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)

Next time:
• Concurrency control by timestamps (18.8)
• Concurrency control by validation (18.9)

Some additional material not in the book
The Problem

- Multiple transactions running concurrently $T_1, T_2, \ldots$
- They read/write 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
  – Inconsistent reads
  – Unrepeateable reads
  – Lost updates

Many other things can go wrong too
Conflicts

- Write-Read – WR
- Read-Write – RW
- Write-Write – WW
Dirty Reads

Write-Read Conflict

T₁: WRITE(A)

T₂: READ(A)

T₁: ABORT
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

$T_1$: WRITE(A)

$T_2$: READ(A);

$T_2$: READ(A);
Lost Update

Write-Write Conflict

T₁: READ(A)
T₁: A := A + 5
T₁: WRITE(A)

T₂: READ(A);
T₂: A := A * 1.3
T₂: WRITE(A);
Schedules

• Given multiple transactions
  – A schedule is a sequence of interleaved actions from all transactions
  – A serial schedule is one in which transactions appear one after the other in some order with no overlap
## 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>READ(A, s)</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>s := s*2</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>WRITE(A, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>WRITE(B, s)</td>
</tr>
<tr>
<td>WRITE(B, t)</td>
<td>s := s*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B, s)</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)     | READ(B,s)                         |
| t := t+100     | s := s*2                          |
| WRITE(B,t)     | WRITE(B,s)                        |

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></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td></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 $T_i$:

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

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

$\begin{align*}
\text{r}_i(X); & \text{w}_i(Y) \\
\text{w}_i(X); & \text{w}_j(X) \\
\text{w}_i(X); & \text{r}_j(X) \\
\text{r}_i(X); & \text{w}_j(X)
\end{align*}$
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:

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

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

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

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

Lost write

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

Equivalent, but can’t swap
View Equivalent

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1(X)</td>
<td>W2(X)</td>
<td>W1(Y)</td>
</tr>
<tr>
<td>W2(Y)</td>
<td>CO2</td>
<td>CO1</td>
</tr>
<tr>
<td>W1(Y)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Lost**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1(X)</td>
<td>W2(X)</td>
<td>W3(Y)</td>
</tr>
<tr>
<td>W1(Y)</td>
<td>W2(Y)</td>
<td>CO2</td>
</tr>
<tr>
<td>CO1</td>
<td></td>
<td>CO3</td>
</tr>
</tbody>
</table>

Serializable, but not conflict serializable
View Equivalence

Two schedules $S$, $S'$ are *view equivalent* if:

- If $T$ reads an initial value of $A$ in $S$, then $T$ also reads the initial value of $A$ in $S'$
- If $T$ reads a value of $A$ written by $T'$ in $S$, then $T$ also reads a value of $A$ written by $T'$ in $S'$
- If $T$ writes the final value of $A$ in $S$, then it writes the final value of $A$ in $S'$
Schedules with Aborted Transactions

• When a transaction aborts, the recovery manager undoes its updates
• But some of its updates may have affected other transactions!
Schedules with Aborted Transactions

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
<tr>
<td></td>
<td>Abort</td>
</tr>
</tbody>
</table>

Cannot abort T1 because cannot undo T2
Recoverable Schedules

• A schedule is *recoverable* if whenever a transaction T commits, all transactions who have written elements read by T have already committed
Recoverable Schedules

\[
\begin{array}{c|c|c|c|c}
\text{T1} & \text{T2} & \text{R(A)} & \text{W(A)} \\
\hline
\text{R(A)} & \text{W(A)} & \text{R(A)} & \text{W(A)} \\
\text{W(A)} & \text{R(B)} & \text{W(B)} & \text{Commit} \\
\text{Abort} & \text{Abort} & \text{Commit} & \text{Commit} \\
\end{array}
\]
Cascading Aborts

- If a transaction T aborts, then we need to abort any other transaction T’ that has read an element written by T

- A schedule is said to avoid cascading aborts if whenever a transaction read an element, the transaction that has last written it has already committed.
Avoiding Cascading Aborts

T1    T2
R(A)   R(A)
W(A)   W(A)

R(A)   R(A)
W(A)   W(A)

R(B)   R(B)
W(B)   W(B)

R(A)   R(A)
W(A)   W(A)

R(B)   R(B)
W(B)   W(B)

With cascading aborts

Without cascading aborts
Review of Schedules

Serializability
- Serial
- Serializable
- Conflict serializable
- View equivalent to serial

Recoverability
- Recoverable
- Avoiding cascading deletes
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)
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>$L_1(A); \text{READ}(A, t)$</td>
<td>$L_2(A); \text{READ}(A, s)$</td>
</tr>
<tr>
<td>$t := t+100$</td>
<td>$s := s\times2$</td>
</tr>
<tr>
<td>WRITE(A, t); U_1(A); L_1(B)</td>
<td>WRITE(A, s); U_2(A); L_2(B)</td>
</tr>
<tr>
<td>DENIED</td>
<td>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); U_1(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>\textbf{…GRANTED}; \text{READ}(B, s)</td>
</tr>
<tr>
<td></td>
<td>$s := s\times2$</td>
</tr>
<tr>
<td></td>
<td>WRITE(B, s); U_2(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

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1(A); L_1(B); \text{READ}(A, t)$</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);</td>
</tr>
<tr>
<td>$L_2(B); \text{DENIED...}$</td>
<td>$L_2(B); \text{DENIED...}$</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); $U_1(B)$;</td>
<td>WRITE(B, s); $U_2(A); U_2(B)$;</td>
</tr>
</tbody>
</table>

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
A Non-Recoverable Schedule

<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>
</tbody>
</table>

READ(B, t)
| WRITE(B,t); U₁(B); | ...GRANTED; READ(B,s) |
| s := s*2          | WRITE(B,s); U₂(A); U₂(B); |
|                   | Commit       |

Abort
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
Lock Granularity

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

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

- **Alternative techniques**
  - Hierarchical locking (and intentional locks) [commercial DBMSs]
  - Lock escalation
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!
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