Lecture #16

Two-Phase Locking
ADMINISTRIVIA

Project #3 is due Nov 12 19th @ 11:59pm.
→ Start early on this project.

Homework #4 is out, due Nov 12th @ 11:59pm.
LAST CLASS

Conflict Serializable

→ Verify using 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

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

Schedule

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

T₂
BEGIN
LOCK(A)
R(A)
W(A)
UNLOCK(A)
COMMIT

Lock Manager

Granted (T₁→A)
EXECUTING WITH LOCKS

Schedule

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

T₂
BEGIN
LOCK(A)
R(A)
W(A)
UNLOCK(A)
COMMIT

Lock Manager

Granted (T₁→A)

Denied!
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></td>
</tr>
<tr>
<td>W(A)</td>
<td></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>

Color-coded arrows indicate the sequence of operations and interactions with the lock manager.

Lock Manager

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

Schedule

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

T₂
BEGIN
LOCK(A)
R(A)
UNLOCK(A)
COMMIT

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>BEGIN</td>
<td>LOCK(A)</td>
<td>BEGIN</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
<td>LOCK(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
<td>UNLOCK(A)</td>
</tr>
<tr>
<td>R(A)</td>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
<td></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><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>
### Basic Lock Types

- **S-lock**: Shared locks for reads.
- **X-lock**: Exclusive locks for writes.

### Compatibility Matrix

<table>
<thead>
<tr>
<th>Existing Mode</th>
<th>S</th>
<th>IS</th>
<th>IX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>IS</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>IX</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>X</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Table 1.1: Compatibility of page lock and row lock modes**

<table>
<thead>
<tr>
<th>Lock Mode</th>
<th>Existing Mode</th>
<th>S</th>
<th>IS</th>
<th>IX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>IX</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>X</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Table 1.2: Conflicting Lock Modes**

- **PostgreSQL**
- **Microsoft SQL Server**
- **Oracle**
- **MySQL**

**Table 1.3: Table-level lock type compatibility**

- **Conflict**
- **Compatible**
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>W(A)</td>
</tr>
<tr>
<td>S-LOCK(A)</td>
<td>UNLOCK(A)</td>
</tr>
<tr>
<td>R(A)</td>
<td>UNLOCK(A)</td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td>COMMIT</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></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>
<tr>
<td>COMMIT</td>
<td></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

Lock Manager

Granted (T₁→A)

Released (T₁→A)

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

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

Transaction Lifetime

<table>
<thead>
<tr>
<th># of Locks</th>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Growing Phase</td>
<td>Shrinking Phase</td>
</tr>
</tbody>
</table>

**Lock Point**
TWO-PHASE LOCKING

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

**Transaction Lifetime**

Growing Phase

Shrinking Phase

# of Locks

TIME

2PL Violation!
EXECUTING WITH 2PL

Schedule

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

dependency graph
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

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**
STRONG STRICT TWO-PHASE LOCKING

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 with some apps need.
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 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.

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

\[
\begin{align*}
&T_2 \\
&\text{BEGIN} \\
&ECHO A + B \\
&\text{COMMIT}
\end{align*}
\]
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>A=A</td>
</tr>
<tr>
<td>W(A)</td>
<td>UNLOCK(A)</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>R(B)</td>
</tr>
<tr>
<td>W(B)</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

T2 Output

A+B=1900

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></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>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>
<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>UNLOCK(A)</td>
<td></td>
</tr>
<tr>
<td>UNLOCK(B)</td>
<td></td>
</tr>
<tr>
<td>ECHO A+B</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></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>R(A)</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>S-LOCK(B)</td>
</tr>
<tr>
<td>R(B)</td>
<td>R(B)</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>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

**T₂ Output**

A+B=2000

**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

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

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

IT JUST GOT REAL

Granted (T₁→A)
Denied!

Granted (T₂→B)
Denied!

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

Schedule

BEGIN
S-LOCK(A)

BEGIN
S-LOCK(B)

BEGIN
X-LOCK(B)

BEGIN
X-LOCK(C)

BEGIN
S-LOCK(C)

BEGIN
X-LOCK(A)

Waits-For Graph

T_1

T_2

T_3

T_1

T_2

T_3

Waits-For Graph with deadlock detected.
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 “work” done)
→ 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

**Wait-Die**
- $T_2$ aborts

**Wound-Wait**
- $T_2$ 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.
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 (Very Common)**
    - **Page 1 (Very Common)**
      - **Tuple 1 (Very Common)**
        - **Attr 1 (Rare)**
    - **Page 2 (Very Common)**
      - **Tuple 2 (Very Common)**
        - **Attr 2 (Rare)**
    - **Page 3 (Very Common)**
      - **Tuple 3 (Very Common)**
        - **Attr n (Rare)**
  - **Table 2 (Slightly Rare)**
    - **Page n (Common)**
      - **Tuple n (Very Common)**
        - **Attr n (Rare)**
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>√</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>

- **T₂ 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**

$T_1$ – Get the balance of Andy’s shady 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.

Update bookie’s record in R.
EXAMPLE – THREE TRANSACTIONS

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 TRANSACTIONS

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

Table $R$

Tuple 1  Tuple 2  ...  Tuple $n$

$T_1$

SIX

$T_1$

$T_1$

$T_1$
EXAMPLE – THREE TRANSACTIONS

Read a single tuple in $R$. 

Tuple 1

Tuple 2

... 

Tuple $n$ 

Table $R$
EXAMPLE – THREE TRANSACTIONS

Scan all tuples in R.
EXAMPLE – THREE TRANSACTIONS

Scan all tuples in $R$. 

```
Tuple 1  Tuple 2  ...  Tuple n
```

```
Table R
```

```
T_3
```

```
S
```

```
T_3
```
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.
 Explicitly locks a table. Not part of the SQL standard.

→ Postgres/DB2/Oracle Modes: **SHARE, EXCLUSIVE**
→ MySQL Modes: **READ, WRITE**

```sql
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;
```
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