CS 457: Database Management Systems

Transactions: Optimistic Concurrency Control (Timestamp)
Pessimistic v.s. Optimistic

• **Pessimistic Concurrency Control** (locking)
  – Prevents unserializable schedules
  – Never abort for serializability (but may abort for deadlocks)
  – Best for workloads with high levels of contention

• **Optimistic Concurrency Control** (timestamp, multi-version, validation)
  – Assume schedule will be serializable
  – Abort when conflicts detected
  – Best for workloads with low levels of contention
Timestamps

• Each transaction receives unique timestamp \( TS(T) \)

Could be:

- The system’s clock
- A unique counter, incremented by the scheduler
Timestamps

Main invariant:

The timestamp order defines the serialization order of the transaction

Will generate a schedule that is view-equivalent to a serial schedule, and recoverable
Timestamps

With each element $X$, associate

- $RT(X) = \text{the highest timestamp of any transaction } U \text{ that read } X$

- $WT(X) = \text{the highest timestamp of any transaction } U \text{ that wrote } X$

- $C(X) = \text{the commit bit: true when transaction with highest timestamp that wrote } X \text{ committed}$
Main Idea

For any $r_T(X)$ or $w_T(X)$ request, check for conflicts:

- $w_U(X) \ldots r_T(X)$
- $r_U(X) \ldots w_T(X)$
- $w_U(X) \ldots w_T(X)$

Notation:
- $r_T(X)$: Transaction $T$ reads from element $X$
- $w_T(X)$: Transaction $T$ writes to element $X$
Main Idea

For any $r_T(X)$ or $w_T(X)$ request, check for conflicts:

- $w_U(X) \ldots r_T(X)$
- $r_U(X) \ldots w_T(X)$
- $w_U(X) \ldots w_T(X)$

When $T$ requests $r_T(X)$, need to check $TS(U) \leq TS(T)$
Read Too Late

- T wants to read X

START(T) ... START(U) ... \( w_U(X) \) ... \( r_T(X) \)
Read Too Late

• T wants to read X

\[
\text{START(T)} \ldots \text{START(U)} \ldots w_U(X) \ldots r_T(X)
\]

If \( WT(X) > TS(T) \) then need to rollback T !
Write Too Late

• T wants to write X

\[
\text{START}(T) \ldots \text{START}(U) \ldots r_U(X) \ldots w_T(X)
\]
Write Too Late

• T wants to write X

If RT(X) > TS(T) then need to rollback T!
Thomas’ Rule

But we can still handle it:

- T wants to write X

\[
\text{START}(T) \ldots \text{START}(V) \ldots w_V(X) \ldots w_T(X)
\]

If \( RT(X) \leq TS(T) \) and \( WT(X) > TS(T) \)
then don’t write X at all!

Why does this work?
Thomas’ Rule

But we can still handle it:

- T wants to write X

START(T) … START(V) … $w_{V}(X)$ … $w_{T}(X)$

If $RT(X) \leq TS(T)$ and $WT(X) > TS(T)$ then don’t write X at all!

Why does this work?

View-serializable schedule
View-Serializability

• By using Thomas’ rule we do obtain a view-serializable schedule
Summary So Far

Only for transactions that do not abort
Otherwise, may result in non-recoverable schedule

Transaction T wants to read element X
   If $WT(X) > TS(T)$ then ROLLBACK
   Else READ and update $RT(X)$ to larger of $TS(T)$ or $RT(X)$

Transaction wants to write element X
   If $RT(X) > TS(T)$ then ROLLBACK
   Else if $WT(X) > TS(T)$ ignore write & continue (Thomas Write Rule)
   Otherwise, WRITE and update $WT(X) = TS(T)$
Ensuring Recoverable Schedules

Recall:

• Schedule avoids cascading aborts if whenever a transaction reads an element, then the transaction that wrote it must have already committed

• Use the commit bit $C(X)$ to keep track if the transaction that last wrote $X$ has committed
Ensuring Recoverable Schedules

Read dirty data:

- T wants to read X, and WT(X) < TS(T)
- Seems OK, but…

If C(X)=false, T needs to wait for it to become true
Ensuring Recoverable Schedules

Thomas’ rule needs to be revised:

- T wants to write X, and WT(X) > TS(T)
- Seems OK not to write at all, but …(WHY?)

START(T) … START(U)… \( w_U(X) \)… \( w_T(X) \)… ABORT(U)

If C(X)=false, T needs to wait for it to become true
Timestamp-based Scheduling

• When a transaction $T$ requests $r_T(X)$ or $w_T(X)$, the scheduler examines $RT(X)$, $WT(X)$, $C(X)$, and decides one of:
  • To grant the request, or
  • To rollback $T$ (and restart with later timestamp)
  • To delay $T$ until $C(X) = true$
Timestamp-based Scheduling

**Transaction wants to READ element X**
- If $WT(X) > TS(T)$ then ROLLBACK
- Else If $C(X) = false$, then WAIT
- Else READ and update $RT(X)$ to larger of $TS(T)$ or $RT(X)$

**Transaction wants to WRITE element X**
- If $RT(X) > TS(T)$ then ROLLBACK
- Else if $WT(X) > TS(T)$
  - Then If $C(X) = false$ then WAIT
  - else IGNORE write (Thomas Write Rule)
- Otherwise, WRITE, and update $WT(X)=TS(T)$, $C(X)=false$
### Basic Timestamps with Commit Bit

