Introduction to Data Management
CSE 344

Lecture 20: More Transactions
Announcements

• HW7 (final one!) will be released today
  – Some Java programming required
  – Connecting to SQL Azure
  – Due Wednesday, March 8

• WQ7 (final one!) released
  – Due Monday, March 6
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*
Turing Awards in Data Management

Charles Bachman, 1973
*IDS and CODASYL*

Ted Codd, 1981
*Relational model*

Jim Gray, 1998
*Transaction processing*

Michael Stonebraker, 2014
*INGRES and Postgres*
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
Atomic

• **Definition**: A transaction is ATOMIC if all its updates must happen or not at all.

• **Example**: move $100 from A to B
  
  – UPDATE accounts SET bal = bal – 100 WHERE acct = A;
  
  – UPDATE accounts SET bal = bal + 100 WHERE acct = B;
  
  – BEGIN TRANSACTION;
    UPDATE accounts SET bal = bal – 100 WHERE acct = A;
    UPDATE accounts SET bal = bal + 100 WHERE acct = B;
  
  COMMIT;
Isolated

• **Definition** An execution ensures that txns are isolated, if the effect of each txn is as if it were the only txn running on the system.

• More in a few slides
Consistent

• Recall: integrity constraints govern how values in tables are related to each other
  – Can be enforced by the DBMS, or ensured by the app

• How consistency is achieved by the app:
  – App programmer ensures that txns only takes a consistent DB state to another consistent state
  – DB makes sure that txns are executed atomically

• Can defer checking the validity of constraints until the end of a transaction
Durable

- A transaction is durable if its effects continue to exist after the transaction and even after the program has terminated

- How?
  - By writing to disk!
  - More in 444
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

• What are examples of such program states?
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.
A *serial schedule* is one in which transactions are executed one after the other, in some sequential order.

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.
Example

A and B are elements in the database
t and s are variables in txn source code

<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>
### Example of a (Serial) 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>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>
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>WRITE(A, s)</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>WRITE(B, s)</td>
</tr>
</tbody>
</table>

READ(A, t)
t := t+100
WRITE(A, t)
READ(B, t)
t := t+100
WRITE(B, t)
Review: Serializable Schedule

A schedule is **serializable** if it is equivalent to a serial schedule.
A 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></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>

This is a **serializable** schedule.
This is NOT a serial schedule.
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></td>
<td>READ(B, t)</td>
</tr>
<tr>
<td></td>
<td>t := t + 100</td>
</tr>
<tr>
<td></td>
<td>WRITE(B, t)</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?
Conflict Serializability

Conflicts: (i.e., swapping will change program behavior)

Two actions by same transaction $T_i$: $r_i(X); w_i(Y)$

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:

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

Example:

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