Lecture 19: Timestamp Ordering

15-445/645 Database Systems (Fall 2017)
Carnegie Mellon University
Prof. Andy Pavlo

Timestamp Ordering Concurrency Control

Timestamp ordering (T/O) is a 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.

1. Each transaction T_i is assigned a unique fixed timestamp that is monotonically increasing:
   (a) Let TS(T_i) be the timestamp allocated to transaction T_i
   (b) Different schemes assign timestamps at different times during the transaction
2. If TS(T_i) $<$ TS(T_j), then the DBMS must ensure that the execution schedule is equivalent to a serial schedule where T_i appears before T_j.
3. Multiple timestamp allocation implementation strategies:
   (a) System clock
   (b) Logical counter
   (c) Hybrid

Basic Timestamp Ordering (BASIC T/O)

Every database object X is tagged with timestamp of the last transaction that successfully did read/write:

1. W-TS(X): Write timestamp on object X.
2. R-TS(X): Read timestamp on object X.

The DBMS check timestamps for every operation. If transaction tries to access an object "from the future", then the DBMS aborts that transaction and restarts it.

1. **Read Operations:**
   (a) If TS(T_i) $<$ W-TS(X) this violates timestamp order of T_i with regard to the writer of X. Thus you abort T_i and restart it with same TS.
   (b) Else:

i. Allow T_i to read X.
ii. Update R-TS(X) to max(R-TS(X), TS(T_i)).
iii. Have to make a local copy of X to ensure repeatable reads for T_i.
iv. Last step may be skipped in lower isolation levels.

2. **Write Operations:**
   (a) If TS(T_i) $<$ R-TS(X) or TS(T_i) $<$ W-TS(X), abort and restart T_i.
   (b) Else:
      i. Allow T_i to write X and update W-TS(X) to T_i.
      ii. Also have to make a local copy of X to ensure repeatable reads for T_i.
3. **Optimization: Thomas Write Rule**
   (a) If TS(T_i) $<$ R-TS(X): Abort and restart T_i
   (b) If TS(T_i) $<$ W-TS(X):
      i. **Thomas Write Rule:** Ignore the write and allow transaction to continue.
      ii. Note that this violates timestamp order of T_i but this is okay because no other transaction will ever read T_i's write to object X.
   (c) Else: Allow T_i to write X and update W-TS(X)

The Basic T/O protocol generates a schedule that is conflict serializable if you do not use Thomas Write Rule. It cannot have deadlocks because no transaction ever waits. But there is a possibility of starvation for long transactions if short transactions keep causing conflicts.

It also permits schedules that are not recoverable. A schedule is **recoverable** if transactions commit only after all transactions whose changes they read or commit. Otherwise, the DBMS cannot guarantee that transactions read data that will be restored after recovering from a crash.

Potential Issues

1. High overhead from copying data to transaction's workspace and from updating timestamps.
2. Long running transactions can get starved: The likelihood that a transaction will read something from a newer transaction increases.
3. Suffers from the timestamp allocation bottleneck on highly concurrent systems.

Optimistic Concurrency Control (OCC)

If we assume that conflicts between transactions are **rare** and most transactions are **short lived**, it may be a better approach to optimize for the common case that assumes transactions are not going to have conflicts.

OCC works well 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, so again locking is wasteful

1. The DBMS creates a **private workspace** for each transaction:

   (a) All modifications are applied to the workspace.
   (b) Any object read is copied into workspace.
   (c) No other transaction can read the changes made by another transaction in its private workspace.

2. 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

3. OCC Transaction Phases
   (a) **Read Phase:** Track the read/write sets of transactions and store their writes in a private workspace.
   (b) **Validation Phase:** When a transaction commits, check whether it conflicts with other transactions.
   (c) **Write Phase:** If validation succeeds, apply private changes to database. Otherwise abort and restart the transaction.

Validation Phase

This is where the DBMS checks whether a transaction conflicts with other transactions. The DBMS needs to guarantee that only serializable schedules are permitted. The DBMS assigns transactions timestamps when they enter the validation phase.

T_i checks other transactions for RW and WW conflicts and makes sure that all conflicts go one way (from older transactions to younger transactions). The DBMS checks the timestamp ordering of the committing transaction with all other running transactions:

1. If TS(T_i) $<$ TS(T_j), then one of the following three conditions must hold:
   (a) T_i completes all three phases before T_j begins
   (b) T_i completes before T_j starts its Write phase, and T_i does not write to any object read by T_j.
   (c) 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.

Potential Issues

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

Partition-Based T/O

When a transaction commits in OCC, the DBMS has check whether there is a conflict with concurrent transactions across the entire database. This is slow if we have a lot of concurrent transactions because the DBMS has to acquire latches to do all of these checks.

An alternative is to split the database up in disjoint subsets called **partitions** (aka shards) and then only check for conflicts between transactions that are running in the same partition.

Partitions are protected by a single lock. Transactions are assigned timestamps based on when they arrive at the DBMS. Each transaction is queued at the partitions it needs before it starts running:

1. The transaction acquires a partition's lock if it has the lowest timestamp in that partition's queue.
2. The transaction starts when it has all of the locks for all the partitions that it will access during execution.
3. Transactions can read/write anything that they want at the partitions that they have locked. If a transaction tries to access a partition that it does not have the lock, it is aborted + restarted.

Potential Issues

Partition-based T/O protocol is very fast if: (1) the DBMS knows what partitions the transaction needs before it starts and (2) most (if not all) transactions only need to access a single partition.

The protocol only really works if (1) transactions are stored procedures (network communication causes the partition to idle because it has to wait for the next query to execute) and (2) transactions only touch one partition (multi-partition transactions cause partitions to be idle because partitions have to wait for the next query to execute).