Introduction to Data Management

Transactions: Locking

Alyssa Pittman
Based on slides by Jonathan Leang, Dan Suciu, et al
Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

February 3, 2020
Midterm next Monday, 2/10, in class
  • Covers material through today’s lecture
    • e.g. can expect transactions but not isolation levels
    • Notes: 1 sheet, both sides, handwritten

TA-led midterm review
  • Sunday 10 am, room TBA

Engineering Teaching & Learning assessment during next lecture
Recap: Transactions

- Execute all parts of a transaction as a single action
- **Transactions are atomic**

```
BEGIN TRANSACTION
[SQL Statements]
COMMIT -- finalizes execution
```

```
BEGIN TRANSACTION
[SQL Statements]
ROLLBACK -- undo everything
```
### Recap: Serializable Schedule

- **Serializable to T1 then T2**

\[
R_1(A), W_1(A), R_2(A), W_2(A), R_1(B), W_1(B), R_2(B), W_2(B)
\]

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
</tr>
</tbody>
</table>

Looks like T2 finished after T1 for each element.
Recap: Serializable Schedule

- Not serializable to either order

### R₁(A), W₁(A), R₂(A), W₂(A), R₂(B), W₂(B), R₁(B), W₁(B)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
</tbody>
</table>
Recap: Conflict Order Rules

- Observation: Reordering operation of the same element around writes will cause different program behavior

- Inter-transaction conflicts
  - WW conflicts $w_1(x), w_2(x)$
    - Not always the same as $w_2(x), w_1(x)$
  - WR conflicts $w_1(x), r_2(x)$
    - Not always the same as $r_2(x), w_1(x)$
  - RW conflicts $r_1(x), w_2(x)$
    - Not always the same as $w_2(x), r_1(x)$
Recap: Equivalent Behavior Schedules

- A **conflict serializable schedule** is a schedule that can be transformed into a serial schedule by performing a series of swaps of adjacent non-conflicting actions.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
</tbody>
</table>

- A reordered schedule of operations is **guaranteed to be equivalent** when WR, RW, and WW conflicts are preserved.
Recap: Non Conflict Serializable Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
</tbody>
</table>

Conflict rule broken!
Outline

▪ Locks
▪ 2PL and conflict serializability
▪ Deadlocks
▪ Strict 2PL and recoverability
- **Scheduler** a.k.a. **concurrency control manager**
  - Impractical (slow and space inefficient) to issue R, W, ... from a literal schedule
  - Use mechanisms like logs and locks to force ACID properties
Scheduling matters to us because it affects our application behavior and performance!

- Your choice of transaction management should be based on expected workload.
  - **Pessimistic Concurrency Control** (this class) good for **high-contention workloads**
  - **Optimistic Concurrency Control** (CSE 444) good for **low-contention workloads**
Optimistic Scheduler

- Commonly implemented with **Multi Version Concurrency Control**
- “Optimistic” △ Assumes transaction executions will not create conflicts
- Main Idea:
  - Execute first, check later
  - Cheap overhead cost but expensive aborting process
Pessimistic Scheduler

- Commonly implemented with **Locking Scheduler**
- “Pessimistic” Assumes transaction executions will conflict
- Main Idea:
  - Prevent executions that would create conflicts
  - Expensive overhead cost but cheap aborting process
The goal of concurrency control is to ensure isolation (the appearance of serial schedules) and atomicity.

What mechanisms does the DBMS use to make (conflict) serializable schedules?
• Pessimistic CC involves locks

• Binary lock mechanisms:
  • We have locks on objects that specify which transaction can do operations
  • A txn must acquire a lock before reading or writing
    • Notation: txn i acquires lock on element X \( L_i(X) \)
  • A txn must eventually release locks (unlock)
    • Notation: txn i releases lock on element X \( U_i(X) \)
  • If a txn wants an element for which another txn holds the lock, wait for the unlock signal
Element Granularity

- A DBMS (and sometimes user) may specify what granularity of elements are locked
  - Dramatically qualifies expected contention

- SQLite: Database locking only

- MySQL, SQL Server, Oracle, ...: Row locking, table locking

- SQL syntax varies or may not exist explicitly
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**

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

\[ U_i(A) = \text{transaction } T_i \text{ releases lock for element } A \]
A Non-Serializable Schedule

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

T2
READ(A)
A := A*2
WRITE(A)
READ(B)
B := B*2
WRITE(B)

READ(B)
B := B+100
WRITE(B)
Add locking....

