Announcements

- WQ7 released – Due on 5/30
- HW8 will be released later today – Due on 5/30
- These are the last HW assignments for the class!

What can go wrong?

- Manager: balance budgets among projects
  - Remove $10k from project A
  - Add $7k to project B
  - Add $3k to project C

- CEO: check company’s total balance
  - SELECT SUM(money) FROM budget;

- This is called a dirty / inconsistent read aka a WRITE-READ conflict

What can go wrong?

- App 1:
  SELECT inventory FROM products WHERE pid = 1

- App 2:
  UPDATE products SET inventory = 0 WHERE pid = 1

- App 1:
  SELECT inventory * price FROM products WHERE pid = 1

- This is known as an unrepeatable read aka READ-WRITE conflict

What can go wrong?

- Account 1 = $100
- Account 2 = $100
- Total = $200

- App 1:
  - Set Account 1 = $200
  - Set Account 2 = $0

- App 2:
  - Set Account 2 = $200
  - Set Account 1 = $0

- At the end:
  - Total = $200

- App 1: Set Account 1 = $200
- App 2: Set Account 2 = $200

- App 1: Set Account 2 = $0
- App 2: Set Account 1 = $0

- At the end:
  - Total = $0

This is called the lost update aka WRITE-WRITE conflict
What can go wrong?

- Buying tickets to the next Bieber concert:
  - Fill up form with your mailing address
  - Put in debit card number
  - Click submit
  - Screen shows money deducted from your account
  - [Your browser crashes]

Lesson:
Changes to the database should be ALL or NOTHING

Transactions

- Collection of statements that are executed atomically (logically speaking)

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

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

Time

This is a serializable schedule.
This is NOT a serial schedule

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

Review: Serializable Schedule

A schedule is **serializable** if it is equivalent to a serial schedule (in terms of its effects on the DB)
A Non-Serializable Schedule

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

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

How do We Know if a Schedule is Serializable?

Notation:

\[
\begin{align*}
T_1: r_1(A); w_1(A); r_1(B); w_1(B) \\
T_2: r_2(A); w_2(A); r_2(B); w_2(B)
\end{align*}
\]

Key Idea: Focus on conflicting operations

Conflicts

- Write-Read – WR
- Read-Write – RW
- Write-Write – WW
- Read-Read?

Conflict Serializability

- A schedule is \textit{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
- The converse is not true (why?)
Conflict Serializability

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

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

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

This schedule is conflict-serializable

Example 2

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

Implementing Transactions

Example 2

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

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: Snapshot Isolation (SI)

We discuss only locking schedulers in this class
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

Now for something more serious...

More Notations

L(A) = transaction T_i acquires lock for element A
U_i(A) = transaction T_i 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></td>
</tr>
<tr>
<td>A := A+100</td>
<td>WRITE(A)</td>
</tr>
<tr>
<td></td>
<td>READ(A)</td>
</tr>
<tr>
<td>A := A^2</td>
<td>WRITE(A)</td>
</tr>
<tr>
<td>READ(B)</td>
<td></td>
</tr>
<tr>
<td>B := B^2</td>
<td>WRITE(B)</td>
</tr>
<tr>
<td></td>
<td>READ(B)</td>
</tr>
<tr>
<td>B := B+100</td>
<td>WRITE(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></td>
</tr>
<tr>
<td>A := A+100</td>
<td>WRITE(A); U_i(A); L_i(B)</td>
</tr>
<tr>
<td>L_i(A); READ(A)</td>
<td></td>
</tr>
<tr>
<td>A := A^2</td>
<td>WRITE(A); U_i(A); L_i(B)</td>
</tr>
<tr>
<td>READ(B)</td>
<td></td>
</tr>
<tr>
<td>B := B+100</td>
<td>WRITE(B); U_i(B); ...GRANTED; READ(B)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scheduler has ensured a conflict-serializable schedule
But…

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L₁(A); READ(A)</td>
<td></td>
</tr>
<tr>
<td>A := A + 100</td>
<td></td>
</tr>
<tr>
<td>WRITE(A); U₁₁(A);</td>
<td></td>
</tr>
<tr>
<td>L₂₁(B); READ(B)</td>
<td></td>
</tr>
<tr>
<td>B := B + 100</td>
<td></td>
</tr>
<tr>
<td>WRITE(B); U₁₂(B);</td>
<td></td>
</tr>
</tbody>
</table>

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

Example: 2PL transactions

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L₁₁(A); L₁₁(B); READ(A)</td>
<td></td>
</tr>
<tr>
<td>A := A + 100</td>
<td></td>
</tr>
<tr>
<td>WRITE(A); U₁₁(A);</td>
<td></td>
</tr>
<tr>
<td>READ(B)</td>
<td></td>
</tr>
<tr>
<td>B := B + 100</td>
<td></td>
</tr>
<tr>
<td>WRITE(B); U₁₂(B);</td>
<td></td>
</tr>
</tbody>
</table>

...GRANTED; READ(B) B := B*2 WRITE(B); U₂(B); U₁₂(B);

Now it is conflict-serializable