Two-Phase Locking
UPCOMING DATABASE EVENTS

QuasarDB Talk
→ Thursday Nov 2nd @ 12pm
→ CIC 4th Floor

Peloton Hack-a-thon
→ Friday Nov 10th @ 9:30am
→ GHC 8102

TimescaleDB Talk
→ Thursday @ Nov 16th @ 12:00pm
→ CIC 4th Floor
A DBMS's concurrency control and recovery components permeate throughout the design of its entire architecture.
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  T₂

BEGIN
R(A)

W(A)

R(A)

BEGIN
R(A)

W(A)

COMMIT

COMMIT

TIME
How could you guarantee that all resulting schedules are correct (i.e., serializable)?

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

Schedule

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

Lock Manager

Granted (T₁ → A)
EXECUTING WITH LOCKS

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Lock Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td></td>
</tr>
<tr>
<td>BEGIN</td>
<td>Granted ($T_1 \rightarrow A$)</td>
</tr>
<tr>
<td>LOCK(A)</td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td>Denied!</td>
</tr>
<tr>
<td></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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Lock Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_2$</td>
<td></td>
</tr>
<tr>
<td>BEGIN</td>
<td></td>
</tr>
<tr>
<td>LOCK(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></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>

CMU 15-445/645 (Fall 2017)
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></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 LOCK(A)</td>
<td>BEGIN 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></td>
</tr>
<tr>
<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

Locks
→ Protects the index’s logical contents from other txns.
→ Held for txn duration.
→ Need to be able to rollback changes.

Latches
→ Protects the critical sections of the index’s internal data structure from other threads.
→ Held for operation duration.
→ Do not need to be able to rollback changes.
## Locks vs. Latches

<table>
<thead>
<tr>
<th>Locks</th>
<th>Latches</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Separate...</strong></td>
<td>Threads</td>
</tr>
<tr>
<td><strong>Protect...</strong></td>
<td>In-Memory Data Structures</td>
</tr>
<tr>
<td><strong>During...</strong></td>
<td>Critical Sections</td>
</tr>
<tr>
<td><strong>Modes...</strong></td>
<td>Read, Write</td>
</tr>
<tr>
<td><strong>Deadlock</strong></td>
<td>Avoidance</td>
</tr>
<tr>
<td>...by...</td>
<td>Coding Discipline</td>
</tr>
<tr>
<td><strong>Kept in...</strong></td>
<td>Protected Data Structure</td>
</tr>
</tbody>
</table>

- Latches: Threads, In-Memory Data Structures, Critical Sections, Read, Write, Avoidance, Coding Discipline, Protected Data Structure.

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>X</td>
</tr>
</tbody>
</table>

CMU 15-445/645 (Fall 2017)
EXECUTING WITH LOCKS

Transactions request locks (or upgrades)
Lock manager grants or blocks requests
Transactions release locks
Lock manager updates its internal lock-table
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>UNLOCK(A)</td>
<td>UNLOCK(A)</td>
</tr>
<tr>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Lock Manager

Granted (T₁→A)
## Executing with Locks

### Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>BEGIN</td>
</tr>
<tr>
<td></td>
<td>X-LOCK(A)</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>UNLOCK(A)</td>
</tr>
<tr>
<td>T₂</td>
<td>BEGIN</td>
</tr>
<tr>
<td></td>
<td>X-LOCK(A)</td>
</tr>
<tr>
<td></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 (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>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td>UNLOCK(A)</td>
</tr>
<tr>
<td>S-LOCK(A)</td>
<td>S-LOCK(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)
- Released (T1→A)
- Granted (T2→A)
- Released (T2→A)
## Executing with Locks

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Lock Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>Granted (T₁→A)</td>
</tr>
<tr>
<td>X-LOCK (A)</td>
<td>Granted (T₁→A)</td>
</tr>
<tr>
<td>R(A)</td>
<td>Granted (T₁→A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>Granted (T₁→A)</td>
</tr>
<tr>
<td>UNLOCK (A)</td>
<td>Released (T₁→A)</td>
</tr>
<tr>
<td>BEGIN</td>
<td>Granted (T₂→A)</td>
</tr>
<tr>
<td>X-LOCK (A)</td>
<td>Granted (T₂→A)</td>
</tr>
<tr>
<td>UNLOCK (A)</td>
<td>Released (T₂→A)</td>
</tr>
<tr>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>
EXECUTING WITH LOCKS

Schedule

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

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

\[ T_1 \rightarrow A \]

Lock Manager

Granted \((T_1 \rightarrow A)\)

Denied!

