Database System Internals

Optimistic Concurrency Control

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle
Announcements

- Quiz grades on Gradescope

- Lab 3 part 1 due Friday
  - Perfect correctness not important, just first try
Pessimistic vs. Optimistic

- **Pessimistic CC** (locking)
  - Prevents unserializable schedules
  - Never abort for serializability (but may abort for deadlocks)
  - Best for workloads with high levels of contention

- **Optimistic CC** (timestamp, multi-version, validation)
  - Assume schedule will be serializable
  - Abort when conflicts detected
  - Best for workloads with low levels of contention
Concurrent control by timestamps (18.8)

Concurrent control by validation (18.9)

Snapshot Isolation
Each transaction receives unique timestamp $TS(T)$

Could be:

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

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

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

- $C(X) =$ the commit bit: true when transaction with highest timestamp that wrote $X$ committed
Timestamps

With each element $X$, associate

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

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

- $C(X) =$ the commit bit: true when transaction with highest timestamp that wrote $X$ committed

If transactions abort, we must reset the timestamps
For any $r_T(X)$ or $w_T(X)$ request, check for conflicts:

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

How do we check if Read too late?

Write too late?
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 $TS(U) \leq TS(T)$

How do we check if Read too late?
Write too late?
T wants to read X

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

- T wants to read X

If $WT(X) > TS(T)$ then need to rollback T!
T tried to read too late
- T wants to write $X$

```
START(T) … START(U) … $r_U(X)$ … $w_T(X)$
```
Write Too Late

- T wants to write X

If RT(X) > TS(T) then need to rollback T!

T tried to write too late
Thomas’ Rule

But… we can still handle it in one case:

- T wants to write X

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

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

If \( RT(X) \leq TS(T) \) and \( WT(X) > TS(T) \)

then don’t write X at all!

Why does this work?

View-serializable: V will have overwritten T!
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 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 (just a read will not change the commit bit).
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 …

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

If \( C(X) = \text{false} \), T needs to wait for it to become true
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 $C(X) = true$
RULES including commit bit

- There are 4 long rules in Sec. 18.8.4
- You should be able to derive them yourself, based on the previous slides
- Make sure you understand them!

READING ASSIGNMENT:
Garcia-Molina et al. 18.8.4
Transaction wants to READ element X
   If $WT(X) > TS(T)$ then ROLLBACK
   Else If $C(X) = \text{false}$, 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 $C(X) = \text{false}$ then WAIT
           else IGNORE write (Thomas Write Rule)
   Otherwise, WRITE, and update $WT(X)=TS(T)$, $C(X)=\text{false}$
### Basic Timestamps with Commit Bit

<table>
<thead>
<tr>
<th>Time</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>

May 5, 2021
## Basic Timestamps with Commit Bit

<table>
<thead>
<tr>
<th>Time</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WT=2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C=false</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RT=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C=true</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RT=3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WT=4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C=false</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WT=2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C=true</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WT=3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C=false</td>
</tr>
</tbody>
</table>
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
When transaction $T$ requests $r(X)$ but $WT(X) > TS(T)$, then $T$ must rollback.

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

$\text{TS}(X_t) > \text{TS}(X_{t-1}) > \text{TS}(X_{t-2}) > \ldots$
- When $w_T(X)$ occurs,
  if the write is legal then
  create a new version, denoted $X_t$ where $t = TS(T)$
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 \( t \leq TS(T) \)

Notes:
- \( WT(X_t) = t \) and it never changes for that version
- \( RT(X_t) \) 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 $X_3$?
Example w/ Basic Timestamps

<table>
<thead>
<tr>
<th>Timestamps:</th>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
<th>T₄</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>150</td>
<td>200</td>
<td>175</td>
<td>225</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>R₁(A)</td>
<td></td>
<td>R₂(A)</td>
<td></td>
<td>RT=150</td>
</tr>
<tr>
<td></td>
<td>W₁(A)</td>
<td></td>
<td>W₂(A)</td>
<td></td>
<td>WT=150</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R₃(A)</td>
<td></td>
<td></td>
<td>RT=200</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Abort</td>
<td></td>
<td></td>
<td>WT=200</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R₄(A)</td>
<td>RT=225</td>
</tr>
</tbody>
</table>
## Example w/ Multiversion

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$T_2$</td>
<td>$T_3$</td>
<td>$T_4$</td>
<td>$A_0$</td>
<td>$A_{150}$</td>
<td>$A_{200}$</td>
</tr>
<tr>
<td>150</td>
<td>200</td>
<td>175</td>
<td>225</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **$R_1(A)$**
- **$W_1(A)$**

- **$R_2(A)$**
- **$W_2(A)$**

- **$R_3(A)$**
- **$W_3(A)$**
- **abort**

- **$R_4(A)$**

- **$RT=150$**
- **Create**
  - **$RT=200$**
  - **Create**
  - **$RT=200$**
  - **Create**

