COMP 421: Files & Databases

Lecture 18: Optimistic Concurrency Control

UNC
DEPARTMENT OF
COMPUTER SCIENCE

Last Class

We discussed concurrency control protocols for generating conflict serializable schedules without needing to know what queries a txn will execute.

The two-phase locking (2PL) protocol requires txns to acquire locks on database objects before they are allowed to access them.

Observation

If you assume that conflicts between txns are **rare** and that most txns are **short-lived**, then forcing txns to acquire locks adds unnecessary overhead.

A better concurrency control protocol could be one that is optimized for the no-conflict case…

Timestamp Ordering Concurrency Control

Use timestamps to determine the serializability order of txns.

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

Each database object (e.g., tuple) will include additional fields to keep track of timestamp(s) of the txns that last accessed/modified them.

Timestamp Allocation

Each txn T_i is assigned a unique fixed timestamp that is monotonically increasing.

→ Let $TS(T_i)$ be the timestamp allocated to txn T_i.
→ Different schemes assign timestamps at different times during the txn.

Multiple implementation strategies:

→ System/Wall Clock.
→ Logical Counter.
→ Hybrid.

Today's Agenda

Optimistic Concurrency Control

Phantom Reads

Isolation Levels

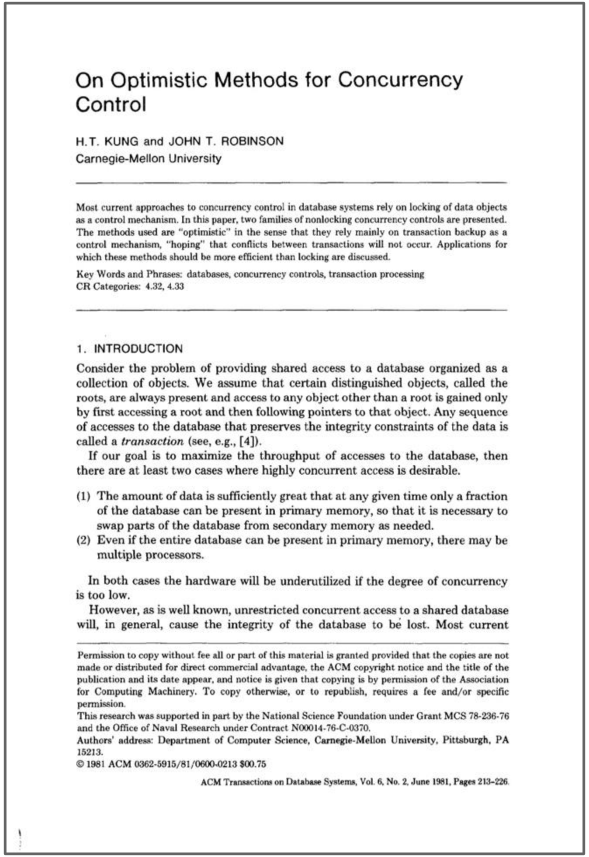
Optimistic Concurrency Control (OCC)

T/O protocol where DBMS creates a private workspace for each txn.

→ Any object read is copied into workspace.
→ Modifications are applied to workspace.

When a txn commits, the DBMS compares workspace write set to see whether it conflicts with other txns.

If there are no conflicts, the write set is installed into the "global" database.



On Optimistic Methods for Concurrency Control

H.T. KUNG and JOHN T. ROBINSON
Carnegie-Mellon University

Most current approaches to concurrency control in database systems rely on locking of data objects as a control mechanism. In this paper, two families of nonlocking concurrency controls are presented. The methods used are "optimistic" in the sense that they rely mainly on transaction backup as a control mechanism, "hoping" that conflicts between transactions will not occur. Applications for which these methods should be more efficient than locking are discussed.

Key Words and Phrases: databases, concurrency controls, transaction processing
CR Categories: 4.32, 4.33

1. INTRODUCTION

Consider the problem of providing shared access to a database organized as a collection of objects. We assume that certain distinguished objects, called the roots, are always present and access to any object other than a root is gained only by first accessing a root and then following pointers to that object. Any sequence of accesses to the database that preserves the integrity constraints of the data is called a *transaction* (see, e.g., [4]).

If our goal is to maximize the throughput of accesses to the database, then there are at least two cases where highly concurrent access is desirable.

(1) The amount of data is sufficiently great that at any given time only a fraction of the database can be present in primary memory, so that it is necessary to swap parts of the database from secondary memory as needed.
(2) Even if the entire database can be present in primary memory, there may be multiple processors.

In both cases the hardware will be underutilized if the degree of concurrency is too low.

However, as is well known, unrestricted concurrent access to a shared database will, in general, cause the integrity of the database to be lost. Most current

This research was supported in part by the National Science Foundation under Grant MCS 78-236-76 and the Office of Naval Research under Contract N00014-76-C-0370.
Authors' address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

OCC Phases

Phase #1 – Read

→ Track the read/write sets of txns and store their writes in a private workspace.
→ DBMS copies every tuple that the txn accesses from the shared database to its workspace ensure repeatable reads.

