Lecture #17
Two-Phase Locking
LAST CLASS

Conflict Serializable
→ Verify using either the “swapping” method or dependency graphs.
→ Any DBMS that says that they support “serializable” isolation does this.

View Serializable
→ No efficient way to verify.
→ No DBMS that supports this.
OBSERVATION

We need a way to guarantee that all execution schedules are correct (i.e., serializable) without knowing the entire schedule ahead of time.

Solution: Use **locks** to protect database objects.
EXECUTING WITH LOCKS

Schedule

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td>LOCK(A)</td>
<td>LOCK(A)</td>
</tr>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td>UNLOCK(A)</td>
</tr>
<tr>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Lock Manager
EXECUTING WITH LOCKS

Schedule

\begin{tabular}{|c|c|}
\hline
T_1 & T_2 \\
\hline
\text{BEGIN} & \text{BEGIN} \\
\text{LOCK}(A) & \text{LOCK}(A) \\
\text{R(A)} & \text{R(A)} \\
\text{W(A)} & \text{W(A)} \\
\text{R(A)} & \text{W(A)} \\
\text{UNLOCK}(A) & \text{UNLOCK}(A) \\
\text{COMMIT} & \text{COMMIT} \\
\hline
\end{tabular}

Lock Manager

Granted (T_1 \rightarrow A)
EXECUTING WITH LOCKS

Schedule

\[
\begin{array}{c|c}
T_1 & T_2 \\
\hline
\text{BEGIN} & \text{BEGIN} \\
\text{LOCK}(A) & \text{LOCK}(A) \\
\text{R}(A) & \text{R}(A) \\
\text{W}(A) & \text{W}(A) \\
\text{R}(A) & \text{W}(A) \\
\text{UNLOCK}(A) & \text{UNLOCK}(A) \\
\text{COMMIT} & \text{COMMIT} \\
\end{array}
\]

Lock Manager

- Granted \((T_1 \rightarrow A)\)
- Denied!
EXECUTING WITH LOCKS

Schedule

\[\begin{array}{c|c}
\text{T}_1 & \text{T}_2 \\
\hline
\text{BEGIN} & \text{BEGIN} \\
\text{LOCK}(A) & \text{LOCK}(A) \\
\text{R(A)} & \text{UNLOCK}(A) \\
\text{W(A)} & \text{R(A)} \\
\text{R(A)} & \text{W(A)} \\
\text{UNLOCK}(A) & \text{UNLOCK}(A) \\
\text{COMMIT} & \text{COMMIT} \\
\end{array}\]

Lock Manager

- Granted (T\(_1\)→A)
- Denied!
EXECUTING WITH LOCKS

Schedule

<table>
<thead>
<tr>
<th>BEGIN LOCK(A)</th>
<th>T1</th>
<th>BEGIN LOCK(A)</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Lock Manager

- Granted (T1→A)
- Denied!
- Released (T1→A)
EXECUTING WITH LOCKS

Schedule

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td>LOCK(A)</td>
<td>LOCK(A)</td>
</tr>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td>UNLOCK(A)</td>
</tr>
<tr>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Lock Manager

- Granted (T₁→A)
- Denied!
- Released (T₁→A)
- Granted (T₂→A)
- Released (T₂→A)
TODAY'S AGENDA

Lock Types
Two-Phase Locking
Deadlock Detection + Prevention
Hierarchical Locking
# Locks vs. Latches

<table>
<thead>
<tr>
<th>Locks</th>
<th>Latches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separate...</td>
<td>User transactions</td>
</tr>
<tr>
<td>Protect...</td>
<td>Database Contents</td>
</tr>
<tr>
<td>During...</td>
<td>Entire Transactions</td>
</tr>
<tr>
<td>Modes...</td>
<td>Shared, Exclusive, Update, Intention</td>
</tr>
<tr>
<td>Deadlock</td>
<td>Detection &amp; Resolution</td>
</tr>
<tr>
<td>...by...</td>
<td>Waits-for, Timeout, Aborts</td>
</tr>
<tr>
<td>Kept in...</td>
<td>Lock Manager</td>
</tr>
</tbody>
</table>

Source: Goetz Graefe
**BASIC LOCK TYPES**

**S-LOCK**: Shared locks for reads.

**X-LOCK**: Exclusive locks for writes.

### Compatibility Matrix

<table>
<thead>
<tr>
<th></th>
<th>Shared</th>
<th>Exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared</td>
<td>✔</td>
<td>X</td>
</tr>
<tr>
<td>Exclusive</td>
<td>X</td>
<td>✔</td>
</tr>
</tbody>
</table>
### Table 1. Compatibility matrix of page lock and row lock modes

