Lecture #18: Timestamp Ordering Concurrency Control

15-445/645 Database Systems (Spring 2025)
https://15445.courses.cs.cmu.edu/spring2025/
Carnegie Mellon University
Jignesh Patel

1 Timestamp Ordering Concurrency Control

Timestamp ordering (T/O) is an optimistic class of concurrency control protocols where the DBMS assumes that transaction conflicts are rare. Instead of requiring transactions to acquire locks before they are allowed to read/write to a database object, the DBMS instead uses timestamps to determine the serializability order of transactions.

Each transaction T_i is assigned a unique fixed timestamp TS(T_i) that is monotonically increasing. Different schemes assign timestamps at different times during the transaction. Some advanced schemes even assign multiple timestamps per transaction.

If TS(T_i) $<$ TS(T_j), then the DBMS must ensure that the execution schedule is equivalent to the serial schedule where T_i appears before T_j.

There are multiple timestamp allocation implementation strategies. The DBMS can use the system clock as a timestamp, but issues arise with edge cases like daylight savings. Another option is to use a logical counter. However, this has issues with overflow and with maintaining the counter across a distributed system with multiple machines. There are also hybrid approaches that use a combination of both methods.

2 Optimistic Concurrency Control (OCC)

Optimistic concurrency control (OCC) is an optimistic concurrency control protocol which uses timestamps to validate transactions. OCC works best when the number of conflicts is low. This is when either all of the transactions are read-only or when transactions access disjoint subsets of data. If the database is large and the workload is not skewed, then there is a low probability of conflict, making OCC a good choice.

In OCC, the DBMS creates a *private workspace* for each transaction. All modifications of the transaction are applied to this workspace. Any object read is copied into workspace and any object written is copied to the workspace and modified there. No other transaction can read the changes made by another transaction in its private workspace.

When a transaction commits, the DBMS compares the transaction's workspace *write set* to see whether it conflicts with other transactions. If there are no conflicts, the write set is installed into the "global" database.

OCC consists of three phases:

1. **Read Phase:** Here, the DBMS tracks the read/write sets of transactions and stores their writes in a private workspace. The DBMS copies every tuple accessed to its private workspace to ensure repeatable reads.
2. **Validation Phase:** When a transaction commits, the DBMS checks whether it conflicts with other transactions.

3. **Write Phase:** If validation succeeds, the DBMS applies the private workspace's changes to the database. Otherwise, it aborts and restarts the transaction.

Validation Phase

The DBMS assigns transactions timestamps when they enter the validation phase. To ensure only serializable schedules are permitted, the DBMS checks T_i against other transactions for RW and WW conflicts and makes sure that all conflicts go one way.

- **Approach 1:** Backward validation (from younger transactions to older transactions, slide 48)
- **Approach 2:** Forward validation (from older transactions to younger transactions, slide 47)

In forward validation, the DBMS checks the timestamp ordering of the committing transaction with all other running transactions. Transactions that have not yet entered the validation phase are assigned a timestamp of ∞.

If TS(T_i) < TS(T_j), then one of the following three conditions must hold:

1. T_i completes all three phases before T_j begins its execution (serial ordering) (slide 25).
2. T_i completes its Write phase before T_j starts its Write phase, and T_i does not write to any object read by T_j (slide 26).
   - WriteSet(T_i) $\cap$ ReadSet(T_j) $= \varnothing$.
3. T_i completes its Read phase before T_j completes its Read phase, and T_i does not write to any object that is either read or written by T_j (slide 40).
   - WriteSet(T_i) $\cap$ ReadSet(T_j) $= \varnothing$, and WriteSet(T_i) $\cap$ WriteSet(T_j) $= \varnothing$.

In backward validation, the DBMS checks the timestamp ordering of the committing transaction with the Read/Write sets of transactions that have already committed, using the same three conditions as above.

Potential Issues:

- High overhead for copying data locally into the transaction's private workspace.
- Validation/Write phase bottlenecks.
- Aborts are potentially more wasteful than in other protocols because they only occur after a transaction has already executed.
- Suffers from timestamp allocation bottleneck.

3 Dynamic Databases and The Phantom Problem

In our previous discussions, we have considered transactions that operate on a static set of objects within the database. However, when transactions perform insertions, updates, and deletions, we encounter a new set of complications.

The phantom problem arises when transactions only lock existing records, neglecting those that are in the process of being created. This oversight can lead to non-serializable executions because the set of objects in the database is not fixed (slide 57). **Approaches to Address the Phantom Problem:**

1. **Re-Execute Scans:**
   Transactions may re-run queries at commit time to check for different results, indicating missed changes due to new or deleted records. The DBMS keeps track of the WHERE clauses for all queries executed by the transaction. At commit time, it re-executes the scans to ensure that the results remain consistent (slide 59).
2. **Predicate Locking:**
   This involves acquiring locks based on the predicates of the queries, ensuring that any data that

satisfies the predicate cannot be modified by other transactions. Originally proposed in System R, this scheme is not widely implemented. However, systems like HyPer utilize a form of precision locking that is akin to predicate locking (slide 61).

3. **Index Locking:**
   Utilizing index keys to protect ranges of data, preventing phantoms by ensuring that no new data can fall within the locked ranges. Different schemes are employed to prevent phantoms using index locking:
   - **Key-Value Locks:** Locks on individual key-values in an index, including virtual keys for non-existent values (slide 63).
   - **Gap Locks:** Locks on the gap following a key-value, preventing insertion in these gaps (slide 64).
   - **Key-Range Locks:** Locks on a range of keys, from one existing key to the next (slide 67).
   - **Hierarchical Locking:** Allows transactions to hold broader key-range locks with different modes, reducing lock manager overhead (slide 70).

   In the absence of a suitable index, transactions must lock every page in the table or the entire table itself to prevent changes that could lead to phantoms.

4 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 retrieves a different result.
- **Lost Updates:** Transaction overwrites data of another concurrent transaction.
- **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.
   - Possible implementation: Strict 2PL + Phantom Protection (e.g., index locks).
2. `REPEATABLE READS`: Phantoms may happen.
   - Possible implementation: Strict 2PL.
3. `READ-COMMITTED`: Phantoms, unrepeatable reads, and lost updates may happen.
   - Possible implementation: Strict 2PL for exclusive locks, immediate release of the shared lock after a read.
4. `READ-UNCOMMITTED`: All anomalies may happen.
   - Possible implementation: Strict 2PL for exclusive locks, no shared locks for reads.

The isolation levels defined as part of SQL-92 standard only focused on anomalies that can occur in a 2PL-based DBMS. An application sets a per-transaction isolation level *before* it starts executing queries.