Transactions

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

```
BEGIN TRANSACTION [SQL Statements]  BEGIN TRANSACTION [SQL Statements]
COMMIT – finalizes execution         ROLLBACK – undo everything
```

Conflict Order Rules

- Observation: Reordering operation of the same element around writes will cause different program behavior
- Inter-transaction conflicts
  - WW conflicts $\rightarrow$ $W(X), W(X)$
  - Not always the same as $W(X), W(X)$
  - WR conflicts $\rightarrow$ $W(X), R(X)$
  - Not always the same as $R(X), W(X)$
  - RW conflicts $\rightarrow$ $R(X), W(X)$
  - Not always the same as $W(X), R(X)$
  - RR is not a conflict!

Equivalent Behavior Schedules

- A reordered schedule of operations is guaranteed to be equivalent when WR, RW, and WW conflicts are preserved

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

```

Outline

- Testing for conflict serializability
- Locks
- 2PL and conflict serializability
- Deadlocks
- Strict 2PL and recoverability

Testing for Conflict-Serializability

Precedence graph core idea:
Compare each pair of operations to see if they could be reordered at some point

Each potential conflict enforces a temporal order among the transactions
Edge from earlier transaction to later one
Testing for Conflict-Serializability

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 $T_j$
- No edge for actions in the same transaction

Theorem:
The schedule is conflict-serializable iff the precedence graph is acyclic

Example 1

Important:
Always draw the full graph, unless ONLY asked if (yes or no) the schedule is conflict serializable

Initial:
- $r_2(A)$
- $r_1(B)$
- $w_2(A)$
- $r_3(A)$
- $w_1(B)$
- $w_3(A)$
- $r_2(B)$
- $w_2(B)

Sequence:
1. $r_2(A)$
2. $r_1(B)$
3. $w_2(A)$
4. $r_3(A)$
5. $w_1(B)$
6. $w_3(A)$
7. $r_2(B)$
8. $w_2(B)$

No edge because no conflict (A != B)
Example 1

\[
\begin{array}{l}
\text{r}_2(\text{A}) \quad \text{w}_2(\text{A}) \\
\text{r}_3(\text{A}) \quad \text{w}_3(\text{A}) \\
\text{r}_1(\text{B}) \quad \text{w}_1(\text{B}) \\
\text{r}_2(\text{B}) \quad \text{w}_2(\text{B}) \\
\end{array}
\]

No edge because same txn (2)

\[
\begin{array}{l}
\text{r}_2(\text{A}) \quad \text{w}_2(\text{A}) \\
\text{r}_3(\text{A}) \quad \text{w}_3(\text{A}) \\
\text{r}_1(\text{B}) \quad \text{w}_1(\text{B}) \\
\text{r}_2(\text{B}) \quad \text{w}_2(\text{B}) \\
\end{array}
\]

No edge because same txn (2)

\[
\begin{array}{l}
\text{r}_2(\text{A}) \quad \text{w}_2(\text{A}) \\
\text{r}_3(\text{A}) \quad \text{w}_3(\text{A}) \\
\text{r}_1(\text{B}) \quad \text{w}_1(\text{B}) \\
\text{r}_2(\text{B}) \quad \text{w}_2(\text{B}) \\
\end{array}
\]

Edge! Conflict from T2 to T3
Example 1

And so on until compared every pair of actions...

Annotations useful but not necessary.

More edges, but repeats of the same directed edge not necessary.

This schedule is conflict-serializable.
Outline

- Testing for conflict serializability
- Locks
- 2PL and conflict serializability
- Deadlocks
- Strict 2PL and recoverability

Scheduling

- **Scheduler** a.k.a. concurrency control manager
  - Impractical (slow and space inefficient) to issue R, W commands from a literal schedule
  - Use mechanisms like logs and locks to force ACID properties
- Why do we care? Application considerations!
  - **Pessimistic Concurrency Control** (this class) good for high-contention workloads
  - **Optimistic Concurrency Control** (CSE 444) good for low-contention workloads

Optimistic Scheduler

- Commonly interchangeable 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
- (This is what happened in our movie reservation example)

Question for Today

What mechanisms do we use to make (conflict) serializable schedules?

Locks

- **Pessimistic Concurrency Control** involves locks
- **Binary lock** mechanisms:
  - We have locks on objects that specify which transaction can do operations
  - A transaction must acquire a lock before reading or writing
    - Notation: \( \text{txn} i \) acquires lock on element \( X \) \( \xrightarrow{L}(X) \)
    - A transaction must eventually release locks (unlock)
      - Notation: \( \text{txn} i \) releases lock on element \( X \) \( \xrightarrow{U}(X) \)
    - If a transaction finds another transaction holds a lock for the desired element, wait for the unlock signal
      - “blocking”
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

A Non-Serializable Schedule

```
T1                     T2
READ(A, t)             READ(A, s)
t := t+100              s := s\times2
WRITE(A, t)            WRITE(A, s)
READ(A, s)             READ(B, s)
s := s\times2           WRITE(B, s)
WRITE(B, s)            READ(B, t)
t := t+100              WRITE(B, t)
```

A Non-Serializable Schedule