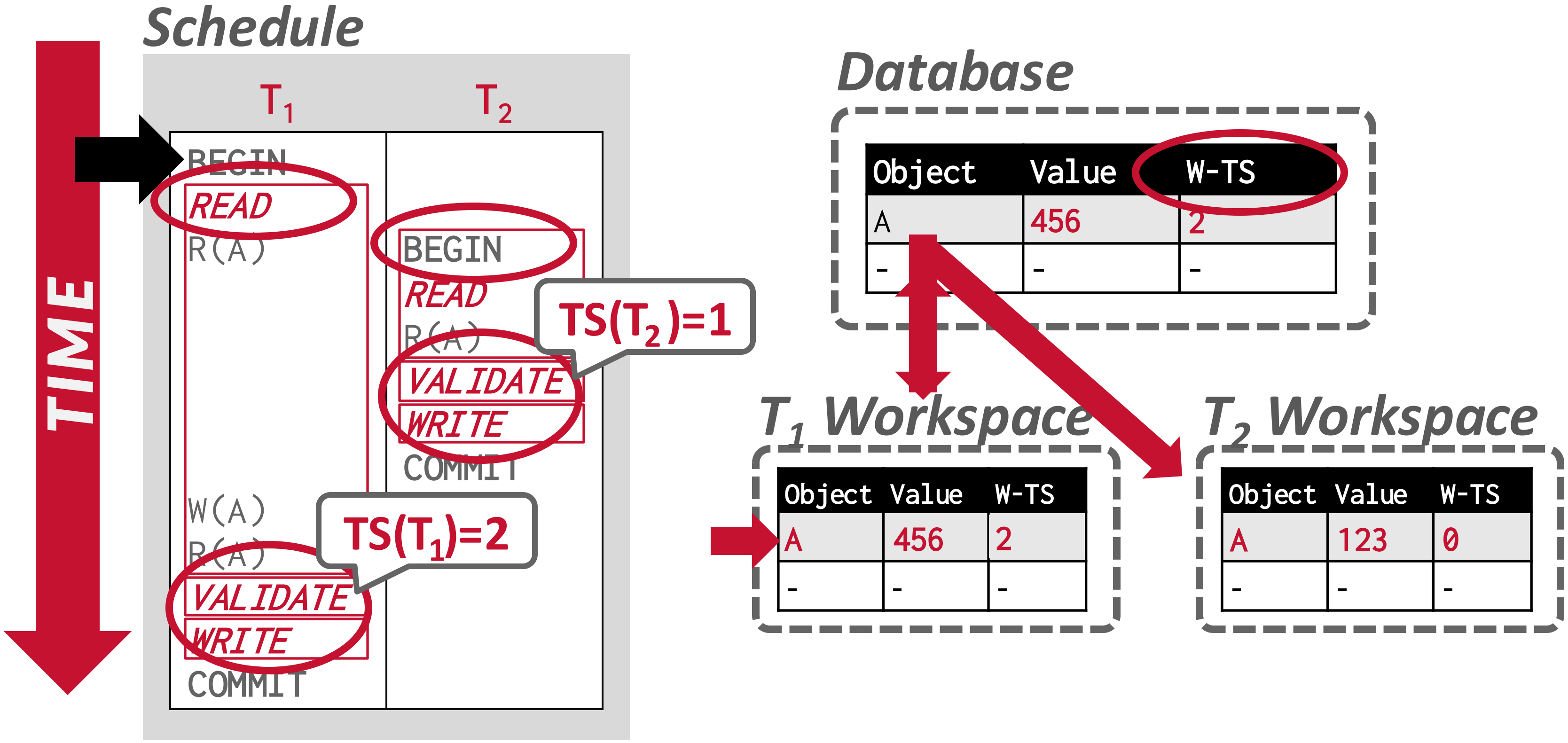
Phase #2 – Validation

→ Assign the txn a unique timestamp (TS) and then check whether it conflicts with other txns.

Phase #3 – Write

→ If validation succeeds, set the write timestamp (W-TS) to all modified objects in private workspace and install them atomically into the global database.
→ Otherwise abort txn.

OCC Example



Schedule
TIME
T1
T2
BEGIN
READ
R(A)
BEGIN
READ
R(A)
TS(T2)=1
VALIDATE
WRITE
COMMIT
W(A)
R(A)
TS(T1)=2
VALIDATE
WRITE
COMMIT
Database
Object Value W-TS
A 456 2
- - -
T1 Workspace
Object Value W-TS
A 456 2
- - -
T2 Workspace
Object Value W-TS
A 123 0
- - -

OCC: Read Phase

Track the read/write sets of txns and store their writes in a private workspace.

