16 Two-Phase Locking
ADMISTRIVIA

**Project #3** is due Sun Nov 14\(^{th}\) @ 11:59pm.

**Recitation** is on Thu Oct 28\(^{th}\) @ 5:00pm over Zoom.

**Homework #4** will be release today. It is due Sun Nov 7\(^{th}\) @ 11:59pm.
UPCOMING DATABASE TALK

The Pinecone Vector Database System
→ Mon Nov 1st @ 4:30pm ET
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.
→ Lin doesn't know of any DBMS that supports this.
BEGIN R(A)
W(A)
R(A) COMMIT

BEGIN R(A)
W(A) COMMIT

Schedule

T_1

T_2

EXAMPLE
EXAMPLE

Schedule

BEGIN  R(A)  COMMIT
R(A)   W(A)   R(A)  COMMIT
BEGIN  R(A)  COMMIT
W(A)   W(A)
We need a way to guarantee that all execution schedules are correct (i.e., serializable) without knowing the entire schedule ahead of time.
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></td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td>UNLOCK(A)</td>
</tr>
<tr>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Lock Manager

TIME
EXECUTING WITH LOCKS

Schedule

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

Lock Manager

Granted \((T_1 \rightarrow A)\)
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>W(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>R(A)</td>
<td></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!
EXECUTING WITH LOCKS

Schedule

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

Lock Manager

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

Schedule

<table>
<thead>
<tr>
<th></th>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOCK(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
<td>COMMIT</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>Threads</td>
</tr>
<tr>
<td>Protect...</td>
<td>Database Contents</td>
</tr>
<tr>
<td>During...</td>
<td>In-Memory Data Structures</td>
</tr>
<tr>
<td>Modes...</td>
<td>Critical Sections</td>
</tr>
<tr>
<td>Shared, Exclusive, Update,</td>
<td>Read, Write</td>
</tr>
<tr>
<td>Intention</td>
<td></td>
</tr>
<tr>
<td>Deadlock</td>
<td>Avoidance</td>
</tr>
<tr>
<td>Detection &amp; Resolution</td>
<td></td>
</tr>
<tr>
<td>...by...</td>
<td>Coding Discipline</td>
</tr>
<tr>
<td>Waits-for, Timeout, Aborts</td>
<td></td>
</tr>
<tr>
<td>Kept in...</td>
<td>Protected Data Structure</td>
</tr>
<tr>
<td>Lock Manager</td>
<td></td>
</tr>
</tbody>
</table>

Source: Goetz Graefe
# Locks vs. Latches

<table>
<thead>
<tr>
<th><strong>Locks</strong></th>
<th><strong>Latches</strong></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>✗</td>
</tr>
<tr>
<td>Exclusive</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
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

<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>UNLOCK(A)</td>
</tr>
<tr>
<td>S-LOCK(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

<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>UNLOCK(A)</td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Lock Manager

Granted (T₁→A)
EXECUTING WITH LOCKS

**Schedule**

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

**Lock Manager**

- Granted (T₁ → A)
- Released (T₁ → A)
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></td>
</tr>
<tr>
<td></td>
<td>X-LOCK(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UNLOCK(A)</td>
<td></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></td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

**Lock Manager**

- Granted (T₁→A)
- Released (T₁→A)
- Granted (T₂→A)
- Released (T₂→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>UNLOCK(A)</td>
</tr>
<tr>
<td>S-LOCK(A)</td>
<td>BEGIN</td>
</tr>
<tr>
<td>R(A)</td>
<td>X-LOCK(A)</td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>COMMIT</td>
<td>UNLOCK(A)</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

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

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

Lock Manager

Granted (T_1 \rightarrow A)

Released (T_1 \rightarrow A)

Granted (T_2 \rightarrow A)

Released (T_2 \rightarrow A)

Granted (T_1 \rightarrow A)

Released (T_1 \rightarrow A)
Two-phase locking (2PL) is a concurrency control protocol that determines whether a txn can access an object in the database on the fly.

