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.
→ Andy doesn't know of any DBMS that supports this.
EXAMPLE

Schedule

T₁

BEGIN
R(A)

W(A)

R(A)

COMMIT

T₂

BEGIN
R(A)

W(A)

COMMIT

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

Lock Manager

 Granted (T₁→A)

TIME
EXECUTING WITH LOCKS

Schedule

<table>
<thead>
<tr>
<th>T_1</th>
<th>T_2</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 (T_1→A)

Denied!
EXECUTING WITH LOCKS

Schedule:

T₁

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

T₂

BEGIN
LOCK(A)

Lock Manager:

 Granted (T₁→A)
 Denied!
 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>BEGIN</td>
</tr>
<tr>
<td></td>
<td>LOCK(A)</td>
<td>LOCK(A)</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>R(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UNLOCK(A)</td>
<td>UNLOCK(A)</td>
</tr>
<tr>
<td></td>
<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
Isolation Levels
## Locks vs. Latches

<table>
<thead>
<tr>
<th>Locks</th>
<th>Latches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separate... User transactions</td>
<td>Threads</td>
</tr>
<tr>
<td>Protect... Database Contents</td>
<td>In-Memory Data Structures</td>
</tr>
<tr>
<td>During... Entire Transactions</td>
<td>Critical Sections</td>
</tr>
<tr>
<td>Modes... Shared, Exclusive, Update, Intention</td>
<td>Read, Write</td>
</tr>
<tr>
<td>Deadlock Detection &amp; Resolution</td>
<td>Avoidance</td>
</tr>
<tr>
<td>...by... Waits-for, Timeout, Aborts</td>
<td>Coding Discipline</td>
</tr>
<tr>
<td>Kept in... Lock Manager</td>
<td>Protected Data Structure</td>
</tr>
</tbody>
</table>

Source: Goetz Graefe
**BASIC LOCK TYPES**

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

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

<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>X</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

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

Lock Manager

Granted \((\text{T}_1 \rightarrow A)\)
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>COMMIT</td>
<td></td>
</tr>
</tbody>
</table>

Lock Manager

- Granted (T₁ → A)
- Released (T₁ → A)

TIME

T₁

T₂
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>S-LOCK(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)
- 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>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)
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)
- 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 is allowed to access an object in the database on the fly.

The protocol does not need to know all of 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 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.
The txn is not allowed to acquire/upgrade locks after the growing phase finishes.
EXECUTING WITH 2PL

---

**Schedule**

<table>
<thead>
<tr>
<th>T&lt;sub&gt;1&lt;/sub&gt;</th>
<th>T&lt;sub&gt;2&lt;/sub&gt;</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></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>W(A)</td>
</tr>
<tr>
<td></td>
<td>UNLOCK(A)</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

---

**Lock Manager**

- Granted (T<sub>1</sub>→A)
EXECUTING WITH 2PL

Schedule

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

Lock Manager

BEGIN
X-LOCK(A)

Granted (T₁→A)

Denied!

 TIME

T₁

T₂
EXECUTING WITH 2PL

Schedule

<table>
<thead>
<tr>
<th></th>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td></td>
<td>BEGIN</td>
</tr>
<tr>
<td>X-LOCK(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
<td>X-LOCK(A)</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 2PL

Schedule

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

T₂
BEGIN
X-LOCK(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.
→ It generates schedules whose precedence graph is acyclic.

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

**Schedule**

<table>
<thead>
<tr>
<th>Schedule</th>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td></td>
<td>BEGIN</td>
</tr>
<tr>
<td>X-LOCK(A)</td>
<td></td>
<td>X-LOCK(A)</td>
</tr>
<tr>
<td>X-LOCK(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABORT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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!**
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: **Strict 2PL**

May lead to deadlocks.
→ Solution: **Detection** or **Prevention**
**STRICT TWO-PHASE LOCKING**

The txn is not allowed to acquire/upgrade locks after the growing phase finishes. Allows only conflict serializable schedules, but it is often stronger than needed for some apps.
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 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>UNLOCK(B)</td>
</tr>
<tr>
<td>X-LOCK(B)</td>
<td>ECHO A+B</td>
</tr>
<tr>
<td>R(B)</td>
<td>COMMIT</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></td>
</tr>
</tbody>
</table>

