CSE 344

AUGUST 13TH

ISOLATION
ADMINISTRIVIA

• WQ7 due today

• HW8 due Wednesday

• Final on Friday
  • strong focus on 2\textsuperscript{nd} half material
    • but first half still fair game (expect some small Qs)
  • more details on Wednesday
FEEDBACK

• Course evaluations out
  • these help us out a lot
  • (may help your grade if participation is high)

• Feedback on Tech Interview talk
  • https://goo.gl/forms/ZxCg8t0ATJ0VU3n8S2
A schedule is conflict serializable if it can be transformed into a serial schedule by a series of swaps of adjacent non-conflicting actions.

Every conflict-serializable schedule is serializable.

The converse is not true (why?)
LOCKING SCHEDULER

Simple idea:

- Each element has a unique lock
- Each transaction must first acquire the lock before reading/writing that element
- The transaction must eventually release the lock(s)
- Until then, another transaction wanting the lock must wait
  - lock delays the second transaction
  - forces the next operation to come after first txn’s release

By using locks scheduler ensures conflict-serializability
MORE NOTATIONS

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

\[ U_i(A) = \text{transaction } T_i \text{ releases lock for element } A \]
A NON-SERIALIZABLE SCHEDULE

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A)</td>
<td>READ(A)</td>
</tr>
<tr>
<td>A := A + 100</td>
<td>A := A * 2</td>
</tr>
<tr>
<td>WRITE(A)</td>
<td>WRITE(A)</td>
</tr>
<tr>
<td></td>
<td>READ(A)</td>
</tr>
<tr>
<td></td>
<td>A := A * 2</td>
</tr>
<tr>
<td></td>
<td>WRITE(A)</td>
</tr>
<tr>
<td></td>
<td>READ(B)</td>
</tr>
<tr>
<td></td>
<td>B := B * 2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B)</td>
</tr>
<tr>
<td>READ(B)</td>
<td>READ(B)</td>
</tr>
<tr>
<td>B := B + 100</td>
<td>WRITE(B)</td>
</tr>
<tr>
<td>WRITE(B)</td>
<td>WRITE(B)</td>
</tr>
</tbody>
</table>
EXAMPLE

T1

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

T2

L₂(A); READ(A)
A := A * 2
WRITE(A); U₂(A);
L₂(B); BLOCKED…

READ(B)
B := B + 100
WRITE(B); U₁(B);

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

Scheduler has ensured a conflict-serializable schedule
BUT...

T1

L_1(A); READ(A)
A := A+100
WRITE(A); U_1(A);

L_1(B); READ(B)
B := B+100
WRITE(B); U_1(B);

T2

L_2(A); READ(A)
A := A*2
WRITE(A); U_2(A);

L_2(B); READ(B)
B := B*2
WRITE(B); U_2(B);

Locks did not enforce conflict-serializability !!! What’s wrong ?
In every transaction, all lock requests must precede all unlock requests.
EXAMPLE: 2PL

TRANSACTIONS

T1

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

READ(B)
B := B + 100
WRITE(B); U₁(B);

T2

L₂(A); READ(A)
A := A * 2
WRITE(A);
L₂(B); BLOCKED...

...GRANTED; READ(B)
B := B * 2
WRITE(B); U₂(A); U₂(B);

Now it is conflict-serializable
TWO PHASE LOCKING (2PL)

Theorem: 2PL ensures conflict serializability
RECALL...

**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 $T_j$
  - ($T_i$ must come before $T_j$ in any equivalent serial ordering)

The schedule is conflict-serializable iff the precedence graph is acyclic
EXAMPLE 1

\[
\text{This schedule is conflict-serializable}
\]
TWO PHASE LOCKING (2PL)

**Theorem:** 2PL ensures conflict serializability

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

![Diagram](image)
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 \textbf{temporal} cycle in the schedule: \(U_1(A) \rightarrow L_2(A)\) 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:
$U_1(A) \rightarrow L_2(A)$ why?

$U_1(A)$ happened strictly before $L_2(A)$
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:

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

why?

