DBS101 Database Systems Fundamentals

Lesson 26

Learning Outcomes

1. Understand the importance of locks in concurrency control.
2. Implement locks in database transactions.
3. Analyse various concurrency control schemes.

Previously On...

When several transactions execute concurrently in the database, however, the isolation property may no longer be preserved. To ensure that it is, the system must control the interaction among the concurrent transactions; this control is achieved through one of a variety of mechanisms called concurrency-control schemes.

No one scheme is clearly the best; each one has advantages. In practice, the most frequently used schemes are two-phase locking and snapshot isolation.

Locks

A lock enables transactions to acquire read or write access to data, ensuring conflict prevention and isolation enforcement.

Why lock resources?

One way to ensure isolation is to require that **data items be accessed in a mutually exclusive manner**; that is, while one transaction is accessing a data item, no other transaction can modify that data item.

The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a lock on that item.

Locks Vs Latches

Locks

There are various modes in which a data item may be locked.We will focus on two:

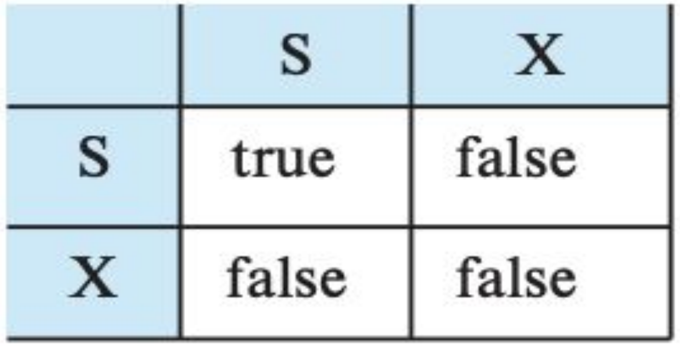
1. Shared: If a transaction Ti has obtained a shared-mode lock (denoted by S) on item Q, then Ti can read, but cannot write, Q.
2. Exclusive: If a transaction Ti has obtained an exclusive-mode lock (denoted by X) on item Q, then Ti can both read and write Q.

Locks

Every transaction must request a lock on data items from the **concurrency-control manager** based on the operations it will perform.

The transaction can only proceed after the concurrency-control manager grants the lock, allowing multiple readers but only one writer at a time.

Locks



	S	X
S	true	false
X	false	false

Figure 18.1 Lock-compatibility matrix comp.

Shared(S)

Exclusive(X)

A true value indicates the row's lock mode is compatible with the column's lock mode for the same data item. Multiple transactions can hold shared (S) locks simultaneously, but an exclusive (X) lock cannot be granted if any other lock exists.

Granting Locks

Locks are granted when no other transaction holds a conflicting lock on the same data item, and there are no earlier pending lock requests for that data item.

This approach prevents starvation, where a transaction is indefinitely blocked from acquiring a lock due to a constant stream of new, compatible lock requests being granted before it.

Two phase locking

One protocol that ensures **serializability** is the two-phase locking protocol. This protocol requires that each transaction issue lock and unlock requests in two phases:

1. Growing phase. A transaction may obtain locks, but may not release any lock.

2. Shrinking phase. A transaction may release locks, but may not obtain any new locks.

Go through Two phase locking protocol: 10 minutes

Implementation of locking

1. A lock manager process handles lock requests, grants, and unlocks from transactions.
2. The lock manager maintains a lock table with linked lists of lock records for each data item, ordered by request arrival time.
3. Lock requests are added to the end of the linked list for that data item. Locks are granted if compatible with existing locks and no earlier pending requests.
4. Unlocks remove the transaction's record from the linked list, potentially allowing later pending requests to be granted.

Implementation of locking

This implementation ensures no starvation by honoring request order and never granting a request while an earlier one is pending.

However **Deadlock detection and handling** is required.

Deadlocks

A deadlock is a cycle of transactions waiting for locks to be released by each other.

