Intro to Database Systems (15-445/645)

# 16 Two-Phase Locking

Carnegie Mellon University

SPRING 2023

Charlie Garrod

# 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

Strict serializable

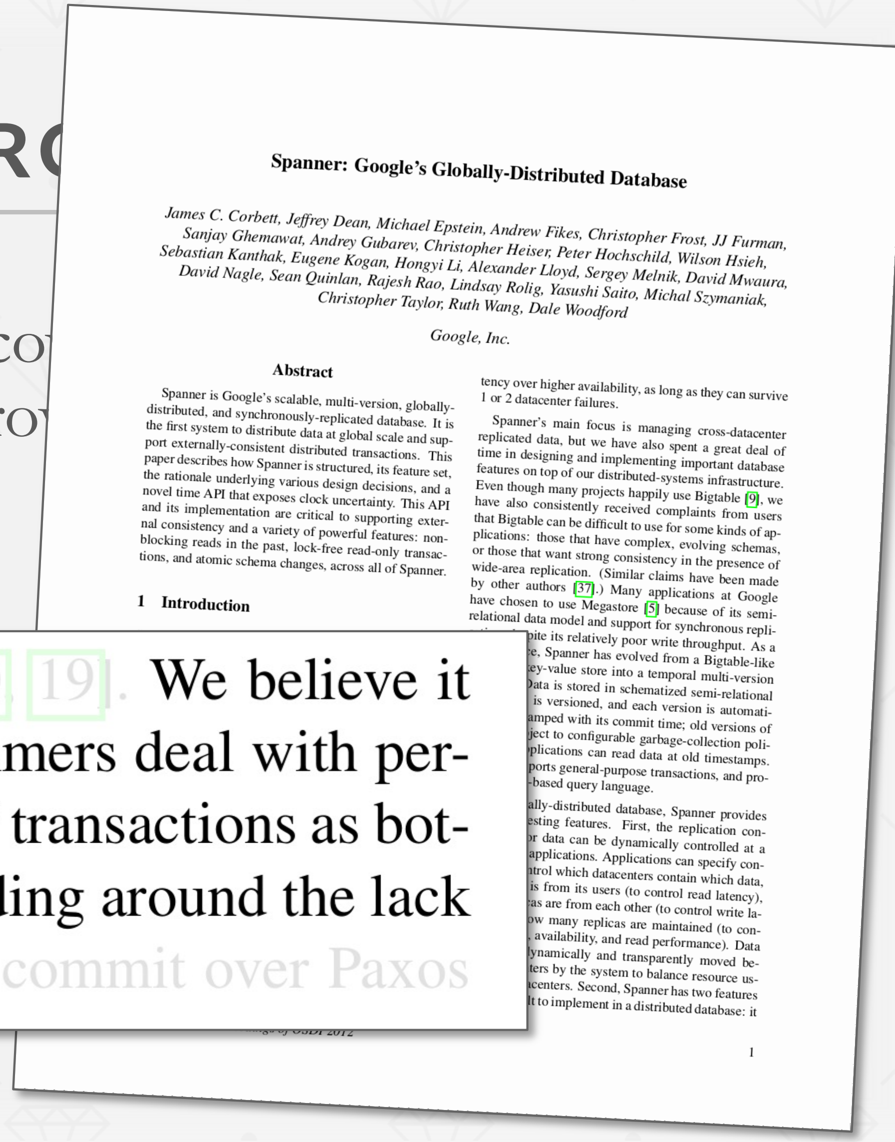
**Durability**

# CONCURRENCY CONTROL CONCLUSIONS

Concurrency control and recovery are among the most important functions provided by a DBMS.

# CONCURRENCY CONTRO

Concurrency control and reco
most important functions pro


Spanner: Google's Globally-Distributed Database

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, Dale Woodford

Google, Inc.

Abstract

Spanner is Google's scalable, multi-version, globally-distributed, and synchronously-replicated database. It is the first system to distribute data at global scale and support externally-consistent distributed transactions. This paper describes how Spanner is structured, its feature set, the rationale underlying various design decisions, and a novel time API that exposes clock uncertainty. This API and its implementation are critical to supporting external consistency and a variety of powerful features: non-blocking reads in the past, lock-free read-only transactions, and atomic schema changes, across all of Spanner.

1 Introduction

tency over higher availability, as long as they can survive 1 or 2 datacenter failures.

Spanner's main focus is managing cross-datacenter replicated data, but we have also spent a great deal of time in designing and implementing important database features on top of our distributed-systems infrastructure. Even though many projects happily use Bigtable [9], we have also consistently received complaints from users that Bigtable can be difficult to use for some kinds of applications: those that have complex, evolving schemas, or those that want strong consistency in the presence of wide-area replication. (Similar claims have been made by other authors [37].) Many applications at Google have chosen to use Megastore [5] because of its semi-relational data model and support for synchronous repli-

1

ability problems that it brings [9, 10, 19]. **We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions.** Running two-phase commit over Paxos

