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

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1(A)$; READ($A$, $t$)</td>
<td>$L_2(A)$; READ($A$, $s$)</td>
</tr>
<tr>
<td>$t := t + 100$</td>
<td>$s := s \times 2$</td>
</tr>
<tr>
<td>WRITE($A$, $t$); $U_1(A)$; $L_1(B)$</td>
<td>WRITE($A$, $s$); $U_2(A)$; $L_2(B)$; DENIED…</td>
</tr>
<tr>
<td>READ($B$, $t$)</td>
<td>WRITE($B$, $s$); $U_2(B)$;</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></td>
</tr>
<tr>
<td></td>
<td>$s := s \times 2$</td>
</tr>
<tr>
<td></td>
<td>WRITE($B$, $s$); $U_2(B)$;</td>
</tr>
</tbody>
</table>

Scheduler has ensured a serializable schedule
### A Non-Serializable Schedule

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

T1

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

T2

\[ L_2(A); \text{READ}(A, s) \]
\[ s := s \times 2 \]
\[ \text{WRITE}(A, s); \]
\[ L_2(B); \text{DENIED...} \]

\[ \text{READ}(B, t) \]
\[ t := t + 100 \]
\[ \text{WRITE}(B, t); U_1(B); \]

\[ \text{...GRANTED; READ}(B, s) \]
\[ s := s \times 2 \]
\[ \text{WRITE}(B, s); U_2(A); U_2(B); \]

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:
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:

$$U_1(A) \rightarrow L_2(A) \quad \text{why?}$$
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 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>( 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*2 )</td>
</tr>
<tr>
<td>( \text{WRITE}(A, t); U_1(A) )</td>
<td>( \text{WRITE}(A,s); )</td>
</tr>
<tr>
<td></td>
<td>( L_2(B); ) ( \text{DENIED\ldots} )</td>
</tr>
<tr>
<td>( \text{READ}(B, t) )</td>
<td>( \text{...GRANTED}; ) ( \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(A); U_2(B); )</td>
</tr>
<tr>
<td></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

\[
\begin{align*}
\text{T1} & \quad \text{T2} \\
L_1(A); \text{READ}(A) & \quad L_2(A); \text{DENIED…} \\
A := A + 100 & \quad \ldots \text{GRANTED}; \text{READ}(A) \\
\text{WRITE}(A); & \quad A := A^*2 \\
\text{READ}(B) & \quad \text{WRITE}(A); \\
B := B + 100 & \quad \text{READ}(B) \\
\text{WRITE}(B); & \quad B := B^*2 \\
U_1(A), U_1(B); \text{Rollback} & \quad \text{WRITE}(B); \\
& \quad U_2(A), U_2(B); \text{Commit}
\end{align*}
\]
Summary of Strict 2PL

• Ensures serializability, recoverability, and avoids cascading aborts

• Issues: implementation, lock modes, granularity, deadlocks, performance
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
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...
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

<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 (‘gizmo’,’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>

Is this schedule serializable?
Phantom Problem

Suppose there are two blue products, $X_1$, $X_2$:

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

**T1**

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

**T2**

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

```
SELECT *  
FROM Product  
WHERE color='blue'
```
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 ('gizmo','blue')</td>
</tr>
<tr>
<td>FROM Product</td>
<td></td>
</tr>
<tr>
<td>WHERE color='blue'</td>
<td></td>
</tr>
<tr>
<td></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>

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

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 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
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