Two ways of dealing with deadlocks:

1. Deadlock Detection
2. Deadlock Prevention

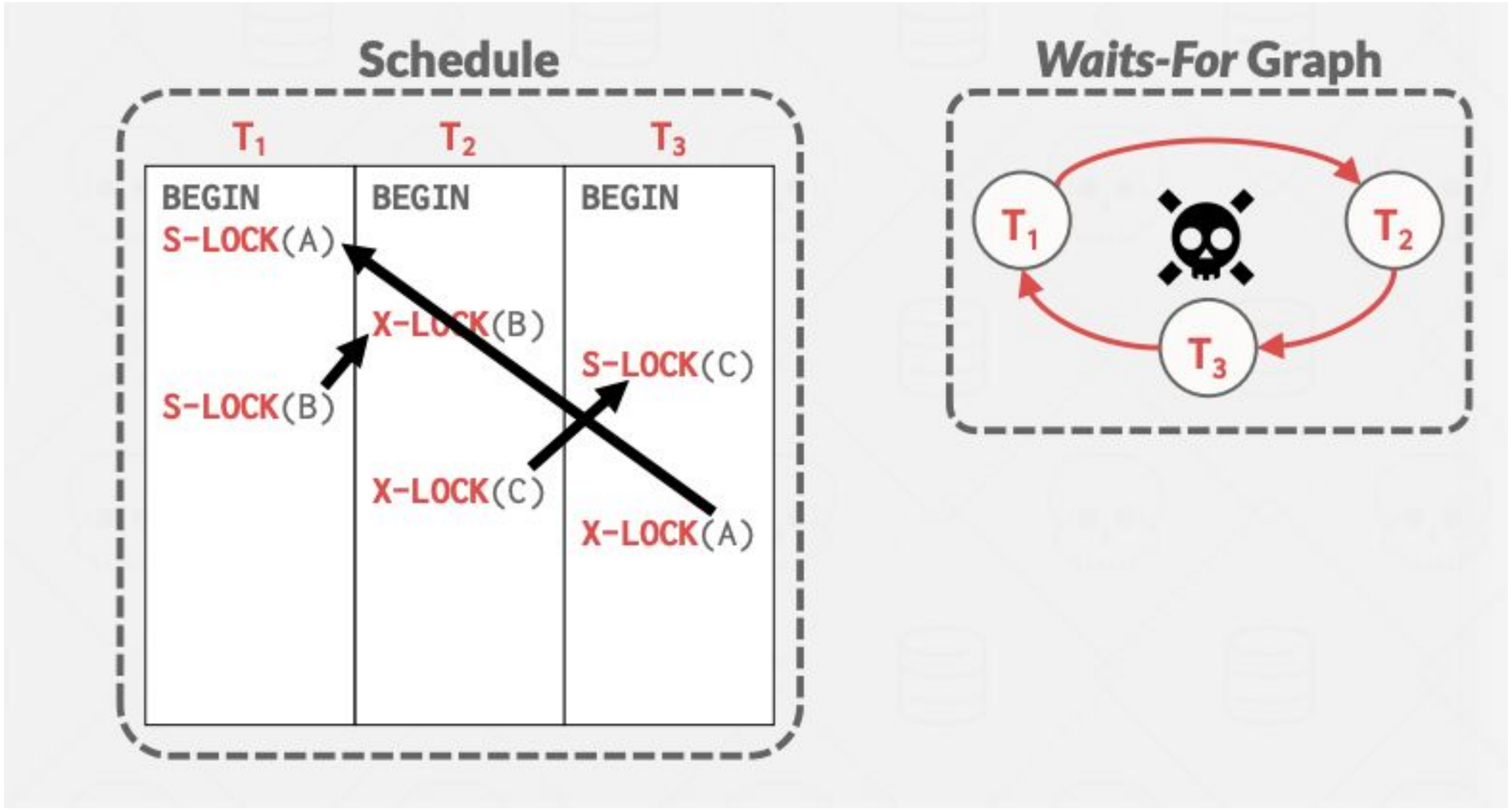
Deadlock Detection

The DBMS creates a **waits-for** graph to keep track of what locks each transaction is waiting to acquire:

- Nodes are transactions
- Edge from Ti to Tj if Ti is waiting for Tj to release a lock.

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

Deadlock Detection



Deadlock Handling

- When the DBMS detects a deadlock, it will select a “victim” txn(transaction) 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 number of items already locked
- By the number of txns that we have to rollback with it

We also should consider the number 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.

