Transactions: Concurrency Control
Reminders

• Last lecture!

• Please fill out the course evaluation form

• Project report due by Tuesday, June 8
   No late days!
Implementing Transactions

Notice: we will discuss about $\frac{1}{2}$ of these slides in class. If you want to learn more details, the skipped slides are easy to read.
Review

• What is a transaction?
• What is a schedule?
• Types:
  – Serializable
  – View serializable
  – Conflict serializable
• Types:
  – Recoverable
  – Avoid cascading aborts
  – Strict (see book)
• What is a transaction?
• What is a schedule?
• Types:
  – Serializable
  – View serializable
  – Conflict serializable
• Types:
  – Recoverable
  – Avoid cascading aborts
  – Strict (see book)
Scheduler

A.k.a. Concurrency Control Manager

• The module that schedules the transaction
• TXN T requests: READ(X) or WRITE(X),
• Scheduler answers one of:
  – Proceed
  – Put in a wait queue, schedule another TXN T’
  – Abort (!!!)
Implementing a Scheduler

Two major approaches:

- **Locking Scheduler**
  - Aka “pessimistic concurrency control”
  - SQLite, SQL Server, DB2

- **Multiversion Concurrency Control (MVCC)**
  - Aka “optimistic concurrency control”
  - Postgres, Oracle: Snapshot Isolation (SI)
Lock-based Implementation of Transactions
Locking Scheduler

Simple idea:

- Each element has a unique lock
- Each transaction must first acquire the lock before reading/writing that element
- If the lock is taken, then wait
- The transaction must release the lock(s)
Actions on Locks

$L_i(A) = \text{transaction } T_i \text{ acquires lock for element } A$

$U_i(A) = \text{transaction } T_i \text{ releases lock for element } A$

Let’s see this in action…
# A Non-Serializable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A)</td>
<td>READ(A)</td>
</tr>
<tr>
<td>A := A+100</td>
<td>A := A*2</td>
</tr>
<tr>
<td>WRITE(A)</td>
<td>WRITE(A)</td>
</tr>
<tr>
<td></td>
<td>READ(B)</td>
</tr>
<tr>
<td></td>
<td>B := B*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B)</td>
</tr>
<tr>
<td></td>
<td>READ(B)</td>
</tr>
<tr>
<td></td>
<td>B := B+100</td>
</tr>
<tr>
<td></td>
<td>WRITE(B)</td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(L_1(A); \text{READ}(A))</td>
<td>(L_2(A); \text{READ}(A))</td>
</tr>
<tr>
<td>A := A+100</td>
<td>A := A*2</td>
</tr>
<tr>
<td>WRITE(A); (U_1(A); L_1(B))</td>
<td>WRITE(A); (U_2(A); L_2(B)) BLOCKED…</td>
</tr>
<tr>
<td>READ(B)</td>
<td>(\ldots\text{GRANTED}; \text{READ}(B))</td>
</tr>
<tr>
<td>B := B+100</td>
<td>B := B*2</td>
</tr>
<tr>
<td>WRITE(B); (U_1(B))</td>
<td>WRITE(B); (U_2(B))</td>
</tr>
</tbody>
</table>
Example

T1
\[ L_1(A); \text{READ}(A) \]
A := A+100
WRITE(A); U_1(A); L_1(B)

READ(B)
B := B+100
WRITE(B); U_1(B);

T2
\[ L_2(A); \text{READ}(A) \]
A := A*2
WRITE(A); U_2(A);
L_2(B); BLOKED...

...GRANTED; READ(B)
B := B*2
WRITE(B); U_2(B);

Schedule is conflict-serializable
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L_1(A); \text{READ}(A) )</td>
<td>( L_2(A); \text{READ}(A) )</td>
</tr>
<tr>
<td>( A := A+100 )</td>
<td>( A := A*2 )</td>
</tr>
<tr>
<td>( \text{WRITE}(A); U_1(A); )</td>
<td>( \text{WRITE}(A); U_2(A); )</td>
</tr>
<tr>
<td>( L_1(B); \text{READ}(B) )</td>
<td>( L_2(B); \text{READ}(B) )</td>
</tr>
<tr>
<td>( B := B+100 )</td>
<td>( B := B*2 )</td>
</tr>
<tr>
<td>( \text{WRITE}(B); U_1(B); )</td>
<td>( \text{WRITE}(B); U_2(B); )</td>
</tr>
</tbody>
</table>

