Introduction to Database Systems
CSE 444

Lectures 17-18:
Concurrency Control

November 5-7, 2007

Outline

• Serial and Serializable Schedules (18.1)
• Conflict Serializability (18.2)
• Locks (18.3)
• Multiple lock modes (18.4)
• The tree protocol (18.7)
• Concurrency control by timestamps 18.8
• Concurrency control by validation 18.9

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

$T_1$: WRITE($A$)  
$T_1$: ABORT

$T_2$: READ($A$)

Lost Update

$T_1$: READ($A$)  
$T_1$: $A := A + 5$

$T_2$: WRITE($A$)

$T_2$: READ($A$);  
$T_2$: $A := A \times 1.3$

$T_1$: WRITE($A$)  
$T_2$: WRITE($A$);
Inconsistent Read

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

$T_1$: WRITE(B)

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

Schedules

Given multiple transactions:
- A **schedule** is a sequence of interleaved actions from all transactions
- A **serial schedule** is one whose actions consist of all those of one transaction, followed by all those of another transaction, etc.

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>t := t+100</td>
</tr>
<tr>
<td>t := t+100</td>
<td>WRITE(A, t)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>WRITE(B, t)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>READ(A, s)</td>
</tr>
<tr>
<td>s := s*2</td>
<td>WRITE(A, s)</td>
</tr>
<tr>
<td>READ(B, s)</td>
<td>WRITE(B, s)</td>
</tr>
<tr>
<td>s := s*2</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>t := t+100</td>
</tr>
<tr>
<td>t := t+100</td>
<td>WRITE(A, t)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>WRITE(B, t)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>READ(A, s)</td>
</tr>
<tr>
<td>s := s*2</td>
<td>WRITE(A, s)</td>
</tr>
<tr>
<td>READ(B, s)</td>
<td>WRITE(B, s)</td>
</tr>
<tr>
<td>s := s*2</td>
<td></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, 0)</td>
<td>READ(A, s)</td>
</tr>
<tr>
<td>( t := t + 100 )</td>
<td>( s := s^2 )</td>
</tr>
<tr>
<td>WRITE(A, 0)</td>
<td>WRITE(A, s)</td>
</tr>
<tr>
<td></td>
<td>( s := s^2 )</td>
</tr>
<tr>
<td>READ(B, 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE(B, s)</td>
</tr>
</tbody>
</table>

## 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

- 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:*  

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

  \[
  \Downarrow
  \]

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

## Conflict Serializability

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

  *Example:*  

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

  \[
  \Downarrow
  \]

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

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_4(B)$

This schedule is conflict-serializable

Example 2

$r_2(A); r_1(B); w_2(A); r_4(B); r_5(A); w_1(B); w_3(A); w_4(B)$

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

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

U_i(A) =$ transaction $T_i$ releases lock for element $A
Example

**T1**

L₁(A); READ(A, t)

`t := t + 100`

WRITE(A, t); U₁(A); L₁(B)

L₂(A); READ(A, s)

`s := s * 2`

WRITE(A, s); U₂(A);

L₂(B); DENIED…

READ(B, t)

`t := t + 100`

WRITE(B, t); U₁(B);

DENIED…

READ(B, s)

`s := s * 2`

WRITE(B, s); U₂(B);

**T2**

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, t)

`t := t + 100`

WRITE(B, t); U₁(B);

DENIED…

READ(B, s)

`s := s * 2`

WRITE(B, s); U₂(B);

**T2**

DENIED…

READ(B, s)

`s := s * 2`

WRITE(B, s); U₂(A); U₂(B);

L₁(B); READ(B, t)

`t := t + 100`

WRITE(B, t); U₁(B);

GRANTED;

READ(B, s)

`s := s * 2`

WRITE(B, s); U₂(B);

The scheduler has ensured a conflict-serializable schedule

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

**T1**

L₁(A); L₂(B); 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); DENIED…

READ(B, t)

`t := t + 100`

WRITE(B, t); U₁(B);

…GRANTED; 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);

…GRANTED; READ(B, s)

`s := s * 2`

WRITE(B, s); U₂(A); U₂(B);

Now it is conflict-serializable

Deadlock

- Transaction T₁ waits for a lock held by T₂;
- But T₂ waits for a lock held by T₃;
- While T₃ waits for . . . . .
- . . .
- . . .and T₇₃ waits for a lock held by T₁!!

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!

