CSE 344
AUGUST 8TH
SCHEDULING
ADMINISTRIVIA

- WQ7 due Monday

- HW8 due Wednesday
  - uses JDBC API
  - (should be easy to figure out see the example code provided)
TRANSACTIONS

We use database transactions everyday

- Bank $$$ transfers
- Online shopping
- Signing up for classes

For this class, a transaction is a series of DB queries & updates

- Read / Write / Update / Delete / Insert
- Unit of work issued by a user that is independent from others
- (Note: we won’t talk about rows much here... transactions are a broader concept than databases)
KNOW YOUR 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

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
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.

**Fact:** nothing can go wrong if the system executes txns **serially**

- (rather, whatever does go wrong is the app’s fault)
- But DBMS don’t do that because we want better overall system performance
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 (i.e., where changing order can change result)
CONFLICT 
SERIALIZABILITY

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

Two actions by same transaction $T_i$:

Two writes by $T_i$, $T_j$ to same element

Read/write by $T_i$, $T_j$ to same element
A schedule is *conflict serializable* if it can be transformed into a serial schedule by a series of swaps of adjacent non-conflicting actions.

Every conflict-serializable schedule is serializable.

The converse is not true (why?)
CONFLICT SERIALIZABILITY

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$
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:

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

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

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

Example:

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

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

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

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

\[ \ldots \]

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

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

This schedule is NOT conflict-serializable
Scheduler = 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
CASE STUDY: 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)
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
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

READ LOCK

first write

RESERVED LOCK

commit requested

no more read locks

commit

EXCLUSIVE LOCK

commit executed
SCHEDULE ANOMALIES

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

\[ T_1: \text{WRITE}(A) \]

\[ T_1: \text{ABORT} \]

\[ T_2: \text{READ}(A) \]
INCONSISTENT READ

Write-Read Conflict

\[ T_1: A := 20; B := 20; \]
\[ T_1: \text{WRITE}(A) \]
\[ T_1: \text{WRITE}(B) \]

\[ T_2: \text{READ}(A); \]
\[ T_2: \text{READ}(B); \]
UNREPEATABLE READ

Read-Write Conflict

$T_1$: WRITE(A)

$T_2$: READ(A);
LOST UPDATE

Write-Write Conflict

$T_1$: READ(A)
$T_1$: A := A + 5
$T_1$: WRITE(A)

$T_2$: READ(A);
$T_2$: A := A * 1.3
$T_2$: WRITE(A);
MORE NOTATIONS

$L_i(A)$ = 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>READ(A)</td>
</tr>
<tr>
<td>A := A+100</td>
<td>A := A*2</td>
</tr>
<tr>
<td>WRITE(A)</td>
<td>WRITE(A)</td>
</tr>
<tr>
<td></td>
<td>READ(A)</td>
</tr>
<tr>
<td></td>
<td>A := A*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(A)</td>
</tr>
<tr>
<td></td>
<td>READ(B)</td>
</tr>
<tr>
<td></td>
<td>B := B*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B)</td>
</tr>
<tr>
<td></td>
<td>READ(B)</td>
</tr>
<tr>
<td></td>
<td>B := B+100</td>
</tr>
<tr>
<td></td>
<td>WRITE(B)</td>
</tr>
</tbody>
</table>
### EXAMPLE

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L₁(A); READ(A)</td>
<td>L₂(A); READ(A)</td>
</tr>
<tr>
<td>A := A + 100</td>
<td>A := A*2</td>
</tr>
<tr>
<td>WRITE(A); U₁(A); L₁(B)</td>
<td>WRITE(A); U₂(A);</td>
</tr>
<tr>
<td></td>
<td>L₂(B); BLOCKED…</td>
</tr>
<tr>
<td>READ(B)</td>
<td></td>
</tr>
<tr>
<td>B := B + 100</td>
<td>…GRANTED; READ(B)</td>
</tr>
<tr>
<td>WRITE(B); U₁(B);</td>
<td>B := B*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B); U₂(B);</td>
</tr>
</tbody>
</table>

Scheduler has ensured a conflict-serializable schedule
BUT...

\begin{tabular}{ll}
T1 & T2 \\
\hline
L_1(A); READ(A) & L_2(A); READ(A) \\
A := A+100 & A := A*2 \\
WRITE(A); U_1(A); & WRITE(A); U_2(A); \\

L_1(B); READ(B) & L_2(B); READ(B) \\
B := B+100 & B := B*2 \\
WRITE(B); U_1(B); & WRITE(B); U_2(B); \\
\end{tabular}

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><strong>L₁(A); L₁(B); READ(A)</strong></td>
<td><strong>L₂(A); READ(A)</strong></td>
</tr>
<tr>
<td>A := A+100</td>
<td>A := A*2</td>
</tr>
<tr>
<td>WRITE(A); U₁(A)</td>
<td>WRITE(A);</td>
</tr>
<tr>
<td><strong>READ(B)</strong></td>
<td><strong>L₂(B); BLOCKED…</strong></td>
</tr>
<tr>
<td>B := B+100</td>
<td></td>
</tr>
<tr>
<td>WRITE(B); U₁(B);</td>
<td></td>
</tr>
</tbody>
</table>

...GRANTED; READ(B)

B := B*2
WRITE(B); U₂(A); U₂(B);

Now it is conflict-serializable
TWO PHASE LOCKING (2PL)

Theorem: 2PL ensures conflict serializability