But…
But…

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(L_1(A);) (\text{READ}(A))</td>
<td>(L_2(A);) (\text{READ}(A))</td>
</tr>
<tr>
<td>(A := A + 100)</td>
<td>(A := A^2)</td>
</tr>
<tr>
<td>(\text{WRITE}(A);) (U_1(A))</td>
<td>(\text{WRITE}(A);) (U_2(A))</td>
</tr>
<tr>
<td>(L_1(B);) (\text{READ}(B))</td>
<td>(L_2(B);) (\text{READ}(B))</td>
</tr>
<tr>
<td>(B := B + 100)</td>
<td>(B := B^2)</td>
</tr>
<tr>
<td>(\text{WRITE}(B);) (U_1(B))</td>
<td>(\text{WRITE}(B);) (U_2(B))</td>
</tr>
</tbody>
</table>

Locks did not enforce conflict-serializability !!! What’s wrong ?
Two Phase Locking (2PL)

The 2PL rule:

In every transaction, all lock requests must precede all unlock requests
Example: 2PL transactions

T1

\[
L_1(A); L_1(B); \text{READ}(A) \\
A := A+100 \\
\text{WRITE}(A); \text{U}_1(A)
\]

T2

\[
L_2(A); \text{READ}(A) \\
A := A*2 \\
\text{WRITE}(A); \\
L_2(B); \text{BLOCKED}…
\]

READ(B)
B := B+100
\text{WRITE}(B); \text{U}_1(B);

\[…\text{GRANTED}; \text{READ}(B)\]
B := B*2
\text{WRITE}(B); \text{U}_2(A); \text{U}_2(B);\]

Conflict-serializable
Two Phase Locking (2PL)

**Theorem:** 2PL ensures conflict serializability
Two Phase Locking (2PL)

**Theorem:** 2PL ensures conflict serializability

**Proof.** Suppose not: then there exists a cycle in the precedence graph.
Two Phase Locking (2PL)

**Theorem:** 2PL ensures conflict serializability

**Proof.** Suppose not: then there exists a cycle in the precedence graph.

Then there is the following **temporal** cycle in the schedule:
Two Phase Locking (2PL)

**Theorem:** 2PL ensures conflict serializability

**Proof.** Suppose not: then there exists a cycle in the precedence graph.

Then there is the following *temporal* cycle in the schedule:

\[ U_1(A) \rightarrow L_2(A) \]

why?

\[ U_1(A) \] happened strictly before \( L_2(A) \)
Two Phase Locking (2PL)

**Theorem:** 2PL ensures conflict serializability

**Proof.** Suppose not: then there exists a cycle in the precedence graph.

Then there is the following **temporal** cycle in the schedule: $	ext{U}_1(\text{A}) \rightarrow \text{L}_2(\text{A})$ why?
Two Phase Locking (2PL)

**Theorem:** 2PL ensures conflict serializability

**Proof.** Suppose not: then there exists a cycle in the precedence graph.

Then there is the following temporal cycle in the schedule:

- $U_1(A) \rightarrow L_2(A)$
- $L_2(A) \rightarrow U_2(B)$

Why?

$L_2(A)$ happened strictly before $U_1(A)$
Two Phase Locking (2PL)

**Theorem:** 2PL ensures conflict serializability

**Proof.** Suppose not: then there exists a cycle in the precedence graph.

Then there is the following *temporal* cycle in the schedule:

\[ U_1(A) \to L_2(A) \]
\[ L_2(A) \to U_2(B) \]

why?
Two Phase Locking (2PL)

**Theorem**: 2PL ensures conflict serializability

**Proof.** Suppose not: then there exists a cycle in the precedence graph.

Then there is the following **temporal** cycle in the schedule:

- $U_1(A) \rightarrow L_2(A)$
- $L_2(A) \rightarrow U_2(B)$
- $U_2(B) \rightarrow L_3(B)$

why?
Two Phase Locking (2PL)

**Theorem:** 2PL ensures conflict serializability

**Proof.** Suppose not: then there exists a cycle in the precedence graph.

Then there is the following **temporal** cycle in the schedule:

- \( U_1(A) \rightarrow L_2(A) \)
- \( L_2(A) \rightarrow U_2(B) \)
- \( U_2(B) \rightarrow L_3(B) \)

......etc......
Two Phase Locking (2PL)

**Theorem**: 2PL ensures conflict serializability

**Proof**. Suppose not: then there exists a cycle in the precedence graph.

Then there is the following **temporal** cycle in the schedule:

- $U_1(A) \rightarrow L_2(A)$
- $L_2(A) \rightarrow U_2(B)$
- $U_2(B) \rightarrow L_3(B)$
- $L_3(B) \rightarrow U_3(C)$
- $U_3(C) \rightarrow L_1(C)$
- $L_1(C) \rightarrow U_1(A)$

