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

T₁: WRITE(A)

T₂: READ(A)

T₁: ABORT
Inconsistent Read

Write-Read Conflict

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

T2: READ(A);
T2: READ(B);
Unrepeatable Read

Read-Write Conflict

T₁: WRITE(A)

T₂: READ(A);

T₂: READ(A);
Lost Update

Write-Write Conflict

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

\[ T_2: \text{READ}(A); \]
\[ T_2: A := A \times 1.3 \]
\[ T_2: \text{WRITE}(A); \]
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(B, t)</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>WRITE(A, s)</td>
</tr>
<tr>
<td>s := s*2</td>
<td>WRITE(B, s)</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>
<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>READ(B, t)</td>
<td>READ(B, t)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>t := t+100</td>
</tr>
<tr>
<td>WRITE(B,t)</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

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

Conflicts:

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 $w_i(X); r_j(X)$
$r_i(X); w_j(X)$
Conflicted 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:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$
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

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

This schedule is conflict-serializable
Example 2

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>L₁(A); 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); L₁(B)</td>
<td>WRITE(A,s); U₂(A); L₂(B); DENIED</td>
</tr>
<tr>
<td><strong>READ(B, t)</strong></td>
<td><strong>...GRANTED; READ(B,s)</strong></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(B);</td>
</tr>
</tbody>
</table>

Scheduler has ensured a conflict-serializable schedule.
## Example

<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>$s := s \times 2$</td>
</tr>
<tr>
<td>WRITE($A,t$); $U_1(A)$;</td>
<td>WRITE($A,s$); $U_2(A)$;</td>
</tr>
<tr>
<td>$L_1(B); \text{READ}(B,t)$</td>
<td>$L_2(B); \text{READ}(B,s)$</td>
</tr>
<tr>
<td>$t := t + 100$</td>
<td>$s := s \times 2$</td>
</tr>
<tr>
<td>WRITE($B,t$); $U_1(B)$;</td>
<td>WRITE($B,s$); $U_2(B)$;</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

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);

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);

Now it is conflict-serializable
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

<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>READ(B, t)</td>
<td>L₂(B); DENIED...</td>
</tr>
<tr>
<td>t := t + 100</td>
<td></td>
</tr>
<tr>
<td>WRITE(B, t); U₁(B)</td>
<td></td>
</tr>
<tr>
<td>Abort</td>
<td>...GRANTED; READ(B, s)</td>
</tr>
<tr>
<td></td>
<td>s := s * 2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B, s); U₂(A); U₂(B);</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>
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 + something)
  - Increment operations commute

Recommended reading: chapter 18.4
The Locking Scheduler

Task 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