16 Two-Phase Locking
ADMINISTRIVIA

Project 2 due tonight!

Project 3 released today

Final exam Monday, May 1st, 8:30 – 11:30 a.m.
LAST TIME: CONCURRENCY CONTROL

Atomicity
Consistency
Isolation
→ Serial execution schedules
→ Serializable
  → Conflict serializable
  → View serializable
  → Linearizable
Durability
  → Strict serializable
Concurrency control and recovery are among the most important functions provided by a DBMS.
Concurrency control and recovery are among the most important functions provided by a DBMS. This paper introduces Spinor, a distributed, globally-concurrent, and highly-reliable database system. Spinnor is designed to handle concurrent transactions across a network of servers. The paper discusses the design and implementation of Spinor, emphasizing its ability to manage global transactions efficiently. Spinor's architecture includes a transaction management protocol that allows for atomic commits across the distributed system.

Abstract
Spinor is a distributed database system that provides global transaction support across a network of servers. The system is designed to handle concurrent transactions efficiently, ensuring data consistency and availability. Spinor uses a novel protocol to manage global transactions, allowing it to achieve high performance and scalability. The paper presents the design and implementation of Spinor, highlighting its ability to handle large-scale concurrent operations.
OBSERVATION

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

One solution: Use locks to protect database objects.
→ System automatically locks & unlocks objects as needed
→ Ensures that resulting execution is equivalent to some serial execution order
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)
 Denied!
 Released (T₁→A)
 Granted (T₂→A)
 Released (T₂→A)
## Locks vs. Latches

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

Source: Goetz Graefe
TODAY'S AGENDA

Lock Types
Two-Phase Locking
Deadlock Detection + Prevention
Hierarchical Locking
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>
### BASIC LOCK TYPES

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

### Compatibility Matrix

#### Table 1: Compatibility matrix of page lock and row lock modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>S (Share)</th>
<th>X (Exclusive)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S (Share)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>X (Exclusive)</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

#### Table 2: Compatibility of table and page lock modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>IS (Intent Shared)</th>
<th>S (Share)</th>
<th>U (Update)</th>
<th>IX (Intent Exclusive)</th>
<th>X (Exclusive)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS (Intent Shared)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>S (Share)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>U (Update)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>IX (Intent Exclusive)</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>X (Exclusive)</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Conflicting Lock Modes

#### Table 3.1: Conflicting Lock Modes

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

#### Table 3.2: Table-level lock type compatibility

<table>
<thead>
<tr>
<th>Lock Type</th>
<th>PostgreSQL</th>
<th>MySQL</th>
<th>Oracle</th>
<th>SQL Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Compatible</td>
<td>Compatible</td>
<td>Compatible</td>
<td>Compatible</td>
</tr>
<tr>
<td>X</td>
<td>Conflict</td>
<td>Conflict</td>
<td>Compatible</td>
<td>Conflict</td>
</tr>
<tr>
<td>IS</td>
<td>Conflict</td>
<td>Conflict</td>
<td>Compatible</td>
<td>Compatible</td>
</tr>
<tr>
<td>IX</td>
<td>Conflict</td>
<td>Conflict</td>
<td>Compatible</td>
<td>Compatible</td>
</tr>
<tr>
<td>S</td>
<td>Conflict</td>
<td>Conflict</td>
<td>Compatible</td>
<td>Compatible</td>
</tr>
<tr>
<td>IS</td>
<td>Conflict</td>
<td>Conflict</td>
<td>Compatible</td>
<td>Compatible</td>
</tr>
</tbody>
</table>

- **PostgreSQL**: Indicates compatibility or conflict depending on the lock type.
- **MySQL**: Indicates compatibility or conflict depending on the lock type.
- **Oracle**: Indicates compatibility or conflict depending on the lock type.
- **SQL Server**: Indicates compatibility or conflict depending on the lock type.
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

T₁

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

T₂

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

<table>
<thead>
<tr>
<th>Time</th>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></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)
Two-phase locking (2PL) is a concurrency control protocol that determines whether a txn can access an object in the database at runtime.

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

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

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

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

Transaction Lifetime

Growing Phase

Shrinking Phase
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)

T₂

BEGIN
X-LOCK(A)
R(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 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 needed for some apps.