The protocol does not need to know all the queries that a txn will execute ahead of time.
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 locks that it previously acquired. It cannot acquire new locks.
The txn is not allowed to acquire/upgrade locks after the growing phase finishes.
The txn is not allowed to acquire/upgrade locks after the growing phase finishes.
The txn is not allowed to acquire/upgrade locks after the growing phase finishes.

**Transaction Lifetime**

- **Growing Phase**
- **Shrinking Phase**

**2PL Violation!**
EXECUTING WITH 2PL

Schedule

\begin{align*}
T_1 & \quad T_2 \\
\text{BEGIN} & \quad \text{BEGIN} \\
\text{X-LOCK}(A) & \quad \text{X-LOCK}(A) \\
\text{R}(A) & \quad \text{R}(A) \\
\text{W}(A) & \quad \text{UNLOCK}(A) \\
\text{R}(A) & \quad \text{UNLOCK}(A) \quad \text{COMMIT} \\
\text{UNLOCK}(A) \quad \text{COMMIT} \\
\text{COMMIT} & \quad \text{COMMIT}
\end{align*}
EXECUTING WITH 2PL

Schedule

\[ T_1 \]

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

BEGIN
X-LOCK(A)

R(A)
UNLOCK(A)
COMMIT

\[ T_2 \]

UNLOCK(A)
COMMIT

17

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)
R(A)
UNLOCK(A)
COMMIT

LOCK Manager

Granted (T₁→A)

Denied!
EXECUTING WITH 2PL

Schedule

T₁

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

T₂

BEGIN
UNLOCK(A)

Lock Manager

Granted (T₁ → A)
Denied!
Released (T₁ → A)

TIME

T₁

T₂
EXECUTING WITH 2PL

Schedule

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

T₂
BEGIN
X-LOCK(A)
W(A)
UNLOCK(A)
COMMIT

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.
→ It generates schedules whose precedence graph is acyclic.

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

Schedule

\[ \begin{array}{c|c}
T_1 & T_2 \\
\hline
\text{BEGIN} & \text{BEGIN} \\
\text{X-LOCK}(A) & \text{X-LOCK}(A) \\
\text{X-LOCK}(B) & \\
\text{R}(A) & \text{R}(A) \\
\text{W}(A) & \text{W}(A) \\
\text{UNLOCK}(A) & : \\
\text{R}(B) & \\
\text{W}(B) & \\
\text{ABORT} & \\
\end{array} \]
## 2PL – Cascading Aborts

### Schedule

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

**TIME**
2PL – CASCADING ABORTS

**Schedule**

<table>
<thead>
<tr>
<th>Time</th>
<th>$T_1$</th>
<th>$T_2$</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>$X$-LOCK(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>UNLOCK(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ABORT</td>
<td></td>
</tr>
</tbody>
</table>

**Example Schedule:**

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

$\rightarrow$ 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.

$\rightarrow$ Any information about $T_1$ cannot be "leaked" to the outside world.

This is all wasted work!
There are potential schedules that are serializable but would not be allowed by 2PL.
→ Locking limits concurrency.

May still have "dirty reads".
→ Solution: Strong Strict 2PL (aka Rigorous 2PL)

May lead to deadlocks.
→ Solution: Detection or Prevention
2PL OBSERVATIONS

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

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.
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 that txn 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 Lin’s account (A) to his promoter’s account (B).

\(T_2\) – Compute the total amount in all accounts and return it to the application.

