if (1) no active transaction in the DBMS can see that version or (2) the version was created by an aborted transaction.

Tuple Level

- Find old versions by examining tuples directly.
- **Background Vacuuming:** Separate threads periodically scan the table and look for reclaimable versions. Works with any version storage technique.
- **Cooperative Cleaning:** Worker threads identify reclaimable versions as they traverse version change. Only works with O2N version chains.

Transaction Level

- Transactions keep track of their old version so the DBMS does not have to scan tuples to determine visibility.
- The DBMS determines when all versions created by a finishing transaction are no longer visible.
- May still maintain multiple threads to reclaim the memory fast enough for the workload.

Index Management

How often the DBMS updates index depends on whether system creates new versions when a tuple is updated.

Primary Key

Primary key indexes always point to the version chain head. If a transaction updates a primary key attribute(s), then this is treated as a `DELETE` followed by an `INSERT`.

Secondary Indexes

Approach #1: Logical Pointer

- Use a fixed identifier per tuple that **does not change**
- Requires an extra indirection layer
- Secondary indexes can store Primary Key or Tuple ID

Approach #2: Physical Address

- Pointer physical points to tuple

Transaction Id Wraparound

If the DBMS reaches the max value for its timestamps, it will have to wrap around and start at zero. This will make all previous versions be in the "future" from new transactions.

Postgres `Txn-ID` Wraparound

- Stop accepting new commands when the system gets close to the max transaction id.
- Set a flag in each tuple header that says that it is "frozen" in the past. Any new transaction id will always be newer than a frozen version.
- Runs the vacuum before the system gets close to this upper limit.

References

[1] H. Berenson, P. Bernstein, J. Gray, M. Jim, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *SIGMOD '95 Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 1–10. URL https://dl.acm.org/citation.cfm?id=223785.

[2] D. P. Reed. Naming and Synchronization in a Decentralized Computer System. *Ph.D. dissertation*, 1978.

[3] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. In *VLDB '17: Proceedings of the VLDB Endowment*, volume 10, pages 781–792, March 2017. URL https://dl.acm.org/citation.cfm?id=3067427.