Cycle in time: Contradiction
A New Problem: 
Non-recoverable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>U1(A)</td>
</tr>
<tr>
<td>L1(A); L1(B); READ(A)</td>
<td>L2(A); READ(A)</td>
</tr>
<tr>
<td>A := A + 100</td>
<td>A := A * 2</td>
</tr>
<tr>
<td>WRITE(A); U1(A);</td>
<td>WRITE(A);</td>
</tr>
<tr>
<td>READ(B)</td>
<td>L2(B); BLOCKED…</td>
</tr>
<tr>
<td>B := B + 100</td>
<td>…GRANTED; READ(B)</td>
</tr>
<tr>
<td>WRITE(B); U1(B);</td>
<td>B := B * 2</td>
</tr>
<tr>
<td>Commit</td>
<td>WRITE(B); U2(A); U2(B);</td>
</tr>
</tbody>
</table>

Rollback
## A New Problem: Non-recoverable Schedule

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L₁(A);</td>
<td>L₁(B); READ(A)</td>
<td>L₂(A); READ(A)</td>
</tr>
<tr>
<td>A := A+100</td>
<td>WRITE(A); U₁(A)</td>
<td>A := A*2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WRITE(A);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L₂(B); BLOCKED…</td>
</tr>
<tr>
<td></td>
<td>READ(B)</td>
<td>...GRANTED; READ(B)</td>
</tr>
<tr>
<td></td>
<td>B := B+100</td>
<td>B := B*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B); U₁(B);</td>
<td>WRITE(B); U₂(A); U₂(B);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

Rollback

Non-recoverable schedule
Strict 2PL

The Strict 2PL rule:

All locks are held until commit/abort:
All unlocks are done with commit/abort.
Strict 2PL

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L₁(A); READ(A)</td>
<td>L₂(A); BLOCKED…</td>
</tr>
<tr>
<td>A := A+100</td>
<td></td>
</tr>
<tr>
<td>WRITE(A);</td>
<td></td>
</tr>
<tr>
<td>L₁(B); READ(B)</td>
<td></td>
</tr>
<tr>
<td>B := B+100</td>
<td></td>
</tr>
<tr>
<td>WRITE(B);</td>
<td></td>
</tr>
<tr>
<td>Rollback &amp; U₁(A); U₁(B);</td>
<td></td>
</tr>
<tr>
<td>…GRANTED; READ(A)</td>
<td></td>
</tr>
<tr>
<td>A := A*2</td>
<td></td>
</tr>
<tr>
<td>WRITE(A);</td>
<td></td>
</tr>
<tr>
<td>L₂(B); READ(B)</td>
<td></td>
</tr>
<tr>
<td>B := B*2</td>
<td></td>
</tr>
<tr>
<td>WRITE(B);</td>
<td></td>
</tr>
<tr>
<td>Commit &amp; U₂(A); U₂(B);</td>
<td></td>
</tr>
</tbody>
</table>
Strict 2PL

• Lock-based systems always use strict 2PL
• Easy to implement:
  – When TXN requests READ(X) or WRITE(X), insert a lock requests on X
  – When the transaction commits/aborts, release all locks
• Conflict-serializable
• Strict
  – Thus: avoids-cascading aborts
Another problem: Deadlocks

- $T_1$: $R(A), W(B)$
- $T_2$: $R(B), W(A)$

- $T_1$ holds the lock on A, waits for B
- $T_2$ holds the lock on B, waits for A

This is a deadlock!
Another problem: Deadlocks

- Deadlock = when waits-for graph has a cycle

- Check the graph periodically; if deadlock is detected then pick a txn T and abort it; recheck more often.
Lock Modes

- **S** = shared lock (for READ)
- **X** = exclusive lock (for WRITE)

Lock compatibility matrix:
Lock Modes

- **S** = shared lock (for READ)
- **X** = exclusive lock (for WRITE)

Lock compatibility matrix:

<table>
<thead>
<tr>
<th></th>
<th>None</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>
Lock Granularity

- **Fine granularity locking** (e.g., tuples)
  - High concurrency
  - High overhead in managing locks
  - E.g., SQL Server

- **Coarse grain locking** (e.g., tables, entire database)
  - Many false conflicts
  - Less overhead in managing locks
  - E.g., SQL Lite

- **Solution**: lock escalation changes granularity as needed
Throughput (TPS)

Lock Performance

# Active Transactions

To avoid, use admission control

TPS = Transactions per second

thrashing
Optimistic concurrency control
Optimistic CC

• Proceeds more aggressively, but in case of conflicts are more likely to require abort
• Three main abstractions:
  – Timestamps
  – Multiversions
  – Validation
• Will illustrate them separately
Timestamps
Timestamps

• Each transaction receives a 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 strict
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
Warning

Confusing notation:

• $r_T(X) = \text{txn } T$ reads element $X$

• $RT(X) = \text{the } \text{“read timestamp” of } X$

• $TS(T) = \text{the } \text{“timestamp” of } \text{txn } T$
Main Idea

- Scheduler receives a request, \( r_T(X) \) or \( w_T(X) \)
- Should it allow it to proceed? Wait? Abort?
- Consider these cases:
  \[
  w_U(X) \ldots r_T(X)
  \]
  Should we allow this?
Main Idea