The DBMS copies every tuple that the txn accesses from the shared database to its workspace ensure repeatable reads.

→ We can ignore for now what happens if a txn reads/writes tuples via indexes.

OCC: Validation Phase

When txn T_i invokes COMMIT, the DBMS checks if it conflicts with other txns.

→ Original OCC algorithm uses serial validation.

→ Parallel validation requires each txn check read/write sets of other txns trying to validate at the same time.

DBMS needs to guarantee only serializable schedules are permitted.

→ **Approach #1: Backward Validation**

→ **Approach #2: Forward Validation**

OCC: Validation Phase

Forward Validation: Check whether the committing txn intersects its read/write sets with any active txns that have **not** yet committed.

Backward Validation: Check whether the committing txn intersects its read/write sets with those of any txns that have **already** committed.

OCC: Forward Validation

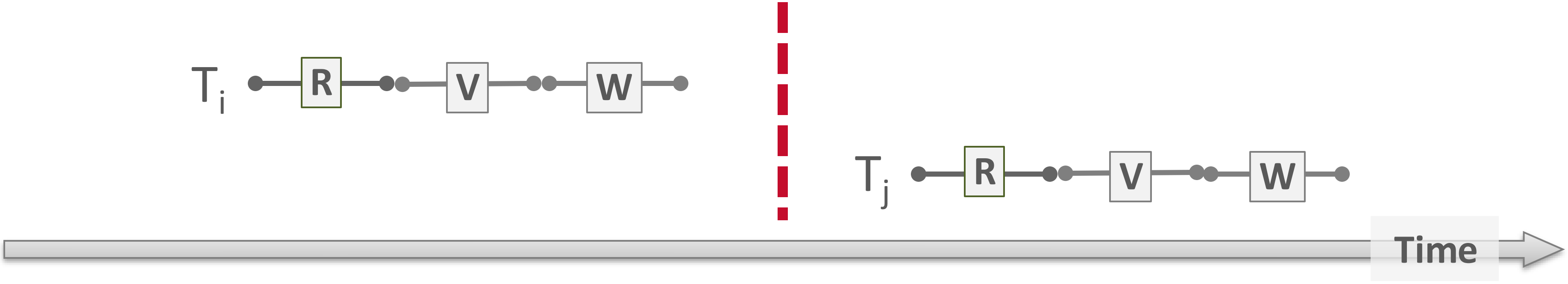
Each txn's timestamp is assigned at the beginning of the validation phase.

Check the timestamp ordering of the committing txn with all other active txns.

If $TS(T_1) < TS(T_2)$, then one of the following three conditions must hold…

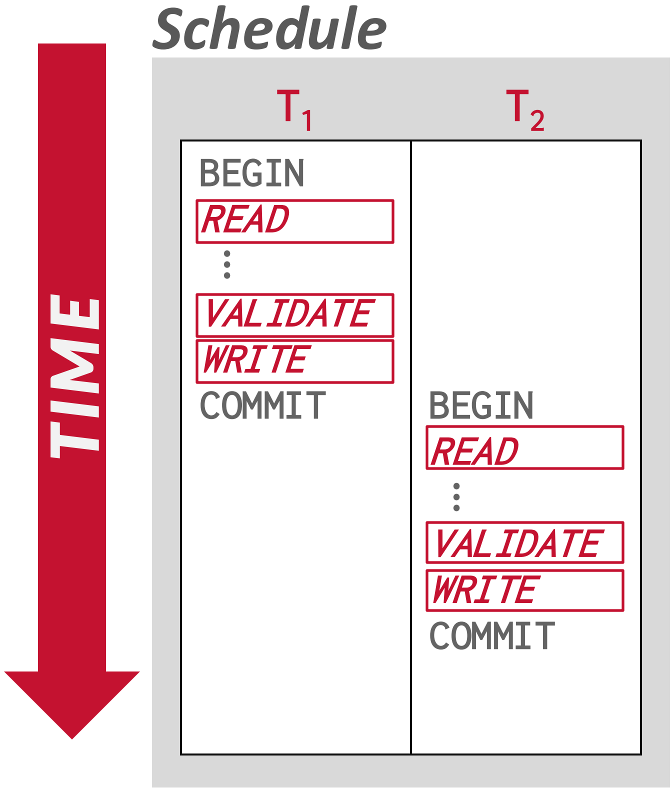
OCC: Validation ($T_i < T_j$) Case #1

Need: T_i completes its write phase before T_j starts its read phase.



No conflict as all of T_i's actions happen before T_j's.