![Diagram showing Growing Phase and Shrinking Phase with a note: Release all locks at end of txn.]
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.

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

\[
\begin{align*}
T_2 &: \text{BEGIN} \quad \text{ECHO} \quad \text{A}+\text{B} \\
    & \quad \text{COMMIT}
\end{align*}
\]
2PL EXAMPLE

**Schedule**

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

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

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
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

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

Waits-For Graph

T₁ → T₂ → T₃ → T₁
DEADLOCK HANDLING

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

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

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

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

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

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

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

Approach #1: Completely
→ Rollback entire txn and tell the application it was aborted.

Approach #2: Partial (Savepoints)
→ DBMS rolls back a portion of a txn (to break deadlock) and then attempts to re-execute the undone queries.
DEADLOCK PREVENTION

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

This approach does not require a \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

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 like an old man at a corrupt senior center.
OBSERVATION

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

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

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

When a txn wants to acquire a "lock", the DBMS can decide the granularity (i.e., scope) of that lock.

→ Attribute? Tuple? Page? Table?

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

Trade-off between parallelism versus overhead.

→ Fewer Locks, Larger Granularity vs. More Locks, Smaller Granularity.
DATABASE LOCK HIERARCHY

Database

Table 1

Table 2

Page 1

Page 2

Page 3

... 

Page n

Tuple 1

Tuple 2

Tuple 3

... 

Tuple n

Attr 1

Attr 2

... 

Attr n

Slightly Rare

Very Common

Common

Very Common

Rare
INTENTION LOCKS

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

If an object is locked in an intention mode, then some txn is doing explicit locking at a lower level.
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.
LOCKING PROTOCOL

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

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

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

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

$T_2$ – Increase Chi’s account balance by 6%.

**What locks should these txns obtain?**

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

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

Read Andy's record in $\mathbf{R}$.

Update Chi’s record in $\mathbf{R}$.
EXAMPLE – THREE QUERIES

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 QUERIES

Scan all tuples in $R$.

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

Read a single tuple in $R$.

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

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.

Read a single tuple in $R$.
<table>
<thead>
<tr>
<th></th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>SIX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IS</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td><strong>IX</strong></td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td><strong>S</strong></td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td><strong>SIX</strong></td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td><strong>X</strong></td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>
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.
LOCKING IN PRACTICE

Applications typically don't acquire a txn's locks manually (i.e., explicit SQL commands).

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

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

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

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

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

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

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

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

Can also set shared locks:

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

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

2PL is used in almost every DBMS.
Automatically generates correct interleaving:
→ Locks + protocol (2PL, SS2PL ...)
→ Deadlock detection + handling
→ Deadlock prevention
PROJECT #3 – QUERY EXECUTION

You will add support for executing queries in BusTub.

BusTub supports (basic) SQL with a rule-based optimizer for converting AST into physical plans.

https://15445.courses.cs.cmu.edu/spring2023/project3/
PROJECT #3 – TASKS

Plan Node Executors
→ Access Methods: Sequential Scan, Index Scan
→ Modifications: Insert, Update, Delete
→ Joins: Nested Loop Join, Hash Join
→ Miscellaneous: Aggregation, Limit, Sort, Top-N

Optimizer Rules:
→ Convert Nested Loop Join into a Hash Join
→ Convert `ORDER BY + LIMIT` into a Top-N
The leaderboard requires you to add additional rules to the optimizer to generate query plans.
→ It will be impossible to get a top ranking by just having the fastest implementations in Project #1 + Project #2.
DEVELOPMENT HINTS

Implement the **Insert** and **Sequential Scan** executors first so that you can populate tables and read from it.

You do **not** need to worry about transactions.

The aggregation and hash join hash tables do not need to be backed by the buffer pool (i.e., use STL).

Gradescope is meant for grading, **not** debugging. Write your own local tests.
THINGS TO NOTE

Do **not** change any file other than the ones that you submit to Gradescope.

Make sure you pull in the latest changes from the BusTub main branch.

Post your questions on Piazza or come to TA office hours.

Compare against our solution in your browser!
NEXT CLASS

Timestamp Ordering Concurrency Control