CSE544
Data Management

Lectures 16-17
Transactions: Concurrency Control
Reminders

• Project milestones due on Friday

• Next Friday 3/12: Project Presentations!
  – 9am – 1pm (may finish earlier)
  – 11 teams
  – Each team gets 10’ presentation + 5’ discussion
  – Contest for the best presentation (stay tuned!)
Implementing Transactions
Scheduler

• **Scheduler a.k.a. Concurrency Control Manager**
  – The module that schedules the transaction’s actions
  – Goal: ensure the schedule is serializable

• **We discuss next how a scheduler may be implemented**
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

$L_i(A)$ = transaction $T_i$ acquires lock for element $A$

$U_i(A)$ = transaction $T_i$ 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><strong>READ(A)</strong></td>
<td><strong>READ(A)</strong></td>
</tr>
<tr>
<td>A := A+100</td>
<td>A := A*2</td>
</tr>
<tr>
<td>WRITE(A)</td>
<td>WRITE(A)</td>
</tr>
</tbody>
</table>

T1: READ(A) → A := A+100 → WRITE(A)

T2: READ(A) → A := A*2 → WRITE(A) → READ(B) → B := B*2 → WRITE(B) → READ(B) → B := B+100 → WRITE(B)
Example

T1

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

T2

\[ L_2(A); \text{READ}(A) \]
\[ A := A \times 2 \]
\[ \text{WRITE}(A); U_2(A); \]
\[ L_2(B); \text{BLOCKED…} \]

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

…GRANTED; READ(B)
\[ B := B \times 2 \]
\[ \text{WRITE}(B); U_2(B); \]

Scheduler has ensured a conflict-serializable schedule
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 \times 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 \times 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

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<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></td>
</tr>
<tr>
<td>WRITE(B); U₁(B);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...GRANTED;</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); U₂(A); U₂(B);</td>
</tr>
</tbody>
</table>

Now it is 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.

![Diagram showing a precedence graph with nodes T1, T2, T3, A, B, and C connected by arrows indicating the order of operations.](image)
Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph. Then there is the following \textit{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?

\[ U_1(A) \text{ happened strictly before } L_2(A) \]
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:

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

\[ L_2(A) \text{ happened strictly before } U_1(A) \]
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:

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

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:

- $U_1(A) \rightarrow L_2(A)$
- $L_2(A) \rightarrow U_2(B)$
- $U_2(B) \rightarrow L_3(B)$
- $\ldots \ldots \text{etc.} \ldots$
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

T1

L₁(A); L₁(B); READ(A)
A := A + 100
WRITE(A); U₁(A)

READ(B)
B := B + 100
WRITE(B); U₁(B);

T2

L₂(A); READ(A)
A := A * 2
WRITE(A);
L₂(B); BLOCKED…

…GRANTED; READ(B)
B := B * 2
WRITE(B); U₂(A); U₂(B);
Commit

Rollback
A New Problem: Non-recoverable Schedule

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

Rollback

Elements A, B written by T1 are restored to their original value.
### A New Problem: Non-recoverable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<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></td>
<td>WRITE(B); U₂(A); U₂(B);</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

---

**Elements A, B written by T1 are restored to their original value.**

**Dirty reads of A, B lead to incorrect writes.**

---

**Rollback**
## A New Problem: Non-recoverable Schedule

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

Rollback

Elements A, B written by T1 are restored to their original value.

Dirty reads of A, B lead to incorrect writes.

Can no longer undo!
Strict 2PL

The Strict 2PL rule:

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

T1

L₁(A); READ(A)
A := A + 100
WRITE(A);

L₁(B); READ(B)
B := B + 100
WRITE(B);
ROLLBACK & U₁(A); U₁(B);

T2

L₂(A); BLOCKED...

L₂(B); READ(B)

…GRANTED; READ(A)
A := A * 2
WRITE(A);
L₂(B); READ(B)
B := B * 2
WRITE(B);
Commit & U₂(A); U₂(B);
Strict 2PL

• Lock-based systems always use strict 2PL
• Easy to implement:
  – Before a transaction reads or writes an element A, insert an L(A)
  – When the transaction commits/aborts, then release all locks
• Ensures both conflict serializability and recoverability
Schedules

- **Recoverable**: whenever a txn commits, all transactions whose values it read have already committed
- **Avoids cascading aborts**: whenever a txn reads an element, the txn that wrote it has already committed
- **Strict**: every value written by a txn T is not read or overwritten* by another txn until after T commits or aborts

*this is the only difference from **avoids cascading aborts**
Strict 2PL

- Every scheduled produced by Strict 2PL is conflict-serializable, and is strict.
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

• Waits-for graph:
  edges \((T_i, T_j)\) if \(T_j\) waits for a lock held by \(T_i\).

• Deadlock = 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:
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) vs. # Active Transactions

- **Throughput (TPS)**
- **# Active Transactions**

**Lock Performance**

TPS = Transactions per second

To avoid **thrashing**, use admission control.
Announcement

Project presentations (see Ed):
- When: Friday, March 12, 9-12
- Zoom link is in Ed
- Schedule: in spreadsheet
- Vote for your favorite project
- Create slides in common presentation
Optimistic concurrency control
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:

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

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

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

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

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

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

Suppose the history was:

\[
\text{START(U), ...,START(T), ..., w}_U(X), ..., r_T(X)\]
Main Idea

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

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

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:

\[
\begin{align*}
&w_U(X) \ldots r_T(X) \\
&r_U(X) \ldots w_T(X) \\
&w_U(X) \ldots w_T(X)
\end{align*}
\]

Suppose the history was:

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

Should we allow this?