Initial Database State

A=1000, B=1000
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>R(B)</td>
</tr>
<tr>
<td>UNLOCK(A)</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(B)</td>
</tr>
<tr>
<td>B=B+100</td>
<td>ECHO A+B</td>
</tr>
<tr>
<td>W(B)</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)

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
**NON-2PL EXAMPLE**

**Initial Database State**

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

**T₂ Output**

\[ A + B = 1100 \]

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

**Initial Database State**

A=1000, B=1000

**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>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(B)</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>
</tbody>
</table>

Initial Database State

A=1000, B=1000
2PL EXAMPLE

Schedule

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

```
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
```
STRONG 2PL EXAMPLE

Schedule

\begin{align*}
T_1 & \quad T_2 \\
\text{BEGIN} & \quad \text{BEGIN} \\
X-Lock(A) & \quad S-Lock(A) \\
R(A) & \quad R(A) \\
A=A-100 & \quad \text{ECHO } A+B \\
W(A) & \quad R(B) \\
X-Lock(B) & \quad \text{ECHO } A+B \\
R(B) & \quad \text{ECHO } A+B \\
B=B+100 & \quad \text{UNLOCK}(A) \\
W(B) & \quad \text{UNLOCK}(B) \\
\text{UNLOCK}(A) & \quad \text{UNLOCK}(A) \\
\text{UNLOCK}(B) & \quad \text{UNLOCK}(B) \\
\text{COMMIT} & \quad \text{COMMIT}
\end{align*}

Initial Database State

\( A=1000, \quad B=1000 \)
**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

**Initial Database State**

- A=1000, B=1000
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>S-LOCK(B)</td>
</tr>
<tr>
<td>W(A)</td>
<td>R(B)</td>
</tr>
<tr>
<td>X-LOCK(B)</td>
<td>B=B+100</td>
</tr>
<tr>
<td>R(B)</td>
<td>W(B)</td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td>UNLOCK(A)</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

T₂ Output

A+B=2000

Initial Database State

A+B=2000

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>B=B+100</td>
</tr>
<tr>
<td>R(B)</td>
<td>W(B)</td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td>UNLOCK(A)</td>
</tr>
<tr>
<td>UNLOCK(B)</td>
<td>COMMIT</td>
</tr>
<tr>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>
UNIVERSE OF SCHEDULES

All Schedules

View Serializable

Conflict Serializable

No Cascading Aborts

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: Strict 2PL

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

Schedule

\[ \begin{align*}
T_1 & : \\
\text{BEGIN} & : \\
\text{X-LOCK}(A) & : \\
\text{R}(A) & : \\
\text{X-LOCK}(B) & : \\
& \\
T_2 & : \\
\text{BEGIN} & : \\
\text{S-LOCK}(B) & : \\
\text{R}(B) & : \\
\text{S-LOCK}(A) & : \\
\end{align*} \]

Lock Manager

Granted \((T_1 \rightarrow A)\)
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)
SHIT JUST GOT REAL, SON

Schedule

<table>
<thead>
<tr>
<th></th>
<th>T_1</th>
<th>T_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>X-LOCK(A)</td>
<td>BEGIN</td>
</tr>
<tr>
<td>R(A)</td>
<td>X-LOCK(B)</td>
<td>S-LOCK(B)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R(B)</td>
</tr>
</tbody>
</table>

Lock Manager

- Granted (T_1→A)
- Granted (T_2→B)
- Denied!

TIME

Carnegie Mellon University
Database Group
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)

Lock Manager

Granted (T₁→A)
Denied!
Granted (T₂→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 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 will periodically check for cycles in **waits-for** graph and then make a decision on how to break it.
DEADLOCK DETECTION

Schedule

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

Waits-For Graph

T1

T2

T3

TIME
DEADLOCK DETECTION

Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>T_1</th>
<th>T_2</th>
<th>T_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>S-LOCK(A)</td>
<td>BEGIN</td>
<td>S-LOCK(C)</td>
</tr>
<tr>
<td>S-LOCK(B)</td>
<td>X-LOCK(B)</td>
<td>X-LOCK(C)</td>
<td></td>
</tr>
<tr>
<td>X-LOCK(C)</td>
<td></td>
<td>X-LOCK(A)</td>
<td></td>
</tr>
</tbody>
</table>

