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

```
<table>
<thead>
<tr>
<th>Schedule</th>
<th>Transaction Type</th>
<th>Transaction ID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$r_2(A)$</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$r_1(B)$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$w_2(A)$</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>$r_3(A)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$w_1(B)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$w_3(A)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$r_2(B)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$w_2(B)$</td>
<td></td>
</tr>
</tbody>
</table>
```

Example 2

```
<table>
<thead>
<tr>
<th>Schedule</th>
<th>Transaction Type</th>
<th>Transaction ID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$r_2(A)$</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$r_1(B)$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$w_2(A)$</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>$r_3(A)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$w_1(B)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$w_3(A)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$r_2(B)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$w_2(B)$</td>
<td></td>
</tr>
</tbody>
</table>
```

This schedule is conflict-serializable.
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”
  - Postgres, Oracle: Snapshot Isolation (SI)

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>
</tbody>
</table>

Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L_1(A); READ(A)</td>
<td>L_2(B); READ(B)</td>
</tr>
<tr>
<td>A := A+100</td>
<td>B := B+100</td>
</tr>
<tr>
<td>WRITE(A); U_1(A); L_1(B);</td>
<td>WRITE(B); U_2(B);</td>
</tr>
<tr>
<td>READ(A)</td>
<td>READ(B)</td>
</tr>
<tr>
<td>A := A*2</td>
<td>B := B*2</td>
</tr>
<tr>
<td>WRITE(A); U_2(A); L_2(B);</td>
<td>WRITE(B); U_2(B);</td>
</tr>
</tbody>
</table>

But what if…

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L_1(A); READ(A)</td>
<td>L_2(B); READ(B)</td>
</tr>
<tr>
<td>A := A+100</td>
<td>B := B+100</td>
</tr>
<tr>
<td>WRITE(A); U_1(A); L_1(B);</td>
<td>WRITE(B); U_2(B);</td>
</tr>
<tr>
<td>READ(A)</td>
<td>READ(B)</td>
</tr>
<tr>
<td>A := A'2</td>
<td>B := B'2</td>
</tr>
<tr>
<td>WRITE(A); U_2(A); L_2(B);</td>
<td>WRITE(B); U_2(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

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L_1(A); L_1(B); READ(A)</td>
<td>L_2(A); READ(A)</td>
</tr>
<tr>
<td>A := A+100</td>
<td>A := A'2</td>
</tr>
<tr>
<td>WRITE(A); U_1(A);</td>
<td>WRITE(A);</td>
</tr>
<tr>
<td>L_2(B); BLOCKED...</td>
<td>L_2(B);</td>
</tr>
<tr>
<td>READ(B)</td>
<td>READ(B)</td>
</tr>
<tr>
<td>B := B+100</td>
<td>B := B*2</td>
</tr>
<tr>
<td>WRITE(B); U_2(B);</td>
<td>WRITE(B); U_2(B);</td>
</tr>
</tbody>
</table>

Scheduler has ensured a conflict-serializable schedule

Locks did not enforce conflict-serializability!!! What's wrong?

Now it is conflict-serializable
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:

Then there is the following temporal cycle in the schedule:

**why?**

Then there is the following temporal cycle in the schedule:

Then there is the following temporal cycle in the schedule:

**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:

\[ U_1(A) \rightarrow L_1(A) \]
\[ L_2(A) \rightarrow U_2(B) \]
\[ U_2(B) \rightarrow L_2(B) \]
\[ U_3(C) \rightarrow L_1(C) \]
\[ L_1(C) \rightarrow U_1(A) \]

Why?

Cycle in time: Contradiction

A New Problem: Non-recoverable Schedule

T1

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

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

\[ \ldots \text{GRANTED}; \text{READ}(B) \]
\[ B := B'2 \]
\[ \text{WRITE}(B); U_1(A); U_1(B); \]
\[ \text{Commit} \]

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

Rollback

CSE 414 - Autumn 2018

T2

\[ L_1(A); \text{READ}(A) \]
\[ A := A'2 \]
\[ \text{WRITE}(A); \]

\[ L_2(A); \text{READ}(A) \]
\[ A := A'2 \]
\[ \text{WRITE}(A); \]

\[ L_3(A); \text{READ}(A) \]
\[ A := A'2 \]
\[ \text{WRITE}(A); \]

Dirty reads of A, B lead to incorrect writes.

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

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

T2
L1(A); READ(A)
A := A * 2
WRITE(A); U(A)

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

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

Rollback & U1(A); U1(B);
Committed

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

Another problem: Deadlocks

T1: R(A), W(B)
T2: R(B), W(A)
T1 holds the lock on A, waits for B
T2 holds the lock on B, waits for A

This is a deadlock!

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

• 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

To detect a deadlocks, search for a cycle in the waits-for graph:

• T1 waits for a lock held by T2;
• T2 waits for a lock held by T1;
• . . .
• Tn waits for a lock held by T1

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

Lock Performance

- Throughput (TPS)
  - # Active Transactions
  - To avoid, use admission control

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:

```
T1
SELECT * FROM Product WHERE color='blue'

T2
INSERT INTO Product(name, color) VALUES ('A3', 'blue')

T1
SELECT * FROM Product WHERE color='blue'
```

Is this schedule serializable?
Suppose there are two blue products, A1, A2:

**Phantom Problem**

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SELECT * FROM Product</td>
<td>INSERT INTO Product(color)</td>
</tr>
<tr>
<td></td>
<td>WHERE color='blue'</td>
<td>VALUES ('A3', 'blue')</td>
</tr>
<tr>
<td></td>
<td>SELECT * FROM Product</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WHERE color='blue'</td>
<td></td>
</tr>
<tr>
<td>R1(A1);R1(A2);W2(A3);R1(A1);R1(A2);R1(A3)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Dealing With Phantoms**

- 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!

**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 !!!

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!