- Scheduler receives a request, $r_T(X)$ or $w_T(X)$
- Should it allow it to proceed? Wait? Abort?
- Consider these cases:

$$w_U(X) \ldots r_T(X)$$

Suppose the history was:

$$\text{START(U), \ldots, START(T), \ldots, } w_U(X), \ldots, r_T(X)$$
Main Idea

• Scheduler receives a request, $r_T(X)$ or $w_T(X)$
• Should it allow it to proceed? Wait? Abort?
• Consider these cases:

\[ w_U(X) \ldots r_T(X) \]

Suppose the history was:

START(U), ..., START(T), ..., $w_U(X)$, ..., $r_T(X)$

Should we allow this?

OK
Main Idea

• Scheduler receives a request, $r_T(X)$ or $w_T(X)$
• Should it allow it to proceed? Wait? Abort?
• Consider these cases:

$$w_U(X) \ldots r_T(X)$$

Suppose the history was:

START(U), ..., START(T), ..., $w_U(X)$, ..., $r_T(X)$

Should we allow this?

OK

$WT(X) \leq TS(T)$
Main Idea

- Scheduler receives a request, $r_T(X)$ or $w_T(X)$
- Should it allow it to proceed? Wait? Abort?
- Consider these cases:

  $w_U(X) \ldots r_T(X)$

Suppose the history was:

START(U), ..., START(T), ..., $w_U(X)$, ..., $r_T(X)$

START(T), ..., START(U), ..., $w_U(X)$, ..., $r_T(X)$

Should we allow this? OK
Main Idea

• Scheduler receives a request, $r_T(X)$ or $w_T(X)$
• Should it allow it to proceed? Wait? Abort?
• Consider these cases:

\[ w_U(X) \ldots r_T(X) \]

Suppose the history was:

\[
\begin{align*}
\text{START(U), } & \ldots, \text{START(T), } \ldots, w_U(X), \ldots, r_T(X) \\
\text{START(T), } & \ldots, \text{START(U), } \ldots, w_U(X), \ldots, r_T(X)
\end{align*}
\]
Main Idea

- Scheduler receives a request, $r_T(X)$ or $w_T(X)$
- Should it allow it to proceed? Wait? Abort?
- Consider these cases:

$w_U(X) \ldots r_T(X)$

Suppose the history was:

START(U), ..., START(T), ..., $w_U(X)$, ..., $r_T(X)$

START(T), ..., START(U), ..., $w_U(X)$, ..., $r_T(X)$

$WT(X) > TS(T)$

OK

Should we allow this?

Too late
Main Idea

• Scheduler receives a request, $r_T(X)$ or $w_T(X)$
• Should it allow it to proceed? Wait? Abort?
• Consider these cases:
  \[ w_U(X) \ldots r_T(X) \]
  \[ r_U(X) \ldots w_T(X) \]
  \[ w_U(X) \ldots w_T(X) \]
• Similarly for the other cases
Details

Read too late:

- T wants to read X, and $WT(X) > TS(T)$

START(T) ... START(U) ... $w_U(X)$ ... $r_T(X)$

Need to rollback T!
Details

Write too late:

- T wants to write X, and \( RT(X) > TS(T) \)

START(T) … START(U) … \( r_U(X) \) . . . \( w_T(X) \)

Need to rollback T!
Details

Write too late, but we can still handle it:

- T wants to write X, and

\[ RT(X) \leq TS(T) \text{ but } WT(X) > TS(T) \]

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

Don’t write X at all!

(Thomas’ rule)
Simplified TS

\begin{align*}
\text{Request is } & r_T(X) \\
\text{Request is } & w_T(X)
\end{align*}

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

\[ w_U(X) \ldots r_T(X) \]
\[ r_U(X) \ldots w_T(X) \]
\[ w_U(X) \ldots w_T(X) \]

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

Request is \( r_T(X) \)
If \( WT(X) > TS(T) \) then ROLLBACK
Else READ and update \( RT(X) \) to larger of \( TS(T) \) or \( RT(X) \)

Request is \( w_T(X) \)
?