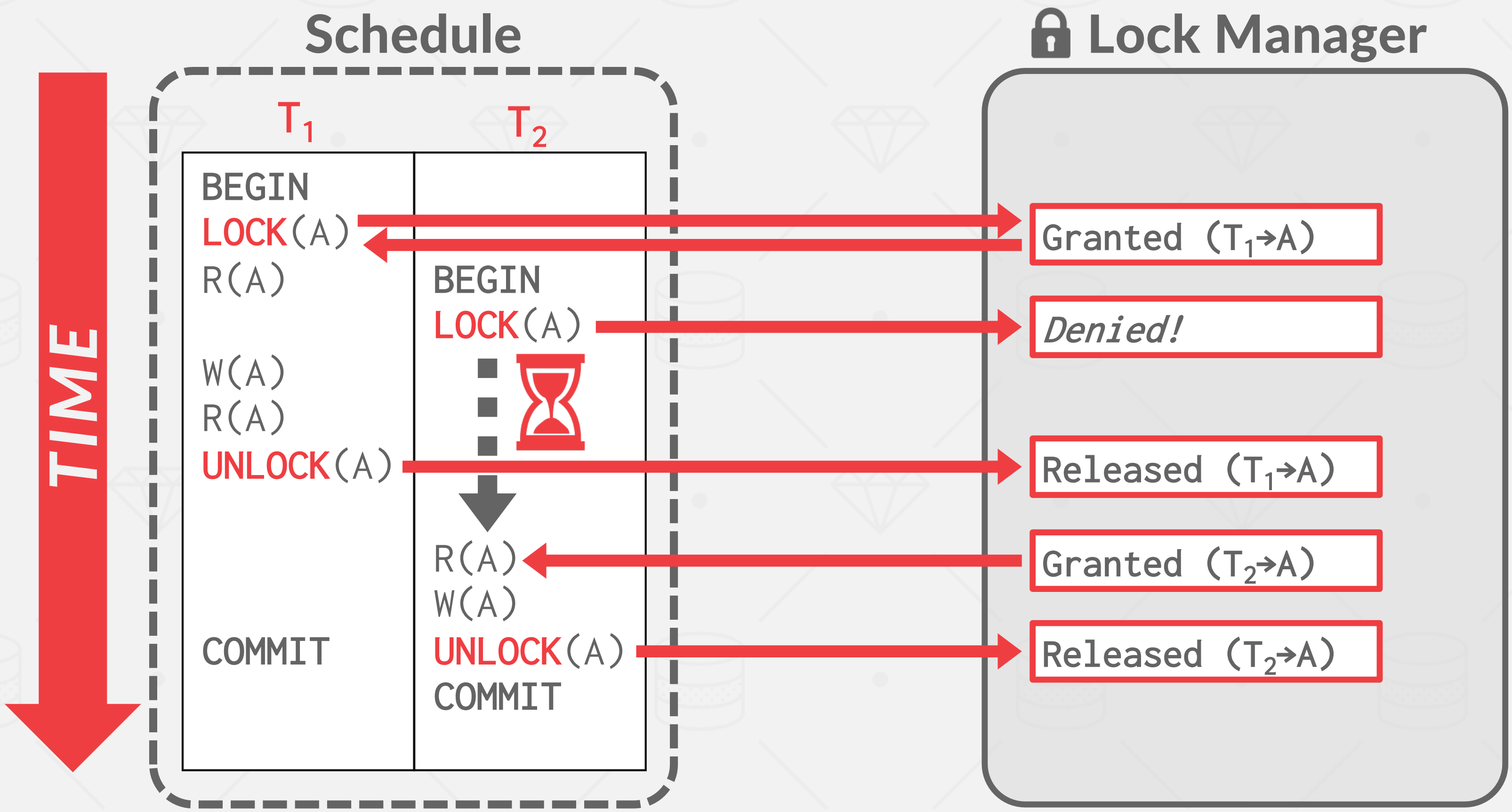
# OBSERVATION

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



# LOCKS VS. LATCHES

| | ***Locks*** | ***Latches*** |
|---|---|---|
| **Separate…** | User transactions | Threads |
| **Protect…** | Database Contents | In-Memory Data Structures |
| **During…** | Entire Transactions | Critical Sections |
| **Modes…** | Shared, Exclusive, Update, Intention | Read, Write |
| **Deadlock** | Detection & Resolution | Avoidance |
| **…by…** | Waits-for, Timeout, Aborts | Coding Discipline |
| **Kept in…** | Lock Manager | Protected Data Structure |

Source: Goetz Graefe

# TODAY'S AGENDA

Lock Types
Two-Phase Locking
Deadlock Detection + Prevention
Hierarchical Locking

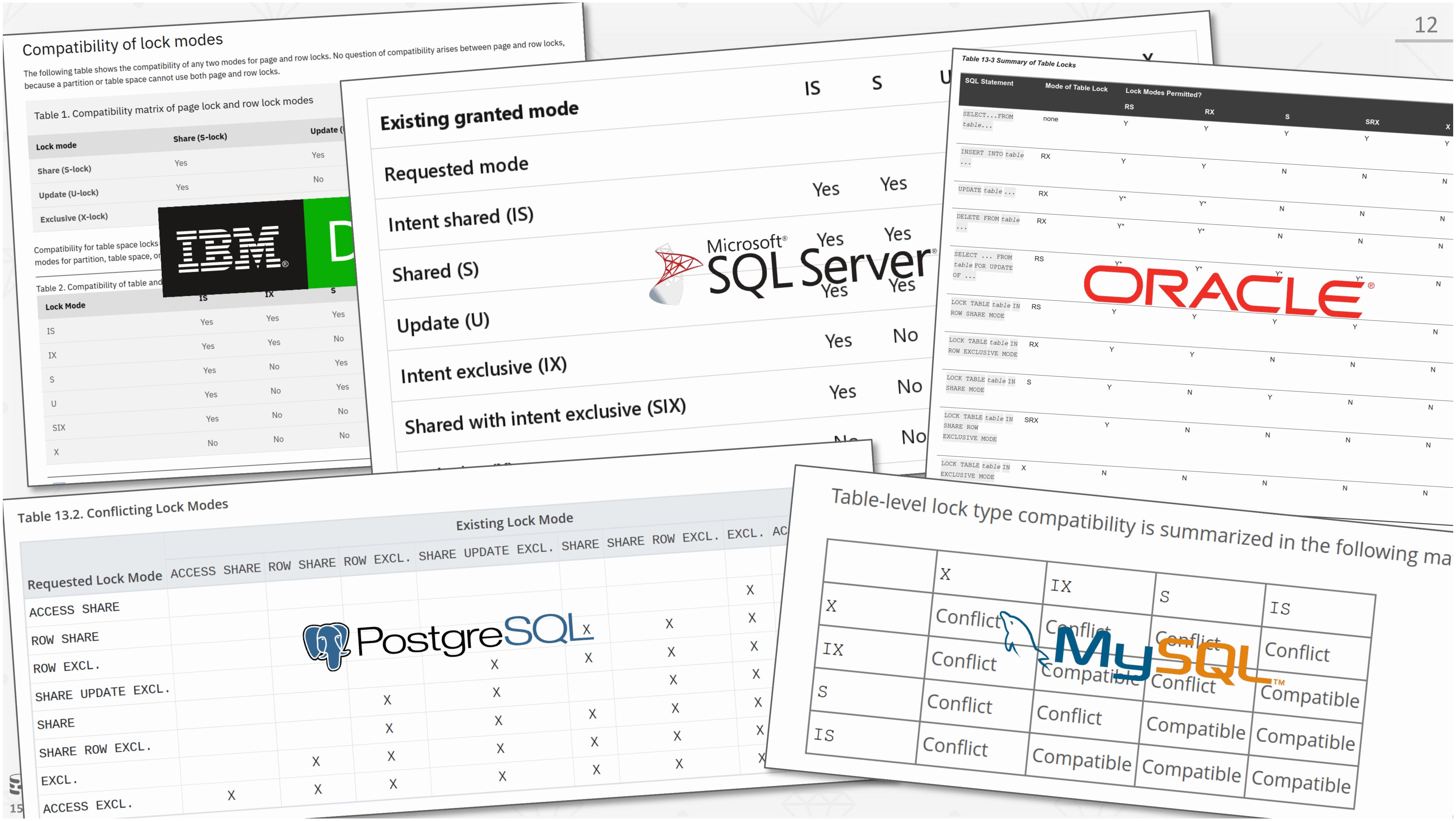
# BASIC LOCK TYPES

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

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

### Compatibility Matrix

| | Shared | Exclusive |
|---|---|---|
| Shared | ✓ | X |
| Exclusive | X | X |


Compatibility of lock modes
The following table shows the compatibility of any two modes for page and row locks. No question of compatibility arises between page and row locks, because a partition or table space cannot use both page and row locks.
Table 1. Compatibility matrix of page lock and row lock modes
Lock mode
Share (S-lock)
Update (U-lock)
Exclusive (X-lock)
Table 2. Compatibility of table and
IBM
Existing granted mode
Requested mode
Intent shared (IS)
Shared (S)
Update (U)
Intent exclusive (IX)
Shared with intent exclusive (SIX)
Microsoft® SQL Server®
Table 13-3 Summary of Table Locks
SQL Statement
Mode of Table Lock
Lock Modes Permitted?
ORACLE®
Table 13.2. Conflicting Lock Modes
Existing Lock Mode
Requested Lock Mode
ACCESS SHARE
ROW SHARE
ROW EXCL.
SHARE UPDATE EXCL.
SHARE
SHARE ROW EXCL.
EXCL.
ACCESS EXCL.
PostgreSQL
Table-level lock type compatibility is summarized in the following ma
Conflict
Compatible
MySQL™

# EXECUTING WITH LOCKS

Transactions request locks (or upgrades).