<table>
<thead>
<tr>
<th>Lock mode</th>
<th>Share (S-lock)</th>
<th>Update (U-lock)</th>
<th>Exclusive (X-lock)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Share</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Update</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

### Table 2. Compatibility of table lock modes

<table>
<thead>
<tr>
<th>Lock Mode</th>
<th>IS</th>
<th>S</th>
<th>IX</th>
<th>IX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>S</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>IX</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>IX</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Table 3. Compatibility of lock modes

- **Existing granted mode**
- **Requested mode**
  - Shared (S): Yes
  - Update (U): Yes
- **Intent shared (IS)**
  - Shared (S): Yes
  - Update (U): Yes
- **Intent exclusive (IX)**
  - Shared with intent exclusive (SIX)

### Table 13.2. Conflicting Lock Modes

<table>
<thead>
<tr>
<th>Requested Lock Mode</th>
<th>ACCESS SHARE ROW SHARE ROW EXCL. SHARE UPDATE EXCL. SHARE SHARE ROW EXCL. EXCL. ACCESS EXCL.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS SHARE</td>
<td>X, X, X</td>
</tr>
<tr>
<td>SHARE SHARE</td>
<td>X, X, X</td>
</tr>
<tr>
<td>ROW EXCL.</td>
<td>X, X, X</td>
</tr>
<tr>
<td>SHARE UPDATE EXCL.</td>
<td>X, X, X</td>
</tr>
<tr>
<td>SHARE SHARE ROW EXCL.</td>
<td>X, X, X</td>
</tr>
<tr>
<td>EXCL.</td>
<td>X, X, X</td>
</tr>
<tr>
<td>ACCESS EXCL.</td>
<td>X, X, X</td>
</tr>
</tbody>
</table>

### Table-level lock type compatibility

- **PostgreSQL**: Conflict
- **MySQL**: Conflict
- **SQL Server**: Conflict
- **Oracle**: Conflict

Table-level lock type compatibility is summarized in the following matrix.
**EXECUTING WITH LOCKS**

Transactions request locks (or upgrades).

Lock manager grants or blocks requests.

Transactions release locks.

Lock manager updates its internal lock-table.

→ It keeps track of what transactions hold what locks and what transactions are waiting to acquire any locks.
EXECUTING WITH LOCKS

Schedule

\[ T_1 \]
BEGIN
\begin{align*}
X\text{-}LOCK & (A) \\
R & (A) \\
W & (A) \\
S\text{-}LOCK & (A) \\
R & (A) \\
UNLOCK & (A) \\
COMMIT & \\
\end{align*}

\[ T_2 \]
BEGIN
\begin{align*}
X\text{-}LOCK & (A) \\
W & (A) \\
UNLOCK & (A) \\
\end{align*}

Commit

Lock Manager
EXECUTING WITH LOCKS

Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td></td>
<td>X-LOCK(A)</td>
<td>X-LOCK(A)</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UNLOCK(A)</td>
<td>UNLOCK(A)</td>
</tr>
<tr>
<td></td>
<td>S-LOCK(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UNLOCK(A)</td>
<td>COMMIT</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Lock Manager

Granted (T₁ → A)
EXECUTING WITH LOCKS

Schedule

T₁  T₂

BEGIN  
X-LOCK(A)  
R(A)  
W(A)  
UNLOCK(A)  
S-LOCK(A)  
R(A)  
UNLOCK(A)  
COMMIT

BEGIN  
X-LOCK(A)  
W(A)  
UNLOCK(A)  
COMMIT

Lock Manager

Granted (T₁→A)  
Released (T₁→A)
EXECUTING WITH LOCKS

Schedule

T<sub>1</sub>
BEGIN
X-LOCK(A)
R(A)
W(A)
S-LOCK(A)
R(A)
UNLOCK(A)
COMMIT

T<sub>2</sub>
BEGIN
X-LOCK(A)
W(A)
UNLOCK(A)
COMMIT

Lock Manager

Granted (T<sub>1</sub>→A)
Released (T<sub>1</sub>→A)
Granted (T<sub>2</sub>→A)
Released (T<sub>2</sub>→A)
EXECUTING WITH LOCKS

Schedule

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td>X-LOCK(A)</td>
<td>X-LOCK(A)</td>
</tr>
<tr>
<td>R(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td>UNLOCK(A)</td>
</tr>
<tr>
<td>S-LOCK(A)</td>
<td>COMMIT</td>
</tr>
<tr>
<td>R(A)</td>
<td>COMMIT</td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td></td>
</tr>
</tbody>
</table>

Lock Manager

- Granted (T₁→A)
- Released (T₁→A)
- Granted (T₂→A)
- Released (T₂→A)
- Granted (T₁→A)
- Released (T₁→A)
EXECUTING WITH LOCKS

Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td>X-LOCK (A)</td>
<td>X-LOCK (A)</td>
</tr>
<tr>
<td>R(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>COMMIT</td>
</tr>
<tr>
<td>UNLOCK (A)</td>
<td>UNLOCK (A)</td>
</tr>
<tr>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Lock Manager

- Granted (T1→A)
- Released (T1→A)
- Granted (T2→A)
- Released (T2→A)
- Granted (T1→A)
- Released (T1→A)
CONCURRENCY CONTROL PROTOCOL

Two-phase locking (2PL) is a concurrency control protocol that determines whether a txn can access an object in the database at runtime.