OCC: Forward Validation Case #1



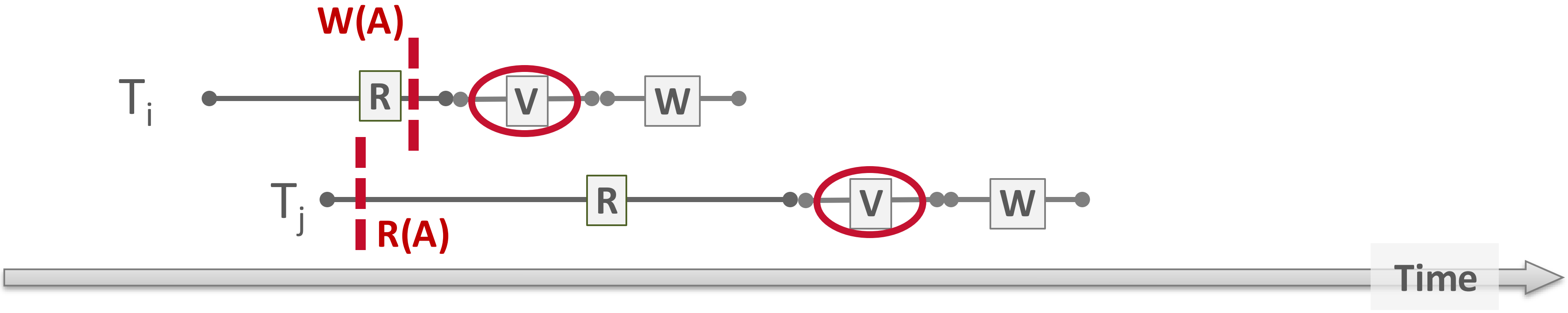
If (T_1 < T_2), check if T_1 completes its **Write** phase before T_2 begins its **Read** phase.

No conflict as all T_1's actions happen before T_2's.

→ This just means that there is serial ordering.

OCC: VALIDATION (T_i < T_j) Case #2

Need: T_i completes its write phase before T_j starts its write phase.



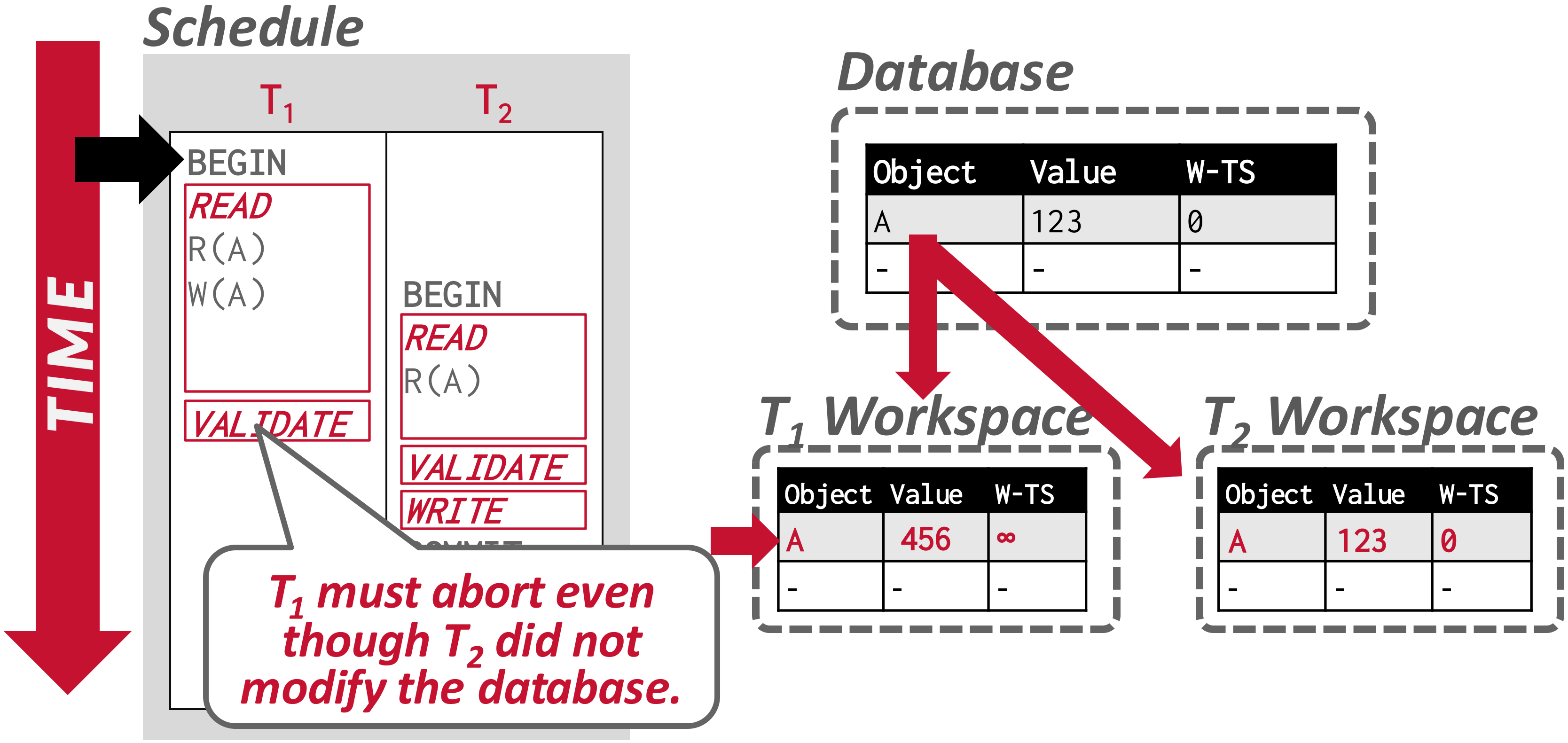
AND Check that the write set of T_i does not intersect the read set of T_j, namely: $WriteSet(T_i) \cap ReadSet(T_j) = \emptyset$