\(T_1\)

\[
\begin{align*}
\text{BEGIN} \\
A &= A - 100 \\
B &= B + 100 \\
\text{COMMIT}
\end{align*}
\]

\(T_2\)

\[
\begin{align*}
\text{BEGIN} \\
\text{ECHO } A + B \\
\text{COMMIT}
\end{align*}
\]
EXAMPLES

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

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

\[
\begin{align*}
T_1 & : \text{BEGIN} \\
& : A = A - 100 \\
& : B = B + 100 \\
& : \text{COMMIT}
\end{align*}
\]

\[
\begin{align*}
T_2 & : \text{BEGIN} \\
& : \text{ECHO } A + B \\
& : \text{COMMIT}
\end{align*}
\]
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

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

Initial Database State

A=1000, B=1000
NON-2PL EXAMPLE

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)

S-LOCK(A)
UNLOCK(A)
S-LOCK(B)

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

Initial Database State

A=1000, B=1000
NON-2PL EXAMPLE

Schedule

$T_1$

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

$T_2$

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

Initial Database State

$A=1000$, $B=1000$
NON-2PL EXAMPLE

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

Initial Database State

A=1000, B=1000

T₂ Output

A+B=1900
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>S-LOCK(B)</td>
</tr>
<tr>
<td>W(A)</td>
<td>R(B)</td>
</tr>
<tr>
<td>X-LOCK(B)</td>
<td>S-LOCK(B)</td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td>R(B)</td>
</tr>
<tr>
<td>R(B)</td>
<td>UNLOCK(A)</td>
</tr>
<tr>
<td>B=B+100</td>
<td>UNLOCK(B)</td>
</tr>
<tr>
<td>W(B)</td>
<td>ECHO A+B</td>
</tr>
<tr>
<td>UNLOCK(B)</td>
<td>COMMIT</td>
</tr>
<tr>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Initial Database State

A=1000, B=1000
**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>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>S-LOCK(B)</td>
</tr>
<tr>
<td>X-LOCK(B)</td>
<td>UNLOCK(A)</td>
</tr>
<tr>
<td>UNLOCK(B)</td>
<td>COMMIT</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></td>
</tr>
<tr>
<td>COMMIT</td>
<td>ECHO A+B</td>
</tr>
</tbody>
</table>

**Initial Database State**

A=1000, B=1000
2PL EXAMPLE

Schedule

Time

\( T_1 \)
BEGIN
\( X\text{-LOCK}(A) \)
\( R(A) \)
\( A=A-100 \)
\( W(A) \)
\( X\text{-LOCK}(B) \)
\( UNLOCK(A) \)
\( R(B) \)
\( B=B+100 \)
\( W(B) \)
\( UNLOCK(B) \)
\( COMMIT \)

\( T_2 \)
BEGIN
\( S\text{-LOCK}(A) \)
\( R(A) \)
\( S\text{-LOCK}(B) \)
\( R(B) \)
\( UNLOCK(A) \)
\( UNLOCK(B) \)
\( ECHO A+B \)
\( COMMIT \)

Initial Database State

\( A=1000, \ B=1000 \)
**2PL EXAMPLE**

### Schedule

<table>
<thead>
<tr>
<th><strong>T₁</strong></th>
<th><strong>T₂</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BEGIN</strong>&lt;br&gt;X-LOCK(A)&lt;br&gt;R(A)&lt;br&gt;A=A-100&lt;br&gt;W(A)&lt;br&gt;X-LOCK(B)&lt;br&gt;UNLOCK(A)&lt;br&gt;R(B)&lt;br&gt;B=B+100&lt;br&gt;W(B)&lt;br&gt;UNLOCK(B)&lt;br&gt;COMMIT</td>
<td><strong>BEGIN</strong>&lt;br&gt;S-LOCK(A)&lt;br&gt;R(A)&lt;br&gt;S-LOCK(B)&lt;br&gt;UNLOCK(A)&lt;br&gt;R(B)&lt;br&gt;UNLOCK(B)&lt;br&gt;ECHO A+B&lt;br&gt;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>$T_1$</th>
<th>$T_2$</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>$A=A-100$</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></td>
</tr>
<tr>
<td>UNLOCK(B)</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
</tr>
<tr>
<td>BEGIN</td>
<td></td>
</tr>
<tr>
<td>$S$-LOCK(A)</td>
<td></td>
</tr>
<tr>
<td>$R$(A)</td>
<td></td>
</tr>
<tr>
<td>$S$-LOCK(B)</td>
<td></td>
</tr>
<tr>
<td>$R$(B)</td>
<td></td>
</tr>
<tr>
<td>ECHO $A+B$</td>
<td></td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td></td>
</tr>
<tr>
<td>UNLOCK(B)</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
</tr>
</tbody>
</table>