The protocol does not need to know all the queries that a txn will execute ahead of time.
TWO-PHASE LOCKING

Phase #1: Growing
→ Each txn requests the locks that it needs from the DBMS’s lock manager.
→ The lock manager grants/denies lock requests.

Phase #2: Shrinking
→ The txn is allowed to only release/downgrade locks that it previously acquired. It cannot acquire new locks.
TWO-PHASE LOCKING

The txn is not allowed to acquire/upgrade locks after the growing phase finishes.

Transaction Lifetime

# of Locks

Growing Phase  Shrinking Phase

TIME
TWO-PHASE LOCKING

The txn is not allowed to acquire/upgrade locks after the growing phase finishes.

Transaction Lifetime

TIME

Growing Phase

Shrinking Phase

# of Locks

2PL Violation!
EXECUTING WITH 2PL

Schedule

T₁
BEGIN
X-LOCK(A)
R(A)
W(A)
R(A)
UNLOCK(A)
COMMIT

T₂
BEGIN
W(A)
UNLOCK(A)
COMMIT

Lock Manager
EXECUTING WITH 2PL

Schedule

\[ T_1 \]
BEGIN
X-LOCK(A)
R(A)
W(A)
R(A)
UNLOCK(A)
COMMIT

\[ T_2 \]
BEGIN
X-LOCK(A)
R(A)
W(A)
UNLOCK(A)
COMMIT

Lock Manager

Granted \((T_1 \rightarrow A)\)
EXECUTING WITH 2PL

Schedule

T₁
BEGIN
X-LOCK(A)
R(A)
W(A)
R(A)
UNLOCK(A)
COMMIT

T₂
BEGIN
X-LOCK(A)

TIME

Lock Manager

Granted (T₁→A)

Denied!
EXECUTING WITH 2PL

Schedule

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td></td>
</tr>
<tr>
<td>X-LOCK(A)</td>
<td></td>
</tr>
<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>UNLOCK(A)</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
</tr>
<tr>
<td>BEGIN</td>
<td></td>
</tr>
<tr>
<td>X-LOCK(A)</td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
</tr>
</tbody>
</table>

Time

LOCK MANAGER

Granted (T₁→A)

Denied!

Released (T₁→A)
EXECUTING WITH 2PL

Schedule

T₁
BEGIN
X-LOCK(A)
R(A)
W(A)
UNLOCK(A)
COMMIT

T₂
BEGIN
X-LOCK(A)
R(A)

Lock Manager

Granted (T₁→A)
Denied!
Released (T₁→A)
Granted (T₂→A)
Released (T₂→A)
TWO-PHASE LOCKING

2PL on its own is sufficient to guarantee conflict serializability because it generates schedules whose precedence graph is acyclic.

But it is subject to cascading aborts.
2PL – CASCADING ABORTS

Schedule

T₁

BEGIN
X-LOCK(A)
X-LOCK(B)
R(A)
W(A)
UNLOCK(A)
R(B)
W(B)
⋮

T₂

BEGIN
X-LOCK(A)
R(A)
W(A)
⋮
2PL – CASCADING ABORTS

Schedule

\[
\text{Begin} \quad \text{X-Lock}(A) \quad \text{X-Lock}(B) \quad \text{R}(A) \quad \text{W}(A) \quad \text{Unlock}(A) \quad \text{R}(B) \quad \text{W}(B) \quad \vdots \quad \text{Abort}
\]

\[
\text{Begin} \quad \text{X-Lock}(A) \quad \text{R}(A) \quad \text{W}(A) \quad \vdots
\]

This is a permissible schedule in 2PL, but the DBMS has to also abort \( T_2 \) when \( T_1 \) aborts. Any information about \( T_1 \) cannot be "leaked" to the outside world.
This is a permissible schedule in 2PL, but the DBMS has to also abort T2 when T1 aborts. Any information about T1 cannot be "leaked" to the outside world.