- **$RT=225$**
**Example w/ Multiversion**

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T₁</td>
<td>T₂</td>
<td>T₃</td>
<td>T₄</td>
<td>A₀</td>
<td>A₁₅₀</td>
</tr>
<tr>
<td></td>
<td>150</td>
<td>200</td>
<td>175</td>
<td>225</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R₁(A)</td>
<td>W₁(A)</td>
<td>R₂(A)</td>
<td>W₂(A)</td>
<td>R₃(A)</td>
<td>W₃(A)</td>
</tr>
</tbody>
</table>

May 5, 2021
## Second Example w/ Multiversion

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>T₂</td>
<td>T₃</td>
<td>T₄</td>
<td>T₅</td>
<td>A₀</td>
<td>A₁</td>
<td>A₂</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>W₄(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Second Example w/ Multiversion

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
<th>T₄</th>
<th>T₅</th>
<th>A₀</th>
<th>A₁</th>
<th>A₂</th>
<th>A₃</th>
<th>A₄</th>
<th>A₅</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>W₁(A)</th>
<th></th>
<th>W₄(A)</th>
<th></th>
<th>Create</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R₂(A)</td>
<td></td>
<td>R₃(A)</td>
<td></td>
<td>Create RT=2</td>
</tr>
<tr>
<td></td>
<td>W₂(A)</td>
<td></td>
<td>R₅(A)</td>
<td></td>
<td>RT=3</td>
</tr>
<tr>
<td></td>
<td>abort</td>
<td></td>
<td>W₅(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R₁(A)</td>
<td></td>
<td></td>
<td>R₄(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</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></td>
</tr>
<tr>
<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>X</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

X means that we can delete this version
Outline

- Concurrency control by timestamps (18.8)
- Concurrency control by validation (18.9)
- 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 $VAL(T)$
Avoid $r_T(X) - w_U(X)$ Conflicts

START(U)  VAL(U)  FIN(U)

U: Read phase  Validate  Write phase

T: Read phase  Validate ?

START(T)  VAL(T)

IF $RS(T) \cap WS(U)$ and $FIN(U) > START(T)$
(U has validated and U has not finished before T begun)
Then ROLLBACK(T)
Avoid $w_T(X) - w_U(X)$ Conflicts

U: Read phase | Validate | Write phase

T: Read phase | Validate | Write phase ?

START(U) → VAL(U) → FIN(U)

START(T) → VAL(T)

IF $WS(T) \cap WS(U)$ and $FIN(U) > VAL(T)$
   (U has validated and U has not finished before T validates)
Then ROLLBACK(T)
Outline

- Concurrency control by timestamps (18.8)
- Concurrency control by validation (18.9)
- **Snapshot Isolation**
  - Not in the book, but good overview in Wikipedia
Snapshot Isolation

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

- Very efficient, and very popular
  - Oracle, PostgreSQL, SQL Server 2005

- Prevents many classical anomalies BUT…

- Not serializable (!), yet ORACLE and PostgreSQL use it even for SERIALIZABLE transactions!
  - But “serializable snapshot isolation” now in PostgreSQL
Each transactions receives a timestamp $\text{TS}(T)$

Transaction $T$ sees snapshot at time $\text{TS}(T)$ of the database

$W/W$ conflicts resolved by “first committer wins” rule
  • Loser gets aborted

$R/W$ conflicts are ignored
Multiversion concurrency control:
  • Versions of X: \( X_{t1}, X_{t2}, X_{t3}, \ldots \)

When T reads X, return \( X_{TS(T)} \).

When T writes X (to avoid lost update):
  • If latest version of X is TS(T) then proceed
  • Else if C(X) = true then abort
  • Else if C(X) = false then wait

• When T commits, write its updates to disk
What Works and What Not

- No dirty reads (Why ?)
  - Start each snapshot with consistent state

- No inconsistent reads (Why ?)
  - Two reads by the same transaction will read same snapshot

- No lost updates (“first committer wins”)

- Moreover: no reads are ever delayed

- However: read-write conflicts not caught!
  - A txn can read and commit even though the value had changed in the middle
Write Skew

T1:
READ(X);
if X >= 50
    then Y = -50; WRITE(Y)
COMMIT

T2:
READ(Y);
if Y >= 50
    then X = -50; WRITE(X)
COMMIT

In our notation:

R₁(X), R₂(Y), W₁(Y), W₂(X), C₁, C₂

Starting with X=50, Y=50, we end with X=-50, Y=-50.
Non-serializable !!!
Write Skews Can Be Serious

- Acidicland had two viceroys, Delta and Rho
- Budget had two registers: taXes, and spendYng
- They had high taxes and low spending...

Delta:
READ(taXes);
if taXes = ‘High’
  then { spendYng = ‘Raise’;
         WRITE(spendYng) }
COMMIT

Rho:
READ(spendYng);
if spendYng = ‘Low’
  then { taXes = ‘Cut’;
         WRITE(taXes) }
COMMIT

... and they ran a deficit ever since.
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 $\rightarrow$ timestamps
  - READ/WRITE transactions $\rightarrow$ locks
Always check documentation!

- **DB2**: Strict 2PL
- **SQL Server**:
  - Strict 2PL for standard 4 levels of isolation
  - Multiversion concurrency control for snapshot isolation
- **PostgreSQL**: SI; recently: serializable SI (!)
- **Oracle**: SI