Scheduler has ensured a conflict serializable schedule

T1

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

READ(B)
B := B+100
WRITE(B); U₁(B);

T2

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

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

The locks didn’t enforce conflict serializability! What happened?

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L₁(A); 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); U₂(A);</td>
</tr>
<tr>
<td></td>
<td>L₂(B); READ(B)</td>
</tr>
<tr>
<td>B := B + 100</td>
<td>B := B * 2</td>
</tr>
<tr>
<td>WRITE(B); U₁(B);</td>
<td>WRITE(B); U₂(B);</td>
</tr>
</tbody>
</table>
But...

The locks didn’t enforce conflict serializability! What happened?

T1 unlocked A too soon... T2 was able to run in full.
Protocol: In every transaction, all lock requests must precede all unlock requests.
2-Phase Locking (2PL)

Protocol: In every transaction, all lock requests must precede all unlock requests

This will ensure conflict serializability
2PL Example

<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></td>
</tr>
<tr>
<td>[A := A + 100]</td>
<td></td>
</tr>
<tr>
<td>[\text{WRITE}(A); U_1(A)]</td>
<td></td>
</tr>
<tr>
<td>[\text{READ}(B)]</td>
<td></td>
</tr>
<tr>
<td>[B := B + 100]</td>
<td></td>
</tr>
<tr>
<td>[\text{WRITE}(B); U_1(B)];</td>
<td></td>
</tr>
<tr>
<td>[\text{L}_2(\text{A}); \text{READ}(\text{A})]</td>
<td></td>
</tr>
<tr>
<td>[\text{A} := \text{A} \times 2]</td>
<td></td>
</tr>
<tr>
<td>[\text{WRITE}(\text{A});]</td>
<td></td>
</tr>
<tr>
<td>[\text{L}_2(\text{B}); \text{BLOCKED}]</td>
<td></td>
</tr>
<tr>
<td>[\ldots \text{GRANTED}; \text{READ}(\text{B})]</td>
<td></td>
</tr>
<tr>
<td>[\text{B} := \text{B} \times 2]</td>
<td></td>
</tr>
<tr>
<td>[\text{WRITE}(\text{B}); U_2(\text{A}); U_2(\text{B});]</td>
<td></td>
</tr>
</tbody>
</table>

Now it is conflict serializable.
Theorem: **2PL ensures conflict serializability**
Theorem: 2PL ensures conflict serializability

Proof by contradiction:
• Suppose a schedule was executed under 2PL that was not conflict serializable.
Conflict Serializability through 2PL

Theorem: **2PL ensures conflict serializability**

Proof by contradiction:
- Suppose a schedule was executed under 2PL that was not conflict serializable.
- Then that schedule must have a *precedence graph* with a *cycle*.
Conflicting Serializability through 2PL

**Theorem:** **2PL ensures conflict serializability**

Proof by contradiction:
- Suppose a schedule was executed under 2PL that was not conflict serializable.
- Then that schedule must have a **precedence graph** with a **cycle**.
- Name the transactions in the cycle as \( T_1, \ldots, T_n \) where:
Theorem: **2PL ensures conflict serializability**

Proof by contradiction:
- Suppose a schedule was executed under 2PL that was not conflict serializable.
- Then that schedule must have a **precedence graph** with a **cycle**.
- Name the transactions in the cycle as $T_1, \ldots, T_n$ where:
  - An edge exists from $T_i$ to $T_{i+1}$ for $i < n$
  - An edge exists from $T_n$ to $T_1$
Theorem: **2PL ensures conflict serializability**