CSEP 544 - Spring 2021 58
Simplified TS

\[ \text{w}_U(X) \ldots \text{r}_T(X) \]
\[ \text{r}_U(X) \ldots \text{w}_T(X) \]

Only for transactions that do not abort

Otherwise, may result in non-recoverable schedule

Request is \text{r}_T(X)

If \text{WT}(X) > \text{TS}(T) then ROLLBACK
Else READ and update \text{RT}(X) to larger of \text{TS}(T) or \text{RT}(X)

Request is \text{w}_T(X)

If \text{RT}(X) > \text{TS}(T) then ROLLBACK
Else if \text{WT}(X) > \text{TS}(T) ignore write & continue (Thomas Write Rule)
Otherwise, WRITE and update \text{WT}(X) = \text{TS}(T)
Simplified TS

- **Fact:** the simplified timestamp-based scheduling with Thomas’ rule ensures that the schedule is view-serializable
Full TS

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

Read dirty data:
• T wants to read X, and \( WT(X) < TS(T) \)
• Seems OK, but…

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

If \( C(X) = \text{false} \), T needs to wait for it to become true
Full TS

Thomas’ rule needs to be revised:

- T wants to write X, and $\text{WT}(X) > \text{TS}(T)$
- Seems OK not to write at all, but …

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

If $C(X)=$false, T needs to wait for it to become true
Full TS

Request is $r_T(X)$
- 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 $RT(X)$

Request is $w_T(X)$
- 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 $WT(X) = TS(T)$, $C(X) = \text{false}$
Full TS

- Fact: full timestamp-based scheduling is view-serializable and avoids cascading aborts
Timestamps

Main takeaway:

• TS defines the serialization order

• Simplifies the scheduler:
  – If action is consistent with serialization order, then proceed
  – Otherwise, ABORT
Multiversions
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}, . . .

$$\text{TS}(X_t) > \text{TS}(X_{t-1}) > \text{TS}(X_{t-2}) > . . .$$

• Let T read an older version, with appropriate timestamp
Details

• When $w_T(X)$ occurs,
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)

\[ X_3 \quad X_9 \quad X_{12} \quad X_{18} \]

\[ R_6(X) \quad -- \text{what happens?} \]

\[ W_{14}(X) \quad -- \text{what happens?} \]

\[ R_{15}(X) \quad -- \text{what happens?} \]

\[ W_5(X) \quad -- \text{what happens?} \]

When can we delete \(X_3\)?
Example (in class)

\[ R_6(X) \quad \text{-- what happens?} \quad \text{Return } X_3 \]

\[ W_{14}(X) \quad \text{-- what happens?} \]
\[ R_{15}(X) \quad \text{-- what happens?} \]
\[ W_5(X) \quad \text{-- what happens?} \]

When can we delete \( X_3 \)?
Example (in class)

\[ X_3 \quad X_9 \quad X_{12} \quad X_{18} \]

\[ R_6(X) \quad \text{-- what happens? Return } X_3 \]
\[ W_{14}(X) \quad \text{-- what happens?} \]
\[ R_{15}(X) \quad \text{-- what happens?} \]
\[ W_5(X) \quad \text{-- what happens?} \]

When can we delete \( X_3 \)?
Example (in class)

\[ X_3 \quad X_9 \quad X_{12} \quad X_{14} \quad X_{18} \]

\[ R_6(X) \quad \text{-- what happens?} \quad \text{Return } X_3 \]
\[ W_{14}(X) \quad \text{-- what happens?} \]
\[ R_{15}(X) \quad \text{-- what happens?} \]
\[ W_5(X) \quad \text{-- what happens?} \]

When can we delete \( X_3 \)?
Example (in class)

\[ X_3 \quad X_9 \quad X_{12} \quad X_{14} \quad X_{18} \]

\( R_6(X) \) -- what happens? Return \( X_3 \)
\( W_{14}(X) \) – what happens?
\( R_{15}(X) \) – what happens?
\( W_5(X) \) – what happens?

When can we delete \( X_3 \)?
Example (in class)

X₃ X₉ X₁₂ X₁₄ X₁₈

R₆(X) -- what happens? Return X₃
W₁₄(X) – what happens?
R₁₅(X) – what happens? Return X₁₄
W₅(X) – what happens?

When can we delete X₃?
Example (in class)

\[X_3 \quad X_9 \quad X_{12} \quad X_{14} \quad X_{18}\]

\[R_6(X) \quad -- \quad \text{what happens?} \quad \text{Return } X_3\]

\[W_{14}(X) \quad -- \quad \text{what happens?}\]

\[R_{15}(X) \quad -- \quad \text{what happens?} \quad \text{Return } X_{14}\]

\[W_5(X) \quad -- \quad \text{what happens?}\]

When can we delete \(X_3\)?
Example (in class)

\[ \text{TS(T)}=6 \]

\[ X_3 \quad X_9 \quad X_{12} \quad X_{14} \quad X_{18} \]

\[ R_6(X) \quad \text{-- what happens?} \quad \text{Return } X_3 \]
\[ W_{14}(X) \quad \text{-- what happens?} \]
\[ R_{15}(X) \quad \text{-- what happens?} \quad \text{Return } X_{14} \]
\[ W_5(X) \quad \text{-- what happens?} \quad \text{ABORT} \]