Timestamps

Every transaction receives a unique timestamp \(TS(T)\)

Could be:
- The system’s clock
- A unique counter, incremented by the scheduler

Main invariant:
- The timestamp order defines the serialization order of the transaction
**Timestamps**

Associate to each element X:
- \( RT(X) \) = the highest timestamp of any transaction that read X
- \( WT(X) \) = the highest timestamp of any transaction that wrote X
- \( C(X) \) = the commit bit: says if the transaction with highest timestamp that wrote X committed

These are associated to each page X in the buffer pool.

**Main Idea**

For any two conflicting actions, ensure that their order is the serialized order:

In each of these cases:
- \( w_U(X) \ldots r_T(X) \)
- \( r_U(X) \ldots w_T(X) \)
- \( w_U(X) \ldots w_T(X) \)

Check that \( TS(U) < TS(T) \)

When \( T \) wants to read X, \( r_T(X) \), how do we know U, and \( TS(U) \)?

**Details**

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

\[ \text{START}(T) \ldots \text{START}(U) \ldots w_U(X) \ldots r_T(X) \]

Need to rollback T!

**Write too late:**
- \( T \) wants to write X, and
- \( WT(X) < TS(T) < RT(X) \)

\[ \text{START}(T) \ldots \text{START}(U) \ldots r_U(X) \ldots w_T(X) \]

Need to rollback T!

Why do we check \( WT(X) < TS(T) \)?

**Details**

**Write too late, but we can still handle it:**
- \( T \) wants to write X, and
- \( TS(T) < RT(X) \) but \( WT(X) > TS(T) \)

\[ \text{START}(T) \ldots \text{START}(V) \ldots w_V(X) \ldots w_T(X) \]

Don’t write X at all!
(but see later…)

**More Problems**

**Read dirty data:**
- \( T \) wants to read X, and \( WT(X) < TS(T) \)
- Seems OK, but…

\[ \text{START}(U) \ldots \text{START}(T) \ldots w_U(X) \ldots r_T(X) \ldots \text{ABORT}(U) \]

If \( C(X) = 1 \), then T needs to wait for it to become 0
More Problems

Write dirty data:
• T wants to write X, and WT(X) > TS(T)
• Seems OK not to write at all, but …

START(T) … START(U) … w_T(X) … w_T(X) … ABORT(U)

If C(X)=1, then T needs to wait for it to become 0

Tensor-based Scheduling

When a transaction T requests r(X) or w(X), the scheduler examines RT(X), WT(X), C(X), and decides one of:
• To grant the request, or
• To rollback T (and restart with later timestamp)
• To delay T until C(X) = 0

Multiversion Timestamp

• When transaction T requests r(X) but WT(X) > TS(T), then T must rollback
• Idea: keep multiple versions of X: X_0, X_1, X_2, …

| TS(X_0) > TS(X_1) > TS(X_2) > … |

• 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 a version X_t such that t < TS(T) and t is the largest such
• WT(X_t) = t and it never changes
• RD(X_t) must also be maintained, to reject certain writes (why?)
• When can we delete X_t: if we have a later version X_{t+1} and all active transactions T have TS(T) > t+1

Tradeoffs

• Locks:
  – Great when there are many conflicts
  – Poor when there are few conflicts
• Timestamps
  – Poor when there are many conflicts (rollbacks)
  – Great when there are few conflicts
• Compromise
  – READ ONLY transactions → timestamps
  – READ/WRITE transactions → locks
Concurrency Control by 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 = \( \text{START}(T) \)
  - Validate (may need to rollback). Time = \( \text{VAL}(T) \)
  - Write all elements in \( WS(T) \). Time = \( \text{FIN}(T) \)

Main invariant: the serialization order is \( \text{VAL}(T) \)

Avoid \( r_T(X) - w_U(X) \) Conflicts

\[
\begin{align*}
\text{START}(U) & \quad \text{VAL}(U) & \quad \text{FIN}(U) \\
\text{T: Read phase} & \quad \text{Validate} & \quad \text{Write phase}
\end{align*}
\]

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 \( w_T(X) - w_U(X) \) Conflicts

\[
\begin{align*}
\text{START}(U) & \quad \text{VAL}(U) & \quad \text{FIN}(U) \\
\text{T: Read phase} & \quad \text{Validate} & \quad \text{Write phase}
\end{align*}
\]

IF \( WS(T) \cap WS(U) \) and \( \text{FIN}(U) > \text{VAL}(T) \) (\( U \) has validated and \( U \) has not finished before \( T \) validates) Then ROLLBACK(\( T \))

Final comments

- Locks and timestamps: SQL Server, DB2
- Validation: Oracle

(more or less)