Proof by contradiction:
- Suppose a schedule was executed under 2PL that was not conflict serializable.
- Then that schedule must have a **precedence graph** with a **cycle**.
- Name the transactions in the cycle as $T_1, \ldots, T_n$ where:
  - An edge exists from $T_i$ to $T_{i+1}$ for $i < n$
  - An edge exists from $T_n$ to $T_1$
  - (An edge means there is a conflict on some element, call it $E_i$)
Theorem: 2PL ensures conflict serializability

Proof by contradiction:

- Suppose a schedule was executed under 2PL that was not conflict serializable.
- Then that schedule must have a precedence graph with a cycle.
- Name the transactions in the cycle as T₁, ..., Tₙ where:
  - An edge exists from Tᵢ to Tᵢ₊₁ for i < n
  - An edge exists from Tₙ to T₁
- (An edge means there is a conflict on some element, call it Eᵢ)
- Under 2PL, we can guarantee the series of locks and unlocks in time:
  - U₁(E₁) then L₂(E₁)

![Diagram of transactions and precedence graph]
Theorem: **2PL ensures conflict serializability**

Proof by contradiction:
- Suppose a schedule was executed under 2PL that was not conflict serializable.
- Then that schedule must have a **precedence graph** with a **cycle**.
- Name the transactions in the cycle as $T_1, ..., T_n$ where:
  - An edge exists from $T_i$ to $T_{i+1}$ for $i < n$
  - An edge exists from $T_n$ to $T_1$
- (An edge means there is a conflict on some element, call it $E_i$)
- Under 2PL, we can guarantee the series of locks and unlocks in time:
  - $U_1(E_1)$ then $L_2(E_1)$
  - $L_2(E_1)$ then $U_2(E_2)$
Theorem: **2PL ensures conflict serializability**

Proof by contradiction:
- Suppose a schedule was executed under 2PL that was not conflict serializable.
- Then that schedule must have a **precedence graph** with a **cycle**.
- Name the transactions in the cycle as $T_1, ..., T_n$ where:
  - An edge exists from $T_i$ to $T_{i+1}$ for $i < n$
  - An edge exists from $T_n$ to $T_1$
  - (An edge means there is a conflict on some element, call it $E_i$)
- Under 2PL, we can guarantee the series of locks and unlocks in time:
  - $U_1(E_1)$ then $L_2(E_1)$
  - $L_2(E_1)$ then $U_2(E_2)$
  - $U_2(E_2)$ then $L_3(E_2)$
  - $L_3(E_2)$ then $U_3(E_3)$
  - $...$

![Diagram showing the precedence graph and the series of locks and unlocks.]
Conflict Serializability through 2PL

Theorem: **2PL ensures conflict serializability**

Proof by contradiction:

- Suppose a schedule was executed under 2PL that was not conflict serializable.
- Then that schedule must have a **precedence graph** with a **cycle**.
- Name the transactions in the cycle as $T_1, \ldots, T_n$ where:
  - An edge exists from $T_i$ to $T_{i+1}$ for $i < n$
  - An edge exists from $T_n$ to $T_1$
- (An edge means there is a conflict on some element, call it $E_i$)
- Under 2PL, we can guarantee the series of locks and unlocks in time:
  - $U_1(E_1)$ then $L_2(E_1)$
  - $L_2(E_1)$ then $U_2(E_2)$
  - $U_2(E_2)$ then $L_3(E_2)$
  - $L_3(E_2)$ then $U_3(E_3)$
  - ... 
  - $U_n(E_n)$ then $L_1(E_n)$

![Diagram](image-url)
Theorem: **2PL ensures conflict serializability**

Proof by contradiction:

- Suppose a schedule was executed under 2PL that was not conflict serializable.
- Then that schedule must have a **precedence graph** with a **cycle**.
- Name the transactions in the cycle as $T_1, \ldots, T_n$ where:
  - An edge exists from $T_i$ to $T_{i+1}$ for $i < n$
  - An edge exists from $T_n$ to $T_1$
- (An edge means there is a conflict on some element, call it $E_i$)
- Under 2PL, we can guarantee the series of locks and unlocks in time:
  - $U_1(E_1)$ then $L_2(E_1)$
  - $L_2(E_1)$ then $U_2(E_2)$
  - $U_2(E_2)$ then $L_3(E_2)$
  - $L_3(E_2)$ then $U_3(E_3)$
  - \[ \ldots \]
  - $U_n(E_n)$ then $L_1(E_n)$
  - $L_1(E_n)$ then $U_1(E_1)$
Conflict Serializability through 2PL