OCC: Forward Validation Case #2

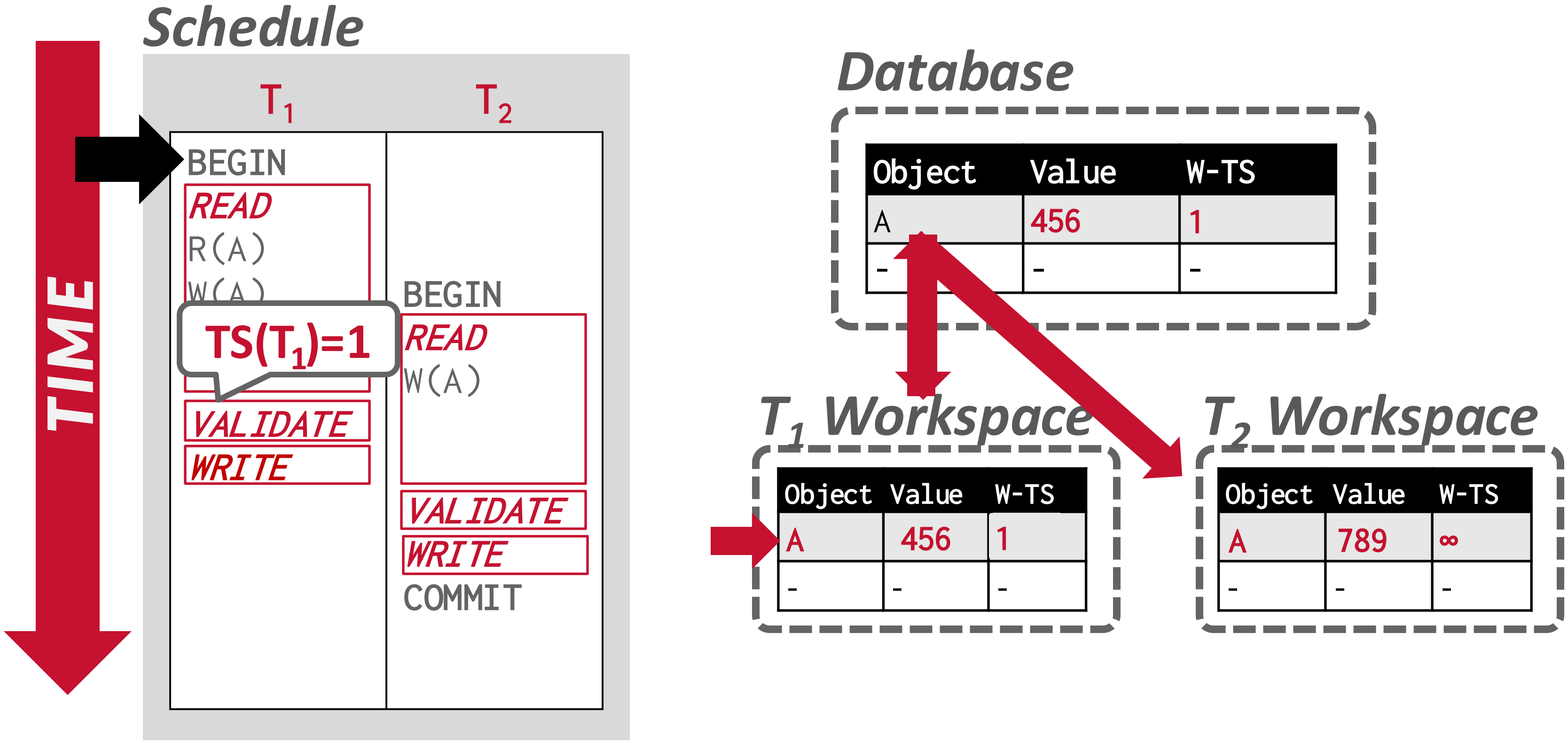
If ($T_1 < T_2$), check if T_1 completes its **Write** phase before T_2 starts its **Write** phase and T_1 does not modify to any object read by T_2.