When can we delete \( X_3 \)?
Example (in class)

\[ X_3 \quad X_9 \quad X_{12} \quad X_{14} \quad X_{18} \]

\( R_6(X) \) -- what happens? Return \( X_3 \)
\( W_{14}(X) \) -- what happens?
\( R_{15}(X) \) -- what happens? Return \( X_{14} \)
\( W_{5}(X) \) -- what happens? ABORT

When can we delete \( X_3 \)?
Example (in class)

\[ X_3 \quad X_9 \quad X_{12} \quad X_{14} \quad X_{18} \]

\[ R_6(X) \quad -- \quad \text{what happens?} \quad \text{Return } X_3 \]
\[ W_{14}(X) \quad -- \quad \text{what happens?} \]
\[ R_{15}(X) \quad -- \quad \text{what happens?} \quad \text{Return } X_{14} \]
\[ W_5(X) \quad -- \quad \text{what happens?} \quad \text{ABORT} \]

When can we delete \( X_3 \)? When \( \min \ TS(T) \geq 9 \)
Multiversion

Takeaways:

• Reduces the number of aborts due to late reads

• Simplifies rollback

• Handles “phantoms”
Validation
Concurrency Control by Validation

• TXN reads elements, performs all updates on local copies

• At commit time:
  – CC manager performs validation
  – If OK, then it writes the local copies to disk
  – If not OK then aborts
Concurreny Control by Validation

• Each transaction T defines:
  – a *read set* RS(T) and
  – a *write set* WS(T)

• Each TXN has three phases:
  – Read elements RS(T): Time = START(T)
  – Validate: Time = VAL(T)
  – Writes elements WS(T). Time = FIN(T)

Main invariant: the serialization order is VAL(T)
Avoid $r_T(X) - w_U(X)$ Conflicts

U: Read phase Validate Write phase

T: Read phase Validate

START(T) → IF $RS(T) \cap WS(U)$ and $FIN(U) > START(T)$ THEN ROLLBACK(T)
Avoid $w_T(X) - w_U(X)$ Conflicts

IF $WS(T) \cap WS(U)$ and $FIN(U) > VAL(T)$
Then ROLLBACK(T)
Validation

Takeaways:

• READs/WRITEs proceed without delay

• Only delay happens at validation time

• May abort aggressively
Snapshot Isolation (SI)

A variant of multiversion/validation

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

Warning: not serializable
• Earlier versions of postgres implemented SI for the SERIALIZABLE isolation level
• Extension of SI to serializable has been implemented recently
• Will discuss only the standard SI (non-serializable)
Snapshot Isolation Rules

- Each transaction receives a timestamp $TS(T)$
- Transaction $T$ sees snapshot at time $TS(T)$ of the database
- When $T$ commits, updated pages are written to disk
- Write/write conflicts resolved by “first committer wins” rule
  - Loser gets aborted
- Read/write conflicts are ignored
Snapshot Isolation (Details)

- Multiversion concurrency control:
  - Versions of X: $X_{t1}, X_{t2}, X_{t3}, \ldots$

- When T reads X, return $X_t$, where $t$ is max s.t. $t \leq TS(T)$

- When T writes X:
  if other transaction updated X, abort
What Works and What Not

• No dirty reads (Why ?)
• No inconsistent reads (Why ?)
  – A: Each transaction reads a consistent snapshot
• No lost updates (“first committer wins”)
• Moreover: no reads are ever delayed
• However: read-write conflicts not caught! “Write skew”
Write Skew

Invariant: $X + Y \geq 0$

T1:
- READ(X);
- if $X \geq 50$
  - then $Y = -50$; WRITE(Y)
- COMMIT

T2:
- READ(Y);
- if $Y \geq 50$
  - then $X = -50$; WRITE(X)
- COMMIT

In our notation: $R_1(X), R_2(Y), W_1(Y), W_2(X), C_1, C_2$
Write Skew

Invariant: $X + Y \geq 0$

T1:
    READ(X);
    if $X \geq 50$
        then $Y = -50$; WRITE(Y)
    COMMIT

T2:
    READ(Y);
    if $Y \geq 50$
        then $X = -50$; WRITE(X)
    COMMIT

In our notation:

$R_1(X), R_2(Y), W_1(Y), W_2(X), C_1, C_2$

$X_0 \quad Y_0$
Write Skew

Invariant: $X + Y \geq 0$

T1:
READ(X);
if $X \geq 50$
then $Y = -50$; WRITE(Y)
COMMIT

T2:
READ(Y);
if $Y \geq 50$
then $X = -50$; WRITE(X)
COMMIT

In our notation:
$R_1(X), R_2(Y), W_1(Y), W_2(X), C_1, C_2$

$X_0 \quad Y_0$
Write Skew

Invariant: $X + Y \geq 0$

T1:

