CSI T5300: Advanced Database Systems

L12: Timestamp-based Protocols

Dr. Kenneth LEUNG

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China
Each transaction is issued a timestamp when it enters the system. If an old transaction $T_i$ has timestamp $TS(T_i)$, a new transaction $T_j$ is assigned timestamp $TS(T_j)$ such that $TS(T_i) < TS(T_j)$.

The protocol manages concurrent execution such that the timestamps determine the serializability order.

In order to assure such behavior, the protocol maintains for each data $Q$ two timestamp values:

- **W-timestamp**(Q) is the largest timestamp of any transaction that executed **write**(Q) successfully.
- **R-timestamp**(Q) is the largest timestamp of any transaction that executed **read**(Q) successfully.
The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order.

Suppose a transaction $T_i$ issues a read($Q$)

1. If $TS(T_i) < W$-timestamp($Q$), then $T_i$ needs to read a value of $Q$ that was already overwritten. Hence, the read operation is rejected, and $T_i$ is rolled back.
   - $T_i$ will restart with a new (larger) timestamp $TS'(T_i)$

2. If $TS(T_i) \geq W$-timestamp($Q$), then the read operation is executed, and R-timestamp($Q$) is set to the maximum of R-timestamp($Q$) and $TS(T_i)$. 

Timestamp-Based Protocols – Read Operation
Suppose that transaction $T_i$ issues $\text{write}(Q)$

1. If $\text{TS}(T_i) < \text{R-timestamp}(Q)$, then the value of $Q$ that $T_i$ is producing was needed previously, and the system assumed that that value would never be produced. Hence, the $\text{write}$ operation is rejected, and $T_i$ is rolled back.

2. If $\text{TS}(T_i) < \text{W-timestamp}(Q)$, then $T_i$ is attempting to write an obsolete value of $Q$. Hence, this $\text{write}$ operation is rejected, and $T_i$ is rolled back.

3. Otherwise, the $\text{write}$ operation is executed, and $\text{W-timestamp}(Q)$ is set to $\text{TS}(T_i)$. 

Timestamp-Based Protocols – Write Operation
Example of a TS Protocol

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

<table>
<thead>
<tr>
<th></th>
<th>(T_1=1)</th>
<th>(T_2=2)</th>
<th>(T_3=3)</th>
<th>(T_4=4)</th>
<th>(T_5=5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>read((Y)) RTS((Y))=2</td>
<td>read((Y)) RTS((Y))=2</td>
<td>write((Y)) WTS((Y))=3</td>
<td>read((X)) RTS((X))=5</td>
<td>read((X)) RTS((X))=5</td>
<td></td>
</tr>
<tr>
<td>read((X)) RTS((X))=5</td>
<td>read((Z or Y)) abort</td>
<td>write((Z)) WTS((Z))=3</td>
<td>read((Z)) RTS((Z))=5</td>
<td>write((Y))</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>write((Z)) abort</td>
<td>write((Z))</td>
<td></td>
</tr>
</tbody>
</table>

kwtleung@cse.ust.hk CSIT5300
Correctness of Timestamp-Ordering Protocols

• The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:

  transaction with smaller timestamp \quad \rightarrow \quad transaction with larger timestamp

• Thus, there will be no cycles in the precedence graph
• Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
• But the schedule may not be recoverable.
Problem with timestamp-ordering protocol:

- Suppose $T_i$ aborts, but $T_j$ has read a data item written by $T_i$
- Then $T_j$ must abort; if $T_j$ had been allowed to commit earlier, the schedule is not recoverable.
- Further, any transaction that has read a data item written by $T_j$ must abort
- This can lead to cascading rollback --- that is, a chain of rollbacks

Solution:

- A transaction is structured such that its writes are all performed at the end of its processing
- All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written
- A transaction that aborts is restarted with a new timestamp
• All protocols that we have seen (e.g., 2PL, TS Ordering) ensure correctness.
• However, a correct schedule may not be permitted by a protocol.
• The more correct schedules allowed by a protocol, the more the degree of concurrency
• The protocols also differ on the way they handle conflicts:
  – Lock-based protocols make transactions wait (thus they can result in deadlocks)
  – TS ordering protocols make transactions abort (thus there are no deadlocks but aborting a transaction may be more expensive).
• **Recoverability** is a necessary property of a schedule, which means that a transaction that has committed should not be rolled back.

• In order to ensure recoverability, a transaction $T_i$ can commit only after all transactions that wrote items which $T_i$ read have committed.

• A cascading rollback happens when an *uncommitted* transaction must be rolled back because it read an item written by a transaction that failed.

• It is desirable to have cascadeless schedules. In order to achieve this property a transaction should only be allowed to read items written by committed operations.
• If a schedule is cascadeless, it is also recoverable.
• Strict 2PL ensures cascadeless schedules by releasing all exclusive locks of transaction $T_i$ after $T_i$ commits (therefore other transactions cannot read the items locked by $T_i$ at the same time)
• TS ordering protocols can also achieve cascadeless schedules by performing all the writes at the end of the transaction as an atomic operation.