**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
### STRONG STRICT 2PL EXAMPLE

#### Schedule

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

#### Initial Database State

- **A** = 1000, **B** = 1000

#### T₂ Output

- **A+B** = 2000
UNIVERSE OF SCHEDULES

All Schedules
UNIVERSE OF SCHEDULES

All Schedules

Serial
UNIVERSE OF SCHEDULES

All Schedules

Conflict Serializable

Serial
UNIVERSE OF SCHEDULES

All Schedules

View Serializable

Conflict Serializable

Serial
UNIVERSE OF SCHEDULES

- All Schedules
  - View Serializable
    - Conflict Serializable
      - No Cascading Aborts
        - Serial
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.
→ Locking limits concurrency.

May still have "dirty reads".
→ Solution: Strong Strict 2PL (Rigorous)

May lead to deadlocks.
→ Solution: Detection or Prevention
SHIT JUST GOT REAL, SON

Schedule

T₁

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

T₂

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

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

Granted (T₁→A)
SHIT JUST GOT REAL, SON

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(B)</td>
</tr>
<tr>
<td>R(A)</td>
<td>R(B)</td>
</tr>
<tr>
<td>X-LOCK(B)</td>
<td>S-LOCK(A)</td>
</tr>
</tbody>
</table>

Lock Manager

- Granted (T₁→A)
- Granted (T₂→B)
SHIT JUST GOT REAL, SON

Schedule

\[ T_1 \]
- BEGIN
- X-LOCK(A)
- R(A)
- X-LOCK(B)

\[ T_2 \]
- BEGIN
- S-LOCK(B)
- R(B)
- S-LOCK(A)

Lock Manager

- Granted (T_1 \rightarrow A)
- Denied!
- Granted (T_2 \rightarrow B)
SHIT JUST GOT REAL, SON

Schedule

BEGIN
X-LOCK(A)

R(A)
X-LOCK(B)

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

Lock Manager

Granted (T₁→A)

Granted (T₂→B)

Denied!

Denied!
SHIT JUST GOT REAL, SON

Schedule

T₁
BEGIN
X-LOCK(A)
R(A)
X-LOCK(B)

T₂
BEGIN

Lock Manager

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

TIME
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**
The DBMS creates a **waits-for** graph to keep track of what locks each txn 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>Time</th>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BEGIN</td>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td></td>
<td>S-LOCK(A)</td>
<td>X-LOCK(B)</td>
<td>S-LOCK(C)</td>
</tr>
<tr>
<td></td>
<td>S-LOCK(B)</td>
<td>X-LOCK(C)</td>
<td>X-LOCK(A)</td>
</tr>
</tbody>
</table>

Waits-For Graph

- T₁
- T₂
- T₃
DEADLOCK DETECTION

Schedule

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

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

BEGIN
X-LOCK(A)

Waits-For Graph

T_1

T_2

T_3
DEADLOCK DETECTION

**Schedule**

- **T_1**
  - BEGIN
  - S-LOCK(A)
  - S-LOCK(B)
- **T_2**
  - BEGIN
  - X-LOCK(B)
  - X-LOCK(C)
- **T_3**
  - BEGIN
  - S-LOCK(C)
  - X-LOCK(A)

**Waits-For Graph**

- **T_1**
- **T_2**
- **T_3**

The diagram illustrates a deadlock scenario where three processes (T_1, T_2, T_3) attempt to acquire locks on resources in a specific order, leading to a deadlock.
DEADLOCK DETECTION

Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BEGIN</td>
<td>S-LOCK(A)</td>
<td>BEGIN</td>
</tr>
<tr>
<td>2</td>
<td>X-LOCK(B)</td>
<td>S-LOCK(C)</td>
<td>X-LOCK(C)</td>
</tr>
</tbody>
</table>

Waits-For Graph

T₁ → T₂ → T₃

T₃ → T₁ → T₂
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 have to wait before deadlocks are broken.
Selecting the proper victim depends on a lot of different variables....
Selecting the proper victim depends on a lot of different variables....
→ By age (lowest timestamp)
Selecting the proper victim depends on a lot of different variables. ...