→ `WriteSet(T1)` ∩ `ReadSet(T2)` = Ø

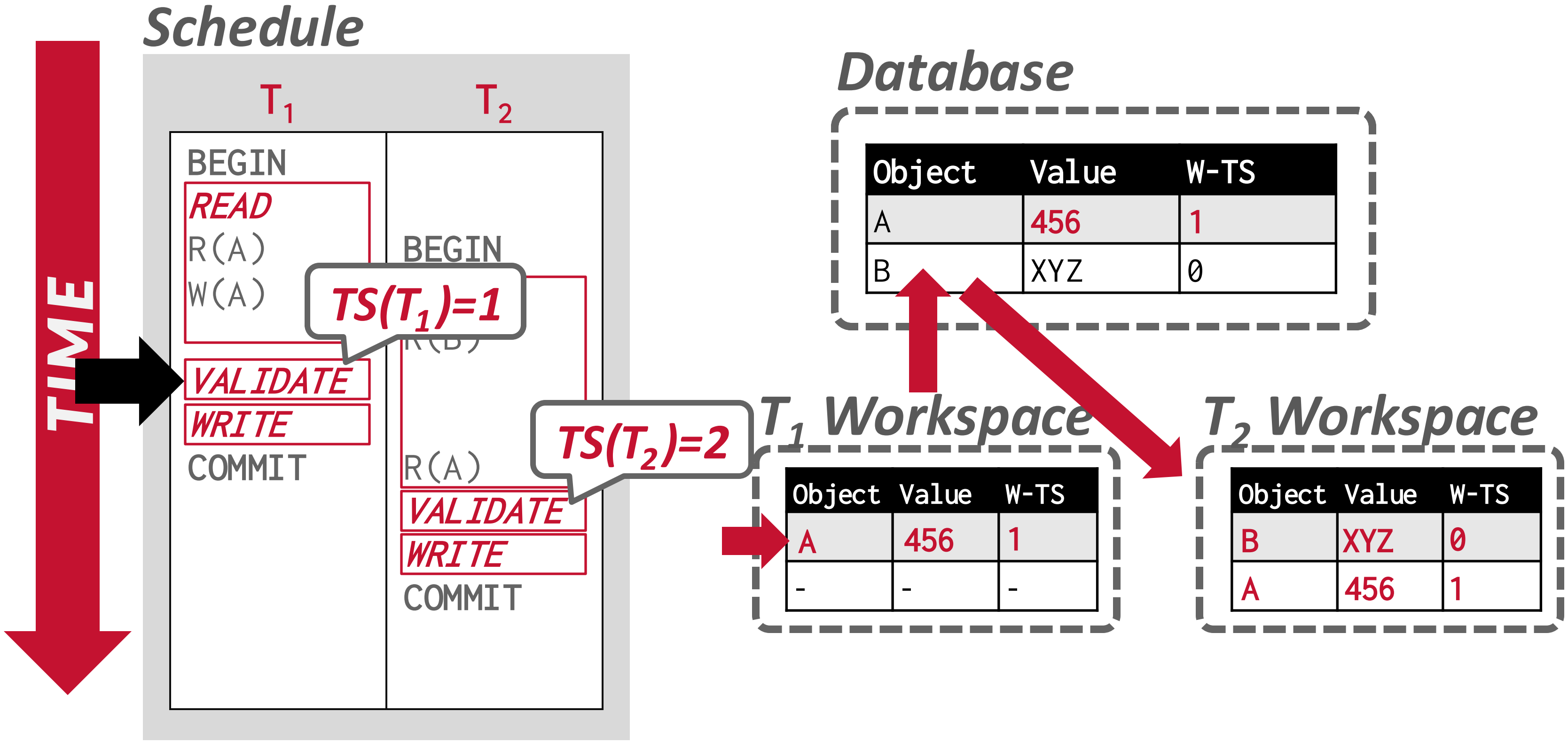
OCC: Forward Validation Case #2



OCC: Forward Validation Case #2



OCC: Forward Validation Case #2



OCC: Forward Validation Case #2

If ($T_1 < T_2$), check if T_1 completes its **Write** phase before T_2 starts its **Write** phase and T_1 does not modify to any object read by T_2.