1. Completely: Rollback entire txn and tell the application it was aborted.
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 transaction tries to acquire a lock that is held by another transaction, 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., T1 > T2)

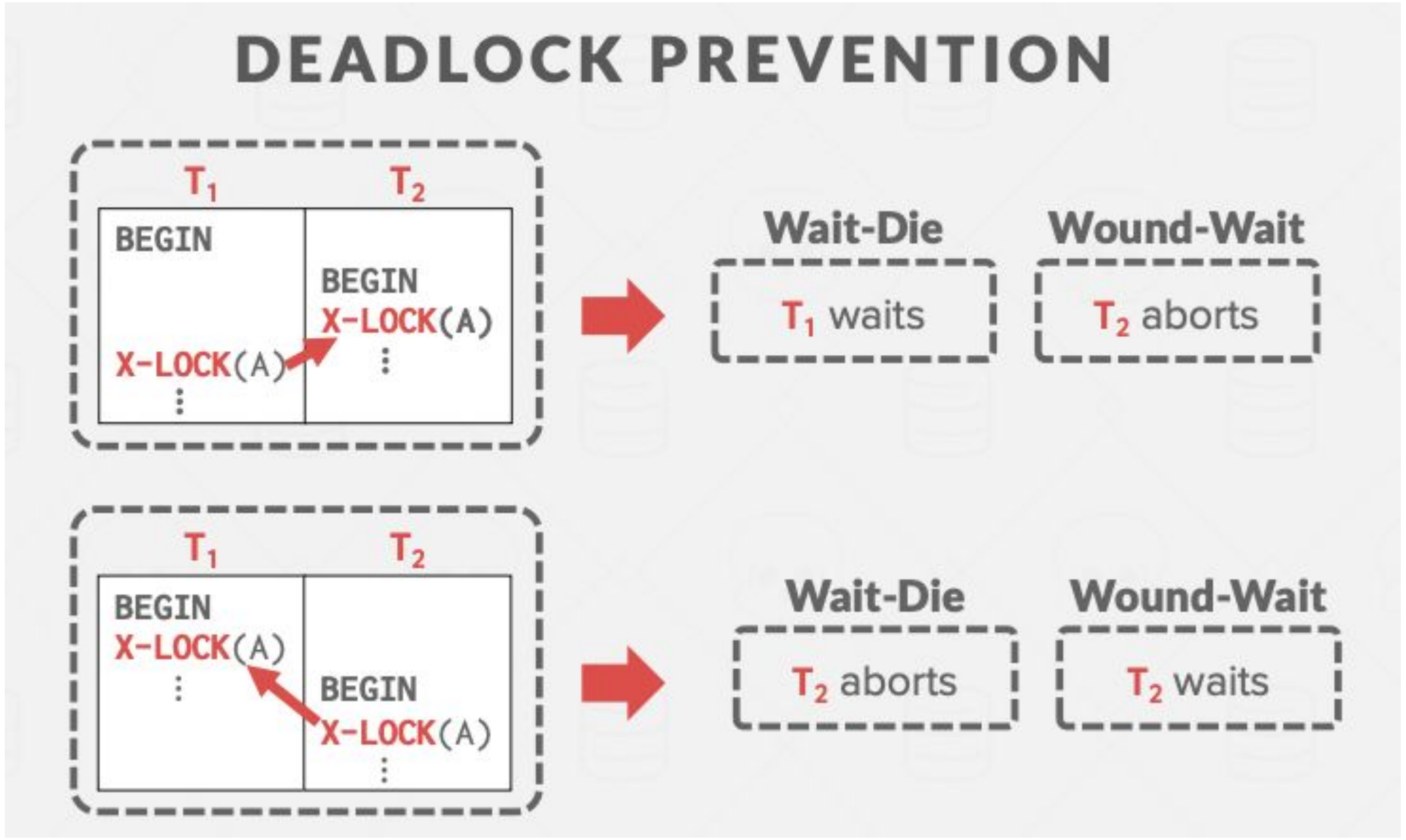
Wait-Die (“Old Waits for Young”)

- If requesting txn(Transaction) 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



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.

Deadlock Prevention

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?