L₂(A) happened strictly before U₂(B)
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:

\[
U_1(A) \rightarrow L_2(A) \\
L_2(A) \rightarrow U_2(B) \\
U_2(B) \rightarrow L_3(B)
\]

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:

- $U_1(A) \rightarrow L_2(A)$
- $L_2(A) \rightarrow U_2(B)$
- $U_2(B) \rightarrow L_3(B)$

......etc.....
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:

- $U_1(A) \rightarrow L_2(A)$
- $L_2(A) \rightarrow U_2(B)$
- $U_2(B) \rightarrow L_3(B)$
- $L_3(B) \rightarrow U_3(C)$
- $U_3(C) \rightarrow L_1(C)$
- $L_1(C) \rightarrow U_1(A)$

Cycle in time: 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)</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);</td>
</tr>
<tr>
<td></td>
<td>$L_2(B)$; BLOCKED…</td>
</tr>
<tr>
<td>READ(B)</td>
<td></td>
</tr>
<tr>
<td>B := B+100</td>
<td></td>
</tr>
<tr>
<td>WRITE(B); $U_1(B)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...GRANTED; READ(B)</td>
</tr>
<tr>
<td></td>
<td>B := B*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B); $U_2(A)$; $U_2(B)$;</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
<tr>
<td>Rollback</td>
<td></td>
</tr>
</tbody>
</table>
A NEW PROBLEM: NON-RECOVERABLE SCHEDULE

T1

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

T2

L₂(A); READ(A)
A := A * 2
WRITE(A);
L₂(B);blocked…

READ(B)
B := B + 100
WRITE(B); U₁(B);

…GRANTED; READ(B)
B := B * 2
WRITE(B); U₂(A); U₂(B);
Commit

Elements A, B written by T1 are restored to their original value.
A NEW PROBLEM: NON-RECOVERABLE SCHEDULE

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1(A); L_1(B); \text{READ}(A)$</td>
<td>$L_2(A); \text{READ}(A)$</td>
</tr>
<tr>
<td>$A := A + 100$</td>
<td>$A := A \times 2$</td>
</tr>
<tr>
<td>WRITE(A); $U_1(A)$</td>
<td>WRITE(A);</td>
</tr>
<tr>
<td>READ(B)</td>
<td>$L_2(B); \text{BLOCKED…}$</td>
</tr>
<tr>
<td>$B := B + 100$</td>
<td>...GRANTED; $\text{READ}(B)$</td>
</tr>
<tr>
<td>WRITE(B); $U_1(B)$</td>
<td>$B := B \times 2$</td>
</tr>
<tr>
<td></td>
<td>WRITE(B); $U_2(A); U_2(B)$</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

Elements A, B written by T1 are restored to their original value.

Dirty reads of A, B lead to incorrect writes.
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)</td>
<td>$L_2(A);$ READ(A)</td>
</tr>
<tr>
<td>$A := A + 100$</td>
<td>$A := A \times 2$</td>
</tr>
<tr>
<td>WRITE(A); $U_1(A)$</td>
<td>WRITE(A);</td>
</tr>
<tr>
<td>READ(B)</td>
<td>$L_2(B);$ BLOCKED…</td>
</tr>
<tr>
<td>$B := B + 100$</td>
<td>...GRANTED; READ(B)</td>
</tr>
<tr>
<td>WRITE(B); $U_1(B)$</td>
<td>$B := B \times 2$</td>
</tr>
<tr>
<td>Commit</td>
<td>WRITE(B); $U_2(A); U_2(B)$; Commit</td>
</tr>
</tbody>
</table>

Rollback
Elements A, B written by T1 are restored to their original value.

Dirty reads of A, B lead to incorrect writes.

Can no longer undo!
**STRICT 2PL**

The Strict 2PL rule:

All locks are held until commit/abort: 
All unlocks are done together with commit/abort.

