ICS 321 Fall 2013
Transactions Processing (i)

Asst. Prof. Lipyeow Lim
Information & Computer Science Department
University of Hawaii at Manoa
Airline Reservation Example

Flights ( fltNo, fltDate, seatNo, seatStatus )

To view available seats:

```
SELECT seatNo
FROM Flights
WHERE fltNo = 123 AND fltDate = DATE '2008-12-25'
    AND seatStatus = ' available ' ;
```

To reserve a particular seat:

```
UPDATE Flights
SET seatStatus = 'occupied'
WHERE fltNo = 123 AND fltDate = DATE '2008-12-25'
    AND seatNo = '22A';
```
Transactions

- A **transaction** is the DBMS’s abstract view of a user program: a sequence of reads and writes.
  - Eg. User 1 views available seats and reserves seat 22A.

- A DBMS supports **multiple users**, ie, multiple transactions may be running **concurrently**.
  - Eg. User 2 views available seats and reserves seat 22A.
  - Eg. User 3 views available seats and reserves seat 23D.
Concurrent Execution

- DBMS tries to execute transactions concurrently – why?

<table>
<thead>
<tr>
<th>Schedule 1</th>
<th>Schedule 2</th>
<th>Schedule 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>U1</strong></td>
<td><strong>U2</strong></td>
<td><strong>U1</strong></td>
</tr>
<tr>
<td>Finds 22A empty</td>
<td>Finds 22A empty</td>
<td>Finds 22A empty</td>
</tr>
<tr>
<td>Reserves 22A</td>
<td>Reserves 22A</td>
<td>Reserves 22A</td>
</tr>
<tr>
<td></td>
<td>Reserves 22A</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

10/30/2013

Lipyeow Lim – University of Hawaii at Manoa
ACID Properties

4 important properties of transactions

• **Atomicity**: all or nothing
  – Users regard execution of a transaction as atomic
  – No worries about incomplete transactions

• **Consistency**: a transaction must leave the database in a good state
  – Semantics of consistency is application dependent
  – The user assumes responsibility

• **Isolation**: a transaction is isolated from the effects of other concurrent transaction

• **Durability**: Effects of completed transactions persists even if system crashes before all changes are written out to disk
Atomicity

• A transaction might
  – *commit* after completing all its actions, or it could
  – *abort* (or be aborted by the DBMS) after executing some actions.

• A very important property guaranteed by the DBMS for all transactions is that they are *atomic*.
  – A user can think of a Xact as always executing all its actions in one step, or not executing any actions at all.

• DBMS *logs* all actions so that it can *undo* the actions of aborted transactions.
Example (Atomicity)

- The first transaction is transferring $100 from B’s account to A’s account.
- The second is crediting both accounts with a 6% interest payment.
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect must be equivalent to these two transactions running serially in some order.
Database View of Transactions

T1: BEGIN
A=A+100
B=B-100
END

T1: BEGIN
Read A from disk
A=A+100
Write A to disk
Read B from disk
B=B-100
Write B to disk
END

T1: BEGIN
R(A)
W(A)
R(B)
W(B)
END
Serial Executions

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=A+100</td>
<td>A=1.06<em>A, B=1.06</em>B</td>
<td>A = 100, B = 200</td>
</tr>
<tr>
<td>B=B-100</td>
<td>A = 100, B = 200</td>
<td></td>
</tr>
<tr>
<td>A=A+100</td>
<td>A = 206, B = 212</td>
<td></td>
</tr>
<tr>
<td>B=B-100</td>
<td>A = 206, B = 112</td>
<td></td>
</tr>
<tr>
<td>A=A+100</td>
<td>A = 106, B = 200</td>
<td></td>
</tr>
<tr>
<td>B=B-100</td>
<td>A = 106, B = 212</td>
<td></td>
</tr>
</tbody>
</table>