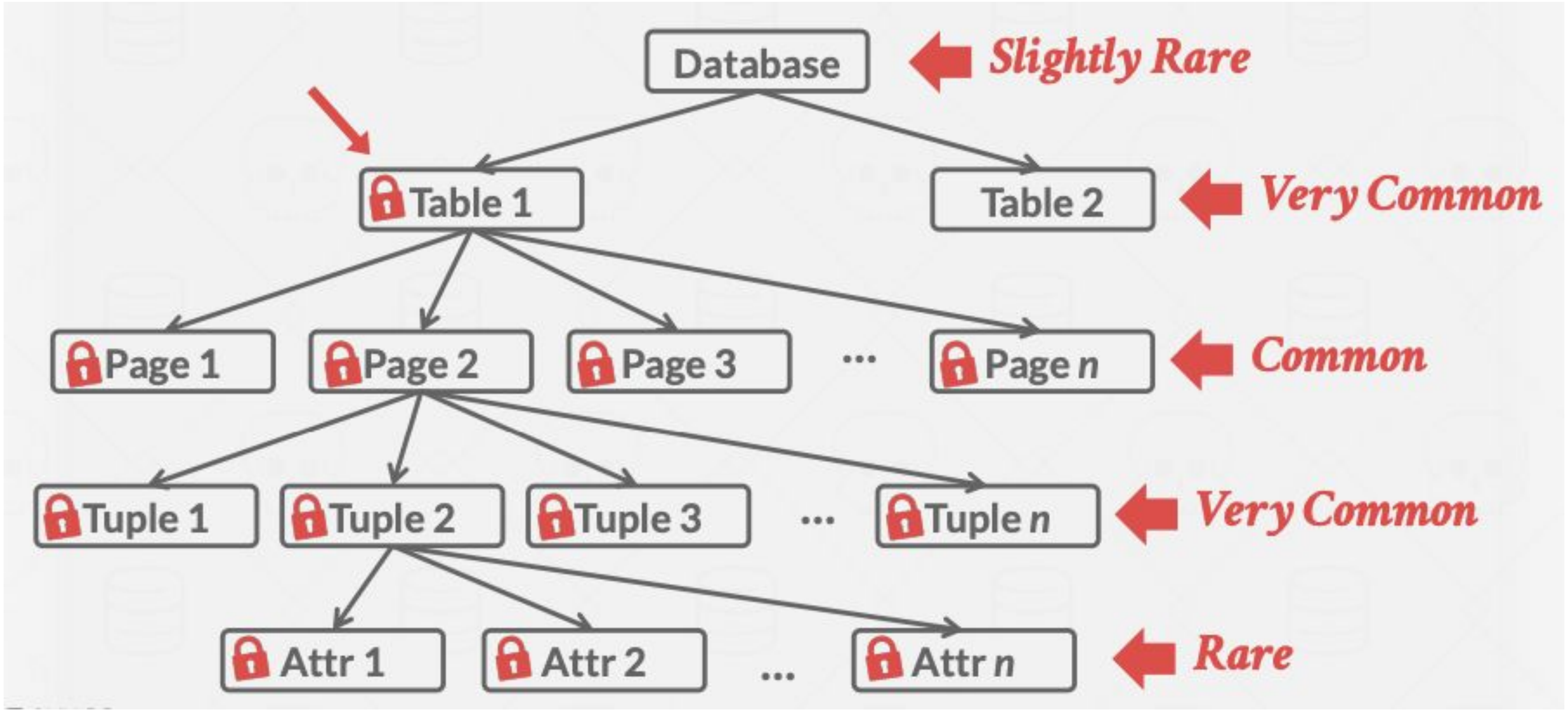
The DBMS should ideally obtain fewest number of locks that a txn needs.

Trade-off between parallelism versus overhead.

- Fewer Locks, Larger Granularity vs. More Locks, Smaller Granularity.

Note: Granularity refers to the level of detail or depth present in a dataset or database.

Database Lock Hierarchy



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.

