Timestamp-Based Protocols

- Idea:
  - Decide in advance ordering of transactions.
  - Ensure concurrent schedule serializes to that serial order.

- Timestamps
  1. $TS(T_i)$ is time $T_i$ entered the system
  2. Data item timestamps:
     1. $W-TS(Q)$: Largest timestamp of any xction that wrote $Q$
     2. $R-TS(Q)$: Largest timestamp of any xction that read $Q$

- Timestamps -> serializability order
Idea: If action $p_i$ of Xact $T_i$ conflicts with action $q_j$ of Xact $T_j$, and $TS(T_i) < TS(T_j)$, then $p_i$ must occur before $q_j$. Otherwise, restart violating Xact.
When Xact T wants to read Object O

- If $TS(T) < W-TS(O)$, this violates timestamp order of T w.r.t. writer of O.
  - So, abort T and restart it with a new, larger TS. (If restarted with same TS, T will fail again!)

- If $TS(T) > W-TS(O)$:
  - Allow T to read O.
  - Reset $R-TS(O)$ to $\max(R-TS(O), TS(T))$

- Change to $R-TS(O)$ on reads must be written to disk! This and restarts represent overhead.

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
When Xact T wants to Write Object O

- If $TS(T) < R - TS(O)$, then the value of $O$ that $T$ is producing was needed previously, and the system assumed that that value would never be produced. **write rejected, $T$ is rolled back.**

- If $TS(T) < W - TS(O)$, then $T$ is attempting to write an obsolete value of $O$. Hence, this **write operation is rejected, and $T$ is rolled back.**

- Otherwise, the **write operation is executed**, and $W - TS(O)$ is set to $TS(T)$. 

```
T start       U start

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
</tr>
</tbody>
</table>

U reads O

T writes O
```
Rollbacks still present
  - On rollback, new timestamp & restart

T1 rollback since $\text{TS}(T1) < W - \text{TS}(O) = \text{TS}(T2)$

Can reduce one rollback situation
When transaction writes an obsolete value, ignore it:
**Thomas’ write-rule** does not rollback T1
Example Use of the Protocol

A partial schedule for several data items for transactions with initial 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>
<th>$T_6$</th>
<th>$T_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($Y$)</td>
<td>read($Y$)</td>
<td>write($Y$)</td>
<td>write($X$)</td>
<td>read($Z$)</td>
<td>write($Y$)</td>
<td>read($Y$)</td>
</tr>
<tr>
<td>read($Q$)</td>
<td>read($X$)</td>
<td>write($Z$)</td>
<td></td>
<td>abort</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>abort</td>
<td>abort</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

15.76
Correctness of Timestamp-Ordering Protocol

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

  ![Diagram](image)

  transaction with smaller timestamp -> 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.
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 have to wait as an appropriate version is returned immediately.
Multiversion Timestamp Ordering

- Each data item $Q$ has a sequence of versions $\langle Q_1, Q_2, \ldots, Q_m \rangle$. 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 $TS(T_i)$.
- R-timestamp of $Q_k$ is updated whenever a transaction $T_j$ reads $Q_k$, and $TS(T_j) > R$-timestamp($Q_k$).
Multiversion Timestamp Ordering (Cont)

Suppose that transaction $T_i$ issues a \texttt{read}(Q) or \texttt{write}(Q) operation. Let $Q_k$ denote the version of $Q$ whose write timestamp is the largest write timestamp less than or equal to $TS(T_i)$.

1. If transaction $T_i$ issues a \texttt{read}(Q), then the value returned is the content of version $Q_k$.
2. If transaction $T_i$ issues a \texttt{write}(Q)
   1. if $TS(T_i) < R$-timestamp($Q_k$), then transaction $T_i$ is rolled back.
   2. if $TS(T_i) = W$-timestamp($Q_k$), the contents of $Q_k$ are overwritten
   3. else a new version of $Q$ is created.

\[
\begin{array}{c}
\text{X} \quad \text{(write Q)} \\
\hline
Q_{k-1} \quad \text{X} \quad \text{(read Q)} \quad Q_k \quad \text{X}
\end{array}
\]

- $T_i$: (write Q) $TS(T_i) < R$-timestamp($Q_k$)
- $T_i$: (read Q) $TS(T_i) > R$-timestamp($Q_k$)
Observe that

- Reads always succeed
- A write by $T_i$ is rejected if 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 $T_i$.

