CS 457: Database Management Systems

Transactions: Locking
Review of Schedules

Serializability

- Serial
- Serializable
- Conflict serializable
- View serializable

Recoverability

- Recoverable
- Avoids cascading aborts
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

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

\( u_i(A) = \text{transaction } T_i \text{ releases lock for element } A \)
Example

T1

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

T2

L₂(A); READ(A,s)
s := s*2
WRITE(A,s); U₂(A);
L₂(B); DENIED…

READ(B, t)
t := t+100
WRITE(B,t); U₁(B);

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

Scheduler has ensured a serializable schedule
A Non-Serializable Schedule

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

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

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

Locks did not enforce serializability !!! What’s wrong?
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)
### Example: 2PL transactions

<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>$L_2(A)$; READ(A,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t); $U_1(A)$</td>
<td>WRITE(A,s);</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>$L_2(B)$; DENIED…</td>
</tr>
<tr>
<td>t := t+100</td>
<td>…GRANTED; READ(B,s)</td>
</tr>
<tr>
<td>WRITE(B,t); $U_1(B)$</td>
<td>s := s*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B,s); $U_2(A)$</td>
</tr>
<tr>
<td></td>
<td>$U_2(B)$;</td>
</tr>
</tbody>
</table>

Now it is 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:
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?
Theorem: 2PL ensures conflict serializability

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

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

\begin{align*}
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)
\end{align*}

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, t))</td>
<td>(L_2(A); \text{READ}(A, s))</td>
</tr>
<tr>
<td>(t := t + 100)</td>
<td>(s := s \times 2)</td>
</tr>
<tr>
<td>(\text{WRITE}(A, t); U_1(A))</td>
<td>(\text{WRITE}(A, s))</td>
</tr>
<tr>
<td>(\text{READ}(B, t))</td>
<td>(L_2(B); \text{DENIED…})</td>
</tr>
<tr>
<td>(t := t + 100)</td>
<td>(s := s \times 2)</td>
</tr>
<tr>
<td>(\text{WRITE}(B, t); U_1(B))</td>
<td>(\text{WRITE}(B, s); U_2(A); U_2(B))</td>
</tr>
<tr>
<td>(\text{…GRANTED; READ}(B, s))</td>
<td>(\text{Commit})</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 **serializable**

• Schedule is **recoverable**

• Schedule **avoids cascading aborts**
Strict 2PL

T1

L₁(A); READ(A)
A := A + 100
WRITE(A);

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

T2

L₂(A); DENIED…

…GRANTED; READ(A)
A := A*2
WRITE(A);
L₂(B); READ(B)
B := B*2
WRITE(B);
U₂(A), U₂(B); Commit
Summary of Strict 2PL

- Ensures serializability, recoverability, and avoids cascading aborts

- Issues: implementation, granularity, performance
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!
The Locking Scheduler

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)

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>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

• **Alternative techniques**
  – Hierarchical locking (and intentional locks) [commercial DBMSs]
  – Lock escalation
Hierarchical Locking

- To enable both coarse- and fine-grained locking
- Consider database as a hierarchy
  - Relations are largest lockable elements
  - Relations consist of blocks/pages
  - Blocks contain tuples
- To place a lock on an element, start at the top
  - If at element to lock, get an S or X lock on it
  - If want to lock an element deeper in the hierarchy
    - Leave an *intentional* lock: IS or IX
Hierarchical Locking

<table>
<thead>
<tr>
<th></th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>SIX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>IX</td>
<td>y</td>
<td>y</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>S</td>
<td>y</td>
<td>n</td>
<td>y</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>SIX</td>
<td>y</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>X</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
</tbody>
</table>

Table 2: Compatibility Matrix for Regular and Intention Locks

<table>
<thead>
<tr>
<th>To Get</th>
<th>Must Have on all Ancestors</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS or S</td>
<td>IS or IX</td>
</tr>
<tr>
<td>IX, SIX, or X</td>
<td>IX or SIX</td>
</tr>
</tbody>
</table>

Table 3: Hierarchical Locking Rules

From Michael Franklin, Concurrency Control and Recovery, 1997
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
  – Think about your OS class…
Lock Performance

Throughput

# Active Transactions

thrashing

Why?
The Tree Protocol

• An alternative to 2PL, for tree structures
• E.g. B-trees (the indexes of choice in databases)

• Because
  – Indexes are hot spots!
  – 2PL would lead to great lock contention
The Tree Protocol (SKIP)

Rules:

• The first lock may be any node of the tree
• Subsequently, a lock on a node A may only be acquired if the transaction holds a lock on its parent B
• Nodes can be unlocked in any order (no 2PL necessary)
• “Crabbing”
  – First lock parent then lock child
  – Keep parent locked only if may need to update it
  – Release lock on parent if child is not full

• The tree protocol is NOT 2PL, yet ensures conflict-serializability!
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
Phantom Problem

Is this schedule serializable? **NO!**

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

SELECT *
FROM Product
WHERE color='blue'

SELECT *
FROM Product
WHERE color='blue'
Phantom Problem

Suppose there are two blue products, X1, X2:

R1(X1), R1(X2), W2(X3), R1(X1), R1(X2), R1(X3)

T1

| SELECT * |
| FROM Product |
| WHERE color=‘blue’ |

T2

| INSERT INTO Product(name, color) |
| VALUES (‘gizmo’,’blue’) |

| SELECT * |
| FROM Product |
| WHERE color=‘blue’ |
Phantom Problem

Suppose there are two blue products, X1, X2:

R1(X1), R1(X2), W2(X3), R1(X1), R1(X2), R1(X3)

This is conflict serializable! What’s wrong??
### Phantom Problem

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

Suppose there are two blue products, X₁, X₂:

R₁(X₁), R₁(X₂), W₂(X₃), R₁(X₁), R₁(X₂), R₁(X₃)

Not serializable due to **phantoms**
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

• 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
• 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
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

May improve performance