Introduction to Database Systems
CSE 444

Lecture 13
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

\( T_1: \text{WRITE}(A) \)
\( T_1: \text{ABORT} \)

\( T_2: \text{READ}(A) \)

Inconsistent Read

\( T_1: A := 20; B := 20; \)
\( T_1: \text{WRITE}(A) \)
\( T_1: \text{WRITE}(B) \)

\( T_2: \text{READ}(A); T_2: \text{READ}(B); \)
Unrepeatable Read

Read-Write Conflict

T₁: WRITE(A)

T₂: READ(A);

T₂: 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

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

T₁

T₂

READ(A, t)

t := t + 100

WRITE(A, t)

READ(B, t)

t := t + 100

WRITE(B, t)

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

T1
READ(A, t)
t := t+100
WRITE(A, t)
READ(A, s)
s := s*2
WRITE(A, s)
READ(B, t)
t := t+100
WRITE(B, t)

Notice:
This is NOT a serial schedule

A Non-Serializable Schedule

T1
READ(A, t)
t := t+100
WRITE(A, t)
READ(A, s)
s := s*2
WRITE(A, s)
READ(B, s)
s := s*2
WRITE(B, s)

T2
READ(B, t)
t := t+100
WRITE(B, t)

Notice:
This is NOT a serial schedule

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

T1: r1(A); w1(A); r1(B); w1(B)
T2: r2(A); w2(A); r2(B); w2(B)

Conflict Serializability

Conflicts:
Two actions by same transaction Ti: r(X); w(Y)
Two writes by Ti, Tj to same element: w(X); w(X)
Read/write by Ti, Tj to same element: w(X); r(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:

- Initial schedule:
  r1(A); w1(A); r2(A); w2(A); r1(B); w1(B); r2(B); w2(B)

- Swapping r1(A) and w1(A): r1(A); w1(A); r2(A); w2(A); r1(B); w1(B); r2(B); w2(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 $T_i$ and comes first
• The test: if the graph has no cycles, then it is conflict serializable!

Example 1

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

This schedule is conflict-serializable

Example 2

$\text{r}_4(\text{A}); \text{r}_4(\text{B}); \text{w}_2(\text{A}); \text{r}_3(\text{A}); \text{w}_1(\text{B}); \text{w}_3(\text{A}); \text{w}_2(\text{B})$

This schedule is NOT conflict-serializable

Conflict Serializability

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

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>( 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^*2 )</td>
</tr>
<tr>
<td>( \text{WRITE}(A, t) ); ( u_1(A) ); ( l_1(B) )</td>
<td>( \text{WRITE}(A, s) ); ( u_1(A) ); ( l_2(B) ); ( \text{DENIED} \ldots )</td>
</tr>
<tr>
<td>( \text{READ}(B, t) )</td>
<td>( \text{READ}(B, s) )</td>
</tr>
<tr>
<td>( t := t+100 )</td>
<td>( s := s^*2 )</td>
</tr>
<tr>
<td>( \text{WRITE}(B, t) ); ( u_1(B) )</td>
<td>( \text{WRITE}(B, s) ); ( u_2(B) )</td>
</tr>
</tbody>
</table>

Scheduler has ensured a conflict-serializable schedule

---

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>( \text{WRITE}(A, t) ); ( u_1(A) )</td>
<td>( \text{WRITE}(A, s) ); ( u_2(A) ); ( l_2(B) ); ( \text{DENIED} \ldots )</td>
</tr>
<tr>
<td>( \text{READ}(B, t) )</td>
<td>( \text{READ}(B, s) )</td>
</tr>
<tr>
<td>( t := t+100 )</td>
<td>( s := s^*2 )</td>
</tr>
<tr>
<td>( \text{WRITE}(B, t) ); ( u_1(B) )</td>
<td>( \text{WRITE}(B, s) ); ( u_2(B) )</td>
</tr>
</tbody>
</table>

Now it is conflict-serializable

---

Two Phase Locking (2PL)

The 2PL rule:

- In every transaction, all lock requests must proceed all unlock requests
- This ensures conflict serializability! (why?)

---

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); L2(B); READ(A, t)
t := t + 100
WRITE(A, t); U1(A)

T2
L2(A); READ(A, s)
s := s + 2
WRITE(A, s); L2(B); DENIED...

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

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 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 + 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