CSE 344

MARCH 23RD – SCHEDULING/LOCKING
ADMINISTRIVIA

• HW7 Due Tonight
• OQ6 Due Tonight
• HW8 Due Friday, June 1
  • Data without quotation marks
  • Extra credit
• OQ7 Due Wednesday, May 30
• Course Evaluations
  • Out over the weekend
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

- Read / Write / Update / Delete / Insert
- Unit of work issued by a user that is independent from others
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
  - 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
SERIAL SCHEDULE

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 transactions serially

- (up to what we have learned so far)
- But DBMS don’t do that because we want better overall system performance
### 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.
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 \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?)
**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
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  
reserved lock  
pending lock  
exclusive lock  
commit executed

commit  
first write  
commit requested  
no more read locks  
commit
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

$T_1$: WRITE(A)

$T_1$: ABORT

$T_2$: 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

T₁: WRITE(A)

T₂: READ(A);

T₂: READ(A);
LOST UPDATE

Write-Write Conflict

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

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

$L_i(A) = \text{transaction } T_i \text{ acquires lock for element } A$

$U_i(A) = \text{transaction } T_i \text{ releases lock for element } A$
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(B)</td>
</tr>
<tr>
<td></td>
<td>READ(B)</td>
</tr>
<tr>
<td></td>
<td>B := B*2</td>
</tr>
<tr>
<td></td>
<td>B := B+100</td>
</tr>
<tr>
<td></td>
<td>WRITE(B)</td>
</tr>
<tr>
<td></td>
<td>WRITE(B)</td>
</tr>
</tbody>
</table>
Scheduler has ensured a conflict-serializable schedule
BUT...

T1

L\(_1\)(A); READ(A)
A := A + 100
WRITE(A); U\(_1\)(A);

T2

L\(_2\)(A); READ(A)
A := A \times 2
WRITE(A); U\(_2\)(A);

L\(_2\)(B); READ(B)
B := B \times 2
WRITE(B); U\(_2\)(B);

L\(_1\)(B); READ(B)
B := B + 100
WRITE(B); U\(_1\)(B);

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

T1
L_1(A); L_1(B); READ(A)
A := A+100
WRITE(A); U_1(A)

T2
L_2(A); READ(A)
A := A*2
WRITE(A);
L_2(B); BLOCKED…

READ(B)
B := B+100
WRITE(B); U_1(B);

…GRANTED; READ(B)
B := B*2
WRITE(B); U_2(A); U_2(B);

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

**Theorem:** 2PL ensures conflict serializability
Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.
Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.

Then there is the following temporal cycle in the schedule:
Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph. Then there is the following temporal cycle in the schedule: $U_1(A) \rightarrow L_2(A)$ why?

$U_1(A)$ happened strictly before $L_2(A)$
**Theorem**: 2PL ensures conflict serializability

**Proof.** Suppose not: then there exists a cycle in the precedence graph.

Then there is the following temporal cycle in the schedule: $U_1(A) \rightarrow L_2(A)$ why?
Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.

Then there is the following \textit{temporal} cycle in the schedule:

\[ U_1(A) \rightarrow L_2(A) \]
\[ L_2(A) \rightarrow U_2(B) \]

why?

\[ L_2(A) \text{ happened strictly before } U_1(A) \]
Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.

Then there is the following temporal cycle in the schedule:

\[ U_1(A) \rightarrow L_2(A) \]
\[ L_2(A) \rightarrow U_2(B) \]

why?
Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.

Then there is the following temporal cycle in the schedule:

\[ U_1(A) \rightarrow L_2(A) \]
\[ L_2(A) \rightarrow U_2(B) \]
\[ U_2(B) \rightarrow L_3(B) \]

why?
**Theorem:** 2PL ensures conflict serializability

**Proof.** Suppose not: then there exists a cycle in the precedence graph.

Then there is the following **temporal** cycle in the schedule:

- $U_1(A) \rightarrow L_2(A)$
- $L_2(A) \rightarrow U_2(B)$
- $U_2(B) \rightarrow L_3(B)$

......etc......
Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.

Then there is the following *temporal* cycle in the schedule:

- $U_1(A) \rightarrow L_2(A)$
- $L_2(A) \rightarrow U_2(B)$
- $U_2(B) \rightarrow L_3(B)$
- $L_3(B) \rightarrow U_3(C)$
- $U_3(C) \rightarrow L_1(C)$
- $L_1(C) \rightarrow U_1(A)$

Cycle in time: Contradiction
A NEW PROBLEM: NON-RECOVERABLE SCHEDULE

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1(A); L_1(B); \text{READ}(A)$</td>
<td>$L_2(A); \text{READ}(A)$</td>
</tr>
<tr>
<td>$A := A+100$</td>
<td>$A := A*2$</td>
</tr>
<tr>
<td>WRITE(A); $U_1(A)$</td>
<td>WRITE(A);</td>
</tr>
<tr>
<td>READ(B)</td>
<td>$L_2(B); \text{BLOCKED…}$</td>
</tr>
<tr>
<td>$B := B+100$</td>
<td></td>
</tr>
<tr>
<td>WRITE(B); $U_1(B)$</td>
<td>...\text{GRANTED}; \text{READ}(B)</td>
</tr>
</tbody>
</table>

Rollback

Commit
A NEW PROBLEM:
NON-RECOVERABLE SCHEDULE

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(L_1(A); L_1(B); \text{READ}(A))</td>
<td>(L_2(A); \text{READ}(A))</td>
</tr>
<tr>
<td>(A := A + 100)</td>
<td>(A := A \times 2)</td>
</tr>
<tr>
<td>\text{WRITE}(A); (U_1(A))</td>
<td>\text{WRITE}(A); (U_2(A))</td>
</tr>
<tr>
<td>(\text{READ}(B))</td>
<td>(\text{READ}(B))</td>
</tr>
<tr>
<td>(B := B + 100)</td>
<td>(B := B \times 2)</td>
</tr>
<tr>
<td>\text{WRITE}(B); (U_1(B));</td>
<td>\text{WRITE}(B); (U_2(A); U_2(B));</td>
</tr>
<tr>
<td></td>
<td>\text{Commit}</td>
</tr>
</tbody>
</table>

Elements A, B written by T1 are restored to their original value.
A NEW PROBLEM:
NON-RECOVERABLE SCHEDULE

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

Commit

Rollback

Elements A, B written by T1 are restored to their original value.

Dirty reads of A, B lead to incorrect writes.
A NEW PROBLEM:
NON-RECOVERABLE SCHEDULE

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L_1(A); L_1(B); )</td>
<td>( L_2(A); )</td>
</tr>
<tr>
<td>READ(A)</td>
<td>READ(A)</td>
</tr>
<tr>
<td>( A := A + 100 )</td>
<td>( A := A \times 2 )</td>
</tr>
<tr>
<td>WRITE(A); ( U_1(A) )</td>
<td>WRITE(A);</td>
</tr>
<tr>
<td>( )</td>
<td>( L_2(B); ) BLOCKED…</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>
<tr>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>( )</td>
<td>( )</td>
</tr>
</tbody>
</table>

Elements A, B written by T1 are restored to their original value.

Dirty reads of A, B lead to incorrect writes.

Can no longer undo!
The Strict 2PL rule:

All locks are held until commit/abort:
All unlocks are done together with commit/abort.

With strict 2PL, we will get schedules that
are both conflict-serializable and recoverable
### STRICT 2PL

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

Lock-based systems always use strict 2PL

Easy to implement:

- Before a transaction reads or writes an element A, insert an L(A)
- When the transaction commits/aborts, then release all locks

Ensures both conflict serializability and recoverability