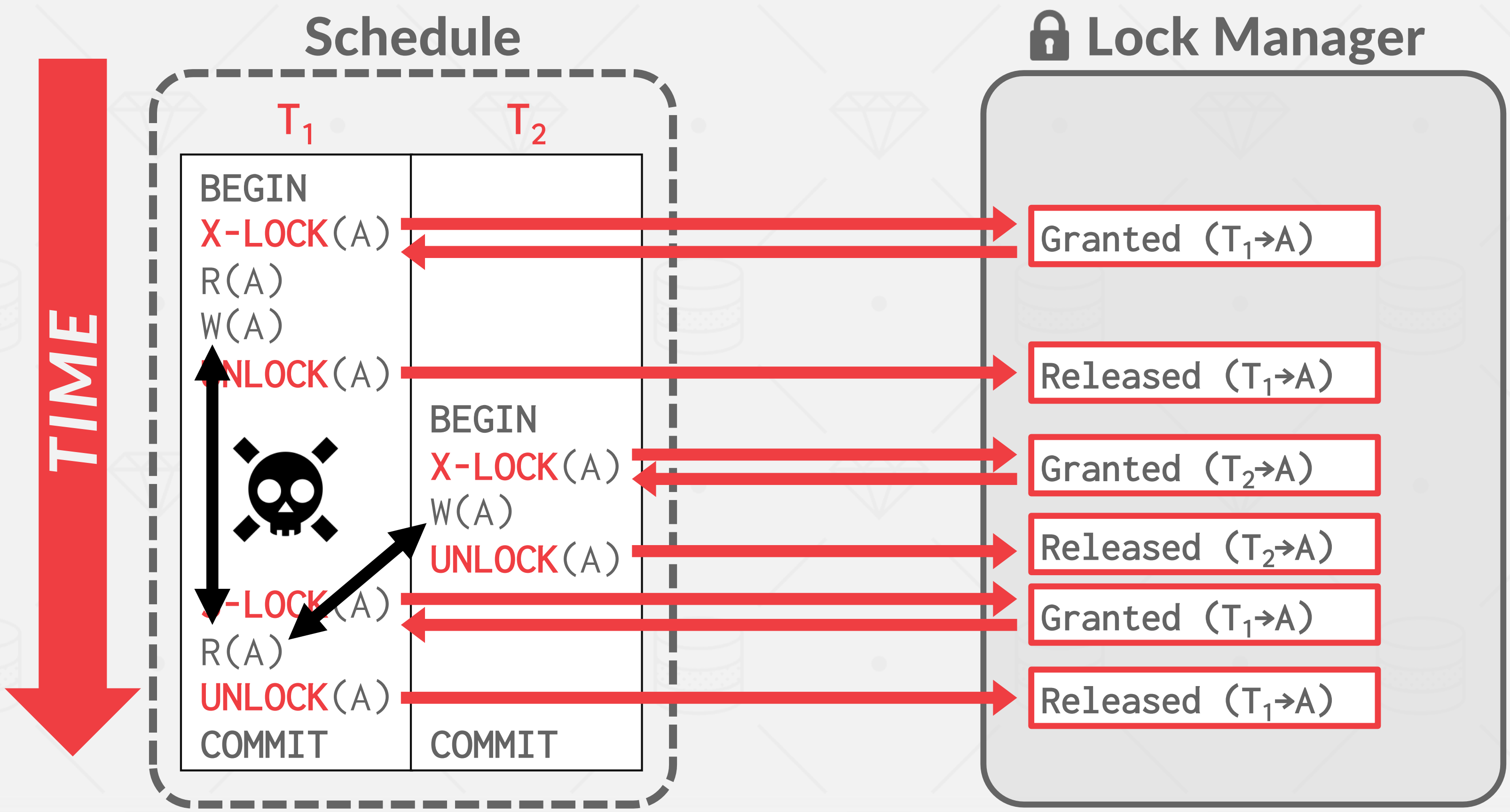
Lock manager grants or blocks requests.

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



# 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**

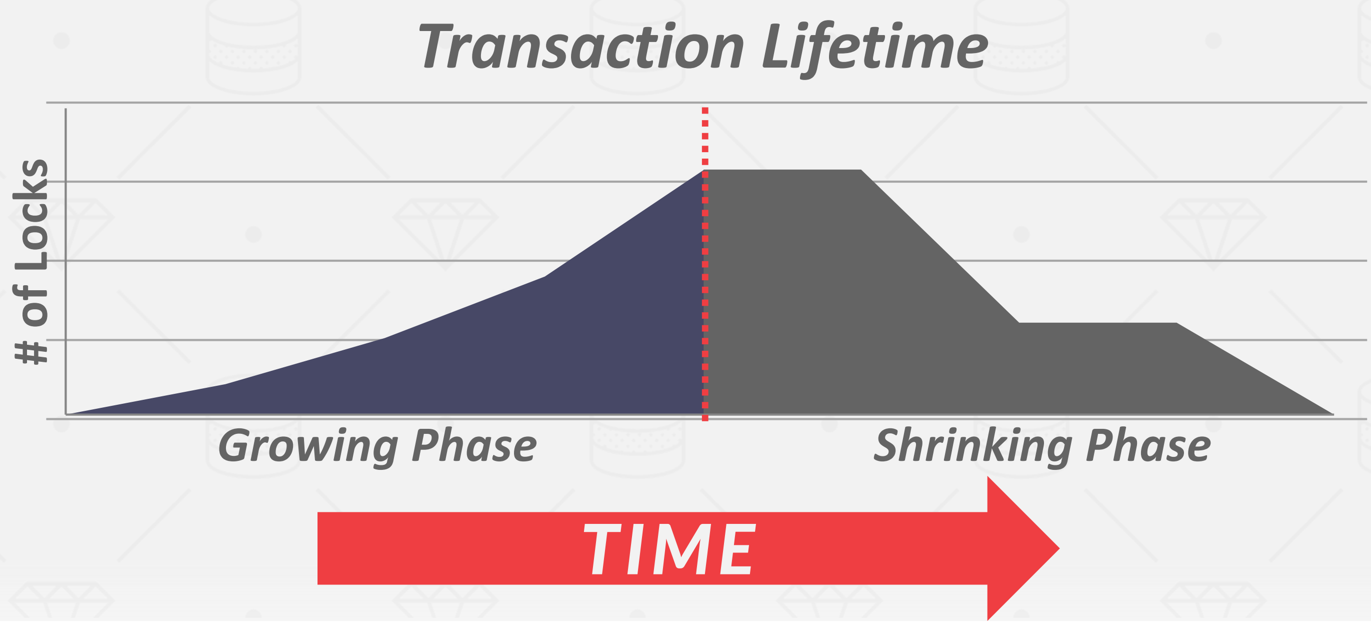
→ Each txn requests the locks that it needs from the DBMS's lock manager.
→ The lock manager grants/denies lock requests.