Waits-For Graph

- T_1 -> T_2
- T_1 -> T_3
- T_2 -> T_3
DEADLOCK DETECTION

Schedule

- T₁:
  - BEGIN
  - S-LOCK(A)
- T₂:
  - BEGIN
  - X-LOCK(B)
- T₃:
  - BEGIN
  - S-LOCK(C)
  - X-LOCK(C)
  - X-LOCK(A)

Waits-For Graph

- T₁
- T₂
- T₃

Waits: T₁ - T₂ - T₃
DEADLOCK DETECTION

Schedule

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

```
BEGIN X-LOCK(B)
X-LOCK(C)
```

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

Waits-For Graph

```
T_1  T_2  T_3
T_1  T_2  T_3
T_3  T_2  T_3
```
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.
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
Approach #2: Minimally
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 \textit{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

**Scenario 1:**
- **T1**: BEGIN X-LOCK(A) ...
- **T2**: BEGIN X-LOCK(A) ...

**Wait-Die**
- **T1** waits

**Wound-Wait**
- **T2** aborts

**Scenario 2:**
- **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. Why?
OBSERVATION

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

If a txn wants to update one billion tuples, then it has to acquire one billion locks.
LOCK GRANULARITIES

When we say that a txn acquires a “lock”, what does that actually mean?
→ On an Attribute? Tuple? Page? Table?

Ideally, each txn should obtain fewest number of locks that is needed...
DATABASE LOCK HIERARCHY

Database

Table 1

Tuple 1

Attr 1

Tuple 2

Attr 2

... Tuple n

... Attr n

CMU 15-445/645 (Fall 2018)
DATABASE LOCK HIERARCHY

Database

Table 1

Tuple 1

Attr 1

Tuple 2

Attr 2

...Tuple n

Attr n

Table 2

Tuple 1

Attr 1

Tuple 2

Attr 2

...Tuple n

Attr n
EXAMPLE

**T₁** – Get the balance of Andy’s shady off-shore bank account.

**T₂** – Increase Lin's bank account balance by 1%.

*What locks should these txns obtain?*

Multiple:

→ **Exclusive** + **Shared** for leaves of lock tree.

→ Special **Intention** locks for higher levels.
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 in an intention mode, then explicit locking is being done at a lower level in the tree.
INTENTION LOCKS

Intention-Shared (IS)
→ Indicates explicit locking at a lower level with shared locks.

Intention-Exclusive (IX)
→ Indicates locking at lower level with exclusive or shared locks.
INTENTION 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>T(_1) 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>
</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>

T\(_2\) Wants

- IS
- IX
- S
- SIX
- X
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 – 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
EXAMPLE – TWO-LEVEL HIERARCHY

Update Lin's record in R.
Update Lin's record in $R$. 

Write
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 $R$. 
SCAN R and update a few tuples.

Example – Threesome

Table R

Tuple 1
Read

Tuple 2
Read

Tuple n
Read+Write
Scan $R$ and update a few tuples.
EXAMPLE – THREESOME

Read a single tuple in $R$. 

Table $R$

Tuple 1

Tuple 2

... 

Tuple $n$
EXAMPLE – THREESOME

Read a single tuple in \( R \).
EXAMPLE – THREESOME

Scan all tuples in $R$. 

Table $R$

Tuple 1  Tuple 2  ...  Tuple $n$

Read  Read  Read

$T_1$  $T_3$  $T_2$

$T_2$  $T_1$

SIX

IS

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$X$

$S$
EXAMPLE – THREESOME

Scan all tuples in $R$.

Table $R$

$T_1$ $T_3$ $T_2$

Tuple 1 $T_1$ $T_3$ $T_2$

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

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 has to 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.
LOCK TABLE

Explicitly locks a table.
Not part of the SQL standard.
→ Postgres/DB2/Oracle Modes: SHARE, EXCLUSIVE
→ MySQL Modes: READ, WRITE

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 DBMS.

Automatically generates correct interleaving:
→ Locks + protocol (2PL, S2PL ...)
→ Deadlock detection + handling
→ Deadlock prevention
NEXT CLASS

Timestamp Ordering Concurrency Control