Lecture 14 and 15 – Transactions: Concurrency Control
Announcements

• Last homework due on Friday

• Last reading assignment next Monday

• Projects
  – Milestones: will send comments by email
  – Posters/demos: next Tuesday, March 6, 10am – 2pm
References

• **Database management systems.**
  Ramakrishnan and Gehrke.
  Third Ed. *Chapters 16 and 17.*
Outline

• Transactions motivation, definition, properties

• Concurrency control and locking

• Optimistic concurrency control
Motivating Example

UPDATE Budget
SET money=money-100
WHERE pid = 1

UPDATE Budget
SET money=money+60
WHERE pid = 2

UPDATE Budget
SET money=money+40
WHERE pid = 3

SELECT sum(money)
FROM Budget

Would like to treat each group of instructions as a unit
Definition

A transaction = one or more operations, single real-world transition

Examples
• Transfer money between accounts
• Purchase a group of products
• Register for a class (either waitlist or allocated)
• ...

CSE 544 - Winter 2018
Transactions

• Major component of database systems
• Critical for most applications; arguably more so than SQL

• Fact: Turing awards to database researchers:
  – Charles Bachman 1973 for CODASYL
  – Edgar Codd 1981 for relational model
  – Jim Gray 1998 for transactions
  – Michael Stonebraker 2015 for postgres
START TRANSACTION

UPDATE Budget
SET money = money - 100
WHERE pid = 1

UPDATE Budget
SET money = money + 60
WHERE pid = 2

UPDATE Budget
SET money = money + 40
WHERE pid = 3

COMMIT
ROLLBACK

• If the application gets to a place where it can’t complete the transaction successfully, it can execute ROLLBACK

• This causes the system to “abort” the transaction

• Database returns to a state without any of the changes made by the transaction
Reasons for Rollback

• User changes their mind ("ctl-C"/cancel)

• Explicit in program, when app program finds a problem
  – e.g., when qty on hand < qty being sold

• System-initiated abort
  – System crash
  – Housekeeping, e.g., due to timeouts, admission control, etc
ACID Properties

- **Atomicity**: Either all changes performed by transaction occur or none occurs.
- **Consistency**: A transaction as a whole does not violate integrity constraints.
- **Isolation**: Transactions appear to execute one after the other in sequence.
- **Durability**: If a transaction commits, its changes will survive failures.
What Could Go Wrong?

• Why is it hard to provide ACID properties?

• Concurrent operations
  – Isolation problems
  – We saw one example earlier

• Failures can occur at any time
  – Atomicity and durability problems
  – Next week

• Transaction may need to abort
What Could Go Wrong

Client 1: \texttt{INSERT INTO SmallProduct(name, price)}
\texttt{SELECT pname, price}
\texttt{FROM Product}
\texttt{WHERE price <= 0.99}

\texttt{DELETE Product}
\texttt{WHERE price <=0.99}

Client 2: \texttt{SELECT count(*)}
\texttt{FROM Product}

\texttt{SELECT count(*)}
\texttt{FROM SmallProduct}

Inconsistent reads
What Could Go Wrong

Client 1:

```
UPDATE Product
SET Price = Price – 1.99
WHERE pname = ‘Gizmo’
```

Client 2:

```
UPDATE Product
SET Price = Price*0.5
WHERE pname='Gizmo'
```

Lost update
What Could Go Wrong

Client 1: UPDATE Account
   SET amount = 1000000
   WHERE number = 1001

Client 2: SELECT Account.amount
          FROM Account
          WHERE Account.number = 1001

Dirty reads
Aborted by system
Summary of What Can Go Wrong

- Concurrent execution problems
  - Write-read conflict: dirty read (includes inconsistent read)
    - A transaction reads a value written by another transaction that has not yet committed
  - Read-write conflict: unrepeateable read
    - A transaction reads the value of the same object twice. Another transaction modifies that value in between the two reads
  - Write-write conflict: lost update
    - Two transactions update the value of the same object. The second one to write the value overwrite the first change

- Failure problems
  - DBMS can crash in the middle of a series of updates
  - Can leave the database in an inconsistent state
ACID Properties