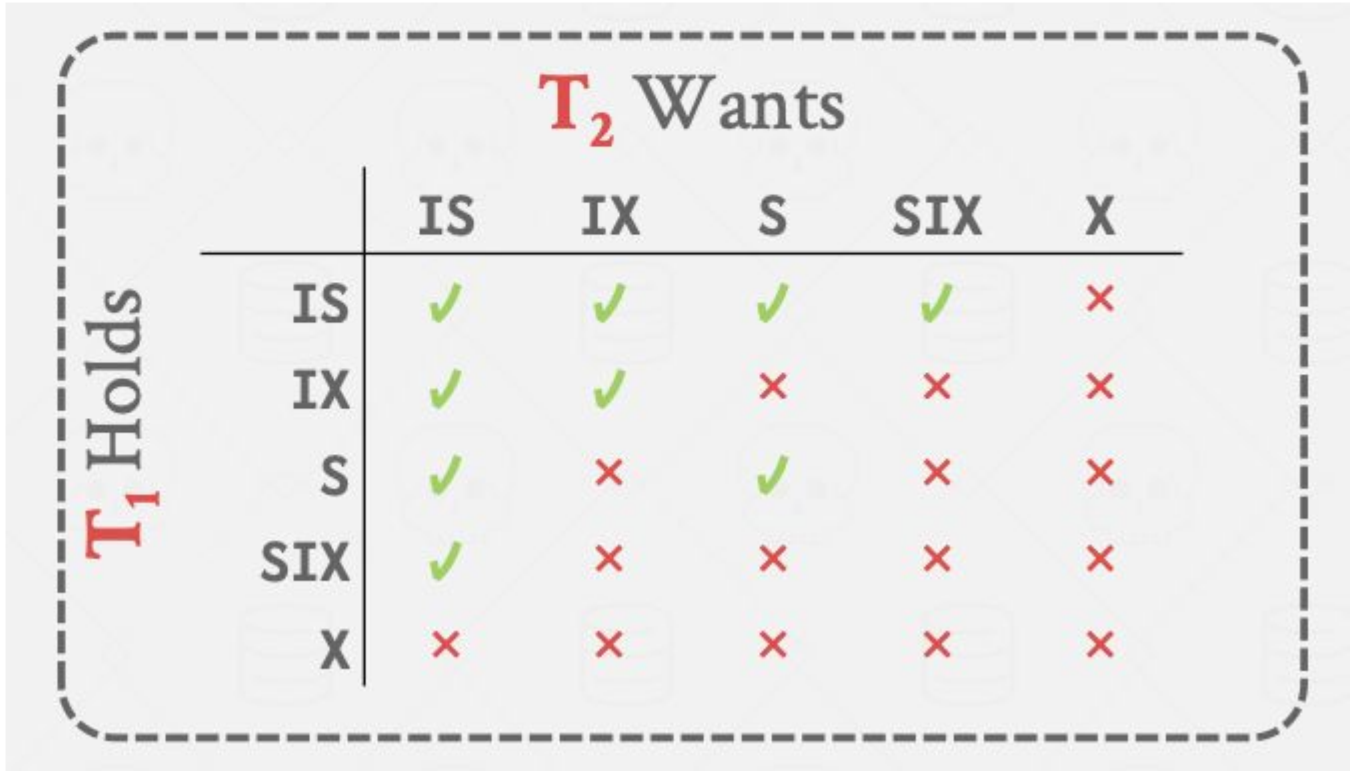
Intention-Exclusive (IX)

- 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



T₂ Wants / T₁ Holds

T₁ Holds \ T₂ Wants	IS	IX	S	SIX	X
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.

