Introduction to Data Management
CSE 344

Lecture 21: More Transactions
Announcements

• HW6, WQ6 due tonight

• HW7 will be released today
  – Some Java programming required
  – Connecting to SQL Azure
  – Due Wednesday, November 30

• WQ7 (final one!) released
  – Due Tuesday, November 29
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
Review: ACID

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

• **Consistent**
  – Txn moves from a state where integrity holds, to another where integrity holds

• **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
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
A schedule is a sequence of interleaved actions from all transactions
Serial Schedule

• 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.
  – But DBMS don’t do that because we want better overall system performance.
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>READ(A, t)</strong></td>
<td><strong>READ(A, s)</strong></td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td><strong>WRITE(A, t)</strong></td>
<td><strong>WRITE(A, s)</strong></td>
</tr>
<tr>
<td><strong>READ(B, t)</strong></td>
<td><strong>READ(B, s)</strong></td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td><strong>WRITE(B, t)</strong></td>
<td><strong>WRITE(B, s)</strong></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>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>
<tr>
<td>READ(A, t)</td>
<td>READ(A, t)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>t := t+100</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A, t)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>READ(B, t)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>t := t+100</td>
</tr>
<tr>
<td>WRITE(B, t)</td>
<td>WRITE(B, t)</td>
</tr>
</tbody>
</table>
A schedule is **serializable** if it is equivalent to a serial schedule.
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>READ(B, t)</td>
<td>WRITE(B, t)</td>
</tr>
<tr>
<td>t := t+100</td>
<td></td>
</tr>
<tr>
<td>WRITE(B, t)</td>
<td></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

• A serializable schedule may not necessarily be conflict-serializable

\[
\begin{align*}
W_1(x, 0) &; W_2(x, 0) &; R_1(x) &; R_2(x) \\
W_1(x, 0) &; R_1(x) &; W_2(x, 0) &; R_2(x)
\end{align*}
\]
Conflict Serializability

Example:

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

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

Example:

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

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

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

1  2  3
Example 1

This schedule is \textit{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

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