BEGIN
X-LOCK(A)
X-LOCK(B)
R(A)
W(A)
UNLOCK(A)
R(B)
W(B)
⋮

BEGIN
X-LOCK(A)
R(A)
W(A)
⋮
ABORT
This is a permissible schedule in 2PL, but the DBMS has to also abort $T_2$ when $T_1$ aborts.

Any information about $T_1$ cannot be “leaked” to the outside world.
This is a permissible schedule in 2PL, but the DBMS has to also abort $T_2$ when $T_1$ aborts.

Any information about $T_1$ cannot be “leaked” to the outside world.

This is all wasted work!
2PL OBSERVATIONS

There are potential schedules that are serializable but would not be allowed by 2PL because locking limits concurrency.

→ Most DBMSs prefer correctness before performance.

May still have “dirty reads”.

→ Solution: **Strong Strict 2PL (aka Rigorous 2PL)**

May lead to deadlocks.

→ Solution: **Detection** or **Prevention**
The txn is only allowed to release locks after it has ended (i.e., committed or aborted).

Allows only conflict serializable schedules, but it is often stronger than needed for some apps.
STRONG STRICT TWO-PHASE LOCKING

A schedule is **strict** if a value written by a txn is not read or overwritten by other txns until thattxn finishes.

Advantages:

→ Does not incur cascading aborts.

→ Aborted txns can be undone by just restoring original values of modified tuples.
EXAMPLES

$T_1$ – Move $100 from Andy’s account ($A$) to his bookie’s account ($B$).

$T_2$ – Compute the total amount in all accounts and return it to the application.

BEGIN
A=A-100
B=B+100
COMMIT

BEGIN
ECHO A+B
COMMIT
NON-2PL EXAMPLE

Schedule

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td>X-LOCK(A)</td>
<td>S-LOCK(A)</td>
</tr>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>A=A-100</td>
<td>UNLOCK(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>S-LOCK(B)</td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td>R(B)</td>
</tr>
<tr>
<td>X-LOCK(B)</td>
<td>UNLOCK(B)</td>
</tr>
<tr>
<td>R(B)</td>
<td>R(B)</td>
</tr>
<tr>
<td>B=B+100</td>
<td>UNLOCK(B)</td>
</tr>
<tr>
<td>UNLOCK(B)</td>
<td>ECHO A+B</td>
</tr>
<tr>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Initial Database State

A=1000, B=1000

Time: 22
NON-2PL EXAMPLE

Initial Database State

\[ A = 1000, \quad B = 1000 \]

Schedule

\[ T_1 \quad T_2 \]

**BEGIN**

- \( X\text{-LOCK}(A) \)
- \( R(A) \)
- \( A = A - 100 \)
- \( W(A) \)
- \( UNLOCK(A) \)
- \( X\text{-LOCK}(B) \)
- \( R(B) \)
- \( B = B + 100 \)
- \( W(B) \)
- \( UNLOCK(B) \)
- **COMMIT**

**BEGIN**

- \( S\text{-LOCK}(A) \)
- **SLEEP**
- \( R(A) \)
- \( UNLOCK(A) \)
- \( S\text{-LOCK}(B) \)
- \( R(B) \)
- \( UNLOCK(B) \)
- **ECHO** \( A + B \)
- **COMMIT**

TIME
NON-2PL EXAMPLE

Initial Database State

A = 1000, B = 1000

Schedule

T1
BEGIN
X-LOCK(A)
R(A)
A = A - 100
W(A)
UNLOCK(A)
X-LOCK(B)
R(B)
B = B + 100
W(B)
UNLOCK(B)
COMMIT

T2
BEGIN
S-LOCK(A)
R(A)
UNLOCK(A)
S-LOCK(B)
R(B)
UNLOCK(B)
ECHO A + B
COMMIT

TIME
NON-2PL EXAMPLE

Initial Database State

\[ A = 1000, \quad B = 1000 \]

T₂ Output

\[ A + B = 1900 \]

Schedule

**T₁**

- **BEGIN**
- **X-LOCK(A)**
- **R(A)**
- \[ A = A - 100 \]
- **W(A)**
- **UNLOCK(A)**
- **X-LOCK(B)**
- **R(A)**
- **W(B)**
- **UNLOCK(B)**
- **COMMIT**

**T₂**

- **BEGIN**
- **S-LOCK(A)**
- **S-LOCK(B)**
- **R(A)**
- **UNLOCK(A)**
- **R(B)**
- **UNLOCK(B)**
- **ECHO A + B**
- **COMMIT**
**2PL EXAMPLE**