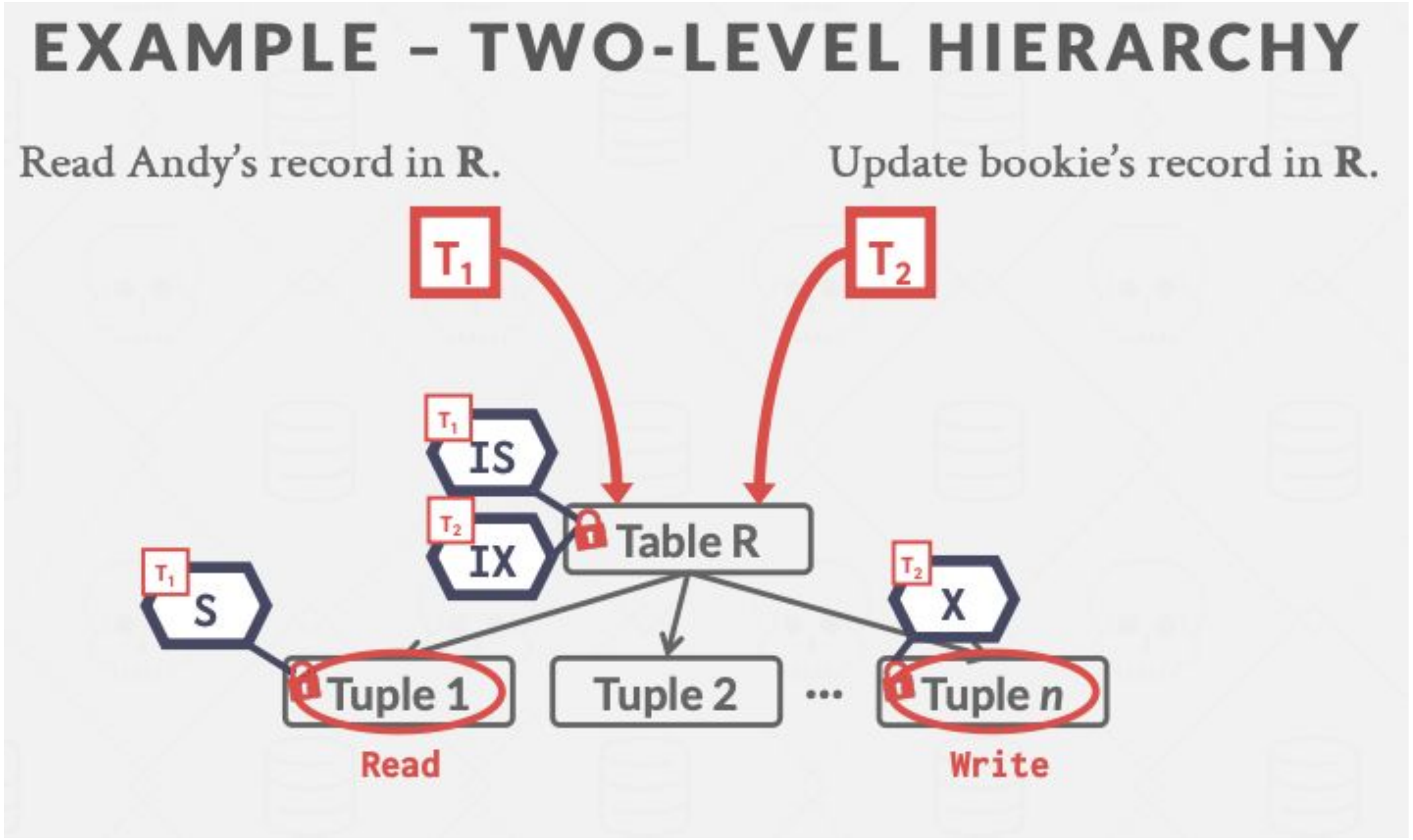
Locking Protocol

```
T1 – Get the balance of Andy’s shady off-shore bank account.
T2 – 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.
```

Locking Protocol



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.

Insert Operations in Concurrency Control

- Treated similar to write operations for concurrency control purposes.

- Under 2-phase locking, transaction gets an exclusive lock on newly inserted data item.

- Under timestamp ordering, R-timestamp and W-timestamp set to transaction timestamp.

Delete Operations

- Conflicts with read, write, delete of same item by another transaction.
- Under 2-phase locking, exclusive lock required before deletion.
- Under timestamp ordering,Delete rejected if older transaction has read/written item

Predicate Reads in Concurrency Control

- Predicate reads (e.g. SELECT based on condition) can conflict with inserts/updates that affect tuples satisfying the predicate.

- Known as the **phantom phenomenon.**

- Prevented by locking index entries/leaf nodes or using predicate locking

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.

Example

```
CREATE DATABASE lock_demo;
\c lock_demo;
CREATE TABLE example_table (
    id SERIAL PRIMARY KEY,
    data TEXT
);
```

Example

```
INSERT INTO example_table (data) VALUES ('Sample Data 1');
INSERT INTO example_table (data) VALUES ('Sample Data 2');
INSERT INTO example_table (data) VALUES ('Sample Data 3');
```

Example

- **SHARE Mode**: Allows read and schema modification operations but not data modification operations within the same transaction.
- **EXCLUSIVE Mode**: Allows full access (read, write, delete) to the table, blocking all other transactions.
- **FOR UPDATE**: Acquires an exclusive lock on the selected rows, allowing modifications on those rows.
- **FOR SHARE**: Acquires a shared lock on the selected rows, preventing other transactions from acquiring exclusive locks but allows updates within the same transaction.