**Phase #2: Shrinking**

→ The txn is allowed to only release/downgrade locks that it previously acquired. It cannot acquire new locks.

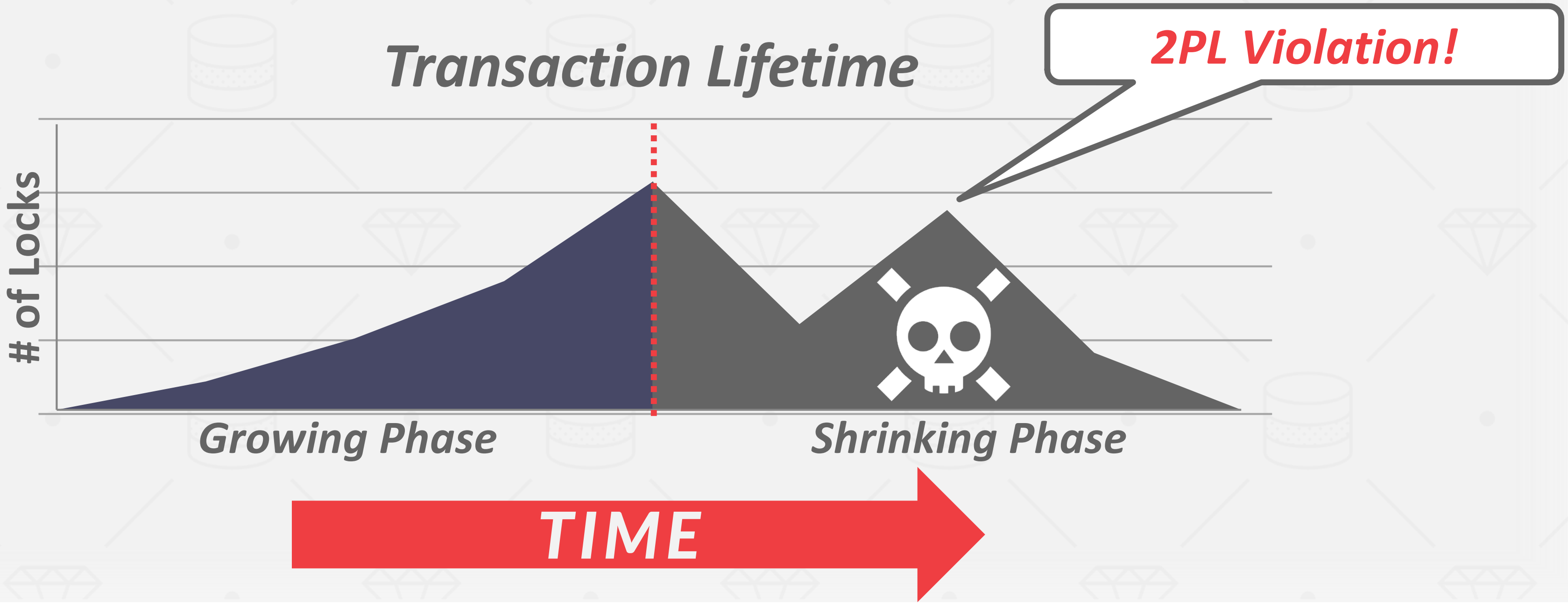
# TWO-PHASE LOCKING

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

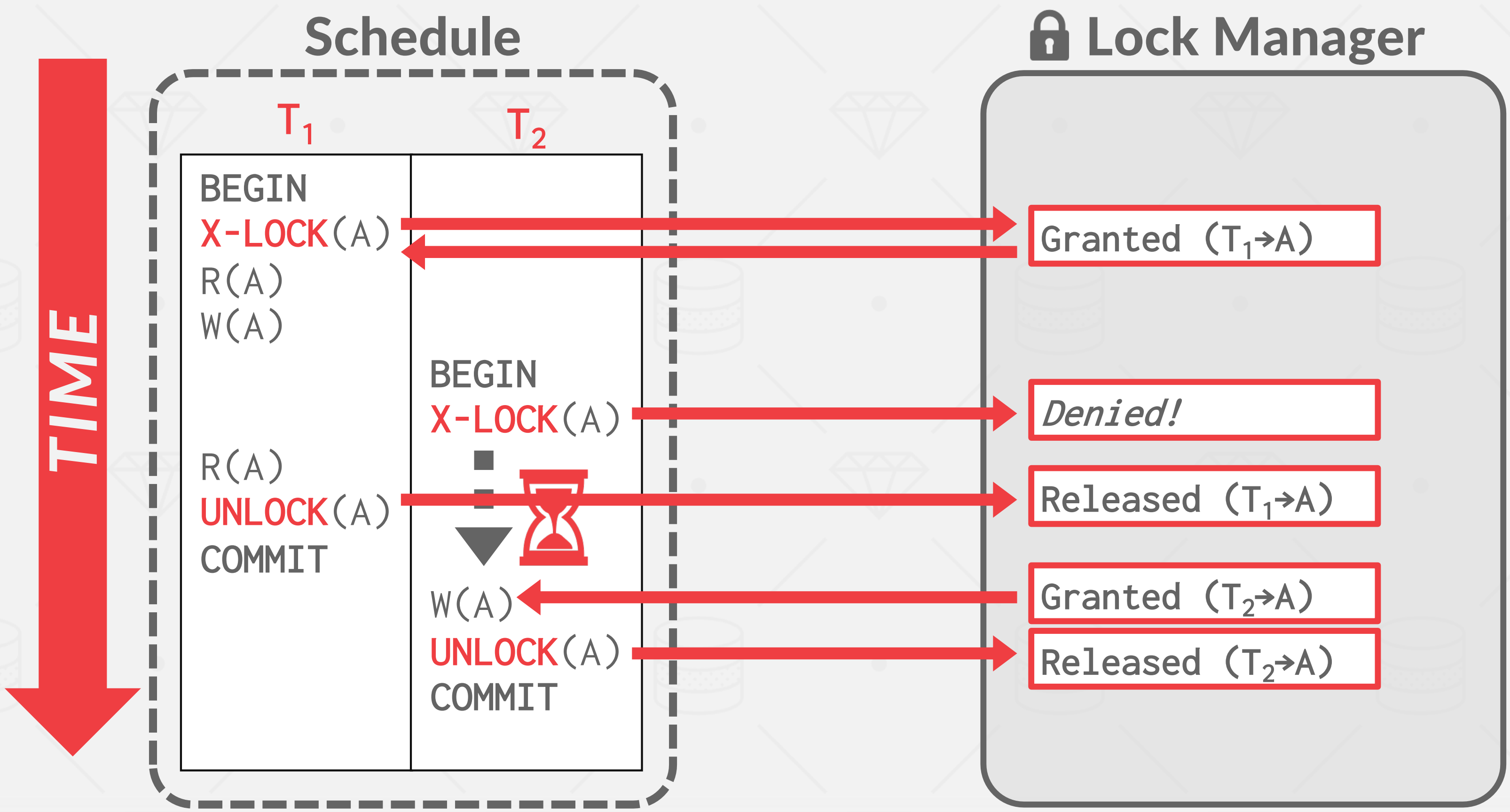


# TWO-PHASE LOCKING

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



