14. Timestamp-based Protocols
Timestamps

• Each transaction is issued a timestamp when it enters the system. If an old transaction $T_i$ has time-stamp $TS(T_i)$, a new transaction $T_j$ is assigned time-stamp $TS(T_j)$ such that $TS(T_i) < TS(T_j)$.

• The protocol manages concurrent execution such that the time-stamps 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 time-stamp of any transaction that executed **write**$(Q)$ successfully.
  – **R-timestamp**$(Q)$ is the largest time-stamp 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 $\text{read}(Q)$

1. If $TS(T_i) < W\text{-timestamp}(Q)$, 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 $TS(T_i) \geq W\text{-timestamp}(Q)$, then the read operation is executed, and $R\text{-timestamp}(Q)$ is set to the maximum of $R\text{-timestamp}(Q)$ and $TS(T_i)$. 
Suppose that transaction $T_i$ issues \text{write}(Q).

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.

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.

Otherwise, the \text{write} operation is executed, and $W$-timestamp$(Q)$ is set to $\text{TS}(T_i)$. 
Example of 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></td>
<td>read($Y$) RTS($Y$)=2</td>
<td>read($Y$) RTS($Y$)=2</td>
<td>write($Y$) W/RTS($Y$)=3</td>
<td>read($X$) RTS($X$)=5</td>
<td>read($Z$) RTS($Z$)=5</td>
</tr>
<tr>
<td></td>
<td>read($X$) RTS($X$)=5</td>
<td>read($Z$ or $Y$) abort</td>
<td>write($Z$) abort</td>
<td>write($Y$)</td>
<td>write($Z$)</td>
</tr>
</tbody>
</table>
The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:

- 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 recoverable.
Recoverability and Cascade Freedom

• 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
Multiversion Schemes

- Multiversion schemes keep old versions of data item to increase concurrency.
  - Multiversion Timestamp Ordering
  - Multiversion Two-Phase Locking
- Each successful **write** results in the creation of a new version of the data item written.
- Use timestamps to label versions.
- When a **read** \( (Q) \) operation is issued, select an appropriate version of \( Q \) based on the timestamp of the transaction, and return the value of the selected version.
- **reads** never fail as an appropriate version can always be found.
Multiversion Timestamp Ordering

- Each data item \( Q \) has a sequence of versions \(<Q_1, Q_2, \ldots, Q_m>\). Each version \( Q_k \) contains three data fields:
  - **Content** -- the value of version \( Q_k \).
  - **W-timestamp**\((Q_k)\) -- timestamp of the transaction that created (wrote) version \( Q_k \)
  - **R-timestamp**\((Q_k)\) -- largest timestamp of a transaction that successfully read version \( Q_k \)

- when a transaction \( T_i \) creates a new version \( Q_k \) of \( Q \), \( Q_k \)'s W-timestamp and R-timestamp are initialized to \( \text{TS}(T_i) \).

- R-timestamp of \( Q_k \) is updated whenever a transaction \( T_j \) reads \( Q_k \), and \( \text{TS}(T_j) > \text{R-timestamp}(Q_k) \).
Suppose that transaction \( T_i \) issues a \texttt{read}(Q) or \texttt{write}(Q) operation. Let \( Q_k \) be the version of \( Q \) whose write timestamp is the largest write timestamp less than or equal to \( \text{TS}(T_i) \).

1. If transaction \( T_i \) issues a \texttt{read}(Q), then the value returned is the content of version \( Q_k \). Reads always succeed.

2. If transaction \( T_i \) issues a \texttt{write}(Q),
   - if \( \text{TS}(T_i) < \text{R-timestamp}(Q_k) \), then transaction \( T_i \) is rolled back. Some other transaction \( T_j \) that (in the serialization order defined by the timestamp values) should read \( T_i \)'s write, has already read a version created by a transaction older than \( T_i \).
   - If \( \text{TS}(T_i) = \text{W-timestamp}(Q_k) \), the contents of \( Q_k \) are overwritten; \( Q_k \) was written before also by \( T_i \).
   - If \( \text{TS}(T_i) > \text{W-timestamp}(Q_k) \) a new version of \( Q \) is created.

Conflicts are resolved through aborting transactions.
Summary

• All protocols that we have seen (e.g., 2PL, TS Ordering, Multiversion protocols) 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
• Multiversion TS protocols also allow schedules that are not conflict serializable, but generate correct results.
• The protocols also differ on the way they handle conflicts: (i) Lock-based protocols make transactions wait (thus they can result in deadlocks); (ii) 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 from 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.
Summary (cont)

• 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.