READ(X);
if $X \geq 50$
then $Y = -50$; WRITE(Y)
COMMIT

T2:

READ(Y);
if $Y \geq 50$
then $X = -50$; WRITE(X)
COMMIT

In our notation:

$R_1(X), R_2(Y), W_1(Y), W_2(X), C_1, C_2$

$X_0, Y_0$
Write Skew

Invariant: $X + Y \geq 0$

T1:
READ($X$);
if $X \geq 50$
then $Y = -50$; WRITE($Y$)
COMMIT

T2:
READ($Y$);
if $Y \geq 50$
then $X = -50$; WRITE($X$)
COMMIT

In our notation:
$R_1(X), R_2(Y), W_1(Y), W_2(X), C_1, C_2$

$X_0 \quad Y_0 \quad Y_1$

Should have aborted T1, but SI doesn’t keep RT($Y$)
Write Skew

Invariant: $X + Y \geq 0$

T1:
- READ($X$);
- if $X \geq 50$
  - then $Y = -50$; WRITE($Y$)
- COMMIT

T2:
- READ($Y$);
- if $Y \geq 50$
  - then $X = -50$; WRITE($X$)
- COMMIT

In our notation:

$R_1(X), R_2(Y), W_1(Y), W_2(X), C_1, C_2$

$X_0 \quad Y_0 \quad Y_1 \quad X_2$

Should have aborted T1, but SI doesn't keep RT($Y$)
Write Skew

Invariant: $X + Y \geq 0$

T1:
- READ(X);
- if $X \geq 50$
  - then $Y = -50$; WRITE(Y)
- COMMIT

T2:
- READ(Y);
- if $Y \geq 50$
  - then $X = -50$; WRITE(X)
- COMMIT

In our notation:

$R_1(X), R_2(Y), W_1(Y), W_2(X), C_1, C_2$

$X_0 \ Y_0 \ Y_1 \ X_2$

Should have aborted T1, but SI doesn’t keep RT(Y)
Write Skew

Invariant: \( X + Y \geq 0 \)

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

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

In our notation:
\[
R_1(X), R_2(Y), W_1(Y), W_2(X), C_1, C_2
\]

Starting with \( X=50, Y=50 \), we end with \( X=-50, Y=-50 \).
Non-serializable !!!

Should have aborted T1, but SI doesn't keep RT(Y)
Discussions

• Snapshot isolation (SI) is like repeatable reads but also avoids some (not all) phantoms

• If DBMS runs SI and the app needs serializable:
  – use dummy writes for all reads to create write-write conflicts… but that is confusing for developers

• Extension of SI to make it serializable is implemented in postgres
Phantom Problem

• So far we have assumed the database to be a static collection of elements (=tuples)

• If tuples are inserted/deleted then the phantom problem appears
Suppose there are two blue products, A1, A2:

**Phantom Problem**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT * FROM Product WHERE color='blue'</td>
<td>INSERT INTO Product(name, color) VALUES ('A3','blue')</td>
</tr>
<tr>
<td>SELECT * FROM Product WHERE color='blue'</td>
<td></td>
</tr>
</tbody>
</table>
Suppose there are two blue products, A1, A2:

**Phantom Problem**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT *</td>
<td>INSERT INTO Product(name, color) VALUES ('A3', 'blue')</td>
</tr>
<tr>
<td>FROM Product</td>
<td></td>
</tr>
<tr>
<td>WHERE color='blue'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SELECT *</td>
</tr>
<tr>
<td></td>
<td>FROM Product</td>
</tr>
<tr>
<td></td>
<td>WHERE color='blue'</td>
</tr>
</tbody>
</table>

Is this schedule serializable?

No: T1 sees a “phantom” product A3
Suppose there are two blue products, A1, A2:

Phantom Problem

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT *</td>
<td>INSERT INTO Product(name, color) VALUES (‘A3’,’blue’)</td>
</tr>
<tr>
<td>FROM Product</td>
<td></td>
</tr>
<tr>
<td>WHERE color=‘blue’</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>SELECT *</td>
<td></td>
</tr>
<tr>
<td>FROM Product</td>
<td></td>
</tr>
<tr>
<td>WHERE color=‘blue’</td>
<td></td>
</tr>
</tbody>
</table>

\[ R_1(A1); R_1(A2); W_2(A3); R_1(A1); R_1(A2); R_1(A3) \]
Suppose there are two blue products, A1, A2:

**Phantom Problem**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT *</td>
<td>INSERT INTO Product(name, color)</td>
</tr>
<tr>
<td>FROM Product</td>
<td>VALUES (‘A3’, ‘blue’)</td>
</tr>
<tr>
<td>WHERE color='blue'</td>
<td></td>
</tr>
</tbody>
</table>

SELECT *
FROM Product
WHERE color='blue'

