CSE 444: Database Internals

Lectures 14
Transactions: Locking

Announcements
- Quiz 1+2 Friday in class
- Last quarter’s quiz linked in calendar
- 1 page (2 sides) of notes allowed
- Most important things to study are labs, including your implementation choices and the related material
- HW 5 due tonight
- Lab 3 part 1 due Saturday 11pm
- Less times for labs in general

Scheduler
- The scheduler:
  - Module that schedules the transaction’s actions, ensuring serializability
- Two main approaches
  - Pessimistic: locks
  - Optimistic: timestamps, multi-version, validation

Pessimistic 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)

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

\[
\begin{align*}
T_1 & \quad T_2 \\
\text{READ}(A, t) & \quad \text{READ}(A, s) \\
\text{WRITE}(A, t) & \quad s := s^2 \\
\text{WRITE}(A, s) & \quad \text{READ}(B, s) \\
\text{READ}(B, s) & \quad s := s^2 \\
\text{WRITE}(B, s) & \quad \text{READ}(B, t) \\
\text{READ}(B, t) & \quad t := t+100 \\
\text{WRITE}(B, t) & 
\end{align*}
\]
Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L_t(A); READ(A, t)</td>
<td></td>
</tr>
<tr>
<td>t := t+100</td>
<td></td>
</tr>
<tr>
<td>WRITE(A, t); U_t(A); L_t(B)</td>
<td></td>
</tr>
<tr>
<td>L_t(A); READ(A, s)</td>
<td></td>
</tr>
<tr>
<td>s := s*2</td>
<td></td>
</tr>
<tr>
<td>WRITE(A, s); U_t(B); L_t(B)</td>
<td></td>
</tr>
</tbody>
</table>

READ(B, t)        

\[ t := t+100 \]

WRITE(B, t)

\[ U_t(B); \]

L_1(B); READ(B, t)

\[ t := t+100 \]

WRITE(B, t)

\[ U_t(B); \]

Scheduler has ensured a conflict-serializable schedule

Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L_t(A); READ(A, t)</td>
<td></td>
</tr>
<tr>
<td>t := t+100</td>
<td></td>
</tr>
<tr>
<td>WRITE(A, t); U_t(A); L_t(B)</td>
<td></td>
</tr>
<tr>
<td>L_t(A); READ(A, s)</td>
<td></td>
</tr>
<tr>
<td>s := s*2</td>
<td></td>
</tr>
<tr>
<td>WRITE(A, s); U_t(B); L_t(B)</td>
<td></td>
</tr>
</tbody>
</table>

READ(B, t)        

\[ t := t+100 \]

WRITE(B, t)

\[ U_t(B); \]

L_1(B); READ(B, t)

\[ t := t+100 \]

WRITE(B, t)

\[ U_t(B); \]

...GRANTED; READ(B, s)

\[ s := s*2 \]

WRITE(B, s)

\[ U_t(B); U_t(B) \]

But...

Example with Multiple Transactions

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L_t(A); L_t(B); READ(A, t)</td>
<td></td>
</tr>
<tr>
<td>t := t+100</td>
<td></td>
</tr>
<tr>
<td>WRITE(A, t); U_t(A); L_t(B)</td>
<td></td>
</tr>
<tr>
<td>L_t(A); READ(A, s)</td>
<td></td>
</tr>
<tr>
<td>s := s*2</td>
<td></td>
</tr>
<tr>
<td>WRITE(A, s); U_t(B); L_t(B)</td>
<td></td>
</tr>
</tbody>
</table>

READ(B, t)        

\[ t := t+100 \]

WRITE(B, t)

\[ U_t(B); \]

L_1(B); READ(B, t)

\[ t := t+100 \]

WRITE(B, t)

\[ U_t(B); \]

L_1(B); L_2(B); DENIED...

Two Phase Locking (2PL)

The 2PL rule:

- In every transaction, all lock requests must precede all unlock requests
- This ensures conflict serializability! (will prove this shortly)

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Example: 2PL transactions

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L_t(A); L_t(B); READ(A, t)</td>
<td></td>
</tr>
<tr>
<td>t := t+100</td>
<td></td>
</tr>
<tr>
<td>WRITE(A, t); U_t(A); L_t(B)</td>
<td></td>
</tr>
<tr>
<td>L_t(A); READ(A, s)</td>
<td></td>
</tr>
<tr>
<td>s := s*2</td>
<td></td>
</tr>
<tr>
<td>WRITE(A, s); U_t(B); L_t(B)</td>
<td></td>
</tr>
</tbody>
</table>

READ(B, t)        

\[ t := t+100 \]

WRITE(B, t)

\[ U_t(B); \]

...GRANTED; READ(B, s)

\[ s := s*2 \]

WRITE(B, s)

\[ U_t(B); U_t(B) \]