Theorem: **2PL ensures conflict serializability**

Proof by contradiction:
- Suppose a schedule was executed under 2PL that was not conflict serializable.
- Then that schedule must have a **precedence graph** with a **cycle**.
- Name the transactions in the cycle as $T_1, \ldots, T_n$ where:
  - An edge exists from $T_i$ to $T_{i+1}$ for $i < n$
  - An edge exists from $T_n$ to $T_1$
- (An edge means there is a conflict on some element, call it $E_i$)
- Under 2PL, we can guarantee the series of locks and unlocks in time:
  - $U_1(E_1)$ then $L_2(E_1)$
  - $L_2(E_1)$ then $U_2(E_2)$
  - $U_2(E_2)$ then $L_3(E_2)$
  - $L_3(E_2)$ then $U_3(E_3)$
  - ...
  - $U_n(E_n)$ then $L_1(E_n)$
  - $L_1(E_n)$ then $U_1(E_1)$
- There is a **cycle in time** which is a contradiction
Conflict Serializability through 2PL

Theorem: **2PL ensures conflict serializability**

Proof by contradiction:

- Suppose a schedule was executed under 2PL that was not conflict serializable.
- Then that schedule must have a precedence graph with a **cycle**.
- Name the transactions in the cycle as $T_1, \ldots, T_n$ where:
  - An edge exists from $T_i$ to $T_{i+1}$ for $i < n$
  - An edge exists from $T_n$ to $T_1$
- (An edge means there is a conflict on some element, call it $E_i$)
- Under 2PL, we can guarantee the series of locks and unlocks in time:
  - $U_1(E_1)$ then $L_2(E_1)$
  - $L_2(E_1)$ then $U_2(E_2)$
  - $U_2(E_2)$ then $L_3(E_2)$
  - $L_3(E_2)$ then $U_3(E_3)$
  - $\ldots$
  - $U_n(E_n)$ then $L_1(E_n)$
  - $L_1(E_n)$ then $U_1(E_1)$
- There is a **cycle in time** which is a contradiction
### 2PL 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>( \text{WRITE}(A); \ U_1(A) )</td>
<td>( \text{WRITE}(A); )</td>
</tr>
<tr>
<td>( \text{READ}(B) )</td>
<td>( \text{L}_2(B); \text{BLOCKED} \ldots )</td>
</tr>
<tr>
<td>( B := B + 100 )</td>
<td>$\ldots \text{GRANTED}; \text{READ}(B)$</td>
</tr>
<tr>
<td>( \text{WRITE}(B); \ U_1(B) )</td>
<td>( B := B \times 2 )</td>
</tr>
<tr>
<td></td>
<td>( \text{WRITE}(B); \ U_2(A); \ U_2(B) )</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

*Rollback*
2PL Non-Recoverable Schedule

LOCKING

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);

ROLLBACK

ROLLBACK will signal the DBMS to revert to original values
**2PL Non-Recoverable Schedule**

T1

\[ L_1(A); L_1(B); \text{READ}(A) \]
A := A + 100
WRITE(A); \text{U}_1(A)

READ(B)
B := B + 100
WRITE(B); \text{U}_1(B);

T2

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

\[ \ldots \text{GRANTED}; \text{READ}(B) \]
B := B \times 2
WRITE(B); \text{U}_2(A); \text{U}_2(B);
Commit

**Rollback**

ROLLBACK will signal the DBMS to revert to original values

T2 already executed under modified A and B values (dirty read)
2PL 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);

Rollback

T2 already executed under modified A and B values (dirty read)

Ti’s ROLLBACK would break the COMMIT promise that T2’s execution was valid

ROLLBACK will signal the DBMS to revert to original values
Strict 2PL

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

With strict 2PL, we get schedules that are both conflict-serializable and recoverable.
Strict 2PL

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

With strict 2PL, we get schedules that are both conflict-serializable and recoverable.