- **Atomicity**: Either all changes performed by transaction occur or none occurs
- **Consistency**: A transaction as a whole does not violate integrity constraints
- **Isolation**: Transactions appear to execute one after the other in sequence
- **Durability**: If a transaction commits, its changes will survive failures
Outline

• Transactions motivation, definition, properties

• Concurrency control and locking

• Optimistic concurrency control
Schedules

- Given multiple transactions

- A *schedule* is a sequence of interleaved actions from all transactions
Example: Two Transactions

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A, s)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>READ(B, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B, t)</td>
<td>WRITE(B, s)</td>
</tr>
</tbody>
</table>
A Serial Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td></td>
</tr>
<tr>
<td>t := t+100</td>
<td></td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td></td>
</tr>
<tr>
<td>READ(B, t)</td>
<td></td>
</tr>
<tr>
<td>t := t+100</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td></td>
</tr>
<tr>
<td>READ(A,s)</td>
<td></td>
</tr>
<tr>
<td>s := s*2</td>
<td></td>
</tr>
<tr>
<td>WRITE(A,s)</td>
<td></td>
</tr>
<tr>
<td>READ(B,s)</td>
<td></td>
</tr>
<tr>
<td>s := s*2</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,s)</td>
<td></td>
</tr>
</tbody>
</table>
Serializable Schedule

- A schedule is **serializable** if it is equivalent to a serial schedule
A Serializable Schedule

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

READ(B, t)

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

Notice:
This is NOT a serial schedule

READ(B,s)
WRITE(B,s)
## A Non-Serializable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A, s)</td>
</tr>
<tr>
<td>t := t + 100</td>
<td>s := s * 2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A, s)</td>
</tr>
<tr>
<td></td>
<td>READ(B, s)</td>
</tr>
<tr>
<td></td>
<td>s := s * 2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B, s)</td>
</tr>
<tr>
<td></td>
<td>READ(B, t)</td>
</tr>
<tr>
<td></td>
<td>t := t + 100</td>
</tr>
<tr>
<td></td>
<td>WRITE(B, t)</td>
</tr>
</tbody>
</table>
Checking Serializability

• Goal: build a scheduler that guarantees serializability

• But how do we know that a schedule is serializable?
  – In general, this is undecidable:
    E.g. T1, T2 compute complex functions, do they commute?

• Two simple sufficient (but not necessary) conditions:
  – Conflict serializability
  – View serializability
Ignoring Details

Capture only the read/write actions
Ignore the computations (assume worst case)

Key Idea: Focus on conflicting operations

\[
T_1: r_1(A); w_1(A); r_1(B); w_1(B) \\
T_2: r_2(A); w_2(A); r_2(B); w_2(B)
\]
Conflict Serializability

Conflicts: (i.e., swapping will change program behavior)

Two actions by same transaction $T_i$: $r_i(X); w_i(Y)$

Two writes by $T_i$, $T_j$ to same element: $w_i(X); w_j(X)$

Read/write by $T_i$, $T_j$ to same element: $w_i(X); r_j(X)$, $r_i(X); w_j(X)$
Conflict Serializability

• A schedule is conflict serializable if it can be transformed into a serial schedule by a series of swappings of adjacent non-conflicting actions

• Every conflict-serializable schedule is serializable
• The converse is not true (why?)
Conflict Serializability

Example:

\[ r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B) \]
Conflict Serializability

Example:

\[ r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B) \]
Conflict Serializability

Example:

```
r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)
```
Conflict Serializability

Example:

\[
\begin{align*}
& r_1(A); \ w_1(A); \ r_2(A); \ \underbrace{w_2(A); \ r_1(B); \ w_1(B); \ r_2(B); \ w_2(B)} \\
& r_1(A); \ w_1(A); \ r_2(A); \ r_1(B); \ w_2(A); \ w_1(B); \ r_2(B); \ w_2(B) \\
& r_1(A); \ w_1(A); \ r_1(B); \ w_1(B); \ r_2(A); \ w_2(A); \ r_2(B); \ w_2(B)
\end{align*}
\]
Conflict Serializability

Example:

\[ r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B) \]
Testing for Conflict-Serializability

