Introduction to Database Systems
CSE 414

Lecture 27:
Implementation of Transactions
Announcements

• Fix quotes in Flights data
  – See email/Piazza post
  – https://piazza.com/class/jmftm54e88t2kk?cid=729

• Final exam Thu, Dec 13 – 2:30 here
  – Will test concepts from entire class but emphasis on post-midterm
  – Previous finals are for reference only, better to study lecture and section materials
Testing for Conflict-Serializability

Precedence graph:
- A node for each transaction $T_i$,
- An edge from $T_i$ to $T_j$ whenever an action in $T_i$ conflicts with, and comes before an action in $T_j$

- The schedule is conflict-serializable iff the precedence graph is acyclic
Example 1

\[ r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B) \]
Example 1

This schedule is conflict-serializable
Example 2

\[ r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B) \]
Example 2

This schedule is NOT conflict-serializable
Implementing Transactions
Scheduler

- **Scheduler** = the module that schedules the transaction’s actions, ensuring serializability

- Also called **Concurrency Control Manager**

- We discuss next how a scheduler may be implemented
Implementing a Scheduler

Major differences between database vendors

- **Locking Scheduler**
  - Aka “pessimistic concurrency control”
  - SQLite, SQL Server, DB2

- **Multiversion Concurrency Control (MVCC)**
  - Aka “optimistic concurrency control”
  - PostgreSQL, Oracle: Snapshot isolation (SI)

We discuss only locking schedulers in this class.
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)

By using locks scheduler ensures conflict-serializability
What Data Elements are Locked?

Major differences between vendors:

• Lock on the entire database
  – SQLite

• Lock on individual records
  – SQL Server, DB2, etc
More Notations

$L_i(A) = \text{transaction } T_i \text{ acquires lock for element } A$

$U_i(A) = \text{transaction } T_i \text{ releases lock for element } A$
## 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 * 2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B)</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>
Scheduler has ensured a conflict-serializable schedule
But what if…

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L₁(A); READ(A)</td>
<td>L₂(A); READ(A)</td>
</tr>
<tr>
<td>A := A + 100</td>
<td>A := A * 2</td>
</tr>
<tr>
<td>WRITE(A); U₁(A);</td>
<td>WRITE(A); U₂(A);</td>
</tr>
<tr>
<td>L₁(B); READ(B)</td>
<td>L₂(B); READ(B)</td>
</tr>
<tr>
<td>B := B + 100</td>
<td>B := B * 2</td>
</tr>
<tr>
<td>WRITE(B); U₁(B);</td>
<td>WRITE(B); U₂(B);</td>
</tr>
</tbody>
</table>

Locks did not enforce conflict-serializability!!! What’s wrong?
Two Phase Locking (2PL)

The 2PL rule:

In every transaction, all lock requests must precede all unlock requests
Example: 2PL transactions

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L₁(A); L₁(B); READ(A)</strong></td>
<td><strong>L₂(A); READ(A)</strong></td>
</tr>
<tr>
<td>A := A+100</td>
<td>A := A*2</td>
</tr>
<tr>
<td>WRITE(A); <strong>U₁(A)</strong></td>
<td>WRITE(A); <strong>L₂(B); BLOCKED…</strong></td>
</tr>
<tr>
<td>READ(B)</td>
<td>…<strong>GRANTED</strong>; READ(B)</td>
</tr>
<tr>
<td>B := B+100</td>
<td>B := B*2</td>
</tr>
<tr>
<td>WRITE(B); <strong>U₁(B)</strong></td>
<td>WRITE(B); <strong>U₂(A); U₂(B)</strong></td>
</tr>
</tbody>
</table>

Now it is conflict-serializable
Two Phase Locking (2PL)

**Theorem:** 2PL ensures conflict serializability
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:
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:

\[ U_1(A) \rightarrow L_2(A) \text{ why?} \]

\[ U_1(A) \text{ happened strictly before } L_2(A) \]
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:

- $U_1(A) \rightarrow L_2(A)$
- $L_2(A) \rightarrow U_2(B)$

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:

\[ U_1(A) \rightarrow L_2(A) \]
\[ L_2(A) \rightarrow U_2(B) \]

why?
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:

- $U_1(A) \rightarrow L_2(A)$
- $L_2(A) \rightarrow U_2(B)$
- $U_2(B) \rightarrow L_3(B)$

why?
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:

\[ U_1(A) \rightarrow L_2(A) \]
\[ L_2(A) \rightarrow U_2(B) \]
\[ U_2(B) \rightarrow L_3(B) \]

......etc.....
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:

- $U_1(A) \rightarrow L_2(A)$
- $L_2(A) \rightarrow U_2(B)$
- $U_2(B) \rightarrow L_3(B)$
- $L_3(B) \rightarrow U_3(C)$
- $U_3(C) \rightarrow L_1(C)$
- $L_1(C) \rightarrow U_1(A)$