Protocol guarantees serializability
MVCC: Implementation Issues

- Creation of multiple versions increases storage overhead
  - Extra tuples
  - Extra space in each tuple for storing version information

- Versions can, however, be garbage collected
  - E.g. if Q has two versions Q5 and Q9, and the oldest active transaction has timestamp > 9, than Q5 will never be required again
Multiversion Two-Phase Locking

- Differentiates between read-only transactions and update transactions

- **Update transactions** acquire read and write locks, and hold all locks up to the end of the transaction. That is, update transactions follow rigorous two-phase locking.
  - Each successful **write** results in the creation of a new version of the data item written.
  - Each version of a data item has a single timestamp whose value is obtained from a counter **ts-counter** that is incremented during commit processing.

- **Read-only transactions** are assigned a timestamp by reading the current value of **ts-counter** before they start execution; they follow the multiversion timestamp-ordering protocol for performing reads.
**Multiversion Two-Phase Locking (Cont.)**

- When an update transaction wants to **read** a data item:
  - it obtains a **shared lock** on it, and reads the **latest version**.

- When it wants to **write** an item
  - it obtains **X lock** on; it then creates a new version of the item and sets this version's timestamp to \( \infty \).

- When update transaction \( T_i \) completes, commit processing occurs:
  - \( T_i \) sets timestamp on the versions it has created to \( \text{ts-counter} + 1 \)
  - \( T_i \) increments \( \text{ts-counter} \) by 1

- Read-only transactions that start after \( T_i \) increments \( \text{ts-counter} \) will see the values updated by \( T_i \).

- Read-only transactions that start before \( T_i \) increments the \( \text{ts-counter} \) will see the value before the updates by \( T_i \).

- Only serializable schedules are produced.
Snapshot Isolation

- Motivation: Decision support queries that read large amounts of data will conflict with OLTP transactions that update a few rows
  - Poor performance results
- Every transaction appears to be given its own snapshot of the database when it begins.
  - All reads are local to this snapshot
  - Updates are kept local until commit time.
  - Thus, a transaction is isolated from updates of other transactions.
  - Subsequent update attempts by other transactions need not wait.
Snapshot Isolation (commit)

- Must test to see if any transaction that was concurrent with T has already written an update to the DB.

- Commit processing for T (*First Committer Wins*):
  - Test if any transaction that was concurrent with T has already written an update for some data item that T intends to write.
  - If found, ABORT T.
  - Otherwise, T commits and its updates are written to DB.

- Lost update problem:
  - Suppose T and T’ execute concurrently.
  - When done, either T should see all updates from T’ or T’ should see all the updates from T. (serial execution)
  - If T reads some data item that T’ updates and T’ reads some data that T updates then both transactions will fail to read the update made by the other.
## Snapshot Isolation

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W(Y := 1)$</td>
<td>Start</td>
<td>$W(X := 2)$</td>
</tr>
<tr>
<td>Commit</td>
<td>$R(X) \rightarrow 0$</td>
<td>$W(Z := 3)$</td>
</tr>
<tr>
<td></td>
<td>$R(Y) \rightarrow 1$</td>
<td>Commit</td>
</tr>
<tr>
<td></td>
<td>$W(Y := 2)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concurrent updates not visible</td>
<td></td>
<td>Serialization error, T2 is rolled back</td>
</tr>
<tr>
<td>Own updates are visible</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not first-committer of X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $R(Z) \rightarrow 0$
- $R(Y) \rightarrow 2$
- $W(X := 3)$
- Commit-Req
- Abort
Skew Write

- $T_i : r(A); r(B): w(B)$
- $T_j : r(A); r(B); w(A)$
- both transactions will be allowed to commit.
- Consider the precedence graph:

$T_i$ reads $A$ before $T_j$ writes $A$

$T_j$ reads $B$ before $T_i$ writes $B$

Non-serializable!

