Announcements

• Final exam will be on Dec. 14 (next Thursday) 14:30-16:20 in class
  – Note the time difference, the exam will last ~2 hours
Recap

• What are transactions?
  – Why do we need them?

• Maintain ACID properties via schedules
  – We focus on the isolation property
  – We briefly discussed consistency & durability
  – We do not discuss atomicity

• Ensure conflict-serializable schedules with locks
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 in 414
Locking Scheduler

Simple idea:
• Each element has a unique lock
• Each transaction must first acquire the lock before reading/writing that element
• If lock is taken by another transaction, then wait
• The transaction must release the lock(s)

By using locks, scheduler can ensure 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.
  – can be even more fine-grained by having different types of locks (allows more txns to run simultaneously)
SQLite

- None ➔ READ LOCK
- first write ➔ RESERVED LOCK
- commit requested ➔ PENDING LOCK
- no more read locks ➔ EXCLUSIVE LOCK
- commit ➔ None
- commit executed ➔ commit requested

CSE 414 - Fall 2017
Locks in the Abstract
Notation

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

T1

L₁(A); READ(A)
A := A + 100
WRITE(A); U₁(A); L₁(B)

READ(B)
B := B + 100
WRITE(B); U₁(B);

T2

L₂(A); READ(A)
A := A * 2
WRITE(A); U₂(A);
L₂(B); BLOCKED…

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

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>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); U₂(A);</td>
</tr>
<tr>
<td>L₂(B); READ(B)</td>
<td>L₂(B); 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(B);</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

2PL approach developed by Jim Gray
Example: 2PL transactions

<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>READ(B)</td>
<td>L₂(B); BLOCKED…</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(A); U₂(B);</td>
</tr>
</tbody>
</table>

Now it is conflict-serializable
Two Phase Locking (2PL)

**Theorem:** 2PL ensures conflict serializability
Two Phase Locking (2PL)

**Theorem:** 2PL ensures conflict serializability

**Proof.** Suppose not: then there exists a cycle in the precedence graph.
Two Phase Locking (2PL)

**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:

![Diagram](image)
Two Phase Locking (2PL)

**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?
Two Phase Locking (2PL)

**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?
Two Phase Locking (2PL)

**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:

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

Contradiction
A New Problem: Non-recoverable Schedule

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

Rollback
Strict 2PL

The Strict 2PL rule:

All locks are held until the transaction commits or aborts.

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; U₁(A), U₁(B)</td>
<td>…GRANTED; 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>L₂(B); 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>COMMIT; U₂(A); U₂(B)</td>
</tr>
</tbody>
</table>
Another problem: Deadlocks

- $T_1$ waits for a lock held by $T_2$;
- $T_2$ waits for a lock held by $T_3$;
- $T_3$ waits for . . . .
- . . .
- $T_n$ waits for a lock held by $T_1$

**SQL Lite:** there is only one exclusive lock; thus, never deadlocks

**SQL Server:** checks periodically for deadlocks and aborts one TXN
Lock Modes

- **S** = shared lock (for READ)
- **X** = exclusive lock (for WRITE)

**Lock compatibility matrix:**

<table>
<thead>
<tr>
<th></th>
<th>None</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lock Modes

- **S** = shared lock (for READ)
- **X** = exclusive lock (for WRITE)

Lock compatibility matrix:

<table>
<thead>
<tr>
<th></th>
<th>None</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>S</td>
<td>✔</td>
<td>✔</td>
<td>✖</td>
</tr>
<tr>
<td>X</td>
<td>✔</td>
<td>✖</td>
<td>✖</td>
</tr>
</tbody>
</table>

CSE 414 - Fall 2017 26
Lock Granularity

• **Fine granularity locking** (e.g., tuples)
  – High concurrency
  – High overhead in managing locks
  – E.g. SQL Server

• **Coarse grain locking** (e.g., tables, entire database)
  – Many false conflicts
  – Less overhead in managing locks
  – E.g. SQLite

• **Solution: lock escalation changes granularity as needed**
Lock Performance

Throughput (TPS) vs. # Active Transactions

TPS = Transactions per second

To avoid, use admission control

Why?

thrashing
Phantom Problem

• So far we have assumed the database to be a *static* collection of elements (=tuples)

• If tuples are inserted/deleted then the *phantom problem* appears
Suppose there are two blue products A1 & A2

