Introduction to Data Management
CSE 344

Lecture 19: More Transactions
Announcements

• HW7 (final one!) will be released today
  – Some Java programming required
  – Connecting to SQL Azure
  – Due Wednesday, Aug 7

• WQ7 (final one!) released
  – Due Monday, Aug 10
Outline

• Serial and Serializable Schedules (18.1)

• Conflict Serializability (18.2)

• Transaction implementation using locks (18.3)
Review: Transactions

• **Problem**: An application must perform several writes and reads to the database, as a unit

• **Solution**: multiple actions of the application are bundled into one unit called a *Transaction*
Review: Transactions in SQL

BEGIN TRANSACTION
   [SQL statements]
COMMIT or
ROLLBACK (=ABORT)

If BEGIN… missing, then TXN consists of a single instruction
Know your chemistry transactions: ACID

- **Atomic**
  - State shows either all the effects of txn, or none of them

- **Consistent**
  - Txn moves from a DBMS state where integrity holds, to another where integrity holds
    - remember integrity constraints?

- **Isolated**
  - Effect of txns is the same as txns running one after another (i.e., looks like batch mode)

- **Durable**
  - Once a txn has committed, its effects remain in the database
Rollback transactions

• If the app gets to a state where it cannot complete the transaction successfully, execute ROLLBACK

• The DB returns to the state prior to the transaction

• Useful for atomicity – ROLLBACK on error.
Isolation: The Problem

• Multiple transactions are running concurrently $T_1, T_2, \ldots$

• They read/write some common elements $A_1, A_2, \ldots$

• How do we prevent unwanted interference?
• The **SCHEDULER** is responsible for that
Schedules

A schedule is a sequence of interleaved actions from all transactions
Review: Serial Schedule

• A *serial schedule* is one in which transactions are executed one after the other, in some sequential order

  *Batch Mode*

• Review: nothing can go wrong if the system executes transactions serially (up to what we have learned so far)
  – But DBMS don’t do that because we want better overall system performance
A must = B

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A, s)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>READ(B, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B, t)</td>
<td>WRITE(B, s)</td>
</tr>
</tbody>
</table>

A and B are elements in the database t and s are variables in txn source code
### Example of a (Serial) Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>WRITE(A, t)</td>
</tr>
<tr>
<td>t := t+100</td>
<td></td>
</tr>
<tr>
<td>WRITE(B, t)</td>
<td></td>
</tr>
<tr>
<td>t := t+100</td>
<td></td>
</tr>
<tr>
<td>WRITE(B, t)</td>
<td></td>
</tr>
</tbody>
</table>

- **A = 50**
- **B = 50**

**Time**

- **A = 150**
- **B = 150**

- **A = 300**
- **B = 300**

CSE 344 - Summer 2017
Another Serial Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A,s)</td>
<td>READ(A,s)</td>
</tr>
<tr>
<td>s := s*2</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A,s)</td>
<td>A := 100</td>
</tr>
<tr>
<td>READ(B,s)</td>
<td>READ(B,s)</td>
</tr>
<tr>
<td>s := s*2</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B,s)</td>
<td>B := 100</td>
</tr>
<tr>
<td>READ(A, t)</td>
<td>WRITE(A, t)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>A := 200</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A, t)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>B := 200</td>
</tr>
<tr>
<td>t := t+100</td>
<td>READ(B, t)</td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>WRITE(B,t)</td>
</tr>
</tbody>
</table>
A schedule is \textit{serializable} if it is equivalent to a serial schedule.
A Serializable Schedule

T1
READ(A, t)
t := t + 100
WRITE(A, t)

T2
READ(A, s)
s := s * 2
WRITE(A, s)

READ(B, t)
t := t + 100
WRITE(B, t)

This is a **serializable** schedule.

This is **NOT** a serial schedule

READ(B, s)
s := s * 2
WRITE(B, s)

No conflicts
### A Non-Serializable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A,s)</td>
</tr>
<tr>
<td></td>
<td>READ(B,s)</td>
</tr>
<tr>
<td></td>
<td>s := s*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B,s)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>READ(B,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>WRITE(B,s)</td>
</tr>
</tbody>
</table>
How do We Know if a Schedule is Serializable?

