Lecture #20: Timestamp Ordering Concurrency Control

15-445/645 Database Systems (Fall 2026)
https://15445.courses.cs.cmu.edu/fall2026/
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, typically using UTC to avoid issues from daylight saving. Some DBMSs can get accurate clock time from satellites. 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 written is copied to the workspace and modified there, any object read is copied to the workspace only if the isolation level guarantees repeatable read. 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, it is assigned a unique timestamp and the DBMS checks whether it conflicts with other transactions.

3. **Write Phase:** If validation succeeds, the write timestamp is assigned to all modified objects in the private workspace, and the DBMS installs the private workspace's changes to the global database atomically. Otherwise, it aborts and restarts the transaction.

The read phase is where the transaction performs all the works, the validation and the write phase consists the commit protocol.

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:** Forward validation (from older transactions to younger transactions)
- **Approach 2:** Backward validation (from younger transactions to older transactions)

In forward validation, the DBMS checks the timestamp ordering of the committing transaction with all other running transactions that have not yet committed. 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 its Write phase before T_j begins its Read phase (serial ordering) (slide 16).
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 17).
   - 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 20).
   - 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 since the current transaction started, using the same three conditions as above. This is the more common approach compared to forward validation.

Write Phase

The DBMS propagates the transaction's write set to the database to make its changes visible to other transactions. There are two approaches to coordinating commits (slide 25):

- **Serial Commits:** Use a global latch so that only one transaction can be in the Validation/Write phases at a time.
- **Parallel Commits:** Use fine-grained write latches to allow parallel Validation/Write phases. Transactions acquire latches in a sequential key order to avoid deadlocks.

Ideal scenario for OCC: OCC works well when the number of conflicts is low, ideally when all transactions are read-only, or when transactions access disjoint subsets of data.

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.

When a transaction scans a range more than once and another transaction inserts or removes tuples that fall within the range between the scans, the scanning transaction can get different results across multitple scans. This is referred to as phantom read.

The phantom problem arises when transactions only lock existing records, neglecting those that are in the process of being created. Conflict serializability on reads and writes of individual items guarantees serializability **only** if the database's set of objects is fixed.

Approaches to Address the Phantom Problem:

1. **Lock Everything:**
   Transactions may take the lock of the entire table or every page, preventing insertions and deletions from other transactions during its execution. This approach is expensive as it locks more than the transaction needs.
2. **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.
3. **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 because of its complexity. However, systems like HyPer utilize a form of precision locking that is akin to predicate locking.
4. **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.
   - **Gap Locks:** Locks on the gap following a key-value, preventing insertion in these gaps.
   - **Key-Range Locks:** Locks on a range of keys, from one existing key to the next.
   - **Hierarchical Locking:** Allows transactions to hold broader key-range locks with different modes, reducing lock manager overhead.

   In the absence of a suitable index, transactions must fall back to the first approach to lock every page in the table or the entire table itself to prevent changes that could lead to phantoms. The Index Locking apporach is the most common among the four.

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 transac-

tions.

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: Strong Strict 2PL + Phantom Protection (e.g., index locks).
2. `REPEATABLE READS`: Phantoms may happen.
   - Possible implementation: Strong 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.