A New Problem: Non-recoverable Schedule

T1
L(A); L(B); READ(A)
A := A + 100
WRITE(A); U(A)
READ(B)
B := B + 100
WRITE(B); U(B);

T2
L(A); READ(A)
A := A^2
WRITE(A); U(A)

...GRANTED; READ(B)
B := B^2
WRITE(B); U(A); U(B);
Commit

Rollback

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

Announcements

• 2 late days for HW 8 are now free
  – No more than 2 late days allowed. Monday Dec. 10 is the hard cut off

• Office hours changes
  – Ryan tomorrow at 11am instead of 10:30
  – Andrew additional office hours Friday
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

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 \( T_3 \);
- \ldots
- \( T_n \) 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)
A “Solution”?: Lock Modes

- S = shared lock (for READ)
- X = exclusive lock (for WRITE)

Lock compatibility matrix:

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Can only fix deadlocks if transactions declare exclusive locks in advance.

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?

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>
<tr>
<td>R1(A1); R1(A2); W2(A3); R1(A1); R1(A2); R1(A3)</td>
<td></td>
</tr>
</tbody>
</table>
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>

\[ R_1(A1);R_1(A2);W_2(A3);R_1(A1);R_1(A2);R_1(A3) \]
\[ W_2(A3);R_1(A1);R_1(A2);R_1(A1);R_1(A2);R_1(A3) \]

**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**

[For better performance]

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

**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

This is not serializable yet !!!

Why ?

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!