**Schedule**

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td>X-LOCK(A)</td>
<td>S-LOCK(A)</td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>A=A-100</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>X-LOCK(B)</td>
<td>R(A)</td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td>S-LOCK(B)</td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>B=B+100</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td>UNLOCK(B)</td>
<td>R(B)</td>
</tr>
<tr>
<td>COMMIT</td>
<td>UNLOCK(A)</td>
</tr>
<tr>
<td></td>
<td>UNLOCK(B)</td>
</tr>
<tr>
<td></td>
<td>ECHO A+B</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

**Initial Database State**

A=1000, B=1000

**T₂ Output**

A+B=2000
STRONG STRICT 2PL EXAMPLE

Schedule

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td>X-LOCK(A)</td>
<td>S-LOCK(A)</td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>A=A-100</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>X-LOCK(B)</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>B=B+100</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>UNLOCK(B)</td>
<td>S-LOCK(B)</td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
</tr>
</tbody>
</table>

Initial Database State

\[ A=1000, \quad B=1000 \]
STRONG STRICT 2PL EXAMPLE

**Schedule**

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td>X-LOCK(A)</td>
<td>S-LOCK(A)</td>
</tr>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>A=A-100</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td>S-LOCK(B)</td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
</tr>
<tr>
<td></td>
<td>B=B+100</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td>ECHO A+B</td>
</tr>
<tr>
<td>UNLOCK(B)</td>
<td>UNLOCK(A)</td>
</tr>
<tr>
<td>COMMIT</td>
<td>UNLOCK(B)</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

**Initial Database State**

A=1000, B=1000

**Time**

T₁ T₂

**Initial Database State**

A=1000, B=1000
STRONG STRICT 2PL EXAMPLE

Schedule

T₁
BEGIN
X-LOCK(A)
R(A)
A=A-100
W(A)
X-LOCK(B)
R(B)
B=B+100
W(B)
UNLOCK(A)
UNLOCK(B)
COMMIT

T₂
BEGIN
S-LOCK(A)
R(A)
S-LOCK(B)
R(B)
ECHO A+B
UNLOCK(A)
UNLOCK(B)
COMMIT

Initial Database State

A=1000, B=1000

T₂ Output

A+B=2000

Schedule

TIME

T₁
T₂

Initial Database State

A=1000, B=1000

T₂ Output

A+B=2000
UNIVERSE OF SCHEDULES

- All Schedules
- View Serializable
- Conflict Serializable
- No Cascading Aborts
- Strong Strict 2PL
- Serial
2PL OBSERVATIONS

There are potential schedules that are serializable but would not be allowed by 2PL because locking limits concurrency.

→ Most DBMSs prefer correctness before performance.

May still have “dirty reads”.

→ Solution: Strong Strict 2PL (aka Rigorous 2PL)

May lead to deadlocks.

→ Solution: Detection or Prevention
IT JUST GOT REAL

Schedule

T₁

BEGIN
X-LOCK(A)

R(A)
X-LOCK(B)

T₂

BEGIN
S-LOCK(B)

R(B)
S-LOCK(A)

Lock Manager
IT JUST GOT REAL

Schedule

T1
BEGIN
X-LOCK(A)
R(A)
X-LOCK(B)

T2
BEGIN
S-LOCK(B)
R(B)
S-LOCK(A)

Lock Manager

Granted (T1\rightarrow A)
IT JUST GOT REAL

Schedule

T₁

BEGIN
X-LOCK(A)
R(A)
X-LOCK(B)

T₂

BEGIN
S-LOCK(B)
R(B)
S-LOCK(A)

Lock Manager

Granted (T₁→A)
Granted (T₂→B)
IT JUST GOT REAL

Schedule

T₁

BEGIN
X-LOCK(A)
R(A)
X-LOCK(B)

T₂

BEGIN
S-LOCK(B)
R(B)
S-LOCK(A)

Lock Manager

Granted (T₁→A)

Granted (T₂→B)

Denied!
**IT JUST GOT REAL**

**Schedule**

- **T₁**
  - BEGIN
  - X-LOCK(A)
  - R(A)
  - X-LOCK(B)

- **T₂**
  - BEGIN
  - S-LOCK(B)
  - R(B)
  - S-LOCK(A)

**Lock Manager**

- Granted (T₁→A)
- Granted (T₂→B)
- Denied!
- Denied!

- **TIME**
IT JUST GOT REAL

BEGIN X-LOCK(A)
R(A)
X-LOCK(B)

BEGIN S-LOCK(B)
R(B)
S-LOCK(A)

IT JUST GOT REAL

Granted (T_1→A)
Denied!

Granted (T_2→B)
Denied!
2PL DEADLOCKS

A **deadlock** is a cycle of transactions waiting for locks to be released by each other.

