Lecture #18: Multi-Version Concurrency Control

15-445/645 Database Systems (Spring 2023)
`https://15445.courses.cs.cmu.edu/spring2023/`
Carnegie Mellon University
Charlie Garrod

1 Isolation Levels

This section is included in the lecture note of lecture 17.

2 Multi-Version Concurrency Control

Multi-Version Concurrency Control (MVCC) is a larger concept than just a concurrency control protocol. It involves all aspects of the DBMS's design and implementation. MVCC is the most widely used scheme in DBMSs. It is now used in almost every new DBMS implemented in last 10 years. Even some systems (e.g., NoSQL) that do not support multi-statement transactions use it.

With MVCC, the DBMS maintains multiple physical versions 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 the transaction started.

The fundamental concept/benefit of MVCC is that writers do not block writers and readers do not block readers. This means that one transaction can modify an object while other transactions read old versions.

One advantage of using MVCC is that read-only transactions can read a consistent **snapshot** of the database without using locks of any kind. Additionally, multi-versioned DBMSs can easily support *time-travel queries*, which are queries based on the state of the database at some other point in time (e.g. performing a query on the database as it was 3 hours ago).

A typical MVCC-based database design will:

1. There is a versioned storage which stores different versions of the same logical object.
2. When a transaction begins, the DBMS takes a snapshot of the database (by copying the transaction status table).
3. The DBMS uses the snapshot to determine which versions of objects are visible to the transaction.

There are four important MVCC design decisions:

1. Concurrency Control Protocol
2. Version Storage
3. Garbage Collection
4. Index Management

The choice of concurrency protocol is between the approaches discussed in previous lectures (two-phase locking, timestamp ordering, optimistic concurrency control).

Snapshot Isolation

Snapshot Isolation involves providing a transaction with a consistent snapshot of the database when the transaction started. Data values from a snapshot consist of only values from committed transactions, and the transaction operates in complete isolation from other transactions until it finishes. This is idea for read-only transactions since they do not need to wait for writes from other transactions. Writes are maintained in a transaction's private workspace or written to the storage with transaction metadata and only become visible to the database once the transaction successfully commits.

Write Conflicts If two transactions update the same object, the first writer wins.

Write Skew Anomaly can occur in Snapshot Isolation when two concurrent transactions modify different objects resulting in non-serializable schedules. For example, if one transaction changes all white marbles to black and the other changes all black marbles to white, the outcome may not correspond to any serializable schedule.

3 Version Storage

This how the DBMS will store the different physical versions of a logical object and how transactions find the newest version visible to them.

The DBMS uses the tuple's pointer field to create a **version chain** per logical tuple, which is essentially a linked list of versions sorted by timestamp. This allows the DBMS to find the version that is visible to a particular transaction at runtime. Indexes always point to the "head" of the chain, which is either the newest or oldest version depending on implementation. A thread traverses chain until it finds the correct version. Different storage schemes determine where/what to store for each version.

Approach #1: Append-Only Storage

All physical versions of a logical tuple are stored in the same table space. Versions are mixed together in the table and each update just appends a new version of the tuple into the table and updates the version chain. The chain can either be sorted *oldest-to-newest* (O2N) which requires chain traversal on look-ups, or *newest-to-oldest* (N2O), which requires updating index pointers for every new version.

Approach #2: Time-Travel Storage

The DBMS maintains a separate table called the time-travel table which stores older versions of tuples. On every update, the DBMS copies the old version of the tuple to the time-travel table and overwrites the tuple in the main table with the new data. Pointers of tuples in the main table point to past versions in the time-travel table.

Approach #3: Delta Storage

Like time-travel storage, but instead of the entire past tuples, the DBMS only stores the deltas, or changes between tuples in what is known as the delta storage segment. Transactions can then recreate older versions by iterating through the deltas. This results in faster writes than time-travel storage but slower reads.

4 Garbage Collection

The DBMS needs to remove *reclaimable* physical versions from the database over time. A version is reclaimable if no active transaction can "see" that version or if it was created by a transaction that was aborted.

Approach #1: Tuple-level GC

With tuple-level garbage collection, the DBMS finds old versions by examining tuples directly. There are two approaches to achieve this:

- **Background Vacuuming**: Separate threads periodically scan the table and look for reclaimable versions. This works with any version storage scheme. A simple optimization is to maintain a "dirty page bitmap," which keeps track of which pages have been modified since the last scan. This allows the threads to skip pages which have not changed.
- **Cooperative Cleaning**: Worker threads identify reclaimable versions as they traverse version chain. This only works with O2N chains. The data will never be cleaned if it is not accessed.

Approach #2: Transaction-level GC

Under transaction-level garbage collection, each transaction is responsible for keeping track of their own old versions so the DBMS does not have to scan tuples. Each transaction maintains its own read/write set. When a transaction completes, the garbage collector can use that to identify which tuples to reclaim. The DBMS determines when all versions created by a finished transaction are no longer visible.

5 Index Management

All primary key (pkey) indexes always point to version chain head. How often the DBMS has to update the pkey index depends on whether the system creates new versions when a tuple is updated. If a transaction updates a pkey attribute(s), then this is treated as a `DELETE` followed by an `INSERT`.

Managing secondary indexes is more complicated. There are two approaches to handling them.

Approach #1: Logical Pointers

The DBMS uses a fixed identifier per tuple that does not change. This requires an extra indirection layer that maps the logical id to the physical location of the tuple. Then, updates to tuples can just update the mapping in the indirection layer.

Approach #2: Physical Pointers

The DBMS uses the physical address to the version chain head. This requires updating every index when the version chain head is updated.