Precedence graph:

• A node for each transaction $T_i$,
• An edge from $T_i$ to $T_j$ whenever an action in $T_i$ conflicts with, and comes before an action in $T_j$

• The schedule is conflict-serializable iff the precedence graph is acyclic
Example 1

\[ r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B) \]
Example 1

This schedule is conflict-serializable
Example 2

\[ r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B) \]
Example 2

This schedule is NOT conflict-serializable

\[ r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B) \]
View Equivalence

• A serializable schedule need not be conflict serializable, even under the “worst case update” assumption

\[ w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y); \]

Is this schedule conflict-serializable?
View Equivalence

- A serializable schedule need not be conflict serializable, even under the “worst case update” assumption.

\[
\begin{align*}
w_1(X); & \ w_2(X); \ w_2(Y); \ w_1(Y); \ w_3(Y); \\
\end{align*}
\]

Is this schedule conflict-serializable? No…
View Equivalence

- A serializable schedule need not be conflict serializable, even under the “worst case update” assumption.

\[
w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y);\]

Lost write

\[
w_1(X); w_1(Y); w_2(X); w_2(Y); w_3(Y);\]

Equivalent, but not conflict-equivalent
View Equivalence

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>W1(X)</td>
<td>W1(Y)</td>
<td>CO1</td>
</tr>
<tr>
<td>W2(X)</td>
<td>W2(Y)</td>
<td>CO2</td>
</tr>
<tr>
<td>W1(Y)</td>
<td></td>
<td>CO1</td>
</tr>
</tbody>
</table>
```

Lost

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>W1(X)</td>
<td>W1(Y)</td>
<td>CO1</td>
</tr>
<tr>
<td>W2(X)</td>
<td>W2(Y)</td>
<td>CO2</td>
</tr>
<tr>
<td>W3(Y)</td>
<td></td>
<td>CO3</td>
</tr>
</tbody>
</table>
```

Serializable, but not conflict serializable
Scheduler

• The scheduler is the module that schedules the transaction’s actions, ensuring serializability

• How? We discuss three techniques in class:
  – Locks
  – Timestamps
  – Validation
Outline

- Transactions motivation, definition, properties
- Concurrency control and locking
- Optimistic concurrency control
Locking Scheduler

Simple idea:
- Each element has a unique lock
- Each transaction must first acquire the lock before reading/writing that element
- If lock is taken by another transaction, then wait
- The transaction must release the lock(s)
Notation

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

$u_i(A)$ = transaction $T_i$ releases lock for element $A$
Scheduler has ensured a conflict-serializable schedule
Is this enough?

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

Locks did not enforce conflict-serializability !!!
Two Phase Locking (2PL)

The 2PL rule:

• In every transaction, all lock requests must precede all unlock requests

• This ensures conflict serializability! (why?)
Example: 2PL transactions

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

Now it is conflict-serializable
Example with Multiple Transactions

Equivalent to each transaction executing entirely the moment it enters shrinking phase
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](image)
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:
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?**
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:
- \( U_1(A) \rightarrow L_2(A) \)
- \( L_2(A) \rightarrow U_2(B) \)
- \( U_2(B) \rightarrow L_3(B) \)
- \( L_3(B) \rightarrow U_3(C) \)
- \( U_3(C) \rightarrow L_1(C) \)
- \( L_1(C) \rightarrow U_1(A) \)

Contradiction
A New Problem: Non-recoverable Schedule

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

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>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>Rollback</td>
<td></td>
</tr>
</tbody>
</table>

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>( L_1(A); L_1(B); ) READ(A)</td>
<td>( L_2(A); ) READ(A)</td>
</tr>
<tr>
<td>A := A + 100</td>
<td>A := A \times 2</td>
</tr>
<tr>
<td>WRITE(A); ( U_1(A) )</td>
<td>WRITE(A);</td>
</tr>
<tr>
<td>READ(B)</td>
<td>( L_2(B); ) BLOCKED…</td>
</tr>
<tr>
<td>B := B + 100</td>
<td>( …GRANTED; ) READ(B)</td>
</tr>
<tr>
<td>WRITE(B); ( U_1(B) )</td>
<td>B := B \times 2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B); ( U_2(A); U_2(B) ); 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 \times 2)</td>
</tr>
<tr>
<td>(\text{WRITE}(A); U_1(A))</td>
<td>(\text{WRITE}(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>(\text{WRITE}(B); U_1(B));</td>
<td>(B := B \times 2)</td>
</tr>
<tr>
<td>Rollback</td>
<td>Can no longer undo!</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.
The Strict 2PL rule:

