Introduction to Database Systems
CSE 414

Lecture 24:
Implementation of Transactions

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

• Every conflict-serializable schedule is serializable

• The converse is not true (why?)

Testing for Conflict-Serializability

Precedence graph:
• A node for each transaction Ti,
• An edge from Ti to Tj whenever an action in Ti conflicts with, and comes before an action in Tj

• The schedule is conflict-serializable iff the precedence graph is acyclic

Example 2

This schedule is NOT conflict-serializable

More Notations

L(A) = transaction Ti acquires lock for element A
U(A) = transaction Ti releases lock for element A

A Non-Serializable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A)</td>
<td>READ(A)</td>
</tr>
<tr>
<td>A := A+100</td>
<td>A := A*2</td>
</tr>
<tr>
<td>WRITE(A)</td>
<td>WRITE(A)</td>
</tr>
<tr>
<td></td>
<td>READ(B)</td>
</tr>
<tr>
<td></td>
<td>B := B+100</td>
</tr>
<tr>
<td></td>
<td>WRITE(B)</td>
</tr>
</tbody>
</table>
Two Phase Locking (2PL)

The 2PL rule:

In every transaction, all lock requests must precede all unlock requests

Example: 2PL transactions

Example

\[ T_1: \text{READ}(A) \]
\[ A := A + 100 \]
\[ \text{WRITE}(A); U_1(A); L_1(B) \]
\[ \text{READ}(B) \]
\[ B := B + 100 \]
\[ \text{WRITE}(B); U_1(B); \]

But…

\[ T_1: \text{READ}(A) \]
\[ A := A + 100 \]
\[ \text{WRITE}(A); U_1(A); L_1(B) \]
\[ \text{READ}(B) \]
\[ B := B + 100 \]
\[ \text{WRITE}(B); U_1(B); \]

Scheduler has ensured a conflict-serializable schedule

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.
**Two Phase Locking (2PL)**

**Theorem:** 2PL ensures conflict serializability

**Proof.** Suppose not: then there exists a cycle in the precedence graph.

Then there is the following temporal cycle in the schedule:

1. $U_1(A) \rightarrow L_2(A)$
2. $L_2(A) \rightarrow U_2(B)$
3. $U_2(B) \rightarrow L_3(B)$

why?

$L_2(A)$ happened strictly before $U_1(A)$
Two Phase Locking (2PL)

**Theorem:** 2PL ensures conflict serializability

**Proof.** Suppose not: then there exists a cycle in the precedence graph.

Then there is the following temporal cycle in the schedule:

- \( U_1(A) \rightarrow L_2(A) \)
- \( L_2(A) \rightarrow U_2(B) \)
- \( U_2(B) \rightarrow L_3(B) \)
- \( L_3(B) \rightarrow U_1(C) \)
- \( U_1(C) \rightarrow L_1(C) \)
- \( L_1(C) \rightarrow U_1(A) \)

Cycle in time: Contradiction

A New Problem:

Non-recoverable Schedule

T1

- \( L_1(A); L_1(B) \)
- \( \text{READ}(A) \)
- \( A := A + 100 \)
- \( \text{WRITE}(A); U_1(A) \)
- \( \text{READ}(B) \)
- \( B := B + 100 \)
- \( \text{WRITE}(B); U_1(B) \)
- \( \text{Commit} \)

T2

- \( L_2(A) \)
- \( \text{READ}(A) \)
- \( A := A' + 100 \)
- \( \text{WRITE}(A); L_2(A) \)
- \( \text{READ}(B) \)
- \( B := B' + 100 \)
- \( \text{WRITE}(B); U_2(A); U_2(B); \text{Commit} \)

Elements A, B written by T1 are restored to their original value.

Can no longer undo!

Dirty reads of A, B lead to incorrect writes.

A New Problem:

Non-recoverable Schedule

T1

- \( L_1(A); L_1(B) \)
- \( \text{READ}(A) \)
- \( A := A + 100 \)
- \( \text{WRITE}(A); U_1(A) \)
- \( \text{READ}(B) \)
- \( B := B + 100 \)
- \( \text{WRITE}(B); U_1(B) \)
- \( \text{Rollback} \)

T2

- \( L_2(A) \)
- \( \text{READ}(A) \)
- \( A := A' + 100 \)
- \( \text{WRITE}(A); L_2(A) \)
- \( \text{READ}(B) \)
- \( B := B' + 100 \)
- \( \text{WRITE}(B); U_2(A); U_2(B); \text{Commit} \)

Elements A, B written by T1 are restored to their original value.

Can no longer undo!

Dirty reads of A, B lead to incorrect writes.

A New Problem:

Non-recoverable Schedule

T1

- \( L_1(A); L_1(B) \)
- \( \text{READ}(A) \)
- \( A := A + 100 \)
- \( \text{WRITE}(A); U_1(A) \)
- \( \text{READ}(B) \)
- \( B := B + 100 \)
- \( \text{WRITE}(B); U_1(B) \)
- \( \text{Rollback} \)

T2

- \( L_2(A) \)
- \( \text{READ}(A) \)
- \( A := A' + 100 \)
- \( \text{WRITE}(A); L_2(A) \)
- \( \text{READ}(B) \)
- \( B := B' + 100 \)
- \( \text{WRITE}(B); U_2(A); U_2(B); \text{Commit} \)

Elements A, B written by T1 are restored to their original value.

