Lecture #03: Multi-Version Concurrency Control (Design Decisions)

15-721 Advanced Database Systems (Spring 2020)
https://15721.courses.cs.cmu.edu/spring2020/
Carnegie Mellon University
Prof. Andy Pavlo

1 Multi-version Concurrency Control (MVCC)

Originally proposed in 1978 MIT dissertation by Reed [1].

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

- Writers do not block readers, unlike 2PL which acquires exclusive writer locks.
- Read-only transactions read a consistent snapshot without acquiring locks and txn ids (e.g., **MySQL**).
- 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: [2]

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

2 Concurrency Control Protocol

A MVCC DBMS uses all of the same concurrency control protocols for single-versioned systems. They are slightly modified to support in-memory databases and multi-versioning, but the high-level concepts are the same.

Tuple Metadata

Disk-oriented DBMSs store metadata like locks in data structures that are separate from the tuples. For an in-memory DBMS, the DBMS will store this metadata within the tuple's header. This removes the need for the DBMS's concurrency control implementation to read separate memory locations to retrieve information about the tuple.

This metadata includes the following 64-bit fields:

- **`Txn-Id`:** Unique Transaction Identifier. This is usually a timestamp.

- **`Begin-TS / End-TS:`** A start/end range that specifies that version's lifetime. The DBMS uses this to determine for a transaction whether this particular physical version of the tuple exists in its consistent snapshot.
- **`Pointer:`** The memory address for the next/previous version in the tuple's *version chain*. The version chain is a singly linked-list of the physical versions for a logical object.

Timestamp Ordering (MV-TO):

The DBMS adds an additional `Read-TS` field in the tuple header to keep track of the timestamp of the last transaction that read it.

For reads, a transaction is allowed to read version if the lock is unset and its transaction id (T_id) is between `Begin-TS` and `End-TS`. Latches are not required for read operations.

For writes, a transaction creates a new version if no other transaction holds lock and T_id is greater than `Read-TS`. The write operation performs CAS on the `Txn-Id` field that provides a latch on this data tuple. After creating a new version, it updates the `End-TS` field with transaction timestamp.

Two-Phase Locking (MV-2PL)

The DBMS adds an additional `Read-Cnt` field to each tuple's header that acts as a shared lock. This is a 64-bit counter that tracks the number of transactions currently holding this lock. For reads, a transaction is allowed to hold the share lock if `Txn-Id` is zero. It then performs a CAS on `Read-Cnt` to increment the counter by one.

For writes, a transaction is allowed to hold the exclusive lock if both `Txn-Id` and `Read-Cnt` are zero. The DBMS uses `Txn-Id` and `Read-Cnt` together as exclusive lock. On commit, `Read-Cnt` and `Txn-Id` are reset to zero.

This design is good for deadlock prevention, but needs extra global data structures for deadlock detection.

3 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 and update pointer.

There are two ways of connecting pointers:

- **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

Instead of storing all the tuples versions in a single table, the DBMS splits a single logical table into two sub-tables: (1) *main table* and (2) *time-travel table*. The main table keeps the latest version of tuples. When a transaction updates a tuple, the DBMS copies current version from the main table to the time-travel table. It then overwrites the master version in main table and updates its pointer to the recently copied entry in the time-travel table.

Garbage collection is fast with this approach since the DBMS can just drop entries from the time-travel entry without scanning the main table. Sequential scans are faster easy since the DBMS can just scan the main table without checking version information.

Delta Storage

With this approach, the main table keeps the latest version of tuples. On every update, the system copies only the values that were modified into a *delta storage* area and overwrite the master version. Transactions recreate old versions by applying the delta in reverse order.

Garbage collection is fast with this approach because the system just has to drop entries from the delta storage. It is also has lower storage overhead than the other two approaches because the system does not have to copy an entire tuple

Non-inline Attributes

Variable-length data can be stored in separate space and be referenced by a pointer in the data tuple. Direct duplication of these data is wasteful, so the DBMS can reuse pointers to variable-length pool for values that do not change between versions. One option is to use reference counters to know when it is safe to free from memory. As a result the DBMS would not be able to relocate memory easily, thus no existing system implements this optimization.

4 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

With this approach, transactions do not maintain additional meta-data about old versions. Thus, the DBMS has to scan tables to find old versions.

- **Background Vacuuming:** Separate threads periodically scan the table and look for reclaimable versions. Works with any version storage technique. To avoid repeatedly scanning through unmodified data, the DBMS can use a *dirty block bitmap* to keep track of what blocks of data have been modified since the last scan.
- **Cooperative Cleaning:** Worker threads identify reclaimable versions as they traverse version change. Only works with O2N version chains. A problem with this is that if there are never any queries that access tuples with reclaimable versions, then these versions will not get cleaned up (i.e., "dusty corners"). Thus, the DBMS still has to periodically scan the table to find old versions.

Transaction Level

With this approach, 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.

5 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

Managing secondary indexes is more complicated than primary key indexes. There are two approaches to storing values that represent the location of a tuple's version chain.

Approach #1: Physical Address

- Use physical address to the version chain head.
- If a databases has many secondary indexes, then updates can become expensive because the DBMS has update all the indexes to the new location (e.g., each update to a N2O version chain requires the DBMS to update every index with the memory address of the new version chain head).

Approach #2: Logical Pointer

- **Primary Key:** Store the tuple's primary key as the value for a secondary index, which will then redirect to the physical address. This approach has high store overhead if the size of the primary key is large.
- **Tuple ID:** Use a fixed identifier per tuple that does not change. This approach requires an extra indirection layer to map the id to a physical address. For example, this could be a hash table that maps Tuple IDs to physical addresses.

References

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

[2] 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`.