Introduction to Data Management
CSE 344

Lecture 21:
Transaction Implementations
Announcements

• WQ7 and HW7 are out
  – Due next Mon and Wed
  – Start early, there is little time!
Review: 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: Schedules, schedules, schedules

• The DBMS scheduler determines the order of operations from txns are executed

• A *serial schedule* is one in which transactions are executed one after the other, in some sequential order

• A schedule is *serializable* if it is equivalent to a serial schedule

• A schedule is *conflict serializable* if it has the same conflicts as a serial schedule
Review: Conflicts

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  
$w_i(X); r_j(X)$

$w_i(X); w_j(X)$
Conflict Serializability

• How to show that a schedule has the same conflicts as a serial schedule?

• Show that it can be transformed into a serial schedule!
  – By moving the non conflicting operations around
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:

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

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

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

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$
Conflict Serializability

Example:

\[
\begin{align*}
&\text{r}_1(A); \text{w}_1(A); \text{r}_2(A); \text{w}_2(A); \text{r}_1(B); \text{w}_1(B); \text{r}_2(B); \text{w}_2(B) \\
&\text{r}_1(A); \text{w}_1(A); \text{r}_2(A); \text{r}_1(B); \text{w}_2(A); \text{w}_1(B); \text{r}_2(B); \text{w}_2(B) \\
&\text{r}_1(A); \text{w}_1(A); \text{r}_1(B); \text{r}_2(A); \text{w}_2(A); \text{w}_1(B); \text{r}_2(B); \text{w}_2(B) \\
&\ldots \\
&\text{r}_1(A); \text{w}_1(A); \text{r}_1(B); \text{w}_1(B); \text{r}_2(A); \text{w}_2(A); \text{r}_2(B); \text{w}_2(B)
\end{align*}
\]
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

This schedule is NOT conflict-serializable
Implementing a Scheduler
Implementing a Scheduler

• Real-world DBMSs runs multiple threads
  – Each thread executes a txn

• How to ensure that the resulting threads implement a conflict serializable schedule?
A Serializable Schedule

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

READ(B, t)
t := t + 100
WRITE(B, t)

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

READ(B, s)
s := s * 2
WRITE(B, s)

Executed by thread #1
Executed by thread #2
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 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
SQLite

- SQLite is very simple

- Lock types
  - READ LOCK (to read)
  - RESERVED LOCK (to write)
  - PENDING LOCK (wants to commit)
  - EXCLUSIVE LOCK (to commit)

More details in the following slides
SQLite

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**
SQLite

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
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

None → READ LOCK
READ LOCK → RESERVED LOCK
RESERVED LOCK → PENDING LOCK
PENDING LOCK → EXCLUSIVE LOCK
EXCLUSIVE LOCK → None

begin transaction → first write
first write → commit requested
commit requested → no more read locks
no more read locks → commit
commit → commit executed

commit executed

CSE 344 - Winter 2017
create table r(a int, b int);
insert into r values (1,10);
insert into r values (2,20);
insert into r values (3,30);
Demonstrating Locking in SQLite

T1:

begin transaction;
select * from r;
-- T1 has a READ LOCK

T2:

begin transaction;
select * from r;
-- T2 has a READ LOCK
Demonstrating Locking in SQLite

T1:

update r set b=11 where a=1;
-- T1 has a RESERVED LOCK

T2:

update r set b=21 where a=2;
-- T2 asked for a RESERVED LOCK: DENIED
Demonstrating Locking in SQLite

T3:

begin transaction;
select * from r;
commit;

-- everything works fine, could obtain READ LOCK
Demonstrating Locking in SQLite

T1:

```sql
commit;
-- SQL error: database is locked
-- T1 asked for PENDING LOCK -- GRANTED
-- T1 asked for EXCLUSIVE LOCK -- DENIED
```
Demonstrating Locking in SQLite

T3':
begin transaction;
select * from r;
-- T3 asked for READ LOCK-- DENIED (due to T1)

T2:
commit;
-- releases the last READ LOCK; T1 can commit
Now for something more serious…
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

\begin{align*}
T1 & \quad T2 \\
READ(A) & \quad \text{READ(A)} \\
A := A + 100 & \quad A := A \times 2 \\
WRITE(A) & \quad WRITE(A) \\
 & \quad \text{READ(B)} \\
B := B + 100 & \quad B := B \times 2 \\
WRITE(B) & \quad WRITE(B)
\end{align*}
Scheduler has ensured a conflict-serializable schedule
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L_1(A); ) READ(A)</td>
<td>( L_2(A); ) 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); ( U_2(A); )</td>
</tr>
<tr>
<td></td>
<td>( L_2(B); ) READ(B)</td>
</tr>
<tr>
<td></td>
<td>B := B*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B); ( U_2(B); )</td>
</tr>
</tbody>
</table>

\[ 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

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

\[ T2 \]
\[ \# \text{locks} \]
\[ L_2(A); \text{READ}(A) \]
\[ A := A \times 2 \]
\[ \text{WRITE}(A); \]
\[ L_2(B); \text{BLOCKED…} \]
\[ …\text{GRANTED}; \text{READ}(B) \]
\[ B := B \times 2 \]
\[ \text{WRITE}(B); U_2(A); U_2(B); \]

Now it is conflict-serializable
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>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>
<tr>
<td></td>
<td>Commit</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

T1

L₁(A); READ(A)
A := A + 100
WRITE(A);

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

T2

L₂(A); BLOCKED…

…GRANTED; READ(A)
A := A * 2
WRITE(A);
L₂(B); READ(B)
B := B * 2
WRITE(B);
Commit
U₂(A); U₂(B);
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 $\ldots$.
- $\ldots$
- $T_n$ 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