Notation:

\[ T_1: r_1(A); w_1(A); r_1(B); w_1(B) \]
\[ T_2: r_2(A); w_2(A); r_2(B); w_2(B) \]

Key Idea: Focus on conflicting operations
Conflicts

- Write-Read – WR
- Read-Write – RW
- Write-Write – WW
- Read-Read?
Conflicts: (i.e., swapping will change program behavior)

- Two actions by same transaction $T_i$: $r_i(X)$; $w_i(Y)$; $w_i(X)$; $w_j(X)$.
- Two writes by $T_i$, $T_j$ to same element: $w_i(X)$; $w_j(X)$.
- Read/write by $T_i$, $T_j$ to same element: $w_i(X)$; $r_j(X)$; $r_i(X)$; $w_j(X)$.
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.
Conflict Serializability

Example:

\[
\begin{align*}
  r_1(A); & \quad w_1(A); \quad r_2(A); \quad w_2(A); \quad r_1(B); \quad w_1(B); \quad r_2(B); \quad w_2(B) \\
  r_1(A); & \quad w_1(A); \quad r_1(B); \quad w_1(B); \quad r_2(A); \quad w_2(A); \quad r_2(B); \quad w_2(B) 
\end{align*}
\]
Conflict Serializability

Example:

\[ r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B) \]
Conflict Serializability

Example:

\[ r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B) \]

\[ r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B) \]

\[ r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B) \]
Conflict Serializability

Example:

\[ r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B) \]

\[ r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B) \]

\[ r_1(A); w_1(A); r_1(B); r_2(A); w_2(A); w_1(B); r_2(B); w_2(B) \]

\[ r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B) \]
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
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

We discuss only locking schedulers in 344
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
Review: SQLite

• SQLite is very simple
• More info: http://www.sqlite.org/atomiccommit.html

• Lock types
  – READ LOCK  (to read)
  – RESERVED LOCK (to write)
  – PENDING LOCK (wants to commit)
  – EXCLUSIVE LOCK (to commit)
SQLite

Step 1: when a transaction begins

- Acquire a READ LOCK (aka "SHARED" lock)
- All these transactions may read happily
- They all read data from the database file
- If the transaction commits without writing anything, then it simply releases the lock
SQLite

Step 2: when one transaction wants to write
- Acquire a **RESERVED LOCK**
- May coexists with many READ LOCKs
- Writer TXN may write; these updates are only in main memory; others don't see the updates
- Reader TXN continue to read from the file
- New readers accepted
- No other TXN is allowed a **RESERVED LOCK**
SQLite

Step 3: when writer transaction wants to commit, it needs *exclusive lock*, which can’t coexists with *read locks*

- Acquire a **PENDING LOCK**
- May coexists with old READ LOCKs
- No new READ LOCKS are accepted
- Wait for all read locks to be released

Why not write to disk right now?
SQLite

Step 4: when all read locks have been released

- Acquire the **EXCLUSIVE LOCK**
- Nobody can touch the database now
- All updates are written permanently to the database file

- Release the lock and **COMMIT**
SQLite

begin transaction

first write

commit requested

no more read locks

None

READ LOCK

RESERVED LOCK

PENDING LOCK

EXCLUSIVE LOCK

commit

Allows new Read Locks

No new Read Locks

Only if no Reserved locks

commit executed

Lecture notes contains a SQLite demo
create table r(a int, b int);
insert into r values (1,10);
insert into r values (2,20);
insert into r values (3,30);

Does not seem to work in Ubuntu for Window
(looks like a file flush issue)
Demonstrating Locking in SQLite

T1:

begin transaction;
select * from r;
-- T1 has a READ LOCK

T2:

begin transaction;
select * from r;
-- T2 has a READ LOCK
Demonstrating Locking in SQLite

T1:

update r set b=11 where a=1;
-- T1 has a RESERVED LOCK

T2:

update r set b=21 where a=2;
-- T2 asked for a RESERVED LOCK: DENIED
Demonstrating Locking in SQLite

T3:

begin transaction;
select * from r;
commit;
-- everything works fine, could obtain READ LOCK
Demonstrating Locking in SQLite

T1:

commit;
-- SQL error: database is locked
-- T1 asked for PENDING LOCK -- GRANTED
-- T1 asked for EXCLUSIVE LOCK -- DENIED
Demonstrating Locking in SQLite

$T_1$: Pudding

$T_3'$:

begin transaction;
select * from r;
-- T3 asked for READ LOCK-- DENIED (due to T1)

T2:

commit;  

Can not commit if any read locks

-- releases the last READ LOCK; T1 can commit
How do anomalies show up in schedules?

- What could go wrong if we didn’t have concurrency control:
  - Dirty reads (including inconsistent reads)
  - Unrepeatable reads
  - Lost updates

Many other things can go wrong too
Dirty Reads

Write-Read Conflict

$T_1$: WRITE(A)

$T_1$: ABORT

$T_2$: READ(A)
Inconsistent Read

Write-Read Conflict

T₁: A := 20; B := 20;
T₁: WRITE(A)
T₁: WRITE(B)

T₂: READ(A);
T₂: READ(B);
Unrepeatable Read

Read-Write Conflict

$T_1$: WRITE(A)

$T_2$: READ(A);

$T_2$: READ(A);
Lost Update

Write-Write Conflict

\[ T_1: \text{READ}(A) \]
\[ T_1: A := A+5 \]
\[ T_1: \text{WRITE}(A) \]

\[ T_2: \text{READ}(A); \]
\[ T_2: A := A*1.3 \]
\[ T_2: \text{WRITE}(A); \]

Next time: handling anomalies with locks