Phantom Problem

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT *</td>
<td>INSERT INTO Product(name, color) VALUES ('A3', 'blue')</td>
</tr>
<tr>
<td>FROM Product</td>
<td></td>
</tr>
<tr>
<td>WHERE color='blue'</td>
<td></td>
</tr>
</tbody>
</table>
Suppose there are two blue products A1 & A2

**Phantom Problem**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT *</td>
<td>INSERT INTO Product(name, color)</td>
</tr>
<tr>
<td>FROM Product</td>
<td>VALUES (‘A3’, ‘blue’)</td>
</tr>
<tr>
<td>WHERE color=‘blue’</td>
<td></td>
</tr>
<tr>
<td>SELECT *</td>
<td></td>
</tr>
<tr>
<td>FROM Product</td>
<td></td>
</tr>
<tr>
<td>WHERE color=‘blue’</td>
<td></td>
</tr>
</tbody>
</table>

R1(A1), R1(A2), W2(A3), R1(A1), R1(A2), R1(A3)
Suppose there are two blue products A1 & A2

Phantom Problem

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT *</td>
<td>INSERT INTO Product(name, color) VALUES ('A3', 'blue')</td>
</tr>
<tr>
<td>FROM Product</td>
<td></td>
</tr>
<tr>
<td>WHERE color='blue'</td>
<td></td>
</tr>
</tbody>
</table>

R1(A1), R1(A2), W2(A3), R1(A1), R1(A2), R1(A3)
Phantom Problem

• A “phantom” is a tuple that is invisible during part of a transaction execution, but not invisible during the entire execution

• In our example:
  – T1: reads list of products
  – T2: inserts a new product
  – T1: re-reads: a new product appears!
Dealing With Phantoms

- Lock the entire table
- Lock the index entry for ‘blue’
  - If index is available
- Or use predicate locks
  - A lock on an arbitrary predicate

Dealing with phantoms is expensive!
Locking & SQL
Isolation Levels in SQL

1. “Dirty reads”
   SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

2. “Committed reads”
   SET TRANSACTION ISOLATION LEVEL READ COMMITTED

3. “Repeatable reads”
   SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

4. Serializable transactions
   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
1. Isolation Level: Dirty Reads

- “Long duration” WRITE locks
  - Strict 2PL
- No READ locks
  - Read-only transactions are never delayed

Possible problems: dirty and inconsistent reads
2. Isolation Level: Read Committed

- “Long duration” WRITE locks
  - Strict 2PL

- “Short duration” READ locks
  - Only acquire lock while reading (not 2PL)

Unrepeateable reads
When reading same element twice, may get two different values
3. Isolation Level: Repeatable Read

- “Long duration” WRITE locks
  - Strict 2PL
- “Long duration” READ locks
  - Strict 2PL

This is not serializable yet !!!
4. Isolation Level Serializable

- “Long duration” WRITE locks
  - Strict 2PL
- “Long duration” READ locks
  - Strict 2PL
- Predicate locking
  - To deal with phantoms
Beware!

In commercial DBMSs:
- Default level is often **NOT** serializable (SQL Server!)
- Default level differs between DBMSs
- Some engines support subset of levels
- Serializable may not be exactly ACID
  - Locking ensures isolation, not atomicity
- Also, some DBMSs do NOT use locking. Different isolation levels can lead to different problems
- **Bottom line:** Read the doc for your DBMS!
Next two slides: try them on Azure
Demonstration with SQL Server

**Application 1:**
create table R(a int);
insert into R values(1);
set transaction isolation level **serializable**;
begin transaction;
select * from R; -- get a shared lock
waitfor delay '00:01'; -- wait for one minute

**Application 2:**
set transaction isolation level **serializable**;
begin transaction;
select * from R; -- get a shared lock
insert into R values(2); -- blocked waiting on exclusive lock
    -- App 2 unblocks and executes insert after app 1 commits/aborts
Demonstration with SQL Server

Application 1:
create table R(a int);
insert into R values(1);
set transaction isolation level repeatable read;
begin transaction;
select * from R; -- get a shared lock
waitfor delay '00:01';  -- wait for one minute

Application 2:
set transaction isolation level repeatable read;
begin transaction;
select * from R; -- get a shared lock
insert into R values(3); -- gets an exclusive lock on new tuple
   -- If app 1 reads now, it blocks because read dirty
   -- If app 1 reads after app 2 commits, app 1 sees new value