This is what SQL Server uses!
### 2PL Deadlocks

<table>
<thead>
<tr>
<th>T1 (A, B)</th>
<th>T2 (B, C)</th>
<th>T3 (C, D)</th>
<th>T4 (D, A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(A)</td>
<td>L(B)</td>
<td>L(C)</td>
<td>L(D)</td>
</tr>
<tr>
<td>L(B) blocked…</td>
<td>L(C) blocked…</td>
<td>L(D) blocked…</td>
<td>L(A) blocked…</td>
</tr>
</tbody>
</table>
## 2PL Deadlocks

<table>
<thead>
<tr>
<th>T1 (A, B)</th>
<th>T2 (B, C)</th>
<th>T3 (C, D)</th>
<th>T4 (D, A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(A)</td>
<td>L(B)</td>
<td>L(C)</td>
<td>L(D)</td>
</tr>
<tr>
<td>L(B) blocked…</td>
<td>L(C) blocked…</td>
<td>L(D) blocked…</td>
<td>L(A) blocked…</td>
</tr>
<tr>
<td>R(A) W(A)</td>
<td>R(B) W(B)</td>
<td>R(C) W(C)</td>
<td>R(D) W(D)</td>
</tr>
</tbody>
</table>
## 2PL Deadlocks

<table>
<thead>
<tr>
<th>T1 (A, B)</th>
<th>T2 (B, C)</th>
<th>T3 (C, D)</th>
<th>T4 (D, A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(A) L(B) blocked…</td>
<td>L(B) L(C) blocked…</td>
<td>L(C) L(D) blocked…</td>
<td>L(D) L(A) blocked…</td>
</tr>
<tr>
<td>R(A) W(A)</td>
<td>R(B) W(B)</td>
<td>R(C) W(C)</td>
<td>R(D) W(D)</td>
</tr>
</tbody>
</table>

Can’t make progress since locking phase is not complete for any txn!
### 2PL Deadlocks

<table>
<thead>
<tr>
<th>T1 (A, B)</th>
<th>T2 (B, C)</th>
<th>T3 (C, D)</th>
<th>T4 (D, A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(A)</td>
<td>L(B)</td>
<td>L(C)</td>
<td>L(D)</td>
</tr>
<tr>
<td>L(B) blocked…</td>
<td>L(C) blocked…</td>
<td>L(D) blocked…</td>
<td>L(A) blocked…</td>
</tr>
</tbody>
</table>

- Lock requests create a precedence/waits-for graph where deadlock cycle (2PL is doing its job!).
- Cycle detection is somewhat expensive \(O(V+E)\), so we check the graph only periodically

![Deadlock Graph](attachment:image.png)
## 2PL Deadlocks

<table>
<thead>
<tr>
<th>T1 (A, B)</th>
<th>T2 (B, C)</th>
<th>T3 (C, D)</th>
<th>T4 (D, A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(A)</td>
<td>L(B)</td>
<td>L(C)</td>
<td>L(D)</td>
</tr>
<tr>
<td>L(B) blocked</td>
<td>L(C) blocked</td>
<td>L(D) blocked</td>
<td>L(A) blocked</td>
</tr>
<tr>
<td>R(A) W(A)</td>
<td>R(B) W(B)</td>
<td>R(C) W(C)</td>
<td>R(D) W(D)</td>
</tr>
</tbody>
</table>

If the DBMS finds a cycle:
- It aborts txns (rollback)
- (Hopefully) makes progress
- Eventually retries the rolledback txns
# 2PL Deadlocks

