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

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 transactions to run simultaneously)

By using locks, scheduler can ensure conflict-serializability
**Notation**

\[ L(A) = \text{transaction } T_i \text{ acquires lock for element } A \]

\[ U(A) = \text{transaction } T_i \text{ releases lock for element } A \]

**A Non-Serializable Schedule**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{READ}(A)</td>
<td>\text{READ}(A)</td>
</tr>
<tr>
<td>A := A + 100</td>
<td>A := A^2</td>
</tr>
<tr>
<td>\text{WRITE}(A)</td>
<td>\text{WRITE}(A)</td>
</tr>
<tr>
<td>\text{READ}(A)</td>
<td>\text{READ}(B)</td>
</tr>
<tr>
<td>B := B + 100</td>
<td>B := B^2</td>
</tr>
<tr>
<td>\text{WRITE}(B)</td>
<td>\text{WRITE}(B)</td>
</tr>
</tbody>
</table>

**Example**

**T1**

\[ L_i(A); \text{READ}(A) \]

A := A + 100

\[ \text{WRITE}(A); U_i(A); L_i(B) \]

\[ \text{READ}(B) \]

B := B + 100

\[ \text{WRITE}(B); U_i(B) \]

**T2**

\[ L_i(A); \text{READ}(A) \]

A := A + 100

\[ \text{WRITE}(A); U_i(A) \]

\[ \text{READ}(A) \]

A := A^2

\[ \text{WRITE}(A); U_i(A); L_i(B); \text{READ}(B) \]

B := B^2

\[ \text{WRITE}(B); U_i(B) \]

\[ \text{LOCK} \]

Scheduler has ensured a conflict-serializable schedule

**But...**

**T1**

\[ L_i(A); \text{READ}(A) \]

A := A + 100

\[ \text{WRITE}(A); U_i(A); L_i(B) \]

\[ \text{READ}(B) \]

B := B + 100

\[ \text{WRITE}(B); U_i(B) \]

**T2**

\[ L_i(A); \text{READ}(A) \]

A := A^2

\[ \text{WRITE}(A); U_i(A); L_i(B); \text{READ}(B) \]

B := B^2

\[ \text{WRITE}(B); U_i(B) \]

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.

2PL approach developed by Jim Gray.

Example: 2PL transactions

T1
\( L_1(A); L_1(B); \text{READ}(A) \)
\( A := A + 100 \)
\( \text{WRITE}(A); U_1(A) \)

T2
\( L_2(A); \text{READ}(A) \)
\( A := A^2 \)
\( \text{WRITE}(A); U_2(A) \)

READ(B)
\( B := B + 100 \)
\( \text{WRITE}(B); U_1(B) \)

Now it is conflict-serializable

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:

\( U_1(A) \rightarrow L_1(A) \) 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:

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

why?

A New Problem: Non-recoverable Schedule

Strand 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

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)

<table>
<thead>
<tr>
<th>Lock compatibility matrix:</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td>X</td>
</tr>
</tbody>
</table>

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

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

T1

```
SELECT * FROM Product
WHERE color='blue'
```

T2

```
INSERT INTO Product(name, color)
VALUES ('A3', 'blue')
```

```
SELECT * FROM Product
WHERE color='blue'
```

Is this schedule serializable?
Suppose there are two blue products A1 & A2

Phantom Problem

T1                     T2
SELECT *              SELECT *
FROM Product          FROM Product
WHERE color = 'blue'  WHERE color = 'blue'

INSERT INTO Product(name, color) VALUES ('A3', 'blue')

SELECT * FROM Product WHERE color = 'blue'

R1(A1), R1(A2), R1(A1), R1(A2), R1(A3)

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!

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

ACID
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)
   Unrepeatable 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