Example

```
BEGIN;
LOCK TABLE example_table IN SHARE MODE;
SELECT * FROM example_table;
COMMIT;
```

Example

```
BEGIN;
LOCK TABLE example_table IN EXCLUSIVE MODE;
INSERT INTO example_table (data) VALUES ('Another New
Sample Data');
UPDATE example_table SET data = 'Updated Data' WHERE id =
1;
DELETE FROM example_table WHERE id = 2;
COMMIT;
```

Example

```
BEGIN;
SELECT * FROM example_table
WHERE data = 'Sample Data 1'
FOR UPDATE;
UPDATE example_table SET data = 'Exclusive Lock Data'
WHERE data = 'Sample Data 1';
COMMIT;
```

Example

```
BEGIN;
SELECT * FROM example_table
WHERE data = 'Sample Data 2'
FOR SHARE;
UPDATE example_table SET data = 'Shared Lock Data' WHERE
data = 'Sample Data 2';
COMMIT;
```

Concurrency Control Approaches

Pessimistic concurrency control:

- These mechanisms assume conflicts will occur and take preemptive measures to prevent them from violating serializability.
- They determine the serialization order of conflicting operations proactively while transactions are executing.
- Two-Phase Locking (2PL) and Timestamp Ordering are examples of pessimistic approaches.

Concurrency Control Approaches

Optimistic concurrency control:

- These mechanisms assume conflicts are rare and allow transactions to execute first.
- They check for serializability violations after transactions have completed their operations.
- If a violation is detected, the mechanism takes remedial actions like restarting transactions.

Concurrency Control Approaches

Two-Phase Locking (2PL) **[Pessimistic]**

- Determine serializability order of conflicting operations at runtime while txns execute.

Timestamp Ordering **[Pessimistic]**

- A serialization mechanism using timestamps.

Optimistic Concurrency Control **[Optimistic]**

- Run then check for serialization violations.

Timestamp Ordering Concurrency Control

Uses timestamps to determine the serializability order of txns.

TS- Timestamp, R- Read, W- Write

If TS(Ti) < TS(Tj), then the DBMS must ensure that the execution schedule is equivalent to the serial schedule where Ti appears before Tj.

Timestamp Ordering Concurrency Control

Imagine there are multiple transactions (let's call them T1, T2, T3, etc.) trying to read and write data items (X, Y, Z, etc.) in a database concurrently. To maintain consistency, each transaction is assigned a timestamp, and data items have read and write timestamps.

The Thomas Write Rule comes into play when a transaction Ti wants to write to a data item X, but the existing write timestamp W-TS(X) of X is smaller than the timestamp TS(Ti) of Ti.

Timestamp Ordering Concurrency Control

In this case, the rule says:

1. If the read timestamp R-TS(X) of X is also smaller than Ti's timestamp, it means another transaction has already read X before Ti. So we have to abort and restart Ti to maintain consistency.
2. However, if only the write timestamp W-TS(X) is smaller than Ti's timestamp, the rule says we can ignore the existing write to X and allow Ti to proceed with writing to X.

Timestamp Ordering Concurrency Control

This seems to violate the usual timestamp ordering, where transactions with smaller timestamps should go first. But the Thomas Write Rule makes an exception here to avoid aborting Ti unnecessarily.

3. After Ti writes to X, we update the write timestamp W-TS(X) of X to reflect Ti's new write.

Timestamp Ordering Concurrency Control

Thomas Write Rule

If TS(Ti) < R-TS(X):

- Abort and restart Ti.

If TS(Ti) < W-TS(X):

- Thomas Write Rule: Ignore the write to allow the txn to continue executing without aborting.
- This violates timestamp order of Ti.

Else: → Allow Ti to write X and update W-TS(X)

Timestamp Ordering Concurrency Control

Basic T/O:

- Generates a schedule that is conflict serializable if you do not use the Thomas Write Rule.
  - No deadlocks because no txn ever waits.
  - Possibility of starvation for long txns if short txns keep causing conflicts.