R₁(A₁); R₁(A₂); W₂(A₃); R₁(A₁); R₁(A₂); R₁(A₃)

W₂(A₃); R₁(A₁); R₁(A₂); R₁(A₁); R₁(A₂); R₁(A₃)
Suppose there are two blue products, A1, A2:

Phantom Problem

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT *</td>
<td>INSERT INTO Product(name, color) VALUES ('A3', 'blue')</td>
</tr>
<tr>
<td>FROM Product</td>
<td></td>
</tr>
<tr>
<td>WHERE color='blue'</td>
<td></td>
</tr>
<tr>
<td>SELECT *</td>
<td></td>
</tr>
<tr>
<td>FROM Product</td>
<td></td>
</tr>
<tr>
<td>WHERE color='blue'</td>
<td></td>
</tr>
</tbody>
</table>

But this is conflict-serializable!
Phantom Problem

• A “phantom” is a tuple that is invisible during part of a transaction execution but not invisible during the entire execution

• In our example:
  – T1: reads list of products
  – T2: inserts a new product
  – T1: re-reads: a new product appears!
Phantom Problem

• In a **static** database:
  – Conflict serializability implies serializability

• In a **dynamic** database, this may fail due to phantoms

• Strict 2PL guarantees conflict serializability, but not serializability
Dealing With Phantoms

• Lock the entire table
• Lock the index entry for ‘blue’
  – If index is available
• Or use predicate locks
  – A lock on an arbitrary predicate

Dealing with phantoms is expensive!
Summary of Serializability

• Serializable schedule = equivalent to a serial schedule
• (strict) 2PL guarantees conflict serializability
  – What is the difference?

• **Static database:**
  – Conflict serializability implies serializability

• **Dynamic database:**
  – Conflict serializability plus phantom management implies serializability
Weaker Isolation Levels

• Serializable are expensive to implement

• SQL allows the application to choose a more efficient implementation, which is not always serializable: weak isolation levels
Isolation Levels in SQL

1. “Dirty reads”
   SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

2. “Committed reads”
   SET TRANSACTION ISOLATION LEVEL READ COMMITTED

3. “Repeatable reads”
   SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

4. Serializable transactions
   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
Lost Update

Write-Write Conflict

\[
\begin{align*}
T_1 &: \text{READ}(A) \\
T_1 &: A := A + 5 \\
T_1 &: \text{WRITE}(A)
\end{align*}
\]

\[
\begin{align*}
T_2 &: \text{READ}(A); \\
T_2 &: A := A \times 1.3 \\
T_2 &: \text{WRITE}(A);
\end{align*}
\]

Never allowed at any level
1. Isolation Level: Dirty Reads

- "Long duration" WRITE locks
  - Strict 2PL
- No READ locks
  - Read-only transactions are never delayed

Possible problems: dirty and inconsistent reads
1. Isolation Level: Dirty Reads

Write-Read Conflict

$T_1$: WRITE(A)

$T_1$: ABORT

$T_2$: READ(A)
1. Isolation Level: Dirty Reads

Write-Read Conflict

T₁: A := 20; B := 20;
T₁: WRITE(A)
T₁: WRITE(B)
T₂: READ(A);
T₂: READ(B);

Inconsistent read
2. Isolation Level: Read Committed

- “Long duration” WRITE locks
  - Strict 2PL
- “Short duration” READ locks
  - Only acquire lock while reading (not 2PL)

Unrepeatable reads:
When reading same element twice, may get two different values
2. Isolation Level: Read Committed

**Read-Write Conflict**

\[ T_1: \text{WRITE}(A) \]  
\[ \text{COMMIT} \]

\[ T_2: \text{READ}(A); \]

\[ T_2: \text{READ}(A); \]

**Unrepeatable read**
3. Isolation Level: Repeatable Read

- “Long duration” WRITE locks
  - Strict 2PL
- “Long duration” READ locks
  - Strict 2PL

This is not serializable yet !!!

Why ?
4. Isolation Level Serializable

• “Long duration” WRITE locks
  – Strict 2PL

• “Long duration” READ locks
  – Strict 2PL

• Predicate locking
  – To deal with phantoms
Beware!

In commercial DBMSs:

• Default level may not be serializable
• Default level differs between DBMSs
• Some engines support subset of levels!
• Also, some DBMSs do NOT use locking and different isolation levels can lead to different pbs

Bottom line: Read the doc for your DBMS!
Final Thoughts on Transactions

• Benchmarks: TPC/C; typical throughput: x100’s TXN/second

• New trend: multicores
  – Current technology can scale to x10’s of cores, but not beyond!
  – Major bottleneck: latches that serialize the cores

• New trend: distributed TXN
  – NoSQL: give up serialization
  – Serializable: very difficult e.g. Spanner w/ Paxos