→ By age (lowest timestamp)
→ By progress (least/most queries executed)
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
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
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.
After selecting a victim txn to abort, the DBMS can also decide on how far to rollback the txn's changes.

**Approach #1: Completely**

**Approach #2: Minimally**
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

\[
\begin{array}{c|c}
T_1 & T_2 \\
\hline
\text{BEGIN} & \text{BEGIN} \\
\text{X-LOCK}(A) & \text{X-LOCK}(A) \\
\vdots & \vdots \\
\end{array}
\]
DEADLOCK PREVENTION

\[ T_1 \quad T_2 \]

\[
\begin{array}{c}
\text{BEGIN} \\
\text{X-LOCK}(A) \\
\vdots \\
\end{array}
\]

\[
\begin{array}{c}
\text{BEGIN} \\
\text{X-LOCK}(A) \\
\vdots \\
\end{array}
\]
DEADLOCK PREVENTION

**Wait-Die**

- $T_1$ waits

**Wound-Wait**

- $T_2$ aborts
DEADLOCK PREVENTION

Wait-Die

\[ T_1 \text{ waits} \]

Wound-Wait

\[ T_2 \text{ aborts} \]
DEADLOCK PREVENTION

Wait-Die

- $T_1$ waits
- $T_2$ aborts

Wound-Wait

- $T_2$ aborts
- $T_2$ waits
Why do these schemes guarantee no deadlocks?

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

Tuple 1

Attr 1

Table 2

Tuple 2

Attr 2

... 

Tuple n

Attr n
DATABASE LOCK HIERARCHY

- Database
  - Table 1
    - Tuple 1
      - Attr 1
      - Attr 2
      - Attr 3
    - Tuple 2
      - Attr 1
      - Attr 2
      - Attr 3
  - Table 2
    - Tuple 1
      - Attr 1
      - Attr 2
      - Attr 3
    - Tuple 2
      - Attr 1
      - Attr 2
      - Attr 3
    - ...
DATABASE LOCK HIERARCHY

T_1

Database

Table 1

Table 2

Tuple 1

Tuple 2

... (ellipsis)

Tuple n

Attr 1

Attr 2

... (ellipsis)

Attr n
DATABASE LOCK HIERARCHY

Database

Table 1

Tuple 1

Attr 1

Tuple 2

Attr 2

...

Tuple n

Attr 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 shared locks.

Intention-Exclusive (IX)
→ Indicates explicit locking at lower level with exclusive locks.

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

<table>
<thead>
<tr>
<th></th>
<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>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>IX</td>
<td>✔️</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>
<td>✗</td>
</tr>
<tr>
<td>SIX</td>
<td>✔️</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>
<td>✗</td>
</tr>
</tbody>
</table>
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 Lin's shady off-shore bank account.

\[ T_2 \] – Increase Andrew's bank account balance by 1%.

**What locks should these txns obtain?**
EXAMPLE

$T_1$ – Get the balance of Lin's shady off-shore bank account.

$T_2$ – Increase Andrew's bank 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

Table R

Tuple 1  Tuple 2  ...  Tuple n
EXAMPLE – TWO-LEVEL HIERARCHY

Read Andy's record in $R$.

Table $R$

Tuple 1  Tuple 2  ...  Tuple $n$
EXAMPLE – TWO-LEVEL HIERARCHY

Read Andy's record in R.
EXAMPLE – TWO-LEVEL HIERARCHY

Read Andy's record in $R$. 