$\rightarrow$ `WriteSet(`T_1`)` $\cap$ `ReadSet(`T_2`)` $= \emptyset$

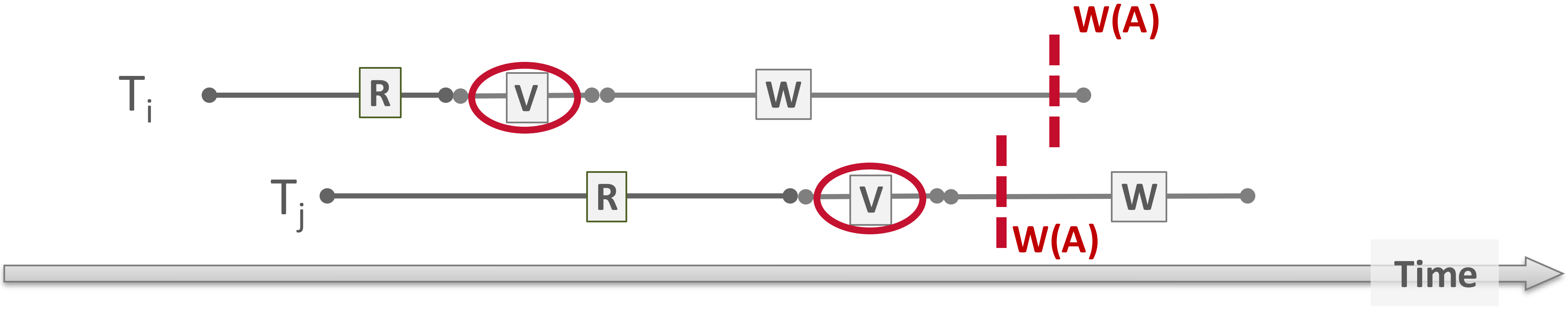
Previous examples had to know write phase ordering at validation

Can enforce by making **Validation+Write** atomic (using locks)

Is this necessary?

OCC: VALIDATION ($T_i < T_j$) Case #3

Need: T_i completes its read phase before T_j completes its read phase.



AND Check that the write set of T_i does not intersect the read set of T_j, namely: `WriteSet(Ti)` ∩ `ReadSet(Tj)` = Ø

AND Check that the write set of T_i does not intersect the write set of T_j, namely: `WriteSet(Ti)` ∩ `WriteSet(Tj)` = Ø

