TIMESTAMP AND VALIDATION PROTOCOLS

TIMESTAMP-BASED PROTOCOLS

- 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.

TIMESTAMP-BASED PROTOCOLS (CONT.)

- 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. The read operation is rejected, and $T_i$ is rolled back.
  2. If $TS(T_i) \geq W$-timestamp($Q$), then the read operation is executed, and $R$-timestamp($Q$) is set to $\max(R$-timestamp($Q$), $TS(T_i)$).
TIMESTAMP-BASED PROTOCOLS (CONT.)

- Suppose that transaction \( T_i \) issues write(\( Q \)).
  - If \( TS(T_i) < R\)-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.
    - The write operation is rejected, and \( T_i \) is rolled back.
  - If \( TS(T_i) < W\)-timestamp(\( Q \)), then \( T_i \) is attempting to write an obsolete value of \( Q \).
    - This write operation is rejected, and \( T_i \) is rolled back.
  - Otherwise, the write operation is executed, and W-timestamp(\( Q \)) is set to TS(\( T_i \)).

EXAMPLE USE OF THE PROTOCOL

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

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

CORRECTNESS OF TIMESTAMP-ORDERING PROTOCOL

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

  ![Diagram](https://via.placeholder.com/150)

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 cascade-free, and may not even be 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.

THOMAS' WRITE RULE

- Modified version of the timestamp-ordering protocol in which obsolete write operations may be ignored under certain circumstances.
- When $T_i$ attempts to write data item $Q$, if $TS(T_i) < W$-timestamp($Q$), then $T_i$ is attempting to write an obsolete value of $Q$.
  - Rather than rolling back $T_i$ as the timestamp ordering protocol would have done, this write operation can be ignored.
- Otherwise this protocol is the same as the timestamp ordering protocol.
- Thomas' Write Rule allows greater potential concurrency.
  - Allows some view serializable schedules that are not conflict serializable.

EXAMPLE OF THOMAS WRITE RULE

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($Y$)</td>
<td>RTS($Y$)=2</td>
<td>read($Y$)</td>
<td>RTS($Y$)=2</td>
<td>read($Z$ or $Y$)</td>
</tr>
<tr>
<td>read($Z$ or $Y$)</td>
<td>write($Y$)</td>
<td>WTS($Y$)=2</td>
<td>write($Z$)</td>
<td>WTS($Z$)=3</td>
</tr>
</tbody>
</table>
MULTIVERSION SCHEMES

- Multiversion schemes keep old versions of data item to increase concurrency.
- Multiversion Timestamp Ordering
- Each successful write results in the creation of a new version of the data item written.
- Use timestamps to label versions.
- When a read 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 have to wait as an appropriate version is returned immediately.

MULTIVERSION TIMESTAMP ORDERING

- Each data item Q has a sequence of versions <Q1, Q2, ..., Qm>. Each version Qk contains three data fields:
  - Content – the value of version Qk.
  - W-timestamp(Qk) – timestamp of the transaction that created (wrote) version Qk.
  - R-timestamp(Qk) – largest timestamp of a transaction that successfully read version Qk.
- When a transaction Ti creates a new version Qk of Q, Qk’s W-timestamp and R-timestamp are initialized to TS(Ti).
- R-timestamp of Qk is updated whenever a transaction Tj reads Qk, and TS(Tj) > R-timestamp(Qk).

MULTIVERSION TIMESTAMP ORDERING (CONT)

- Suppose that transaction Ti issues a read(Q) or write(Q) operation. Let Qk denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(Ti).
  - If transaction Ti issues a read(Q), then the value returned is the content of version Qk.
  - If transaction Ti issues a write(Q):
    - if TS(Ti) < R-timestamp(Qk), then transaction Ti is rolled back.
    - if TS(Ti) = W-timestamp(Qk), the contents of Qk are overwritten
      - else a new version of Q is created.
- Observe that:
  - Reads always succeed
  - A write by Ti is rejected if some other transaction Tj that (in the serialization order defined by the timestamp values) should read Q’s write, has already read a version created by a transaction older than Ti.
- Protocol guarantees serializability
VALIDATION-BASED PROTOCOL

- Execution of transaction $T_i$ is done in three phases.

1. **Read and execution phase**: Transaction $T_i$ writes only to temporary local variables.

2. **Validation phase**: Transaction $T_i$ performs a "validation test" to determine if local variables can be written without violating serializability.

3. **Write phase**: If $T_i$ is validated, the updates are applied to the database; otherwise, $T_i$ is rolled back.

The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.
- Assume for simplicity that the validation and write phase occur together, atomically and serially.
- I.e., only one transaction executes validation/write at a time.
- Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation.

Each transaction $T_j$ has 3 timestamps:
- $\text{Start}(T_j)$: the time when $T_j$ started its execution
- $\text{Validation}(T_j)$: the time when $T_j$ entered its validation phase
- $\text{Finish}(T_j)$: the time when $T_j$ finished its write phase

Serializability order is determined by timestamp given at validation time, to increase concurrency.
- Thus $TS(T_j)$ is given the value of $\text{Validation}(T_j)$.

This protocol is useful and gives greater degree of concurrency if probability of conflicts is low.
- because the serializability order is not pre-decided, and
- relatively few transactions will have to be rolled back.

If for all $T_i$ with $TS(T_i) < TS(T_j)$ either one of the following condition holds:
- $\text{finish}(T_i) < \text{start}(T_j)$
- $\text{start}(T_i) < \text{finish}(T_j) < \text{validation}(T_j)$ and the set of data items written by $T_i$ does not intersect with the set of data items read by $T_j$.

then validation succeeds and $T_j$ can be committed. Otherwise, validation fails and $T_j$ is aborted.

Justification: Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and
- the writes of $T_i$ do not affect reads of $T_j$ since they occur after $T_j$ has finished its reads.
- the writes of $T_i$ do not affect reads of $T_j$ since $T_j$ does not read any item written by $T_i$.
**SUMMARY**

- All protocols that we have seen (e.g., 2PL, TS Ordering, Multiversion protocols) ensure correctness (i.e. do not violate the ACID properties).
- However, it does not mean that if a schedule is correct it is always permitted by a protocol.
- The more correct schedules allowed by a protocol, the more the degree of concurrency (i.e., multiversion TS protocols allow more concurrency than simple TS protocols).
- 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 and validation-based protocols make transactions abort (thus there are no deadlocks but aborting a transaction may be more expensive).

**SUMMARY (CONT)**

- **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.
If a schedule is cascadeless, it is also recoverable.

Strict 2PL ensures cascadeless schedules by releasing all exclusive locks of transaction Tᵢ after Tᵢ commits (therefore other transactions cannot read the items locked by Tᵢ 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.