<table>
<thead>
<tr>
<th>T1 (A, B)</th>
<th>T2 (B, C)</th>
<th>T3 (C, D)</th>
<th>T4 (D, A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(A)</td>
<td>L(B)</td>
<td>L(C)</td>
<td>L(D)</td>
</tr>
<tr>
<td>L(B) blocked…</td>
<td>L(C) blocked…</td>
<td>L(D) blocked…</td>
<td>L(A) blocked…</td>
</tr>
<tr>
<td>R(A) W(A)</td>
<td>R(B) W(B)</td>
<td>R(C) W(C)</td>
<td>R(D) W(D)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>…granted L(D)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>abort U(D)</td>
</tr>
</tbody>
</table>

If the DBMS finds a cycle:
- It aborts txns (rollback)
- (Hopefully) makes progress
- Eventually retries the rolledback txns
# 2PL Deadlocks

If the DBMS finds a cycle:
- It aborts txns \((\text{rollback})\)
- \((\text{Hopefully})\) makes progress
- Eventually retries the rolledback txns

<table>
<thead>
<tr>
<th>T1 (A, B)</th>
<th>T2 (B, C)</th>
<th>T3 (C, D)</th>
<th>T4 (D, A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(A)</td>
<td>L(B)</td>
<td>L(C)</td>
<td>L(D)</td>
</tr>
<tr>
<td>L(B) blocked…</td>
<td>L(C) blocked…</td>
<td>L(D) blocked…</td>
<td>L(A) blocked…</td>
</tr>
<tr>
<td>R(A) W(A)</td>
<td>R(B) W(B)</td>
<td>R(C) W(C)</td>
<td>R(D) W(D)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>…granted L(D)</td>
<td>abort U(D)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R(D) W(D)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
### 2PL Deadlocks

<table>
<thead>
<tr>
<th>T1 (A, B)</th>
<th>T2 (B, C)</th>
<th>T3 (C, D)</th>
<th>T4 (D, A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(A)</td>
<td>L(B)</td>
<td>L(C)</td>
<td>L(D)</td>
</tr>
<tr>
<td>L(B)</td>
<td>L(C)</td>
<td>L(D)</td>
<td>L(A)</td>
</tr>
<tr>
<td>blocked…</td>
<td>blocked…</td>
<td>blocked…</td>
<td>blocked…</td>
</tr>
<tr>
<td>R(A) W(A)</td>
<td>R(B) W(B)</td>
<td>R(C) W(C)</td>
<td>R(D) W(D)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>…granted L(D)</td>
<td>abort U(D)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R(D) W(D)</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>…granted L(C)</td>
<td>U(D) U(C)</td>
</tr>
</tbody>
</table>

If the DBMS finds a cycle:
- It aborts txns (rollback)
- (Hopefully) makes progress
- Eventually retries the rolledback txns
# 2PL Deadlocks

<table>
<thead>
<tr>
<th>T1 (A, B)</th>
<th>T2 (B, C)</th>
<th>T3 (C, D)</th>
<th>T4 (D, A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(A)</td>
<td>L(B)</td>
<td>L(C)</td>
<td>L(D)</td>
</tr>
<tr>
<td>L(B) blocked…</td>
<td>L(C) blocked…</td>
<td>L(D) blocked…</td>
<td>L(A) blocked…</td>
</tr>
<tr>
<td>R(A) W(A)</td>
<td>R(B) W(B)</td>
<td>R(C) W(C)</td>
<td>R(D) W(D)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>…granted L(D)</td>
<td>abort U(D)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R(D) W(D)</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>…granted L(C)</td>
<td>U(D) U(C)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>retry</td>
</tr>
</tbody>
</table>

If the DBMS finds a cycle:
- It aborts txns (rollback)
- (Hopefully) makes progress
- Eventually retries

On the application level:
- Can lock resources in a defined order
- Can retry transactions that were aborted due to deadlock
Conservative 2PL

- Protocol: All locks are acquired before the transaction begins
Do I need to implement any of this?

Short Answer: No
“Do I need to implement any of this?”

Long Answer:
These mechanisms are internal to the DBMS. The DBMS manages locks with a locking protocol. The DBMS creates the precedence graph. The DBMS checks for deadlocks.
As an application programmer / database user you only need to (and should only need to) specify transactions and think about application-level consistency.
Next Time

- Phantom reads
- Isolation levels
- Hierarchical locking