Cycle in time: Contradiction
A New Problem: Non-recoverable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L_1(A); L_1(B); \text{READ}(A) )</td>
<td>( L_2(A); \text{READ}(A) )</td>
</tr>
<tr>
<td>( A := A + 100 )</td>
<td>( A := A \times 2 )</td>
</tr>
<tr>
<td>( \text{WRITE}(A); U_1(A) )</td>
<td>( \text{WRITE}(A); )</td>
</tr>
<tr>
<td>( \text{READ}(B) )</td>
<td>( L_2(B); \text{BLOCKED…} )</td>
</tr>
<tr>
<td>( B := B + 100 )</td>
<td>( \text{…GRANTED}; \text{READ}(B) )</td>
</tr>
<tr>
<td>( \text{WRITE}(B); U_1(B); )</td>
<td>( B := B \times 2 )</td>
</tr>
<tr>
<td></td>
<td>( \text{WRITE}(B); U_2(A); U_2(B); )</td>
</tr>
<tr>
<td></td>
<td>( \text{Commit} )</td>
</tr>
</tbody>
</table>

Rollback
# A New Problem: Non-recoverable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L₁(A); L₁(B); READ(A)</td>
<td>L₂(A); READ(A)</td>
</tr>
<tr>
<td>A := A+100</td>
<td>A := A*2</td>
</tr>
<tr>
<td>WRITE(A); U₁(A)</td>
<td>WRITE(A);</td>
</tr>
<tr>
<td></td>
<td>L₂(B); BLOCKED…</td>
</tr>
<tr>
<td></td>
<td>…GRANTED; READ(B)</td>
</tr>
<tr>
<td>READ(B)</td>
<td>B := B*2</td>
</tr>
<tr>
<td>B := B+100</td>
<td>WRITE(B); U₂(A); U₂(B);</td>
</tr>
<tr>
<td>WRITE(B); U₁(B);</td>
<td>Commit</td>
</tr>
</tbody>
</table>

Elements A, B written by T1 are restored to their original value.
A New Problem: Non-recoverable Schedule

T1

L_1(A); L_1(B); READ(A)
A := A + 100
WRITE(A); U_1(A)

READ(B)
B := B + 100
WRITE(B); U_1(B); 

T2

L_2(A); READ(A)
A := A * 2
WRITE(A);
L_2(B); BLOCKED…

…GRANTED; READ(B)
B := B * 2
WRITE(B); U_2(A); U_2(B);
Commit

Dirty reads of A, B lead to incorrect writes.

Rollback

Elements A, B written by T1 are restored to their original value.
A New Problem: Non-recoverable Schedule

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.

Rollback
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.
Strict 2PL

T1

L₁(A); READ(A)
A := A + 100
WRITE(A);

L₁(B); READ(B)
B := B + 100
WRITE(B);
Rollback & U₁(A); U₁(B);

T2

L₂(A); BLOCKED…

L₂(B); READ(B)

…GRANTED; READ(A)
A := A*2
WRITE(A);
L₂(B); READ(B)
B := B*2
WRITE(B);
Commit & U₂(A); U₂(B);
Strict 2PL

• Lock-based systems always use strict 2PL

• Easy to implement:
  – Before a transaction reads or writes an element A, insert an L(A)
  – When the transaction commits/aborts, then release all locks

• Ensures both conflict serializability and recoverability
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:

```
None   S   X
None   |    |    |
S      |    |    |
X      |    |    |
```
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
Throughput (TPS) vs. # Active Transactions

**Lock Performance**

Thrashing

**Why?**

To avoid, use admission control

**TPS = Transactions per second**

CSE 414 - Autumn 2018
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 *</td>
<td>INSERT INTO Product(name, color)</td>
</tr>
<tr>
<td>FROM Product</td>
<td>VALUES (‘A3’,’blue’)</td>
</tr>
<tr>
<td>WHERE color=‘blue’</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SELECT *</td>
</tr>
<tr>
<td></td>
<td>FROM Product</td>
</tr>
<tr>
<td></td>
<td>WHERE color=‘blue’</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 *</td>
<td>INSERT INTO Product(name, color) VALUES (‘A3’,’blue’)</td>
</tr>
<tr>
<td>FROM Product</td>
<td></td>
</tr>
<tr>
<td>WHERE color=‘blue’</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>SELECT *</td>
<td></td>
</tr>
<tr>
<td>FROM Product</td>
<td></td>
</tr>
<tr>
<td>WHERE color=‘blue’</td>
<td></td>
</tr>
</tbody>
</table>

\[ R_1(A1);R_1(A2);W_2(A3);R_1(A1);R_1(A2);R_1(A3) \]
Suppose there are two blue products, A1, A2:

**Phantom Problem**

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

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

\[ W_2(A3); R_1(A1); R_1(A2); R_1(A1); R_1(A2); R_1(A3)^44 \]
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
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!