Note: The code is executed in order from T1 to T2, with the result of T1 being the input for T2.
Example (Serializability)

\[
\begin{array}{c|c}
T1 & T2 \\
A = A + 100 & A = A + 100 \\
B = B - 100 & B = B \cdot 1.06 \\
\end{array}
\]

\[
\begin{array}{c|c}
T1 & T2 \\
A = A + 100 & A = 200, B = 200 \\
B = B - 100 & A = 212, B = 200 \\
A = A \cdot 1.06 & A = 212, B = 106 \\
B = B \cdot 1.06 & A = 212, B = 100 \\
\end{array}
\]

Equivalent:

\[
\begin{array}{c|c}
T1 & T2 \\
A = A + 100 & A = 100, B = 200 \\
B = B - 100 & A = 200, B = 200 \\
A = A \cdot 1.06 & A = 212, B = 200 \\
B = B \cdot 1.06 & A = 212, B = 212 \\
\end{array}
\]

\[
\begin{array}{c|c}
T1 & T2 \\
A = A + 100 & A = 212, B = 112 \\
B = B - 100 & A = 212, B = 112 \\
A = A \cdot 1.06 & A = 212, B = 200 \\
B = B \cdot 1.06 & A = 212, B = 212 \\
\end{array}
\]
Scheduling Transactions

• *Serial schedule:* Schedule that does not interleave the actions of different transactions.

• *Equivalent schedules:* For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.

• *Serializable schedule:* A schedule that is equivalent to some serial execution of the transactions.

(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)
Transactions in SQL

• After connection to a database, a transaction is automatically started
  – Different connections -> different transactions
• Within a connection, a transaction is ended by
  – COMMIT or COMMIT WORK
  – ROLLBACK (= “abort”)
• DBMS can also initiate rollback and return an error.
• SAVEPOINT <savepoint name>
• ROLLBACK TO SAVEPOINT <savepoint name>
  – Locks obtained after savepoint can be released after rollback to that savepoint
• Using savepoints vs sequence of transactions
  – Transaction rollback is to last transaction only
Isolation levels in SQL

- SQL supports 4 isolation levels

<table>
<thead>
<tr>
<th>SQL Isolation Levels</th>
<th>DB2 Isolation Levels</th>
<th>Dirty read</th>
<th>Unrepeatable Read</th>
<th>Phantom</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>UNCOMMITTED READ (UR)</td>
<td>Maybe</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>CURSOR STABILITY * (CS)</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>READ STABILITY (RS)</td>
<td>No</td>
<td>No</td>
<td>Maybe</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>REPEATABLE READ (RR)</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**SET TRANSACTION ISOLATION LEVEL** SERIALIZABLE

**SELECT** *

**FROM** Reserves

**WHERE** SID=100

**WITH UR**
Anomaly: Dirty Reads

- T1 reads uncommitted data from T2 which may abort

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = 1.06 * A Commit</td>
<td>A = A + 100</td>
</tr>
<tr>
<td>B = B - 100 Abort</td>
<td></td>
</tr>
</tbody>
</table>

- A = 20
- A = 120
- A = 127.2

With T2 aborted correct value of A = 21.2
Anomaly: Unrepeatable Reads

- T1 sees two different values of A, because updates are committed from another transaction (T2)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Print A</td>
<td>A = 20</td>
</tr>
<tr>
<td>A = 100</td>
<td>A = 20</td>
</tr>
<tr>
<td>Commit</td>
<td>A = 21.2</td>
</tr>
<tr>
<td>Print A</td>
<td>A = 21.2</td>
</tr>
</tbody>
</table>

T1 sees two different values of A even though T1 did not change A!
# Anomaly: Phantom Reads

- Multiple reads from the same transaction sees different set of tuples

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find all ics321 students</td>
<td>Enroll student D into ics321</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
<tr>
<td>Find all ics321 students</td>
<td>Insert D</td>
</tr>
<tr>
<td>Commit</td>
<td></td>
</tr>
</tbody>
</table>

T1 sees two different results of the query even though T1 did not change the table!