Two ways of dealing with deadlocks:
→ Approach #1: Deadlock Detection
→ Approach #2: Deadlock Prevention
DEADLOCK DETECTION

The DBMS creates a **waits-for** graph to keep track of what locks eachtxn is waiting to acquire:

→ Nodes are transactions

→ Edge from $T_i$ to $T_j$ if $T_i$ is waiting for $T_j$ to release a lock.

The system periodically checks for cycles in **waits-for** graph and then decides how to break it.
DEADLOCK DETECTION

Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
</table>
| BEGIN
S-LOCK(A) | BEGIN
X-LOCK(B) | BEGIN
S-LOCK(C) |
| S-LOCK(B) | X-LOCK(C) | X-LOCK(A) |

Waits-For Graph

- T1
- T2
- T3
DEADLOCK DETECTION

Schedule

T₁
BEGIN
S-LOCK(A)

S-LOCK(B)

T₂
BEGIN

X-LOCK(B)

T₃
BEGIN

S-LOCK(C)

X-LOCK(C)

X-LOCK(A)

Waits-For Graph

T₁

T₂

T₃

Graph 29
DEADLOCK DETECTION

Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td>S-LOCK(A)</td>
<td>S-LOCK(B)</td>
<td>S-LOCK(B)</td>
</tr>
<tr>
<td></td>
<td>X-LOCK(B)</td>
<td>X-LOCK(C)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X-LOCK(A)</td>
</tr>
</tbody>
</table>

Waits-For Graph

- T1
- T2
- T3

Graph shows the dependencies:
- T1 waits for T2
- T2 waits for T3
- T3 waits for T1
DEADLOCK DETECTION

Schedule

T_1
BEGIN
S-LOCK(A)
S-LOCK(B)

T_2
BEGIN
X-LOCK(B)

T_3
BEGIN
S-LOCK(C)
X-LOCK(C)
X-LOCK(A)

Waits-For Graph

T_1  T_2  T_3

29
DEADLOCK HANDLING

When the DBMS detects a deadlock, it will select a “victim” txn to rollback to break the cycle.

The victim txn will either restart or abort (more common) depending on how it was invoked.

There is a trade-off between the frequency of checking for deadlocks and how long txns wait before deadlocks are broken.
DEADLOCK HANDLING: VICTIM SELECTION

Selecting the proper victim depends on a lot of different variables....

- By age (lowest timestamp)
- By progress (least/most queries executed)
- By the # of items already locked
- By the # of txns that we have to rollback with it

We also should consider the # of times a txn has been restarted in the past to prevent starvation.
DEADLOCK HANDLING: ROLLBACK LENGTH

After selecting a victim txn to abort, the DBMS can also decide on how far to rollback the txn's changes.

**Approach #1: Completely**

→ Rollback entire txn and tell the application it was aborted.

**Approach #2: Partial (Savepoints)**

→ DBMS rolls back a portion of a txn (to break deadlock) and then attempts to re-execute the undone queries.
DEADLOCK PREVENTION

When a txn tries to acquire a lock that is held by another txn, the DBMS kills one of them to prevent a deadlock.

This approach does not require a waits-for graph or detection algorithm.
DEADLOCK PREVENTION

Assign priorities based on timestamps:
→ Older Timestamp = Higher Priority (e.g., \( T_1 > T_2 \))

**Wait-Die (“Old Waits for Young”)**
→ If requesting txn has higher priority than holding txn, then requesting txn waits for holding txn.
→ Otherwise requesting txn aborts.

**Wound-Wait (“Young Waits for Old”)**
→ If requesting txn has higher priority than holding txn, then holding txn aborts and releases lock.
→ Otherwise requesting txn waits.
**DEADLOCK PREVENTION**

**Wait-Die**
- $T_1$ waits

**Wound-Wait**
- $T_2$ aborts
DEADLOCK PREVENTION

T₁
BEGIN
X-LOCK(A)
⋮
T₂
BEGIN
X-LOCK(A)
⋮

Wait-Die
T₁ waits

Wound-Wait
T₂ aborts

T₁
BEGIN
X-LOCK(A)
⋮
T₂
BEGIN
X-LOCK(A)
⋮

Wait-Die
T₁ waits

Wound-Wait
T₂ aborts

T₁
BEGIN
X-LOCK(A)
⋮
T₂
BEGIN
X-LOCK(A)
⋮

Wait-Die
T₂ aborts

Wound-Wait
T₂ waits
DEADLOCK PREVENTION

Why do these schemes guarantee no deadlocks?

Only one “type” of direction allowed when waiting for a lock.

When a txn restarts, what is its (new) priority?

Its original timestamp to prevent it from getting starved for resources like an old man at a corrupt senior center.
OBSERVATION

All these examples have a one-to-one mapping from database objects to locks.

If a txn wants to update one billion tuples, then it must acquire one billion locks.