# EXECUTING WITH 2PL

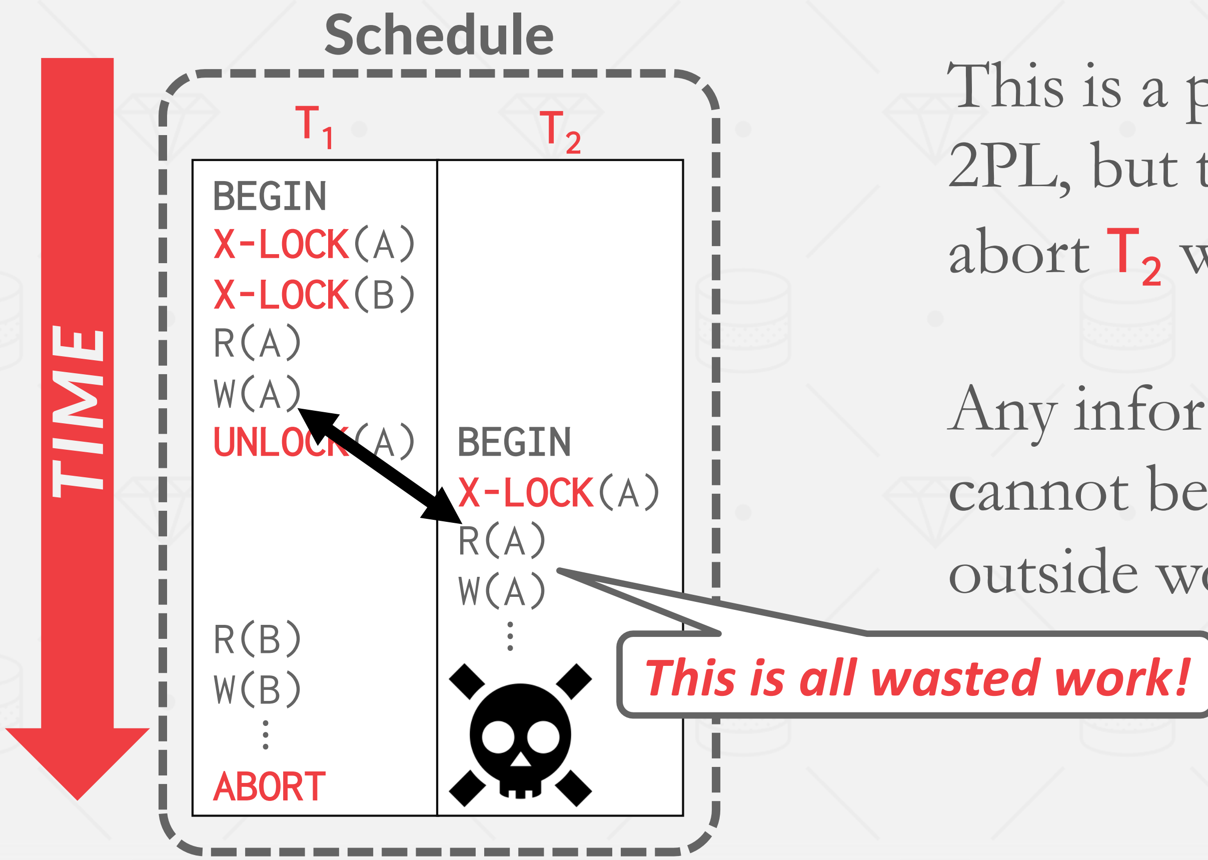


# TWO-PHASE LOCKING

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.

# 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)**

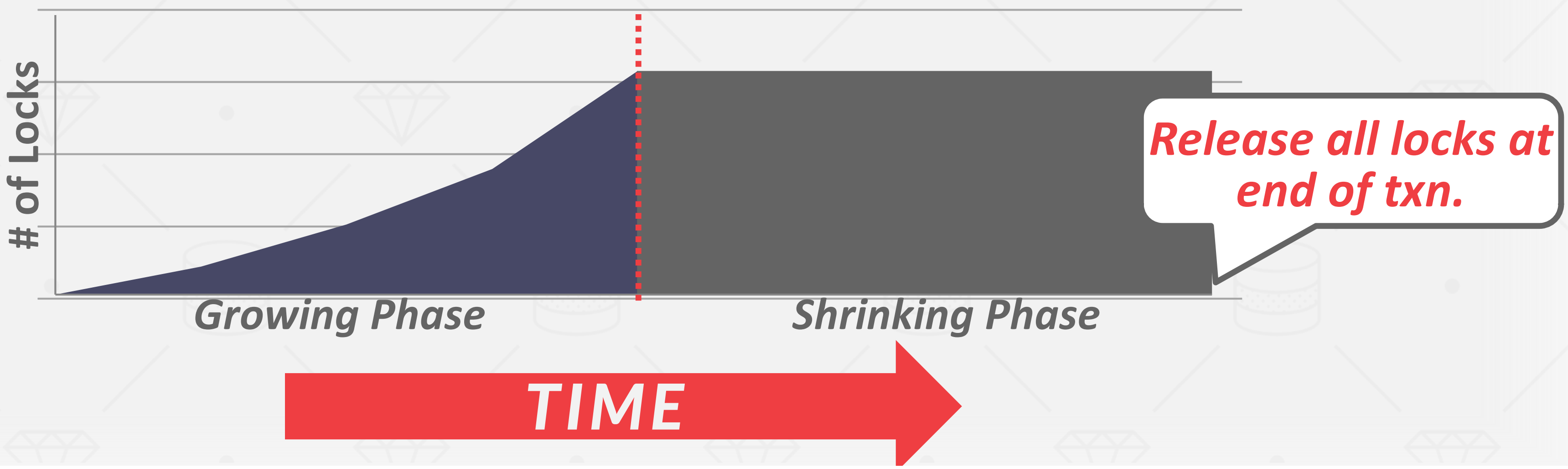
May lead to deadlocks.

→ 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.



