Lecture #04: Multi-Version Concurrency Control (Protocols)

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

1 Microsoft Hekaton

Hekaton started as an internal at Microsoft in 2008. The goal was to create new in-memory OLTP engine for **Microsoft SQL Server** (MSSQL). The project was led by database experts Paul Larson and Mike Zwilling. It was important that the new engine integrated with **MSSQL** ecosystem. It was also necessary that the engine support all possible OLTP workloads with predictable performance.

Key Lessons from Hekaton on building a high-performance OLTP DBMS:

- Use only lock-free data structures. This means no latches, spin locks, or critical sections for indexes, transaction map, memory allocator, garbage collector. Not necessarily true for indexes; systems using traditional latching methods for indexing more often than not outperform lock-free implementations.
- Only one single serialization point in the DBMS to get the transaction's begin and commit timestamp using an atomic addition (i.e., compare-and-swap – CaS). At high parallelism levels this could be a bottleneck. In this case we can improve performance with batching techniques.

The Hekaton concurrency control protocol relies on CaS to install new versions [3].

- Each transaction is assigned a timestamp when they begin (`BeginTS`) and when they commit (`EndTS`). Hekaton uses a special bit for transaction's `BeginTS` field to indicate whether it represents an uncommitted transaction.
- Each tuple contains two timestamps that represents their visibility and current state.
  - **`BEGIN-TS`**: The `BeginTS` of the active transaction or the `EndTS` of the committed transaction that created it.
  - **`END-TS`**: The `BeginTS` of the active transaction that created the next version or infinity or the `EndTS` of the committed transaction that created it

Hekaton allows transactions to speculatively read versions of transactions of uncommitted transactions. Then checks in validations if the transactions that it read uncommitted data committed. The DBMS does not allow speculative write; first transaction to write a new version succeeds, subsequent transactions that attempt to write to the same tuple are aborted.

Transaction State Map

Global map of all transactions states in the system. Transactions have to consult this map to determine the state of a transaction.

- `ACTIVE`: The transaction is executing read/write operations.
- `VALIDATING`: The transaction has invoked commit and the DBMS is checking whether it is valid.
- `COMMITTED`: The transaction is finished, but may not have updated its version's timestamps.
- `TERMINATED`: The transaction has updated the timestamps for all of the versions that it created.

Transaction Life-Cycle

1. **Begin**: Get `BeginTS`, set state to ACTIVE. The transaction then begins its normal execution for read-/write queries. In addition to updating version information, the DBMS also tracks each transaction's read set, scan set and write set.
2. **Pre-Commit**: Assign the transaction its `EndTS` and set its state in the transaction map to `VALIDATING`. During validation the DBMS checks whether the transaction's reads and scans produce the same result (to ensure serializability). If validation passes, then the DBMS writes new versions to redo log.
3. **Post-Processing**: Update the timestamps for the versions that the transaction either created or invalidated. The transaction updates the `BeginTS` in new versions, `CommitTS` in old versions.
4. **Terminate**: Set transaction state to TERMINATED and remove from map.

Transaction Meta-Data

The DBMS maintains additional meta-data about what transactions did during execution.

- **Read Set**: Pointers to every version that the transaction read.
- **Write Set:** Pointers to the versions that the transaction updated (old and new), deleted (old), and inserted (new).
- **Scan Set:** Stores enough information needed to perform each scan operation.
- **Commit Dependencies:** List of transactions that are waiting for this transaction to finish.

Transaction Validation

Hekaton used both optimistic / pessimistic schemes for transactions. Optimistic schemes outperforms pessimistic ones as the number of cores and parallelism increase.

- **Optimistic Transactions**: Check whether a version read is still visible at the end of a transaction. Repeat all index scans to check for phantoms.
- **Pessimistic Transactions**: Use shared and exclusive locks on records and buckets. Do not need any validations and use a separate background thread to perform deadlock detection. Under this setting, the DBMS does not need to perform validation.

The DBMS needs to scan transactions to provide its transactional guarantees.

- **First-Writer Wins:** The version vector always points to the last committed version. Do not need to check whether write-sets overlap.
- **Read Stability:** Check that each version read is still visible as of the end of the transaction.
- **Phantom Avoidance:** Repeat each scan to check whether new versions have become visible since the transaction began.
- Extend of validation depends on isolation level:
  - `SERIALIZABLE`: Read Stability + Phantom Avoidance.
  - `REPEATABLE READS`: Read Stability.
  - `SNAPSHOT ISOLATION`: None
  - `READ COMMITTED`: None

2 HyPer

Read/scan set validations are expensive if transactions access a lot of data. Appending new versions hurts the performance of OLAP scans due to pointer chasing and branching. Record level conflict checks may be too coarse-grained and incur false positives.