Acquiring locks is a more expensive operation than acquiring a latch even if that lock is available.
LOCK GRANULARITIES

When a txn wants to acquire a “lock”, the DBMS can decide the granularity (i.e., scope) of that lock.
→ Attribute? Tuple? Page? Table?

The DBMS should ideally obtain fewest number of locks that a txn needs.

Trade-off between parallelism versus overhead.
→ Fewer Locks, Larger Granularity vs. More Locks, Smaller Granularity.
DATABASE LOCK HIERARCHY

Database

Table 1

Table 2

Page 1

Page 2

Page 3

Page n

Tuple 1

Tuple 2

Tuple 3

Tuple n

Attr 1

Attr 2

Attr n
DATABASE LOCK HIERARCHY

Table 1

Table 2

Page 1

Page 2

Page 3

... Page n

Tuple 1

Tuple 2

Tuple 3

... Tuple n

Attr 1

Attr 2

... Attr n
DATABASE LOCK HIERARCHY

**Table 1**

- **Tuple 1**
- **Tuple 2**
- **Tuple 3**

- **Page 1**
- **Page 2**
- **Page 3**

---

**Table 2**

- **Tuple 1**
- **Tuple 2**
- **Tuple 3**

- **Page 1**
- **Page 2**
- **Page 3**

---

**Database**

- **Attr 1**
- **Attr 2**
- **Attr n**
DATABASE LOCK HIERARCHY

- **Database**
  - **Table 1**
    - **Page 1**
      - **Tuple 1**
        - **Attr 1**
    - **Page 2**
      - **Tuple 2**
        - **Attr 2**
    - **Page 3**
      - **Tuple 3**
        - **Attr 3**
    - ... (Continued)
  - **Table 2**
    - **Page n**
      - **Tuple n**
        - **Attr n**

**Frequency Labels**:
- **Very Common**
- **Common**
- **Slightly Rare**
- **Rare**

**Page Numbers**:
- Page 1
- Page 2
- Page 3
- ... (Continued)
- Page n

**Attributes**:
- Attr 1
- Attr 2
- Attr n

**Tuples**:
- Tuple 1
- Tuple 2
- Tuple 3
- ... (Continued)
- Tuple n
INTENTION LOCKS

An intention lock allows a higher-level node to be locked in shared or exclusive mode without having to check all descendent nodes.

If a node is locked in an intention mode, then some txn is doing explicit locking at a lower level in the tree.
INTENTION LOCKS

Intention-Shared (IS)
→ Indicates explicit locking at lower level with S locks.
→ Intent to get S lock(s) at finer granularity.

Intention-Exclusive (IX)
→ Indicates explicit locking at lower level with X locks.
→ Intent to get X lock(s) at finer granularity.

Shared+Intention-Exclusive (SIX)
→ The subtree rooted by that node is locked explicitly in S mode and explicit locking is being done at a lower level with X locks.
# COMPATIBILITY MATRIX

<table>
<thead>
<tr>
<th>T₁ Holds</th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>SIX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>IX</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SIX</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>X</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
LOCKING PROTOCOL

Each txn obtains appropriate lock at highest level of the database hierarchy.

To get S or IS lock on a node, the txn must hold at least IS on parent node.

To get X, IX, or SIX on a node, must hold at least IX on parent node.
EXAMPLE

\( T_1 \) – Get the balance of Andy’s off-shore bank account.

\( T_2 \) – Increase bookie’s account balance by 1%.

**What locks should these txns obtain?**

→ **Exclusive** + **Shared** for leaf nodes of lock tree.

→ Special **Intention** locks for higher levels.
EXAMPLE – TWO-LEVEL HIERARCHY

Read Andy’s record in \( R \).

\[ T_1 \]

\[ \text{Table } R \]

\[ \text{Tuple 1} \quad \text{Tuple 2} \quad \ldots \quad \text{Tuple } n \]
EXAMPLE - TWO-LEVEL HIERARCHY

Read Andy’s record in R.

Table R

Tuple 1

Tuple 2

... Tuple n

Read
EXAMPLE – TWO-LEVEL HIERARCHY

Read Andy’s record in \( R \).

Read

\[ T_1 \]

Table \( R \)

Tuple 1

Tuple 2

\( \cdots \)

Tuple \( n \)
EXAMPLE – TWO-LEVEL HIERARCHY

Read Andy’s record in R.

Table R

Tuple 1
Tuple 2
... Tuple n

Read

<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>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
</tr>
<tr>
<td>IX</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>S</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>SIX</td>
<td>✔</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>X</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>
EXAMPLE – TWO-LEVEL HIERARCHY

Read Andy’s record in R.