# 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.

$T_1$

```
BEGIN
A=A-100
B=B+100
COMMIT
```

$T_2$

```
BEGIN
ECHO A+B
COMMIT
```

# 2PL EXAMPLE

TIME

## Schedule

| $T_1$ | $T_2$ |
|---|---|
| BEGIN | BEGIN |
| X-LOCK(A) | |
| R(A) | S-LOCK(A) |
| A=A-100 | |
| W(A) | |
| X-LOCK(B) | |
| UNLOCK(A) | R(A) |
| | S-LOCK(B) |
| R(B) | |
| B=B+100 | |
| W(B) | |
| UNLOCK(B) | R(B) |
| COMMIT | UNLOCK(A) |
| | UNLOCK(B) |
| | ECHO A+B |
| | COMMIT |

## Initial Database State

A=1000, B=1000

## $T_2$ Output

A+B=2000

# STRONG STRICT 2PL EXAMPLE

TIME

## Schedule

| $T_1$ | $T_2$ |
| --- | --- |
| BEGIN | BEGIN |
| X-LOCK(A) | |
| R(A) | S-LOCK(A) |
| A=A-100 | |
| W(A) | |
| X-LOCK(B) | |
| R(B) | |
| B=B+100 | |
| W(B) | |
| UNLOCK(A) | |
| UNLOCK(B) | R(A) |
| COMMIT | S-LOCK(B) |
| | R(B) |
| | ECHO A+B |
| | UNLOCK(A) |
| | UNLOCK(B) |
| | COMMIT |

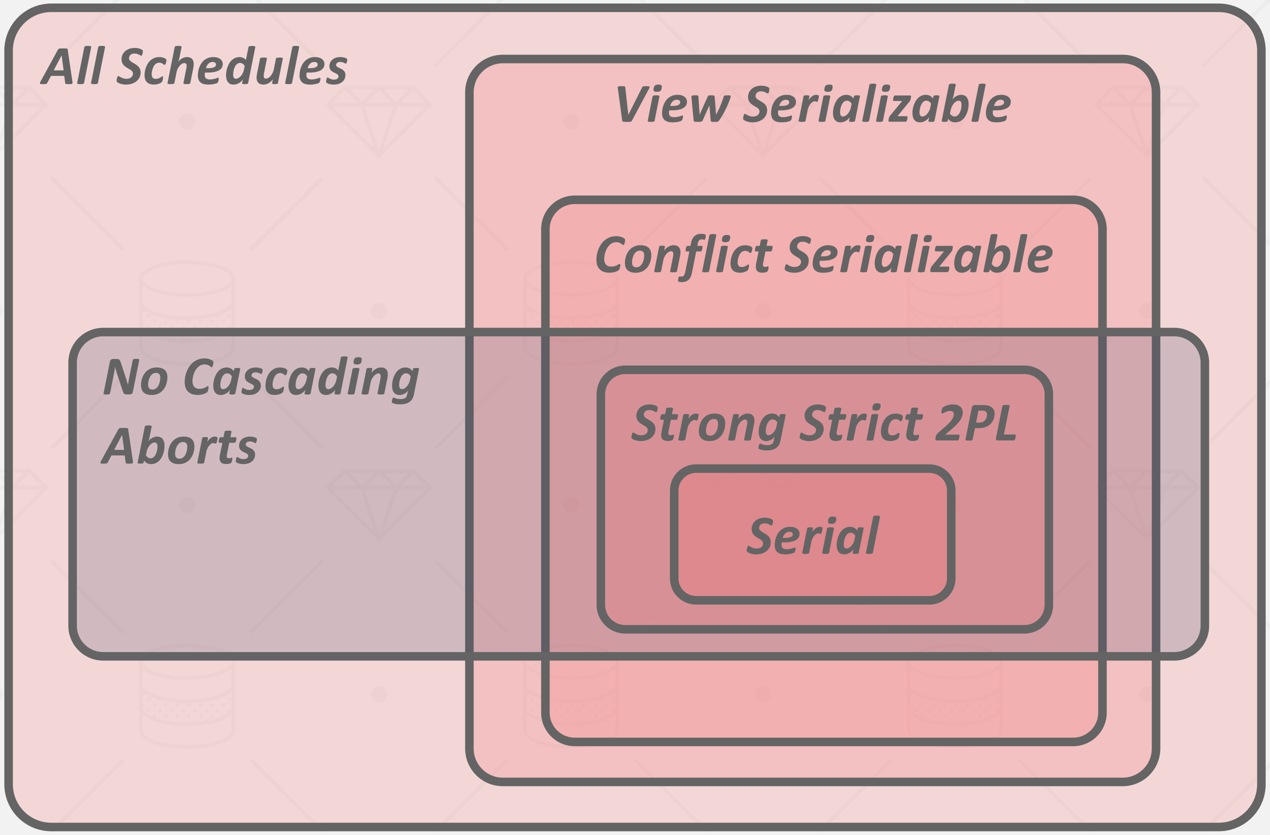
## Initial Database State

A=1000, B=1000

## $T_2$ Output

A+B=2000

# UNIVERSE OF SCHEDULES



# 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.

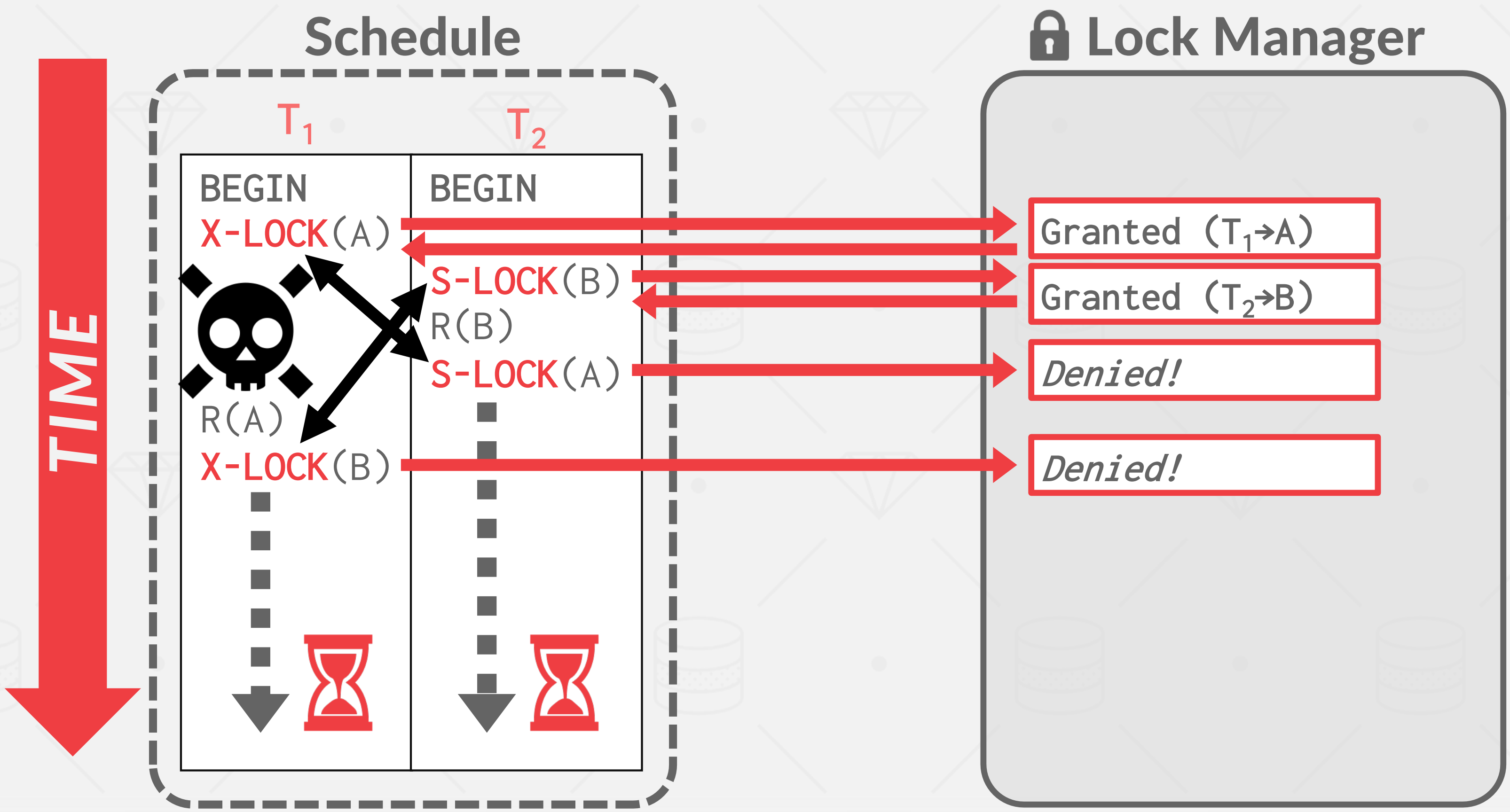
May still have "dirty reads".