```
T1                     T2
L_1(A); READ(A, t)    L_1(A); READ(A, s)
t := t+100              s := s\times2
WRITE(A, t); U_1(A)   WRITE(A, s); U_1(A);
L_2(A); READ(A, s)    L_2(A); READ(B, s)
s := s\times2           s := s\times2
WRITE(A, s)            WRITE(B, s); U_1(B);
L_2(B); READ(B, t)    L_2(B); READ(B, t)
t := t+100              t := t+100
WRITE(B, t)            WRITE(B, t); U_1(B);
```

2-Phase Locking (2PL)

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

Example: 2PL transactions

```
T1                     T2
L_1(A); L_1(B); READ(A, t) L_1(A); READ(A, s)
t := t+100              s := s\times2
WRITE(A, t); U_1(A)    WRITE(A, s);
...GRANTED: READ(B, t) L_2(B); DENIED...
t := t+100              WRITE(B, s);
WRITE(B, t); U_1(B);
```

Example with Multiple Transactions

```
T1              T2              T3              T4
Growing phase  Unlocks second so Unlock first  Unlocks second Unlocks third
Shrinking phase perhaps we waiting for T3  Does not waiting for anyone
```

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

• $U(E_1)$ then $L(E_2)$

• $U(E_2)$ then $L(E_3)$

• $U(E_3)$ then $L(E_4)$

• $\ldots$

• $U(E_n)$ then $L(E_1)$

• $U(E_1)$ then $L(E_n)$

There is a cycle in time which is a contradiction.

Deadlocks!

Byproduct of dealing with groups of locks
2PL Deadlocks

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

If the DBMS finds a cycle:
- We rollback the transaction(s) involved in the cycle.
- (Hopefully) make progress.
- Eventually retry the rolled back transaction(s).

Can’t make progress since locking phase is not complete for any transaction!
## 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>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>...</td>
<td>...</td>
</tr>
</tbody>
</table>

If the DBMS finds a cycle:
- We rollback txns
- (Hopefully) make progress
- Eventually retry the rollback txns

## 2PL Problems

- **Deadlocks!**
  - Byproduct of dealing with groups of locks
- **Recoverability**
  - Transactions might want an abort feature if not all consistency constraints are enforced by the DB
  - Can’t always abort txns under vanilla 2PL

## Non-Recoverable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(A)</td>
<td>L(A) blocked...</td>
</tr>
<tr>
<td>L(B)</td>
<td>L(B) blocked...</td>
</tr>
<tr>
<td>R(A) W(A)</td>
<td>... granted L(A)</td>
</tr>
<tr>
<td>R(B) W(B)</td>
<td>... granted L(B)</td>
</tr>
<tr>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

ROLLBACK will try to signal the DBMS to revert to original values
Non-Recoverable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(A)</td>
<td>L(A) blocked...</td>
</tr>
<tr>
<td>L(B)</td>
<td>L(B) blocked...</td>
</tr>
<tr>
<td>R(A) W(A)</td>
<td>R(A) W(A)</td>
</tr>
<tr>
<td>U(A)</td>
<td>granted L(A)</td>
</tr>
<tr>
<td>U(B)</td>
<td>granted L(B)</td>
</tr>
<tr>
<td>R(B) W(B)</td>
<td>R(B) W(B)</td>
</tr>
<tr>
<td>U(A)</td>
<td>granted L(A)</td>
</tr>
<tr>
<td>U(B)</td>
<td>granted L(B)</td>
</tr>
</tbody>
</table>

ROLLBACK will try to signal the DBMS to revert to original values.

ROLLBACK will try to signal the DBMS to revert to original values.

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

Strict 2PL

Protocol: 2PL + All unlocks are done together with the COMMIT or ROLLBACK

Strict 2PL guarantees schedule conflict serializability and recoverability.

Recoverable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(A)</td>
<td>L(A) blocked...</td>
</tr>
<tr>
<td>L(B)</td>
<td>L(B) blocked...</td>
</tr>
<tr>
<td>R(A) W(A)</td>
<td>R(A) W(A)</td>
</tr>
<tr>
<td>R(B) W(B)</td>
<td>R(B) W(B)</td>
</tr>
<tr>
<td>ROLLBACK</td>
<td>ROLLBACK</td>
</tr>
<tr>
<td>U(A)</td>
<td>U(A)</td>
</tr>
<tr>
<td>U(B)</td>
<td>U(B)</td>
</tr>
<tr>
<td>R(A) W(A)</td>
<td>R(A) W(A)</td>
</tr>
<tr>
<td>R(B) W(B)</td>
<td>R(B) W(B)</td>
</tr>
<tr>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
<tr>
<td>U(A)</td>
<td>U(B)</td>
</tr>
</tbody>
</table>

“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 strict 2PL. The DBMS creates the precedence graph. The DBMS does the deadlock retry.
As an application programmer / database user you only need to (and should only need to) specify transactions and think about application-level consistency.