<table>
<thead>
<tr>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>SIX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>✔</td>
<td>×</td>
<td>✔</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>✔</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>
EXAMPLE – TWO-LEVEL HIERARCHY

Read Andy’s record in R.

<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>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
</tr>
<tr>
<td>IX</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>S</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>SIX</td>
<td>✔</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>X</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>
EXAMPLE – TWO-LEVEL HIERARCHY

Update bookie’s record in R.

<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>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>IX</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>SIX</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>X</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>
EXAMPLE - TWO-LEVEL HIERARCHY

Update bookie’s record in R.

<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>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>IX</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>SIX</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>X</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Write
EXAMPLE – THREE TXNS

Assume three txns execute at same time:

→ **T_1** – Scan all tuples in **R** and update one tuple.
→ **T_2** – Read a single tuple in **R**.
→ **T_3** – Scan all tuples in **R**.
EXAMPLE – THREE TXNS

Table R

Tuple 1  Tuple 2  ...  Tuple n

<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>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>IX</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>SIX</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>X</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>
EXAMPLE – THREE TXNS

Scan all tuples in $R$ and update one tuple.

Table $R$

<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>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>IX</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SIX</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Tuple 1

Tuple 2

...
EXAMPLE – THREE TXNS

Scan all tuples in $R$ and update one tuple.

Table $R$

<table>
<thead>
<tr>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>SIX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Tuple 1: Read

Tuple 2: Read

Tuple $n$: Read+Write
EXAMPLE – THREE TXNS

Scan all tuples in R and update one tuple.

Tuple 1

Tuple 2

... 

Tuple n

Table R

<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>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
</tr>
<tr>
<td>IX</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>S</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>SIX</td>
<td>✔</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>X</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>
Scan all tuples in $R$ and update one tuple.

EXAMPLE – THREE TXNS

<table>
<thead>
<tr>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>SIX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>✔</td>
<td>×</td>
<td>✔</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>✔</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>
EXAMPLE – THREE TXNS

Scan all tuples in $R$ and update one tuple.

<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>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>IX</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>SIX</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>X</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>
EXAMPLE – THREE TXNS

Read a single tuple in $\mathbf{R}$. 

<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>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>IX</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>SIX</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>X</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>
EXAMPLE – THREE TXNS

Read a single tuple in $\mathbf{R}$. 

<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>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>IX</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>S</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>SIX</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>X</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
EXAMPLE – THREE TXNS

Read a single tuple in $R$.

<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>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>IX</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>SIX</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>X</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>
EXAMPLE – THREE TXNS

Read a single tuple in $R$.

<table>
<thead>
<tr>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>SIX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
EXAMPLE – THREE TXNS

Scan all tuples in $R$.

<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>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>IX</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SIX</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>X</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
EXAMPLE – THREE TXNS

Scan all tuples in R.

Table R

<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>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>IX</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>SIX</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>X</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>
Scan all tuples in $\mathbf{R}$. 

<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>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
</tr>
<tr>
<td>IX</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>S</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>SIX</td>
<td>✔</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>X</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>
EXAMPLE – THREE TXNS

Scan all tuples in \( R \).

<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>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>∙</td>
</tr>
<tr>
<td>IX</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>S</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>SIX</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>X</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
EXAMPLE – THREE TXNS

Scan all tuples in \( R \).
LOCK ESCALATION

The DBMS can automatically switch to coarser-grained locks when a txn acquires too many low-level locks.

This reduces the number of requests that the lock manager must process.
Applications typically don't acquire a txn's locks manually (i.e., explicit SQL commands).

Sometimes you need to provide the DBMS with hints to help it to improve concurrency.
→ Update a tuple after reading it.

Explicit locks are also useful when doing major changes to the database.
LOCK TABLE

Explicitly locks a table. Not part of the SQL standard.

→ Postgres/DB2/Oracle Modes: SHARE, EXCLUSIVE

→ MySQL Modes: READ, WRITE

**LOCK TABLE**

```
LOCK TABLE <table> IN <mode> MODE;
```

```
SELECT 1 FROM <table> WITH (TABLOCK, <mode>);
```

```
LOCK TABLE <table> <mode>;
```
SELECT...FOR UPDATE

Perform a select and then sets an exclusive lock on the matching tuples.

Can also set shared locks:

→ Postgres: FOR SHARE
→ MySQL: LOCK IN SHARE MODE

```
SELECT * FROM <table>
WHERE <qualification> FOR UPDATE;
```
CONCLUSION

2PL is used in almost every DBMS.
Automatically generates correct interleaving:
→ Locks + protocol (2PL, SS2PL ...)
→ Deadlock detection + handling
→ Deadlock prevention
NEXT CLASS

Timestamp Ordering Concurrency Control