Lecture 17: Concurrency Control

Friday, February 17, 2006

Outline

• Serial and Serializable Schedules (18.1)
• Conflict Serializability (18.2)
• Locks (18.3)
• Multiple lock modes (18.4)
• The tree protocol (18.7)
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

Three Famous Anomalies

What can go wrong if we didn’t have concurrency control:

• Dirty reads
• Lost updates
• Inconsistent reads

Many other things may go wrong, but have no names
Dirty Reads

T₁: WRITE(A)
T₁: ABORT

T₂: READ(A)

Lost Update

T₁: READ(A)
T₁: A := A+5
T₁: WRITE(A)

T₂: READ(A);
T₂: A := A*1.3
T₂: WRITE(A);
Inconsistent Read

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

T₂: READ(A);
T₂: READ(B);

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></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>
<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

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

T2
READ(A,s)
s := s*2
WRITE(A,s)
READ(B,s)
s := s*2
WRITE(B,s)

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

Ignoring Details

• Sometimes transactions’ actions may commute accidentally because of specific updates
  – Serializability is undecidable!
• The scheduler shouldn’t look at the transactions’ details
• Assume worst case updates, only care about reads r(A) and writes w(A)
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
\[ w_i(X); r_j(X) \]
\[ r_i(X); w_j(X) \]
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:

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

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

Conflict Serializability

• Any conflict serializable schedule is also a serializable schedule (why?)

• The converse is not true, even under the “worst case update” assumption

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

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

This schedule is conflict-serializable
Example 2

\[\text{Example 2}\]

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

This schedule is NOT conflict-serializable

Scheduler

- The scheduler is the module that schedules the transaction’s actions, ensuring serializability
- How? Three techniques:
  - Locks
  - Time stamps
  - Validation
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

\[ l_i(A) = \text{transaction } T_i \text{ acquires lock for element } A \]
\[ u_i(A) = \text{transaction } T_i \text{ releases lock for element } A \]
### Example

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

The scheduler has ensured a conflict-serializable schedule

---

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1(A)$; READ(A, t)</td>
<td></td>
</tr>
<tr>
<td>$t := t + 100$</td>
<td>$L_2(A)$; READ(A, s)</td>
</tr>
<tr>
<td>WRITE(A, t); $U_1(A)$;</td>
<td>$s := s \times 2$</td>
</tr>
<tr>
<td>$L_1(B)$; READ(B, t)</td>
<td>WRITE(A, s); $U_2(A)$</td>
</tr>
<tr>
<td>$t := t + 100$</td>
<td>$L_2(B)$; READ(B, s)</td>
</tr>
<tr>
<td>WRITE(B, t); $U_1(B)$;</td>
<td>$s := s \times 2$</td>
</tr>
<tr>
<td></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 preceed 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>READ(B, t)</td>
<td></td>
</tr>
<tr>
<td>t := t+100</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t); U₁(B);</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Now it is conflict-serializable</td>
<td>...GRANTED; READ(B,s)</td>
</tr>
</tbody>
</table>
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 plus 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
- READ 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!

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
The Tree Protocol

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

Rules:
- The first lock may be any node of the tree
- Subsequently, a lock on a node A may only be acquired if the transaction holds a lock on its parent B
- Nodes can be unlocked in any order (no 2PL necessary)

The tree protocol is NOT 2PL, yet ensures conflict-serializability!