\( W_T(X) \leq T_S(T) \)

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:

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

Suppose the history was:

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

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

OK

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:

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

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

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:

\[
\begin{align*}
&w_U(X) \ldots r_T(X) \\
&r_U(X) \ldots w_T(X) \\
&w_U(X) \ldots w_T(X)
\end{align*}
\]

Suppose the history was:

\[
\begin{align*}
&\text{START(U), ...,START(T), ..., w}_U(X), ..., r_T(X) \\
&\text{START(T), ...,START(U), ..., w}_U(X), ..., r_T(X)
\end{align*}
\]

\( WT(X) > TS(T) \)

Should we allow this?

OK

Too late
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 \(r_T(X)\) ?

Request is \(w_T(X)\) ?
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 \( w_T(X) \)
  ?
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 \( w_T(X) \)
- If \( RT(X) > TS(T) \) then ROLLBACK
- Else if \( WT(X) > TS(T) \) ignore write & continue (Thomas Write Rule)
- Otherwise, WRITE and update \( WT(X) = TS(T) \)
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 \( RT(X) > TS(T) \)

Need to rollback T!
Write too late, but we can still handle it:

- T wants to write X, and
  \[ RT(X) \leq TS(T) \text{ but } WT(X) > TS(T) \]

START(T) … START(V) … \[ w_{V}(X) \ldots w_{T}(X) \]

Don’t write X at all!

(Thomas’ rule)
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…

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

If $C(X)=$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 strict
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

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

- 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 \( t \leq TS(T) \)

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 \( X_{t_1} \) and all active transactions \( T \) have \( TS(T) > t_1 \)
Example (in class)

$X_3 \quad X_9 \quad X_{12} \quad X_{18}$

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

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

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

\[ R_6(X) \] -- 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)

$X_3 \quad X_9 \quad X_{12} \quad X_{18}$

$R_6(X)$ -- 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)

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

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

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)

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

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

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?

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)

$X_3 \quad X_9 \quad X_{12} \quad X_{14} \quad X_{18}$

$R_6(X)$ -- 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)

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

\[ R_{6}(X) \quad \text{-- what happens?} \quad \text{Return } X_3 \]
\[ W_{14}(X) \quad \text{-- what happens?} \]
\[ R_{15}(X) \quad \text{-- what happens?} \quad \text{Return } X_{14} \]
\[ W_{5}(X) \quad \text{-- what happens?} \quad \text{ABORT} \]

When can we delete \( X_3 \)? When \( \min \text{TS}(T) \geq 9 \)
Concurrency Control by Validation

Even more optimistic than timestamp validation

- Each transaction T defines a read set $RS(T)$ and a write set $WS(T)$
- Each transaction proceeds in three phases:
  - Read all elements in $RS(T)$. Time = $START(T)$
  - Validate (may need to rollback). Time = $VAL(T)$
  - Write all elements in $WS(T)$. Time = $FIN(T)$

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

\[ \text{START}(U) \]

\[ \text{VAL}(U) \]

\[ \text{FIN}(U) \]

**U:**

- Read phase
- Validate
- Write phase

**T:**

- Read phase
- Validate?

**IF** \( RS(T) \cap WS(U) \) and \( \text{FIN}(U) > \text{START}(T) \)

(\( U \) has validated and \( U \) has not finished before \( T \) begun)

**Then** ROLLBACK(T)
Avoid $w_U(X) - w_T(X)$ Conflicts

START(U) ➔ VAL(U) ➔ FIN(U)

U: Read phase | Validate | Write phase

T: Read phase | Validate ? | Write phase ?

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)
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 $X_{TS(T)}$.

- When T writes X: if other transaction updated X, abort
  - Not faithful to “first committer” rule, because the other transaction U might have committed after T. But once we abort T, U becomes the first committer 😊
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:

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

Starting with $X=50, Y=50$, we end with $X=-50, Y=-50$. Non-serializable !!!
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 *</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>
</tbody>
</table>

**Is this schedule serializable?**
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>

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>
</tbody>
</table>

*R₁(A1); R₁(A2); W₂(A3); 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>
</tbody>
</table>

\[
\begin{align*}
R_1(A1); R_1(A2); W_2(A3); R_1(A1); R_1(A2); R_1(A3) \\
W_2(A3); R_1(A1); R_1(A2); R_1(A1); R_1(A2); R_1(A3)
\end{align*}
\]
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!

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

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

\[ T_1: \text{READ}(A) \]
\[ T_1: A := A + 5 \]
\[ T_1: \text{WRITE}(A) \]

\[ T_2: \text{READ}(A); \]
\[ T_2: A := A \times 1.3 \]
\[ T_2: \text{WRITE}(A); \]

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

T₁: WRITE(A)

T₁: ABORT

T₂: READ(A)
1. Isolation Level: Dirty Reads

Write-Read Conflict

\[ T_1: \ A := 20; \ B := 20; \]
\[ T_1: \ WRITE(A) \]
\[ T_1: \ WRITE(B) \]

\[ T_2: \ READ(A); \]
\[ T_2: \ READ(B); \]

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

T₁: WRITE(A)
    COMMIT

T₂: 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