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

With strict 2PL, we will get schedules that are both conflict-serializable and recoverable.
### 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></td>
<td>L₂(B); READ(B)</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></td>
<td></td>
</tr>
<tr>
<td></td>
<td>…GRANTED; READ(A)</td>
</tr>
<tr>
<td></td>
<td>A := A*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(A);</td>
</tr>
<tr>
<td></td>
<td>L₂(B); 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>Commit &amp; U₂(A); U₂(B);</td>
</tr>
</tbody>
</table>
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
Deadlock

• Transaction $T_1$ waits for a lock held by $T_2$;
• $T_2$ waits for $T_3$;
• $T_3$ waits for $T_4$;
• . . .
• $T_n$ waits for $T_1$

• A deadlock is when two or more transactions are waiting for each other to complete
Handling Deadlock

• **Deadlock avoidance**
  – Acquire locks in pre-defined order
  – Acquire all locks at once before starting

• **Deadlock detection**
  – Timeouts (but hard to pick the right threshold)
  – Wait-for graph; this is what commercial systems use (they check graph periodically)
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>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>S</td>
<td>OK</td>
<td>OK</td>
<td>Conflict</td>
</tr>
<tr>
<td>X</td>
<td>OK</td>
<td>Conflict</td>
<td>Conflict</td>
</tr>
</tbody>
</table>

Others:
- **U** = update lock: Initially like S, later may be upgraded to X
- **I** = increment lock (for A := A + something): Increment operations commute
Lock Granularity

• **Fine granularity locking** (e.g., tuples)
  – High concurrency
  – High overhead in managing locks

• **Coarse grain locking** (e.g., tables)
  – Many false conflicts
  – Less overhead in managing locks

• **Alternative techniques**
  – Hierarchical locking (and intentional locks) [commercial DBMSs]
  – Lock escalation
The Tree Protocol

• An alternative to 2PL, for tree structures
• E.g. B+ trees (the indexes of choice in databases)

• Because
  – Indexes are hot spots!
  – 2PL would lead to huge lock contention for the root node
  – Also, unlike data, the index is not directly visible to transactions
  – So only need to guarantee that index returns correct values
The Tree Protocol

Rules:
• A lock on a node A may only be acquired if TXN holds a lock on its parent B
• Nodes can be unlocked in any order (no 2PL necessary)
• Cannot relock a node for which already released a lock
• “Crabbing”
  – First lock parent then lock child
  – Keep parent locked only if may need to update it
  – Release lock on parent if child is not full

• The tree protocol is NOT 2PL, yet ensures conflict-serializability!
• (More in the textbook)
Lock Performance

Throughput vs. # Active Transactions

thrashing

Why?
Phantom Problem

- Static database = a fixed collection of elements (records or blocks)
  - So far we considered serializability only for a static database

- Dynamic database = elements may be inserted/deleted
  - New problem: phantoms
Phantom Problem

Is this schedule serializable?

<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 ('gizmo','blue')</td>
</tr>
</tbody>
</table>
| SELECT * FROM Product WHERE color='blue' | }
## Phantom Problem

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

Suppose there are two blue products, X1, X2:

R1(X1),R1(X2),W2(X3),R1(X1),R1(X2),R1(X3)
Suppose there are two blue products, X₁, X₂:

R₁(X₁), R₁(X₂), W₂(X₃), R₁(X₁), R₁(X₂), R₁(X₃)

This is conflict serializable! What’s wrong??
## Phantom Problem

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

Suppose there are two blue products, X1, X2:

R1(X1),R1(X2),W2(X3),R1(X1),R1(X2),R1(X3)

Not serializable due to **phantoms**
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 view serializability
  implies serializability

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

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

Is expensive!!