HyPer uses a MVCC implementation that is designed for HTAP workloads [5]. It is a column-store with delta record versioning. Avoids write-write conflicts by aborting transactions that try to update an uncommitted object.

- In-Place updates for non-indexed attributes
- Delete/Insert updates for indexed attributes
- N2O version chains
- No Predicate Locks and No Scan Checks

Transaction Validation

Like **Hekaton**, **HyPer** uses the first-writer wins policy. Although **HyPer**'s MVCC protocol supports multiple writers, the actual implementation uses single-threaded execution for write transactions.

It also uses a technique called *precision locking* to enforce serializability [2]. The system only stores each transaction's read predicates and not its entire read set. For each validating transactions, the DBMS evaluates its `WHERE` clause based on each delta record of any transactions that committed *after* the committing transaction has started.

- If the predicate evaluates to false, then the already committed transaction did not create a version that the committing transaction should have read.
- If the predicate evaluates to true, then this means that the committing transaction should have read the version created by the other transaction. Thus, the DBMS will abort our validating transactions and rollback its changes.

Version Synopsis

Checking the version vector for every tuple during long table scans is expensive. This is wasted work for cold data that is unlikely to have previous versions. Thus, to avoid this work, **HyPer** maintains special *version synopsis* tags per block to keep track of ranges of tuples that do not have versions. This is a separate column that tracks the position of the first and last versioned tuple in a block of tuples. When scanning tuples, the DBMS can check for strides of tuples without older versions and execute more efficiently.

3 SAP HANA

SAP HANA is an in-memory HTAP DBMS with time-travel version storage (N2O). [1].

- Supports both optimistic and pessimistic MVCC.
- Latest versions are stored in time-travel space.
- Hybrid storage layout (row + columnar).
- Based on **P*TIME**, **TREX**, and **MaxDB**.

Version Storage

The DBMS stores the oldest version in the main data table. Each tuple in the main data table contains a flag to denote whether there exists newer versions in the version space (i.e., time-travel table). For the time-travel table, the DBMS maintains a separate hash table that maps the logical record id for the tuple to the head of the version chain. This avoids storing additional metadata about versioning in the main table.

Instead of embedding begin/end timestamps in each tuple, **HANA** instead stores a pointer in each tuple to a context object. This object contains all the meta-data about the transaction that created a version with the data. This allows the DBMS to update the timestamps for multiple just by writing to this context instead of having to update each tuple individually.

The system has another layer of indirection for the meta-data about whether a transaction has committed.

This separate context object keeps track of when transactions commit and when their log records have been flushed to disk.

4 Cicada

Cicada is an academic in-memory OLTP engine based on optimistic MVCC with append-only storage (N2O) [4]. It was designed with specific optimizations for supporting both low and high contention workloads:

- Best-effort in-lining
- Loosely Synchronized Clocks
- Contention Aware validation
- Index Nodes Stored in Tables

Best-Effort In-lining

- Record meta-data is stored in a fixed location.
- Threads will attempt to inline read-mostly version within this meta-data to reduce version chain traversals.

Transaction Validation

- **Contention-aware Validation:** Validate access to recently modified (high contention) records first. Instead of comparing transactions' read sets, **Cicada** tries to reorder them based on their write timestamps. The DBMS keeps track of what transactions are mostly likely to conflict for a transaction attempting to commit, so it then checks for conflicts with them first.
- **Early Consistency Check:** Pre-validated access set before making global writes. Rather than waiting until the end of a transaction to perform validation, we can preemptively kill transactions that might abort later on.
- **Incremental Version Search:** Resume from last search location in version list to improve cache performance.

If the DBMS knows that most of the recently executed transactions committed successfully, then it can skip Contention-Aware Validation and Early Consistency check

References

[1] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The sap hana database–an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[2] J. R. Jordan, J. Banerjee, and R. B. Batman. Precision locks. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, SIGMOD '81, pages 143–147, 1981. doi: 10.1145/582318.582340. URL http://doi.acm.org/10.1145/582318.582340.

[3] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. In *VLDB '11: Proceedings of the VLDB Endowment*, volume 5, pages 298–309, December 2011. URL https://dl.acm.org/citation.cfm?id=2095686.2095689.

[4] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *SIGMOD '17: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 21–35, 2017. doi: https://dl.acm.org/citation.cfm?id=2749436.

[5] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *SIGMOD '15: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 677–689, 2015. doi: https://dl.acm.org/citation.cfm?id=2749436.