Can no longer undo!

Strict 2PL

T1

- \( L_1(A) \)
- \( \text{READ}(A) \)
- \( A := A + 100 \)
- \( \text{WRITE}(A); L_1(A) \)
- \( \text{READ}(B) \)
- \( B := B + 100 \)
- \( \text{WRITE}(B); \text{Rollback} \)

T2

- \( L_2(A) \)
- \( \text{READ}(A) \)
- \( A := A' + 100 \)
- \( \text{WRITE}(A); L_2(A) \)
- \( \text{READ}(B) \)
- \( B := B' + 100 \)
- \( \text{WRITE}(B); \text{Commit} \)

...GRANTED; \text{READ}(A)

Rollback & \text{U}_1(A); \text{U}_1(B);
Strict 2PL

The Strict 2PL rule:

All locks are held until commit/abort:
All unlocks are done together with commit/abort.

With strict 2PL, we will get schedules that are both conflict-serializable and recoverable

Another problem: Deadlocks

• $T_1$: R(A), W(B)
• $T_2$: R(B), W(A)

$T_1$ holds the lock on A, waits for B
$T_2$ holds the lock on B, waits for A

This is a deadlock!

Another problem: Deadlocks

To detect a deadlocks, search for a cycle in the waits-for graph:
• $T_1$ waits for a lock held by $T_2$;
• $T_2$ waits for a lock held by $T_3$;
• ...
• $T_n$ waits for a lock held by $T_1$

Relatively expensive: check periodically, if deadlock is found, then abort one transaction.
need to continuously re-check for deadlocks

A “Solution”: Lock Modes

• $S$ = shared lock (for READ)
• $X$ = exclusive lock (for WRITE)

Lock compatibility matrix:

<table>
<thead>
<tr>
<th></th>
<th>None</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>S</td>
<td>✔</td>
<td>✔</td>
<td>✖</td>
</tr>
<tr>
<td>X</td>
<td>✖</td>
<td>✖</td>
<td>✖</td>
</tr>
</tbody>
</table>

Lock Granularity

• Fine granularity locking (e.g., tuples)
  – High concurrency
  – High overhead in managing locks
  – E.g., SQL Server

• Coarse grain locking (e.g., tables, entire database)
  – Many false conflicts
  – Less overhead in managing locks
  – E.g., SQL Lite

• Solution: lock escalation changes granularity as needed
Phantom Problem

- So far we have assumed the database to be a static collection of elements (=tuples)
- If tuples are inserted/deleted then the *phantom problem* appears

Suppose there are two blue products, A1, A2:

**Phantom Problem**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT * FROM Product WHERE color=’blue’</td>
<td>INSERT INTO Product(name, color) VALUES (’A3’, ’blue’)</td>
</tr>
<tr>
<td>SELECT * FROM Product WHERE color=’blue’</td>
<td></td>
</tr>
</tbody>
</table>

**Is this schedule serializable?**

**Phantom Problem**

- A "phantom" is a tuple that is invisible during part of a transaction execution but not invisible during the entire execution

- In our example:
  - T1: reads list of products
  - T2: inserts a new product
  - T1: re-reads: a new product appears!
Dealing With Phantoms

- Lock the entire table
- Lock the index entry for 'blue'
  - If index is available
- Or use predicate locks
  - A lock on an arbitrary predicate

Dealing with phantoms is expensive!

Summary of Serializability

- Serializable schedule = equivalent to a serial schedule
- (strict) 2PL guarantees conflict serializability
  - What is the difference?
- Static database:
  - Conflict serializability implies serializability
- Dynamic database:
  - This no longer holds

Isolation Levels in SQL

1. “Dirty reads”
   SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
2. “Committed reads”
   SET TRANSACTION ISOLATION LEVEL READ COMMITTED
3. “Repeatable reads”
   SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
4. Serializable transactions
   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

ACID

1. Isolation Level: Dirty Reads

- “Long duration” WRITE locks
  - Strict 2PL
- No READ locks
  - Read-only transactions are never delayed

Possible problems: dirty and inconsistent reads

2. Isolation Level: Read Committed

- “Long duration” WRITE locks
  - Strict 2PL
- “Short duration” READ locks
  - Only acquire lock while reading (not 2PL)

Unrepeatable reads:
When reading same element twice, may get two different values

3. Isolation Level: Repeatable Read

- “Long duration” WRITE locks
  - Strict 2PL
- “Long duration” READ locks
  - Strict 2PL

Why ?
This is not serializable yet !!!
4. Isolation Level Serializable

- "Long duration" WRITE locks
  - Strict 2PL
- "Long duration" READ locks
  - Strict 2PL
- Predicate locking
  - To deal with phantoms

Beware!

In commercial DBMSs:
- Default level is often NOT serializable
- Default level differs between DBMSs
- Some engines support subset of levels!
- Serializable may not be exactly ACID
  - Locking ensures isolation, not atomicity
- Also, some DBMSs do NOT use locking and different isolation levels can lead to different pbs
- Bottom line: RTFM for your DBMS!