Timestamp Ordering Concurrency Control

Read Blogs 10 minutes:
https://www.tutorialspoint.com/concurrency-control-based-on-timestamp-ordering

https://www.geeksforgeeks.org/timestamp-based-concurrency-control/

Multi-Version Concurrency Control

The DBMS maintains multiple physical versions of a single logical object in the database:

- When a txn writes to an object, the DBMS creates a new version of that object.
- When a txn reads an object, it reads the newest version that existed when the txn st

Multi-Version Concurrency Control

- Protocol was first proposed in 1978 MIT PhD dissertation.
- First implementations was Rdb/VMS and InterBase at DEC in early 1980s. Both were by Jim Starkey, co-founder of NuoDB.
- DEC Rdb/VMS is now “Oracle Rdb”.
- InterBase was open-sourced as Firebird.

Multi-Version Concurrency Control

- Writers do not block readers.
- Readers do not block writers.

Read-only txns can read a consistent snapshot without acquiring locks.

- Use timestamps to determine visibility.
- Easily support time-travel queries.

Snapshot Isolation(SI)

When a txn starts, it sees a consistent snapshot of the database that existed when that the txn started.

- No torn writes from active txns.
- If two txns update the same object, then first writer wins. SI is susceptible to the Write Skew Anomaly.

Snapshot Isolation(SI)

In a write skew anomaly, two transactions (T1 and T2) concurrently read an overlapping data set (e.g. values V1 and V2), concurrently make disjoint updates (e.g. T1 updates V1, T2 updates V2), and finally concurrently commit, neither having seen the update performed by the other.

Read Aloud Session Start

Traditional locking techniques like two-phase locking aim to ensure serializability of operations by using different lock modes like shared and exclusive locks.

This means that for transactions doing read/write operations, serializable execution would give the same result as if the transactions ran one-by-one in some sequential order.

However, in certain situations, serializable execution may not be ideal;

1. Long-running transactions where concurrency can be increased by allowing higher degrees of concurrency control.
2. Operations where the semantics of the operation can be used to increase concurrency, like increment, even if not fully serializable.
3. Applications where weak levels of consistency are acceptable, like multi-player online games.

The traditional two-phase locking protocol cannot be used in such cases, and more advanced concurrency control techniques are required.

For example, consider maintaining a materialized view which records the total sales each day. Every transaction updating sales needs to increment the materialized view. Performing this in a fully serializable way is inefficient, as all transactions would be forced to update the view in some serial order. It's better to allow concurrent increments, with a mechanism to ensure the final state is as if the increments occurred serially.

Advanced Concurrency Control Techniques

Weak Levels of Consistency:

- Degree 2 consistency: No validations, only holds exclusive locks until commit
- Cursor stability: No validations, no two-phase
- Optimistic concurrency control: Execute without blocking, validate at commit

Advanced Concurrency Control Techniques

Main-Memory Databases

- With data memory-resident, index operations are very fast so coarse-grained latches on entire indexes can be preferable to fine-grained locking.
- Latch-free or lock-free data structures use atomic hardware instructions like compare-and-swap to synchronize correctly without using latches/locks.
- This avoids the overhead of latch acquisition and releases for the very fast indexing operations on in-memory data.

Advanced Concurrency Control Techniques

Concurrency Control with Operations

- Concurrency can be increased by defining new operations beyond read/write relevant to the application, such as increment or conditional increment.
- Concurrency control protocols can be designed considering the semantics of these operations rather than just treating as read/write.
- This may allow increased concurrency compared to strict serializability, though application-specific constraints must still be satisfied.

Advanced Concurrency Control Techniques

Real-Time Databases

- In real-time databases, transactions have deadlines that must be given priority in concurrency control decisions.
- The concurrency control manager may decide to abort/restart or wait for a lock based on the deadlines of the involved transactions.
- Long running lower priority transactions may be preempted by aborting them to allow higher priority transactions to meet their deadline.
- Optimistic concurrency control protocols tend to reduce the number of missed deadlines compared to locking protocols.

Read Aloud Session Over

Unit XII : Recovery System

12.1 Failure Classification
12.2 Storage
12.3 Recovery and Atomicity
12.4 Recovery Algorithm
12.5 Buffer Management
12.6 Failure with Loss of Non-Volatile Storage
12.7 High Availability Using Remote Backup Systems
12.8 Early Lock Release and Logical Undo Operations
12.9 ARIES
12.10 Recovery in Main-Memory Databases