Now it is conflict-serializable

Example with Multiple Transactions

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Growing phase</td>
<td></td>
</tr>
<tr>
<td>Shrinking phase</td>
<td></td>
</tr>
</tbody>
</table>

T1          

Unlocks second so perhaps was waiting for T3

T2          

Unlocks first Was not waiting for anyone

T3          

Equivalent to each transaction executing entirely the moment it enters shrinking phase

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

CSE 444 - Spring 2019
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:

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

**Contradiction**

A New Problem:
Non-recoverable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1(A)$; $L_1(B)$; READ($A$, t)</td>
<td></td>
</tr>
<tr>
<td>$t := t+100$</td>
<td></td>
</tr>
<tr>
<td>WRITE($A$, t); $U_1(A)$</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>READ($B$, t)</td>
<td></td>
</tr>
<tr>
<td>$s := s*2$</td>
<td></td>
</tr>
<tr>
<td>WRITE($B$, t); $U_1(B)$; $U_2(B)$; Commit</td>
<td></td>
</tr>
</tbody>
</table>

Abort
Strict 2PL

- Strict 2PL: All locks held by a transaction are released when the transaction is completed; release happens at the time of COMMIT or ROLLBACK
- Schedule is recoverable
- Schedule avoids cascading aborts

Summary of Strict 2PL

- Ensures serializability, recoverability, and avoids cascading aborts
- Issues?

The Locking Scheduler

Task 1: -- act on behalf of the transaction
- Add lock/unlock requests to transactions
- Examine all READ(A) or WRITE(A) actions
- Add appropriate lock requests
- On COMMIT/ROLLBACK release all locks
- Ensures Strict 2PL!

Task 2: -- act on behalf of the system
- Execute the locks accordingly
  - Lock table: a big, critical data structure in a DBMS!
  - When a lock is requested, check the lock table
    - Grant, or add the transaction to the element’s wait list
  - When a lock is released, re-activate a transaction from its wait list
  - When a transaction aborts, release all its locks
  - Check for deadlocks occasionally
## Lock Modes
- **S** = shared lock (for READ)
- **X** = exclusive lock (for WRITE)

<table>
<thead>
<tr>
<th></th>
<th>None</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>S</td>
<td>OK</td>
<td>OK</td>
<td>Conflict</td>
</tr>
<tr>
<td>X</td>
<td>OK</td>
<td>Conflict</td>
<td>Conflict</td>
</tr>
</tbody>
</table>

## Lock Granularity
- **Fine granularity locking** (e.g., tuples)
  - High concurrency
  - High overhead in managing locks
- **Coarse grain locking** (e.g., tables, predicate locks)
  - Many false conflicts
  - Less overhead in managing locks

## Deadlocks
- **Cycle in the wait-for graph:**
  - T1 waits for T2
  - T2 waits for T3
  - T3 waits for T1
- **Deadlock detection**
  - Timeouts
  - Wait-for graph
- **Deadlock avoidance**
  - Acquire locks in pre-defined order
  - Acquire all locks at once before starting

## Lock Performance
- Throughput vs. # Active Transactions
- Thrashing
- Why?
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, X1, X2:

\[ R_1(X_1), R_1(X_2), W_2(X_3), R_1(X_1), R_1(X_2), R_1(X_3) \]

### Is this schedule serializable?

**Phantom Problem**

T1

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

T2

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

Is this schedule serializable?

**Phantom Problem**

T1

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

T2

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

This is conflict serializable! What's wrong??

**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!

**Phantom Problem**

Not serializable due to **phantoms**
Phantom Problem

- In a static database:
  - Conflict serializability implies serializability

- In a dynamic database, this may fail due to phantoms

- Strict 2PL guarantees conflict serializability, but not serializability

Dealing With Phantoms

- Lock the entire table, or
- 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

Possible pbs: dirty and inconsistent reads

1. Isolation Level: Dirty Reads

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

2. Isolation Level: Read Committed

- "Long duration" WRITE locks
  - Strict 2PL
- "Short duration" READ locks
  - Only acquire lock while reading (not 2PL)

If 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

READ-ONLY Transactions

Client 1:
START TRANSACTION
INSERT INTO SmallProduct(name, price)
SELECT pname, price
FROM Product
WHERE price <= 0.99
DELETE FROM Product
WHERE price <=0.99
COMMIT

Client 2:
SET TRANSACTION READ ONLY
START TRANSACTION
SELECT count(*)
FROM Product
SELECT count(*)
FROM SmallProduct
COMMIT

Commercial Systems

Always check documentation!
- **DB2**: Strict 2PL
- **SQL Server**:
  - Strict 2PL for standard 4 levels of isolation
  - Multiversion concurrency control for snapshot isolation
- **PostgreSQL**: Snapshot isolation; recently: serializable Snapshot isolation (!)
- **Oracle**: Snapshot isolation

May improve performance