With strict 2PL, we will get schedules that are both conflict-serializable and recoverable.
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(L_1(A)); READ(A)</td>
<td></td>
</tr>
<tr>
<td>(A := A + 100)</td>
<td></td>
</tr>
<tr>
<td>WRITE(A);</td>
<td></td>
</tr>
<tr>
<td>(L_1(B)); READ(B)</td>
<td></td>
</tr>
<tr>
<td>(B := B + 100)</td>
<td></td>
</tr>
<tr>
<td>WRITE(B);</td>
<td></td>
</tr>
<tr>
<td>Rollback &amp; U_1(A); U_1(B);</td>
<td></td>
</tr>
<tr>
<td>(L_2(A)); BLOCKED…</td>
<td></td>
</tr>
<tr>
<td>(\ldots GRANTED; READ(A))</td>
<td></td>
</tr>
<tr>
<td>(A := A^*2)</td>
<td></td>
</tr>
<tr>
<td>WRITE(A);</td>
<td></td>
</tr>
<tr>
<td>(L_2(B)); READ(B)</td>
<td></td>
</tr>
<tr>
<td>(B := B^*2)</td>
<td></td>
</tr>
<tr>
<td>WRITE(B);</td>
<td></td>
</tr>
<tr>
<td>Commit &amp; U_2(A); U_2(B);</td>
<td></td>
</tr>
</tbody>
</table>
Lock-based systems always use strict 2PL

Easy to implement:

- Before a transaction reads or writes an element A, insert an L(A)
- When the transaction commits/aborts, then release all locks
  - locks accumulate until the end

Ensures both conflict serializability and recoverability
ANOTHER PROBLEM: DEADLOCKS

T_1: R(A), W(B)
T_2: R(B), W(A)

T_1 holds the lock on A, waits for B
T_2 holds the lock on B, waits for A

This is a deadlock!
ANOTHER PROBLEM: DEADLOCKS

To detect a deadlocks, search for a cycle in the waits-for graph:

\( T_1 \) waits for a lock held by \( T_2 \);
\( T_2 \) waits for a lock held by \( T_3 \);
\ldots
\( T_n \) waits for a lock held by \( T_1 \)

Relatively expensive: check periodically
If deadlock is found, then abort one TXN
**LOCK MODES**

\[ S = \text{shared lock (for READ)} \]
\[ X = \text{exclusive lock (for WRITE)} \]

**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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>X</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
Real systems have many types of locks
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., SQL Lite

Solution: lock escalation changes granularity as needed
LOCK PERFORMANCE

Throughput (TPS)

# Active Transactions

Thrashing

TPS = Transactions per second

To avoid, use admission control

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, A1, A2:

<table>
<thead>
<tr>
<th>PHANTOM PROBLEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
</tr>
<tr>
<td>SELECT *</td>
</tr>
<tr>
<td>FROM Product</td>
</tr>
<tr>
<td>WHERE color='blue'</td>
</tr>
<tr>
<td>T2</td>
</tr>
<tr>
<td>INSERT INTO Product(name, color)</td>
</tr>
<tr>
<td>VALUES ('A3', 'blue')</td>
</tr>
<tr>
<td>SELECT *</td>
</tr>
<tr>
<td>FROM Product</td>
</tr>
<tr>
<td>WHERE color='blue'</td>
</tr>
</tbody>
</table>
Suppose there are two blue products, A1, A2:

**PHANTOM PROBLEM**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT *</td>
<td></td>
</tr>
<tr>
<td>FROM Product</td>
<td></td>
</tr>
<tr>
<td>WHERE color='blue'</td>
<td></td>
</tr>
<tr>
<td>INSERT INTO Product(name, color)</td>
<td></td>
</tr>
<tr>
<td>VALUES ('A3','blue')</td>
<td></td>
</tr>
</tbody>
</table>

SELECT *
FROM Product
WHERE color='blue'

R₁(A1);R₁(A2);W₂(A3);R₁(A1);R₁(A2);R₁(A3)
Suppose there are two blue products, A1, A2:

**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 (‘A3’,’blue’) |
| SELECT *  
FROM Product  
WHERE color='blue' |   |

$R_1(A1) ; R_1(A2) ; W_2(A3) ; R_1(A1) ; R_1(A2) ; R_1(A3)$

$W_2(A3) ; R_1(A1) ; R_1(A2) ; R_1(A1) ; R_1(A2) ; R_1(A3)$
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!
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!
SUMMARY OF SERIALIZABILITY

Serializable schedule = equivalent to a serial schedule

(strict) 2PL guarantees conflict serializability

• What is the difference?

Static database:

• Conflict serializability implies serializability

Dynamic database:

• This no longer holds
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
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
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 and different isolation levels can lead to different problems
Bottom line: Read the doc for your DBMS!