CMU 15-445/645 (Fall 2017)
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)

CMU 15-445/645 (Fall 2017)
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.
This is a permissible schedule in 2PL, but the DBMS has to also abort $T_2$. 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 actually stronger than needed.

![Graph showing growing and shrinking phases with instruction to release all locks at end of txn.](image-url)
**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 $50 from Andy’s account to his bookie’s account.

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

Legend:

→ $A$ ➔ Andy’s account.

→ $B$ ➔ The bookie’s account.
BEGIN X-LOCK(A)  
R(A)  
A=A-50  
W(A)  
UNLOCK(A)  
X-LOCK(B)  
R(B)  
B=B+50  
W(B)  
UNLOCK(B)  
COMMIT

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

Schedule
**NON-2PL EXAMPLE**

### Schedule

<table>
<thead>
<tr>
<th>T_1</th>
<th>T_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN&lt;br&gt;x-lock(A)&lt;br&gt;r(A)&lt;br&gt;A=A-50&lt;br&gt;w(A)&lt;br&gt;unlock(A)</td>
<td>BEGIN&lt;br&gt;s-lock(A)&lt;br&gt;r(A)&lt;br&gt;unlock(A)&lt;br&gt;s-lock(B)</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+50&lt;br&gt;w(B)&lt;br&gt;unlock(B)</td>
<td>unlock(B)</td>
</tr>
<tr>
<td>commit</td>
<td>commit</td>
</tr>
</tbody>
</table>

### Initial Database State

A=100, B=100

### T_2 Output

A+B=150
2PL EXAMPLE

Schedule

T₁
BEGIN
X-LOCK(A)
R(A)
A=A-50
W(A)
X-LOCK(B)
UNLOCK(A)
R(B)
B=B+50
W(B)
UNLOCK(B)
COMMIT

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

Initial Database State
A=100, B=100

T₂ Output
A+B=200
**STRICT 2PL EXAMPLE**

**Initial Database State**

A=100, B=100

**T2 Output**

A+B=200

---

**Schedule**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN X-LOCK(A) R(A) A=A-50 W(A) X-LOCK(B) R(B) B=B+50 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>
UNIVERSE OF SCHEDULES

All Schedules

View Serializable

Conflict Serializable

No Cascading Aborts

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: **Strict 2PL**

May lead to deadlocks.
→ Solution: **Detection** or **Prevention**
BEGIN X-LOCK(A)
R(A)
X-LOCK(B)

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

Granted (T₁→A)
BEGIN X-LOCK(A) BEGIN
R(A) S-LOCK(B)
X-LOCK(B) R(B)
S-LOCK(A)

Granted (T₁→A)
Granted (T₂→B)
BEGIN X-LOCK(A)
R(A)
X-LOCK(B)

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

Granted (T$_1$→A)

Denied!

Granted (T$_2$→B)
BEGIN
X-LOCK(A)

R(A)

X-LOCK(B)

T_1

BEGIN
S-LOCK(B)

R(B)

S-LOCK(A)

T_2

Granted (T_1→A)

Granted (T_2→B)

Denied!

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:
→ 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:

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

T₁ waits for T₂
T₂ waits for T₃
T₃ waits for T₁
DEADLOCK DETECTION

Schedule

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

Waits-For Graph

T₁

→

T₂

→

T₃
DEADLOCK DETECTION

Schedule

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

Waits-For Graph

T₁ - T₂ - T₃ - T₁

CMU 15-445/645 (Fall 2017)
DEADLOCK DETECTION

How often should we run the algorithm?

How many txns are typically involved?

What do we do when we find a deadlock?
**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 depending on how the application invoked it.
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.
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, kill one of them to prevent a deadlock.

