Database Systems
CSE 414

Lecture 22:
Transaction Implementations

Announcements

• WQ7 (last!) due on Sunday

• HW7:
  – due on Wed, May 24
  – using JDBC to execute SQL from Java
  – using SQL Server via Azure

Recap

• What are transactions?
  – Why do we need them?

• Maintain ACID properties via schedules
  – We focus on the isolation property
  – We briefly discussed consistency & durability
  – We do not discuss atomicity

• Ensure conflict-serializable schedules with locks

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

We discuss only locking in 414

Locking Scheduler

Simple idea:

• Each element has a unique lock
• Each transaction must first acquire the lock before reading/writing that element
• If lock is taken by another transaction, then wait
• The transaction must release the lock(s)

By using locks, scheduler can ensure 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.
  – can be even more fine-grained by having different types of locks (allows more bns to run simultaneously)
Notation

\[ \text{L}(A) = \text{transaction } T_i \text{ acquires lock for element } A \]
\[ \text{U}(A) = \text{transaction } T_i \text{ releases lock for element } A \]

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); \text{L}(A)</td>
<td>WRITE(A); \text{U}(A); L(B)</td>
</tr>
<tr>
<td>READ(B)</td>
<td>READ(B)</td>
</tr>
<tr>
<td>B := B + 100</td>
<td>B := B' + 100</td>
</tr>
<tr>
<td>WRITE(B); \text{U}(B)</td>
<td>WRITE(B); \text{U}(B);</td>
</tr>
</tbody>
</table>

Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(A); READ(A)</td>
<td>L(A); READ(A)</td>
</tr>
<tr>
<td>A := A + 100</td>
<td>A := A' + 2</td>
</tr>
<tr>
<td>WRITE(A); \text{U}(A)</td>
<td>WRITE(A); \text{U}(A);</td>
</tr>
<tr>
<td>READ(B)</td>
<td>READ(B)</td>
</tr>
<tr>
<td>B := B + 100</td>
<td>B := B' + 100</td>
</tr>
<tr>
<td>WRITE(B); \text{U}(B)</td>
<td>WRITE(B); \text{U}(B);</td>
</tr>
</tbody>
</table>

But...

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(A); READ(A)</td>
<td>L(A); READ(A)</td>
</tr>
<tr>
<td>A := A + 100</td>
<td>A := A' + 2</td>
</tr>
<tr>
<td>WRITE(A); \text{U}(A)</td>
<td>WRITE(A); \text{U}(A);</td>
</tr>
<tr>
<td>L(B); READ(B)</td>
<td>L(B); READ(B)</td>
</tr>
<tr>
<td>B := B + 100</td>
<td>B := B' + 100</td>
</tr>
<tr>
<td>WRITE(B); \text{U}(B)</td>
<td>WRITE(B); \text{U}(B);</td>
</tr>
</tbody>
</table>

Scheduler has ensured a conflict-serializable schedule

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

2PL approach developed by Jim Gray

Example: 2PL transactions

T1
L1(A); L1(B); READ(A)
A := A + 100
WRITE(A); U1(A)

T2
L2(A); READ(A)
A := A * 2
WRITE(A)
L2(B); BLOCKED...

READ(B)
B := B + 100
WRITE(B); U1(B); U2(B); ...GRANTED; READ(B)
B := B * 2
WRITE(B); U2(A); U2(B);

Now it is conflict serializable

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:

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

Then there is the following temporal cycle in the schedule:

U1(A) => L2(A) 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:

\[ T_1 \rightarrow L_2(A) \rightarrow U_2(B) \rightarrow L_3(B) \rightarrow U_3(C) \rightarrow L_1(C) \rightarrow U_1(A) \]

Contradiction

另一个问题：死锁

- \( T_1 \) 等待由 \( T_2 \) 持有的锁；
- \( T_2 \) 等待由 \( T_3 \) 持有的锁；
- \( T_3 \) 等待 . . .
- . . .
- \( T_n \) 等待由 \( T_1 \) 持有的锁

SQL Lite：只有唯一的排它锁；因此，从不出现死锁

SQL Server：周期性检查死锁并中止一个事务

**Strict 2PL**

The Strict 2PL rule:

All locks are held until the transaction commits or aborts.

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

**A New Problem:** Non-recoverable Schedule

\[
T_1 \\
L_1(A); L_1(B) \quad \text{READ}(A) \\
A := A + 100 \\
\text{WRITE}(A); U_1(A) \\
\begin{align*}
\text{READ}(B) \\
B := B + 100 \\
\text{WRITE}(B); U_1(B); \quad \text{Rollback}
\end{align*}
\]

**Strict 2PL**

\[
T_1 \\
L_1(A) \quad \text{READ}(A) \\
A := A + 100 \\
\text{WRITE}(A); L_1(A) \quad \text{BLOCKED} \\
\begin{align*}
\text{READ}(B) \\
B := B + 100 \\
\text{WRITE}(B); U_1(A); U_1(B); \quad \text{ROLLBACK} \\
\text{U_1(A); U_1(B)}
\end{align*}
\]

**Another problem:** Deadlocks

- \( T_1 \) 等待由 \( T_2 \) 持有的锁；
- \( T_2 \) 等待由 \( T_3 \) 持有的锁；
- \( T_3 \) 等待 . . . .
- . . .
- \( T_n \) 等待由 \( T_1 \) 持有的锁

SQL Lite：只有唯一的排它锁；因此，从不出现死锁

SQL Server：周期性检查死锁并中止一个事务
Lock Modes

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

<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 compatibility matrix:

---

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

---

Phantom Problem

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

- 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!

Isolation Levels in SQL

- “Dirty Reads”
  
  ```sql
  SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
  ```

- “Committed Reads”
  
  ```sql
  SET TRANSACTION ISOLATION LEVEL READ COMMITTED
  ```

- “Repeatable Reads”
  
  ```sql
  SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
  ```

- Serializable Transactions
  
  ```sql
  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 (SQL Server!)
   - 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 probs
   - Bottom line: Read the doc for your DBMS!

Next two slides: try them on Azure
Demonstration with SQL Server

Application 1:
create table R(a int);
insert into R values(1);
set transaction isolation level serializable;
begin transaction;
select * from R; -- get a shared lock
waitfor delay '00:01'; -- wait for one minute

Application 2:
set transaction isolation level serializable;
begin transaction;
select * from R; -- get a shared lock
insert into R values(2); -- blocked waiting on exclusive lock
-- App 2 unblocks and executes insert after app 1 commits/aborts

Application 2:
set transaction isolation level repeatable read;
begin transaction;
select * from R; -- get a shared lock
insert into R values(3); -- gets an exclusive lock on new tuple
-- If app 1 reads now, it blocks because read dirty
-- If app 1 reads after app 2 commits, app 1 sees new value

CSE 414 - Spring 2017