Introduction to Database Systems
CSE 444

Lecture 11
Transactions: concurrency control
(part 1)
Outline

• Serial and Serializable Schedules (18.1)
• Conflict Serializability (18.2)
• Locks (18.3)
The Problem

• Multiple transactions are running concurrently $T_1, T_2, \ldots$

• They read/write some common elements $A_1, A_2, \ldots$

• How can we prevent unwanted interference?
• The SCHEDULER is responsible for that
Some Famous Anomalies

• What could go wrong if we didn’t have concurrency control:
  – Dirty reads (including inconsistent reads)
  – Unrepeatable reads
  – Lost updates

Many other things can go wrong too
Dirty Reads

Write-Read Conflict

$T_1$: WRITE(A)

$T_1$: ABORT

$T_2$: READ(A)
Inconsistent Read

Write-Read Conflict

$T_1: A := 20; B := 20;$
$T_1: WRITE(A)$
$T_1: WRITE(B)$

$T_2: READ(A);$  
$T_2: READ(B);$
Unrepeatable Read

Read-Write Conflict

T₁: WRITE(A)

T₂: READ(A);

T₂: READ(A);
Lost Update

Write-Write Conflict

$T_1$: READ($A$)

$T_1$: $A := A + 5$

$T_1$: WRITE($A$)

$T_2$: READ($A$); 

$T_2$: $A := A \times 1.3$

$T_2$: WRITE($A$);
Schedules

• Given multiple transactions
• A schedule is a sequence of interleaved actions from all transactions
Example

<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>WRITE(A, t)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>READ(B, t)</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>READ(B, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>READ(B, s)</td>
</tr>
<tr>
<td>WRITE(B, t)</td>
<td>WRITE(B, s)</td>
</tr>
</tbody>
</table>

READ(A,s)

s := s*2

WRITE(A,s)

READ(B,s)

s := s*2

WRITE(B,s)
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>
<tr>
<td></td>
<td></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>

Notice:
This is NOT a serial schedule
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>
Ignoring Details

- Sometimes transactions’ actions can commute accidentally because of specific updates
  - Serializability is undecidable!

- Scheduler should not look at transaction details

- Assume worst case updates
  - Only care about reads \( r(A) \) and writes \( w(A) \)
  - Not the actual values involved
Notation

\begin{align*}
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)
\end{align*}
Conflict Serializability

Conflicts:

Two actions by same transaction $T_i$:

Two writes by $T_i$, $T_j$ to same element

Read/write by $T_i$, $T_j$ to same element

$r_i(X); w_i(Y)$

$w_i(X); w_j(X)$

$w_i(X); r_j(X)$

$r_i(X); w_j(X)$
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

Example:

\[
\begin{align*}
&\text{r}_1(\text{A}); \; \text{w}_1(\text{A}); \; \text{r}_2(\text{A}); \; \text{w}_2(\text{A}); \; \text{r}_1(\text{B}); \; \text{w}_1(\text{B}); \; \text{r}_2(\text{B}); \; \text{w}_2(\text{B}) \\
\end{align*}
\]
The Precedence Graph Test

Is a schedule conflict-serializable?
Simple test:
• Build a graph of all transactions $T_i$

• Edge from $T_i$ to $T_j$ if $T_i$ makes an action that conflicts with one of $T_j$ and comes first

• The test: if the graph has no cycles, then it is conflict serializable!
Example 1

\begin{align*}
    r_2(A) &; r_1(B) &; w_2(A) &; r_3(A) &; w_1(B) &; w_3(A) &; r_2(B) &; w_2(B)
\end{align*}

This schedule is conflict-serializable
Example 2

\[ \text{r}_2(A); \text{r}_1(B); \text{w}_2(A); \text{r}_2(B); \text{r}_3(A); \text{w}_1(B); \text{w}_3(A); \text{w}_2(B) \]

This schedule is NOT conflict-serializable
Conflict Serializability

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

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

Lost write

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

Equivalent, but can’t swap
Scheduler

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

• How? We discuss three techniques in class:
  – Locks
  – Time stamps (next lecture)
  – Validation (next lecture)
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 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 $A$
Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>L_1(A); READ(A, t)</code></td>
<td><code>L_2(A); READ(A,s)</code></td>
</tr>
<tr>
<td><code>t := t+100</code></td>
<td><code>s := s*2</code></td>
</tr>
<tr>
<td><code>WRITE(A, t); U_1(A); L_1(B)</code></td>
<td><code>WRITE(A,s); U_2(A); L_2(B); DENIED...</code></td>
</tr>
</tbody>
</table>

`READ(B, t)`

`t := t+100`

`WRITE(B,t); U_1(B);`

`...GRANTED; READ(B,s)`

`s := s*2`

`WRITE(B,s); U_2(B);`

Scheduler has ensured a conflict-serializable schedule
Example

T1
L₁(A); READ(A, t)
t := t+100
WRITE(A, t); U₁(A);

L₂(A); READ(A,s)
s := s*2
WRITE(A,s); U₂(A);
L₂(B); READ(B,s)
s := s*2
WRITE(B,s); U₂(B);

T2

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

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>L_1(A); L_1(B); READ(A, t)</td>
<td>L_2(A); 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);</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>L_2(B); DENIED…</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B, t); U_1(B);</td>
<td>WRITE(B, s);</td>
</tr>
</tbody>
</table>

Now it is conflict-serializable

…GRANTED; READ(B, s) | WRITE(B, s); U_2(A); U_2(B);
What about Aborts?

• 2PL enforces conflict-serializable schedules
• But what if a transaction releases its locks and then aborts?

• Serializable schedule definition only considers transactions that commit
  – Relies on assumptions that aborted transactions can be undone completely
Example with Abort

T1

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

READ(B, t)
t := t+100
WRITE(B, t); U₁(B);

Abort

T2

L₂(A); READ(A, s)
s := s*2
WRITE(A, s);
L₂(B); DENIED...

...GRANTED; READ(B, s)
s := s*2
WRITE(B, s); U₂(A); U₂(B);

Commit
Strict 2PL

- Strict 2PL: All locks held by a transaction are released when the transaction is completed

- Ensures that schedules are recoverable
  - Transactions commit only after all transactions whose changes they read also commit

- Avoids cascading rollbacks
Deadlock

- Transaction $T_1$ waits for a lock held by $T_2$;
- But $T_2$ waits for a lock held by $T_3$;
- While $T_3$ waits for . . . .
- . . .
- . . . and $T_{73}$ waits for a lock held by $T_1$ !!

- Could be avoided, by ordering all elements (see book); or deadlock detection + rollback
Lock Modes

• $S$ = shared lock (for READ)
• $X$ = exclusive lock (for WRITE)
• $U$ = update lock
  – Initially like $S$
  – Later may be upgraded to $X$
• $I$ = increment lock (for $A := A + \text{something}$)
  – Increment operations commute

Recommended reading: chapter 18.4
The Locking Scheduler

Taks 1:
  Add lock/unlock requests to transactions
• Examine all READ(A) or WRITE(A) actions
• Add appropriate lock requests
• Ensure 2PL!

Recommended reading: chapter 18.5
The Locking Scheduler

Task 2:
- Execute the locks accordingly
  - Lock table: a big, critical data structure in a DBMS!
  - When a lock is requested, check the lock table
    - Grant, or add the transaction to the element’s wait list
  - When a lock is released, re-activate a transaction from its wait list
  - When a transaction aborts, release all its locks
  - Check for deadlocks occasionally

Recommended reading: chapter 18.5