Table $R$  

Tuple 1  Tuple 2  ...  Tuple $n$ 

Read
Read Andy's record in R.
EXAMPLE – TWO-LEVEL HIERARCHY

Read Andy's record in R.
EXAMPLE – TWO-LEVEL HIERARCHY

Update Biden's record in R.

Write
Update Biden's record in \( R \).
EXAMPLE – THREESOME

Assume three txns execute at same time:
- $T_1$ – Scan $R$ and update a few tuples.
- $T_2$ – Read a single tuple in $R$.
- $T_3$ – Scan all tuples in $R$. 

Table R

Tuple 1  Tuple 2  ...  Tuple n
EXAMPLE – THREESOME
EXAMPLE – THREESOME

Scan $R$ and update a few tuples. $T_1$
Scan **R** and update a few tuples.

```
  T1
```

```
Table R
```

```
Tuple 1
  Read
```

```
Tuple 2
  Read
```

```
Tuple n
  Read+Write
```
Scan $R$ and update a few tuples.

**EXAMPLE – THREESOME**

- $T_1$
- Table $R$
- Tuple 1
- Tuple 2
- ...
- Tuple $n$
EXAMPLE – THREESOME

Scan $R$ and update a few tuples.
Scan $R$ and update a few tuples.
EXAMPLE – THREESOME

Read a single tuple in R.
EXAMPLE – THREESOME

Read a single tuple in $R$. 

SIX

$T_1$

Table $R$

$T_2$

$T_1$

Tuple 1

Read

Tuple 2

... 

Tuple $n$
EXAMPLE – THREESOME

Read a single tuple in $R$. 

- $T_1$
- $T_2$
- SIX
- IS

Table R

- $T_1$
- $T_2$

Tuple 1

Tuple 2

... 

Tuple n
EXAMPLE – THREESOME

Read a single tuple in $R$. 

Table $R$ 

Tuple 1 

Tuple 2 

... 

Tuple $n$
EXAMPLE – THREESOME

Scan all tuples in $R$.

T_1 \rightarrow \text{SIX} \rightarrow \text{Tuple 1} \rightarrow \text{Read}

T_2 \rightarrow \text{IS} \rightarrow \text{Tuple 2} \rightarrow \text{Read}

T_3 \rightarrow \text{Table R} \rightarrow \text{Tuple n} \rightarrow \text{Read}
EXAMPLE – THREESOME

Scan all tuples in \( R \).
EXAMPLE – THREESOME

Scan all tuples in \( R \).

\[ T_1 \rightarrow T_3 \rightarrow T_2 \]

Table \( R \)

\[ T_1 \rightarrow T_2 \rightarrow S \]

\[ T_1 \rightarrow T_2 \rightarrow IS \]

Tuple 1

Tuple 2

\( \ldots \)
EXAMPLE – THREESOME

Scan all tuples in \( R \).

![Diagram showing a database table with tuples and a process for scanning all tuples.](image-url)
EXAMPLE – THREESOME

Scan all tuples in $R$. 

Table $R$

Tuple 1

Tuple 2

... 

Tuple $n$
Hierarchical locks are useful in practice as each txn only needs a few locks.

Intention locks help improve concurrency:
→ **Intention-Shared (IS):** Intent to get $S$ lock(s) at finer granularity.
→ **Intention-Exclusive (IX):** Intent to get $X$ lock(s) at finer granularity.
→ **Shared+Intention-Exclusive (SIX):** Like $S$ and $IX$ at the same time.
Lock escalation dynamically asks for coarser-grained locks when too many low-level locks acquired.

This reduces the number of requests that the lock manager must process.
LOCKING IN PRACTICE

You typically don't set locks manually in txns.

Sometimes you will need to provide the DBMS with hints to help it to improve concurrency. Explicit locks are also useful when doing major changes to the database.
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>;
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;
```
2PL is used in almost DBMS.

Automatically generates correct interleaving:
→ Locks + protocol (2PL, SS2PL ...)
→ Deadlock detection + handling
→ Deadlock prevention
NEXT CLASS

Timestamp Ordering Concurrency Control