No waits-for graph or detection algorithm.
DEADLOCK PREVENTION

Assign priorities based on timestamps:
→ Older ⇨ higher priority (e.g., $T_1 > T_2$)

Wait-Die ("Old Waits for Young")
→ If $T_1$ has higher priority, $T_1$ waits for $T_2$.
→ Otherwise $T_1$ aborts.

Wound-Wait ("Young Waits for Old")
→ If $T_1$ has higher priority, $T_2$ aborts.
→ Otherwise $T_1$ 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 transaction 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

T_1

Diagram showing the hierarchy of database locks with nodes labeled: Database, Table 1, Tuple 1, Attr 1, Table 2, Tuple 2, Attr 2, and Tuple n, Attr n.
DATABASE LOCK HIERARCHY

- $T_1$
- Database
- Table 1
- Table 2
- Tuple 1
- Tuple 2
- ... Tuple $n$
- Attr 1
- Attr 2
- ... Attr $n$
DATABASE LOCK HIERARCHY

$T_1$

- Database
  - Table 1
    - Tuple 1
      - Attr 1
    - Tuple 2
      - Attr 2
    - ... (ellipsis)
    - Tuple $n$
      - Attr $n$
EXEMPLARY

$T_1$ – Get the balance of Andy’s shady off-shore bank account.

$T_2$ – Increase Joy's bank account balance by 1%.

What locks should they obtain?
EXAMPLE

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

\(T_2\) – Increase Joy's bank account balance by 1%.

What locks should they 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 descendendent 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.
<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>T_1</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>S</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>IS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
</tbody>
</table>

**T\_2 Wants**
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

Update Joy's record in R.
EXAMPLE – TWO-LEVEL HIERARCHY

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

Assume three txns execute at the 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 $R$. 
EXAMPLE – THREESOME

Scan \( R \) and update a few tuples.
EXAMPLE – THREESOME

Scan $R$ and update a few tuples.

Table $R$

Tuple 1

Tuple 2

Tuple $n$
EXAMPLE – THREE SOMES

Read a single tuple in R.
EXAMPLE – THREESOME

Read a single tuple in R.
EXAMPLE – THREESOME

Scan all tuples in $\mathbf{R}$. 

Tuples $T_1$, $T_2$, and $T_3$ are accessing the table $\mathbf{R}$. Each tuple reads a tuple: Tuple 1, Tuple 2, ..., Tuple $n$. 

The diagram illustrates the read operations on the tuples and the interconnections between them and the table $\mathbf{R}$. 

---

CMU 15-445/645 (Fall 2017)
EXAMPLE – THREESOME

Scan all tuples in \( R \).

\begin{align*}
T_1 & \quad T_3 & \quad T_2 \\
SIX & \quad S & \\
IS & \quad \text{Table } R & \\
T_2 & \quad T_3 & \quad T_1 \\
\text{Tuple 1} & \quad \text{Tuple 2} & \quad \ldots & \quad \text{Tuple } n \\
S & \quad X &
\end{align*}
MULTIPLE LOCK GRANULARITIES

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.
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.
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. Sometimes you will need to provide the DBMS with hints to help it to improve concurrency. Also useful for doing major changes.
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

```sql
SELECT * FROM <table>
WHERE <qualification> FOR UPDATE;
```
DYNAMIC DATABASES

Recall that so far we have only dealing with transactions that read and update data.

But now if we have insertions, updates, and deletions, we have new problems...
THE PHANTOM PROBLEM

Schedule

T₁

BEGIN

SELECT MAX(age)
FROM people
WHERE status='lit'

COMMIT

T₂

BEGIN

INSERT INTO people
(age=96, status='lit')

72

SELECT MAX(age)
FROM people
WHERE status='lit'

96

COMMIT

CREATE TABLE people (id SERIAL, name VARCHAR, age INT, status VARCHAR);

CMU 15-445/645 (Fall 2017)
WTF?

How did this happen?
→ Because $T_1$ locked only existing records and not ones under way!

Conflict serializability on reads and writes of individual items guarantees serializability only if the set of objects is fixed.

We will solve this problem in the next class.
CONCLUSION

2PL is used in almost DBMS.

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

Two-Phase Locking
Isolation Levels