Happens when a pair of transaction read data that is written by the other, but there is no data that is written by both.
Example

- Constraint: Checking + Savings can never go negative.
- Start: checking := $200 and savings := $100
- T1: checking := checking - $200

- T2: savings := savings - $200

- Both transactions can commit because they do not update the same data items.
- But, constraint is violated! (checking + savings = -$100)

- => test integrity constraints at commit.
Indices are unlike other database items in that their only job is to help in accessing data.

Index-structures are typically accessed very often, much more than other database items.

- Treating index-structures like other database items, e.g. by 2-phase locking of index nodes can lead to low concurrency.
- It is acceptable to have non-serializable concurrent access to an index as long as the accuracy of the index is maintained.
  - In particular, the exact values read in an internal node of a B⁺-tree are irrelevant so long as we end up in the correct leaf node.

There are several index concurrency protocols where locks on internal nodes are released early, and not in a two-phase fashion.
Concurrency in Index Structures (Cont.)

- Example of index concurrency protocol:
  - Use **crabbing** instead of two-phase locking on the nodes of the B\(^+\)-tree, as follows. During search/insertion/deletion:
    - First lock the root node in shared mode.
    - After locking all required children of a node in shared mode, release the lock on the parent.
    - During insertion/deletion, upgrade leaf node locks to exclusive mode.
    - When splitting or coalescing requires changes to a parent, lock the parent in exclusive mode.
  - Above protocol can cause excessive deadlocks
    - Searches coming down the tree deadlock with updates going up the tree
    - Can abort and restart search, without affecting transaction
  - Better protocols are available; e.g., the B-link tree protocol
High-performance hardware and parallelism help improve the rate of transaction processing, but are insufficient to obtain high performance:

- Disk I/O is a bottleneck — I/O time (10 milliseconds) has no decreased at a rate comparable to the increase in processor speeds.
- Parallel transactions may attempt to read or write the same data item, resulting in data conflicts that reduce effective parallelism.

We can reduce the degree to which a database system is disk bound by increasing the size of the database buffer.
Commercial 64-bit systems can support main memories of hundreds of gigabytes.

Memory resident data allows faster processing of transactions.

Disk-related limitations:
- Logging is a bottleneck when transaction rate is high.
- Use group-commit to reduce number of output operations (Will study two slides ahead.)
- If the update rate for modified buffer blocks is high, the disk data-transfer rate could become a bottleneck.
- If the system crashes, all of main memory is lost.
Main-Memory Database Optimizations

- To reduce space overheads, main-memory databases can use structures with pointers crossing multiple pages. In disk databases, the I/O cost to traverse multiple pages would be excessively high.
- No need to pin buffer pages in memory before data are accessed, since buffer pages will never be replaced.
- Design query-processing techniques to minimize space overhead - avoid exceeding main memory limits during query evaluation.
- Improve implementation of operations such as locking and latching, so they do not become bottlenecks.
- Optimize recovery algorithms, since pages rarely need to be written out to make space for other pages.
Group Commit

- Idea: Instead of performing output of log records to stable storage as soon as a transaction is ready to commit, wait until
  - log buffer block is full, or
  - a transaction has been waiting sufficiently long after being ready to commit
- Results in fewer output operations per committed transaction, and correspondingly a higher throughput.
- However, commits are delayed until a sufficiently large group of transactions are ready to commit, or a transaction has been waiting long enough—leads to slightly increased response time.
- Above delay acceptable in high-performance transaction systems since log buffer blocks will fill up quickly.
In systems with real-time constraints, correctness of execution involves both database consistency and the satisfaction of deadlines.

- **Hard deadline** – Serious problems may occur if task is not completed within deadline
- **Firm deadline** - The task has zero value if it completed after the deadline.
- **Soft deadline** - The task has diminishing value if it is completed after the deadline.

The wide variance of execution times for read and write operations on disks complicates the transaction management problem for time-constrained systems

- main-memory databases are thus often used
- Waits for locks, transaction aborts, contention for resources remain as problems even if data is in main memory

Design of a real-time system involves ensuring that enough processing power exists to meet deadline without requiring excessive hardware resources.