CS54200: Distributed Database Systems

Timestamp Ordering

28 January 2009

Prof. Chris Clifton

Indiana Center for Database Systems

Timestamp Ordering

- The key idea for serializability is to ensure that conflicting operations are not executed in an inconsistent order.
- 2PL ensures this by not allowing new locks to be acquired once a lock is released.
- In timestamp ordering (TO), we predetermine an order and enforce it for conflicting operations.
- The order is based upon timestamps assigned to each txn.

2

Timestamp Ordering

- The TM assigns each txn, T_i, a unique timestamp, $ts(Ti)$.
- No two txns share a timestamp.
- A TO scheduler enforces:
- TO Rule: if $p_i[x]$ and $q_j[x]$ are conflicting operations, then the DM processes $p_i[x]$ before $q_j[x]$ iff $ts(T_i) < ts(T_j)$.

3

Serializability

- Theorem: If H is a history representing an execution produced by a TO scheduler, then H is serializable.
- Proof: Consider SG(H).
- If $T_i \rightarrow T_j$ is an edge in SG(H), then there must exist conflicting operations $p_i[x]$ and $q_j[x]$ in H such that $p_i[x] < q_j[x]$.
- Hence by the TO rule, $ts(T_i) < ts(T_j)$.
- If there is a cycle $T1 \rightarrow T2 \rightarrow \ldots \rightarrow Tn \rightarrow T1$ in SG(H), then by induction, $ts(T_1) < ts(T_1)$!!!

4

Basic TO

- For each operation, we pass it to the DM as long as it is not too late!
- An operation is too late if a conflicting operation with a larger timestamp has already been sent to the DM.
- If an operation is too late, the earlier operation cannot be undone, then the txn is aborted.
- The aborted txn is restarted with a new timestamp – why?
- This avoids cyclic restart.

5

Implementing Basic TO

- How to determine that an operation is too late?
- Maintain for each data item *x*, the maximum timestamp of a txn whose Read (Write) for *x* has been sent to the DM.
- Let this be stored in *max_r(w)_scheduled[x].*
- When $p_i[x]$ is received, check $ts(T_i)$ with *max_q_scheduled[x]* for all operations *q* that conflict with *p*.
- If $ts(T_i)$ is less than any of these, T_i is too late.
- Otherwise, schedule $p_i[x]$, update *max_p_scheduled[x]* to $ts(T_i)$.

6

Timing

- It is important for the scheduler to ensure that all scheduled operations on a given object are processed in the correct order.
- It must ensure that the DM acknowledges the completion of all conflicting operations before scheduling the next one.
- The scheduler maintains counts of pending operations of each type, and a queue of pending operations for each object.

7

Basic TO

- An operation $p_i[x]$ is accepted for scheduling if $ts(T_i) \geq max\_q\_scheduled[x]$ for all *q* that conflict with *p*.
- Otherwise, $p_i[x]$ is rejected, and T_i is aborted.
- If for all types *q* that conflict with *p*, there is no pending operation on *x*, and there are no waiting *q* type operations on *x*, then *pi[x]* is scheduled.
- Otherwise $p_i[x]$ is inserted into the waiting Q.
- When the DM acks an operation's completion, schedule all possible opns on *x* at the head of Q.

8

Strict TO

- TO does not even ensure recoverability!
- How can we enforce strictness?
- In the check for pending operations being processed by the DM, for write operations, we consider them pending until the DM acknowledges the abort or commit.
- Thus a write operation “locks” the item until the txn commits or aborts.
- TO does NOT suffer from deadlocks.

9

Strict TO = Strict 2PL?

- How do these two compare?
- They are not equal.
- E.g. $r_2[x]\ w_3[x]\ c_3\ w_1[y]\ c_1\ r_2[y]\ w_2[z]\ c_2$
- This history can be produced by a Strict TO scheduler if $ts(T_1) < ts(T_2) < ts(T_3)$.
- This cannot be produced by a 2PL scheduler: T_2 must release its read lock on x before $w_3[x]$ but may not set its read lock on y until after $w_1[y]$ – not allowed by 2PL!

10

TO Variants

- Distributed TO: How can TO be modified for distributed sites?
- Simple – nothing special needed as long as ….
- *Timestamps are unique across sites!*
- Easy to enforce this.
- Much better than distributed 2PL – no need for inter-site communication, unlike 2PL which requires communication for deadlocks.
- Conservative TO: delay operations. Make assumptions about the system or timestamps.

11

Serialization Graph Testing

- Maintain a version of the SG and check for acyclicity.
- Basis SGT: Upon receiving $p_i[x]$ add a node for T_i if necessary; add $T_i \rightarrow T_j$ for each $q_j[x]$ that conflicts and has been scheduled previously.
- If graph has a cycle – must reject p_i, abort T_i, remove the node for T_i.
- Otherwise, if all conflicting operations have been processed by DM, schedule $p_i[x]$.
- Must keep track of what operations have been scheduled for each transaction!!

12

Deleting Nodes

- When can nodes be deleted?
- Upon commitment? NO
- $r_{k+1}[x]w_1[x]w_1[y1]c_1w_2[x]w_2[y2]c_2\ldots w_k[x]w_k[yk]c_k$ followed by $w_{k+1}[z]$.
- In order to accept $w_{k+1}[z]$, z must not be any of $x, y1, y2, \ldots, yk$. Thus the scheduler has to remember all writes of $T1,\ldots,Tk$!
- Can delete a committed txn if it is a source ➔ it cannot be involved in any cycles. WHY?

13

Certifiers

- Extremely aggressive schedulers – no checks are done until absolutely necessary.
- Can be based upon 2PL, TO, or SGT.
- Schedule without checks until a txn wants to commit, at that time determine if allowing the txn to commit makes the execution non-serializable.
- If so, abort, otherwise commit.
- May lead to too many aborts if contention is high.
- Can be very efficient if contention is low.

14