Lecture #19: Multi-Version Concurrency Control

15-445/645 Database Systems (Fall 2024)
https://15445.courses.cs.cmu.edu/fall2024/
Carnegie Mellon University
Andy Pavlo

1 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 readers and readers do not block writers. This means that one transaction can modify an object while other transactions read old versions. Writers may still block other writers if they are writing the same object, since there is still a lock on the versions related to the database object. See slide 8 for an example.

One advantage of using MVCC is that read-only transactions can read a consistent **snapshot** of the database without using locks of any kind, and it naturally supports Snapshot Isolation (SI). In MVCC, timestamps are used to determine visibility of versions to transactions. Additionally, MVCC without garbage collection allows the DBMS to 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. Have a versioned storage which stores different versions of the same logical object. (*Note: Do not do this!*)
2. Takes a snapshot of the database (by copying the transaction status table) when a transaction starts.
3. Use the snapshot to determine which versions of objects are visible to the transaction.

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 ideal 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. See slides 28 to 31 for details.

There are five important MVCC design considerations:

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

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

2 Design consideration: Version Storage

This is 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 (i.e. no need to traverse the chain. This is typically the better approach because most txns only care about the newest version). See slides 37 to 40 for an example.

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. See slides 42 to 46 for an example.

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 in reverse order and applying them. This results in faster writes than time-travel storage but slower reads. See slides 47 to 51 for an example.

3 Design consideration: 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. We can either perform tuple level, or transaction level garbage collection.

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. See slides 54 to 60 for an example.
- **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.
  正常扫描多版本数据的同时，执行清理旧数据的操作。

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. See slides 66 to 73 for an example.

4 Design consideration: 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, which can be very expensive.

Imaging a table with a primary index, and multiple secondary indices, using physical pointers requires each secondary index to point to the tuple version chain's physical address, and each update to the tuple requires and update to all the secondary indices. With a logical pointer approach, the secondary indices point to the location in the primary index corresponding to the tuple, thus requiring less overhead when the tuple is updated.

MVCC duplicate key problem

MVCC DBMS indexes (usually) do not store version information about tuples with their keys. Instead, every index must support duplicate keys from different snapshots, since the same key may point to different logical tuples in different snapshots.

MVCC Duplicate Key Problem describes this need to support duplicate keys in MVCC DBMS indexes when multiple transactions need multiple versions of the same logical tuple. For example, say TXN 1 points to version 0 of a logical tuple, and TXN 2 creates a version 1 of that same logical tuple. Our index would need to point to both of these versions for both TXNs so the proper version is accessible to each transaction at all times.

Workers may get back multiple entries for a single fetch, and they then must follow the pointers to find their proper physical version.

5 Design consideration: Deletes

The DBMS physically deletes a tuple from the database only when all versions of a logically deleted tuple are not visible. If a tuple is a deleted, then there cannot be a new version of that tuple after the newest version. This means no write-write conflicts, and the first-writer wins.

We need a way to denote that a tuple has been logically deleted at some point in time. There are two approaches to this.

Approach #1: Deleted Flag

Maintain a flag to indicate that the logical tuple has been deleted after the newest physical version. This can either be in the tuple header or a separate column.

Approach #2: Tombstone Tuple

Create an empty physical version to indicate that a logical tuple is deleted. Use a separate pool for tombstone tuples with only a special bit pattern in version chain pointer to reduce storage overhead.