→ Solution: **Strong Strict 2PL (aka Rigorous 2PL)**

May lead to deadlocks.

→ Solution: **Detection** or **Prevention**

# IT JUST GOT REAL, SON



# 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.

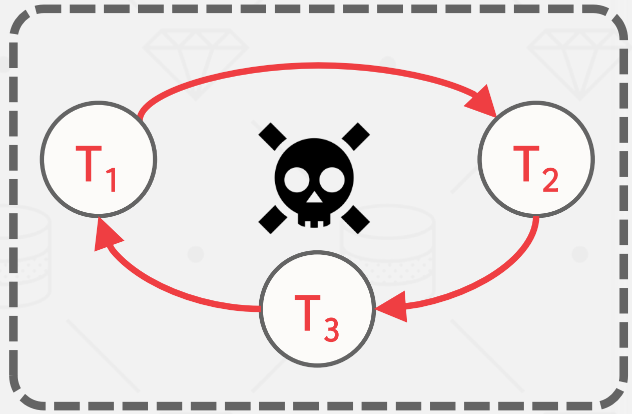
The system periodically checks for cycles in ***waits-for*** graph and then decides how to break it.

# DEADLOCK DETECTION

## Schedule

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

## *Waits-For* Graph



# 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 ***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

| $T_1$ | $T_2$ |
|---|---|
| BEGIN | |
| | BEGIN |
| | X-LOCK(A) |
| X-LOCK(A) | ⋮ |
| ⋮ | |

| Wait-Die | Wound-Wait |
|---|---|
| $T_1$ waits | $T_2$ aborts |

| $T_1$ | $T_2$ |
|---|---|
| BEGIN | |
| X-LOCK(A) | |
| ⋮ | |
| | BEGIN |
| | X-LOCK(A) |
| | ⋮ |

| Wait-Die | Wound-Wait |
|---|---|
| $T_2$ aborts | $T_2$ 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.

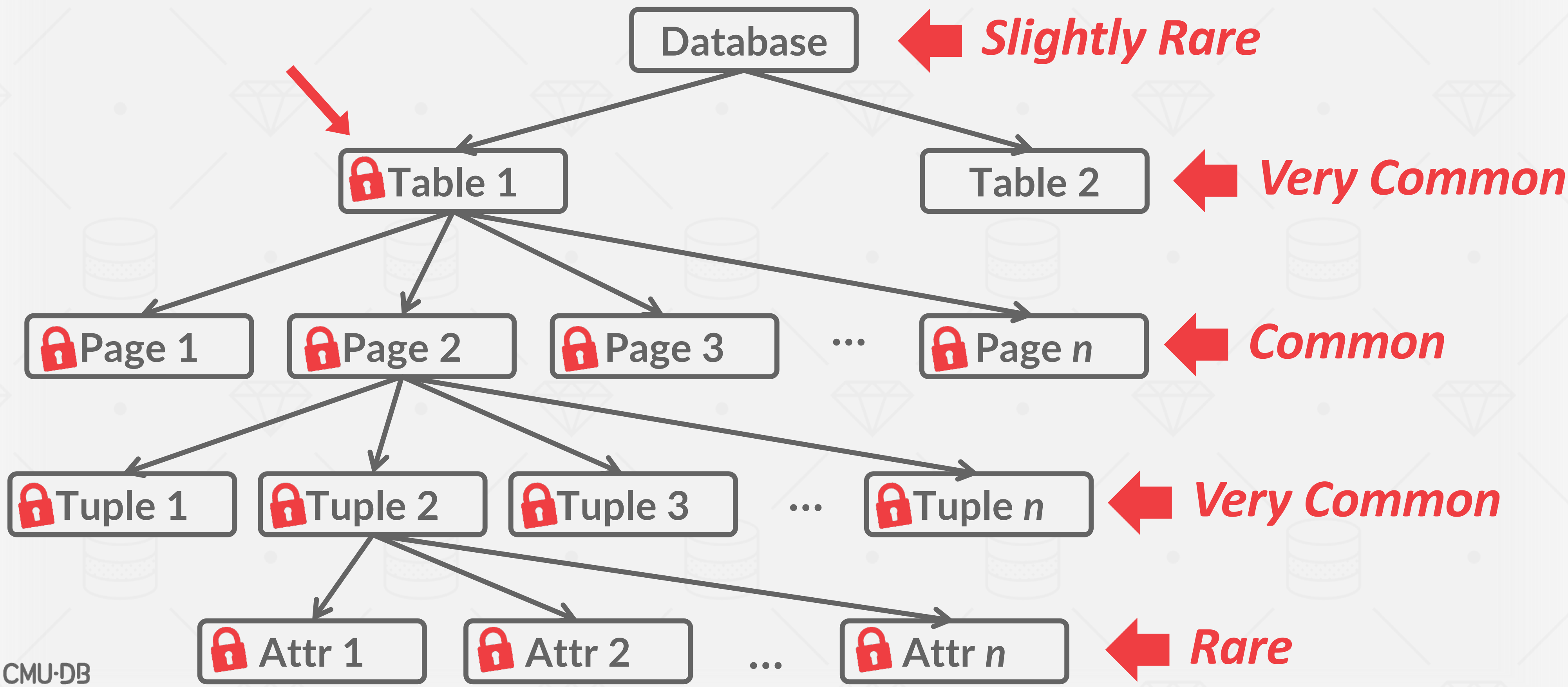
→ Attribute? Tuple? Page? Table?