<table>
<thead>
<tr>
<th></th>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
<th>T₄</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>RT=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WT=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C=true</td>
</tr>
<tr>
<td>W₂(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Time

- Delay

- Abort
### Basic Timestamps with Commit Bit

<table>
<thead>
<tr>
<th>TS (T_i):</th>
<th>T_1</th>
<th>T_2</th>
<th>T_3</th>
<th>T_4</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>RT=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WT=0 C=true</td>
</tr>
<tr>
<td>R_1(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abort</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WT=2 C=false</td>
</tr>
<tr>
<td>W_2(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RT=0</td>
</tr>
<tr>
<td>R_3(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C=true</td>
</tr>
<tr>
<td>Delay</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W_3(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RT=3</td>
</tr>
<tr>
<td>Delay</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C=false</td>
</tr>
<tr>
<td>W_4(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>abort</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WT=2 C=true</td>
</tr>
<tr>
<td>W_3(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WT=3 C=false</td>
</tr>
</tbody>
</table>

**Time**
Summary of Timestamp-based Scheduling

• View-serializable

• Avoids cascading aborts (hence: recoverable)

• Does NOT handle phantoms
  – These need to be handled separately, e.g. predicate locks
Multiversion Timestamp

• When transaction T requests r(X) but $WT(X) > TS(T)$, then T must rollback

• Idea: keep multiple versions of X: $X_t, X_{t-1}, X_{t-2}, \ldots$

\[
TS(X_t) > TS(X_{t-1}) > TS(X_{t-2}) > \ldots
\]
Details

• When $w_T(X)$ occurs, if the write is legal then create a new version, denoted $X_t$ where $t = TS(T)$.

• When $r_T(X)$ occurs, find most recent version $X_t$ such that $t < TS(T)$.

Notes:
  – $WT(X_t) = t$ and it never changes
  – $RT(X_t)$ must still be maintained to check legality of writes

• Can delete $X_t$ if we have a later version $X_{t1}$ and all active transactions $T$ have $TS(T) > t1$.
Example (in class)

Four versions of $X$: $X_3$ $X_9$ $X_{12}$ $X_{18}$

$R_6(X)$ -- Read $X_3$
$W_{21}(X)$ -- Check read timestamp of $X_{18}$
$R_{15}(X)$ -- Read $X_{12}$
$W_5(X)$ -- Check read timestamp of $X_3$

When can we delete $X_3$? (10+)
## Example w/ Basic Timestamps

<table>
<thead>
<tr>
<th></th>
<th>( T_1 )</th>
<th>( T_2 )</th>
<th>( T_3 )</th>
<th>( T_4 )</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Timestamps:</strong></td>
<td>150</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( R_1(A) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( RT=0 )</td>
</tr>
<tr>
<td>( W_1(A) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( WT=0 )</td>
</tr>
<tr>
<td>( R_2(A) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( RT=150 )</td>
</tr>
<tr>
<td>( W_2(A) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( WT=150 )</td>
</tr>
<tr>
<td>( R_3(A) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( RT=200 )</td>
</tr>
<tr>
<td><strong>Abort</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( WT=200 )</td>
</tr>
<tr>
<td>( R_4(A) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( RT=225 )</td>
</tr>
</tbody>
</table>
### Example w/ Multiversion

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
<th>T₄</th>
<th>A₀</th>
<th>A₁₅₀</th>
<th>A₂₀₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>150</td>
<td>200</td>
<td>175</td>
<td>225</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R₁(A)</th>
<th>W₁(A)</th>
<th>R₂(A)</th>
<th>W₂(A)</th>
<th>R₃(A)</th>
<th>W₃(A)</th>
<th>abort</th>
<th>R₄(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>RT=150</td>
<td>Create</td>
<td>RT=200</td>
<td>Create</td>
<td>RT=200</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


Outline

• Concurrency control by timestamps
• Concurrency control by validation
• Snapshot Isolation
Concurrency Control by Validation

• Each transaction $T$ defines:
  – Read set $RS(T)$ = the elements it reads
  – Write set $WS(T)$ = the elements it writes

• Each transaction $T$ has three phases:
  – Read phase; time = $START(T)$
  – Validate phase (may need to rollback); time = $VAL(T)$
  – Write phase; time = $FIN(T)$

Main invariant: the serialization order is $VAL(T)$
Outline

- Concurrency control by timestamps
- Concurrency control by validation
- **Snapshot Isolation**
Snapshot Isolation

- A type of multiversion concurrency control algorithm
- Provides yet another level of isolation

- Very efficient, and very popular
  - Oracle, PostgreSQL, SQL Server
Snapshot Isolation Overview

- Each transactions receives a timestamp TS(T)
- Transaction T sees snapshot at time TS(T) of the database
- Write/write conflicts resolved by “first committer wins” rule
  - Loser gets aborted
- Read/write conflicts are ignored
Discussion: Tradeoffs

• **Pessimistic CC: Locks**
  – Great when there are many conflicts
  – Poor when there are few conflicts

• **Optimistic CC: Timestamps, Validation, SI**
  – Poor when there are many conflicts (rollbacks)
  – Great when there are few conflicts

• **Compromise**
  – READ ONLY transactions → timestamps
  – READ/WRITE transactions → locks
Commercial Systems

Always check documentation!

- **DB2**: Strict 2PL
- **SQL Server**:
  - Strict 2PL
  - Snapshot Isolation (SI)
- **PostgreSQL**: SI; recently: seralizable SI (!)
- **Oracle**: SI