• Lock the entire table, or

• Lock the index entry for ‘blue’
  – If index is available

• Or use predicate locks
  – A lock on an arbitrary predicate
Degrees of Isolation

• Isolation level “serializable” (i.e. ACID)
  – Golden standard
  – Requires strict 2PL and predicate locking
  – But often too inefficient
  – Imagine there are only a few update operations and many long read operations

• Weaker isolation levels
  – Sacrifice correctness for efficiency
  – Often used in practice (often default)
  – Sometimes are hard to understand
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

ACID
1. Isolation Level: Dirty Reads

• “Long duration” WRITE locks
  – Strict 2PL
• No READ locks
  – Read-only transactions are never delayed

Possible pbs: dirty and inconsistent reads
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
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
- Deals with phantoms too
Outline

• Transactions motivation, definition, properties

• Concurrency control and locking
  • Optimistic concurrency control
Locking vs Optimistic

• Locking prevents unserializable behavior from occurring: it causes transactions to wait for locks

• Optimistic methods assume no unserializable behavior will occur: they abort transactions if it does

• Locking typically better in case of high levels of contention; optimistic better otherwise
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 recoverable
Main Idea

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

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

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

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

OK

Too late
Timestamps

With each element $X$, associate

- $\text{RT}(X) =$ the highest timestamp of any transaction $U$ that read $X$
- $\text{WT}(X) =$ the highest timestamp of any transaction $U$ that wrote $X$
- $\text{C}(X) =$ the commit bit: true when transaction with highest timestamp that wrote $X$ committed

If element = page, then these are associated with each page $X$ in the buffer pool
Simplified Timestamp-based Scheduling

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 Timestamp-based Scheduling

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

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

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

Read too late:
- T wants to read X, and $TS(T) < WT(X)$

Need to rollback T!
Write too late:
• T wants to write X, and $TS(T) < RT(X)$

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

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

- T wants to write X, and
  \[ TS(T) \geq RT(X) \] but \[ WT(X) > TS(T) \]

START(T) … START(V) … \( w_V(X) \) … \( w_T(X) \)

Don’t write X at all!
(Thomas’ rule)
View-Serializability

- By using Thomas’ rule we do not obtain a conflict-serializable schedule
- Instead, we obtain a view-serializable schedule
Ensuring Recoverable Schedules

- Recall the definition: if 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 $C(X) = \text{false}$, 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 …

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

Request is \( r_T(X) \)
- If \( TS(T) < WT(X) \) 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 \( TS(T) < RT(X) \) then ROLLBACK
- Else if \( TS(T) < WT(X) \)
  - Then If \( C(X) = \text{false} \) then WAIT
  - else IGNORE write (Thomas Write Rule)
- Otherwise, WRITE, and update \( WT(X) = TS(T) \), \( C(X) = \text{false} \)
Summary of Timestamp-based Scheduling

- Conflict-serializable
- Recoverable
  - Even avoids cascading aborts
- Does NOT handle phantoms
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 $t < TS(T)$

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)

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

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

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

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

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

\[ R_6(X) \] -- 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}

\text{TS(T)}=6

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)

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

\[ R_{6}(X) \] -- 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)

\[
\begin{array}{cccccc}
X_3 & X_9 & X_{12} & X_{14} & X_{18} \\
\end{array}
\]

\(R_6(X)\) -- 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)

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

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

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

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

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

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

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

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

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\ $X_3$? \ When\ $\max TS(T) \geq 9$
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

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

U: Read phase  Validate  Write phase

T: Read phase  Validate?

START(T)

IF $\text{RS}(T) \cap \text{WS}(U)$ 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

Beginnings:
- START(U)
- START(T)

Validations:
- VAL(U)
- VAL(T)

Finishes:
- FIN(U)

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)

**Diagram**: U's and T's processes are shown with key phases:
- Read phase
- Validate
- Write phase

**Conflicts**: Arrows indicate potential conflict scenarios between U and 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 transactions 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: $X_{t1}$, $X_{t2}$, $X_{t3}$, ... 

- 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

• Recent extension of SI to make it serializable was implemented in postgres