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



# 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?***

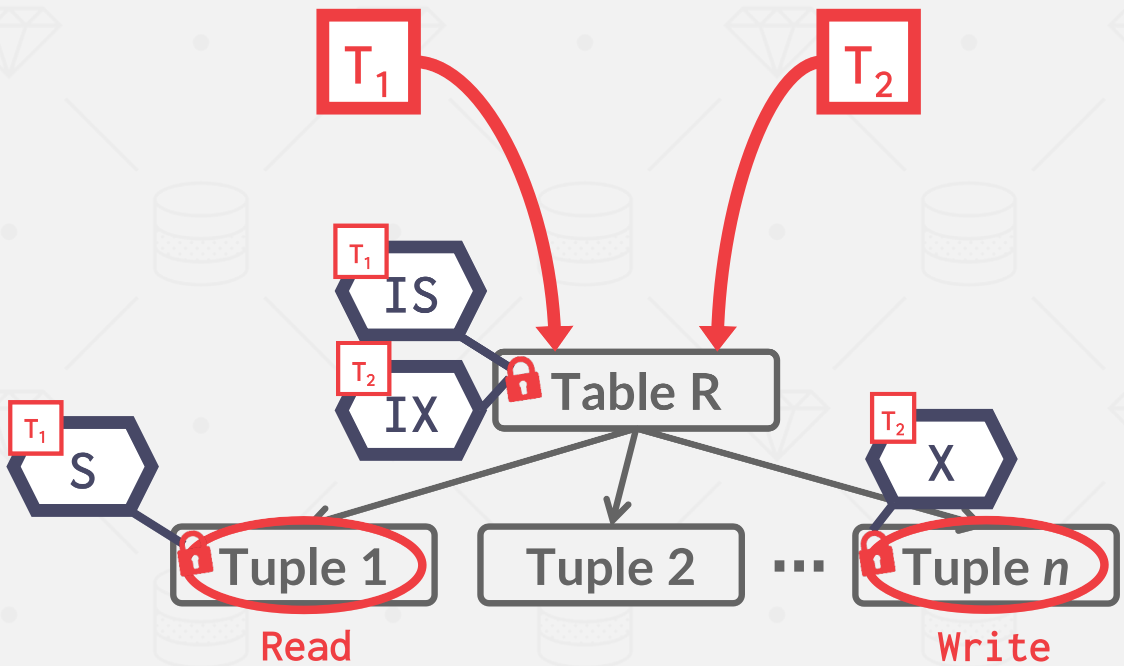
→ **Exclusive** + **Shared** for leaves of lock tree.

→ Special **Intention** locks for higher levels.

# EXAMPLE – TWO-LEVEL HIERARCHY

Read Andy's record in **R**.

Update Chi's record in **R**.



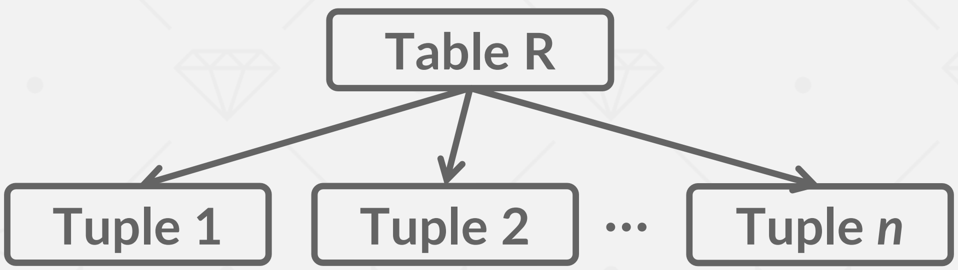
# EXAMPLE – THREE QUERIES

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.

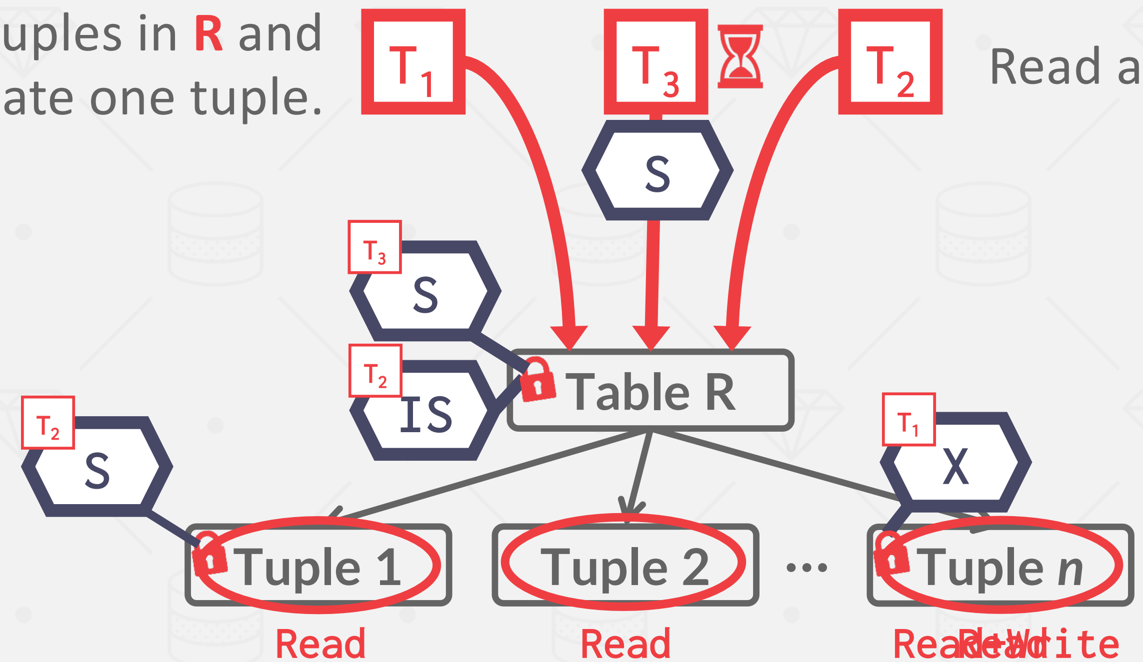


# EXAMPLE – THREE QUERIES

Scan all tuples in **R**.

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

Read a single tuple in **R**.



# COMPATIBILITY MATRIX

$T_2$ Wants / $T_1$ Holds

| $T_1$ Holds \ $T_2$ Wants | IS | IX | S | SIX | X |
|---|---|---|---|---|---|
| IS | ✓ | ✓ | ✓ | ✓ | × |
| IX | ✓ | ✓ | × | × | × |
| S | ✓ | × | ✓ | × | × |
| SIX | ✓ | × | × | × | × |
| X | × | × | × | × | × |

# 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;
```

PostgreSQL, ORACLE, IBM DB2

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

Microsoft SQL Server

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

MySQL

# 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.

***Prompt**: A realistic photo of a bath tub with wheels and cartoon eyes driving down a city street.*

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

# PROJECT #3 - LEADERBOARD

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