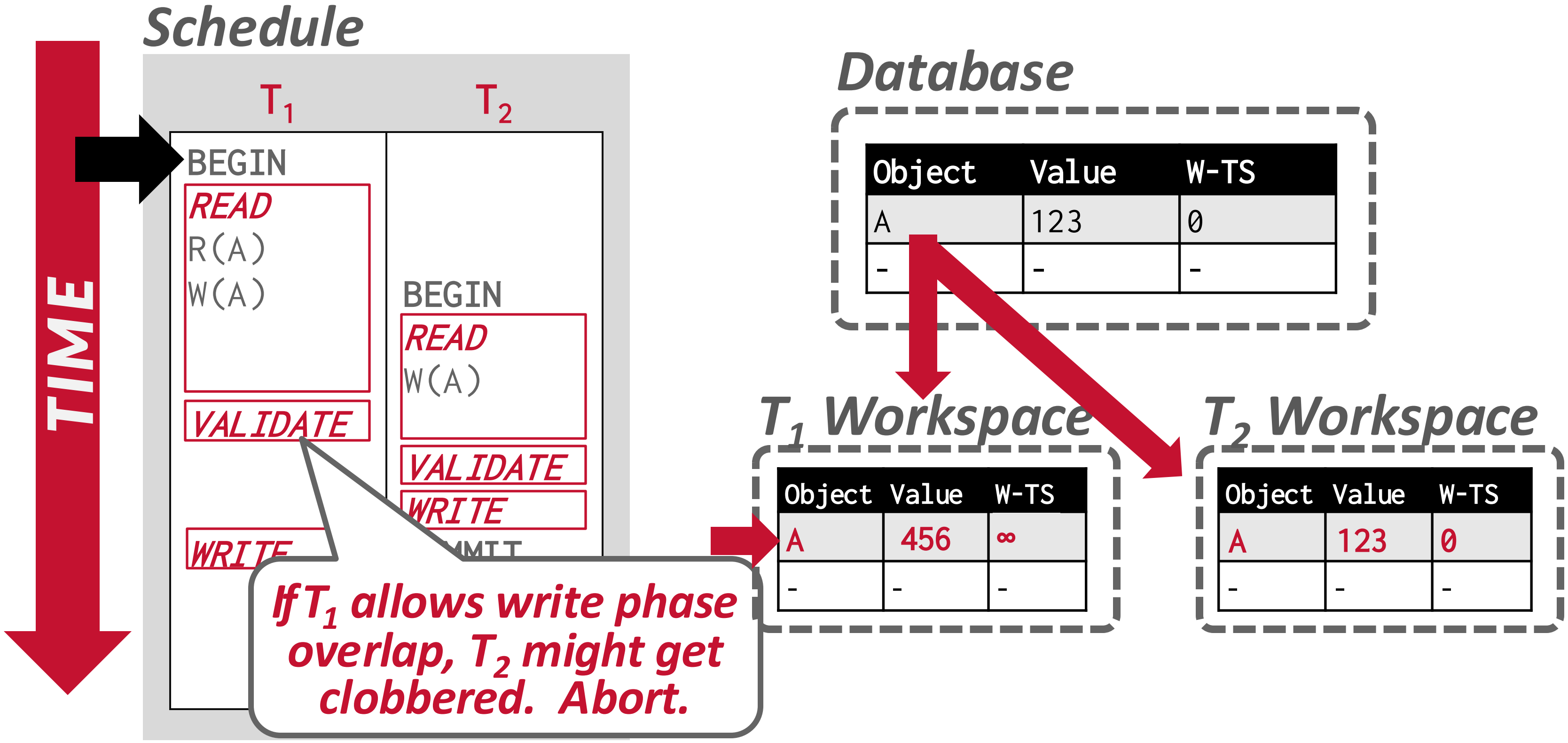
OCC: Forward Validation Case #3

If ($T_1 < T_2$), check if T_1 completes its **Read** phase before T_2 completes its **Read** phase and T_1 does not modify any object either read or written by T_2:

→ WriteSet(T_1) ∩ ReadSet(T_2) = Ø

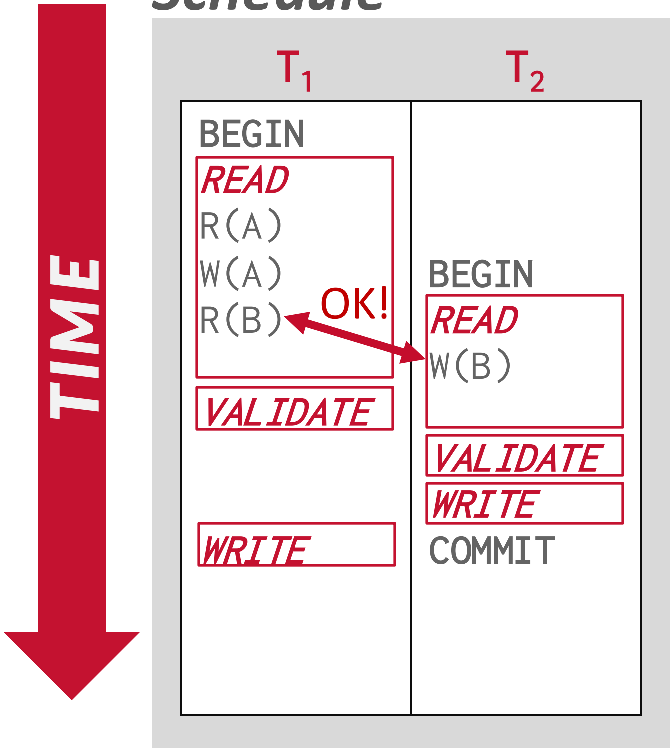
→ WriteSet(T_1) ∩ WriteSet(T_2) = Ø

OCC: Forward Validation Case #3



OCC: Forward Validation Case #3

Schedule



Database

Object	Value	W-TS
A	123	0
-	-	-

T_1 Workspace

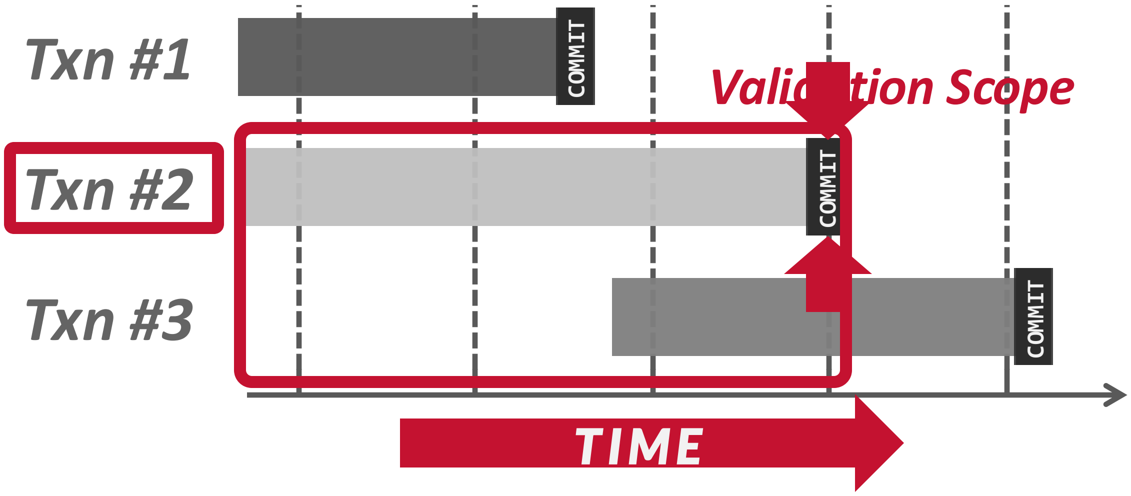
Object	Value	W-TS
-	-	-
-	-	-

T_2 Workspace

Object	Value	W-TS
-	-	-
-	-	-

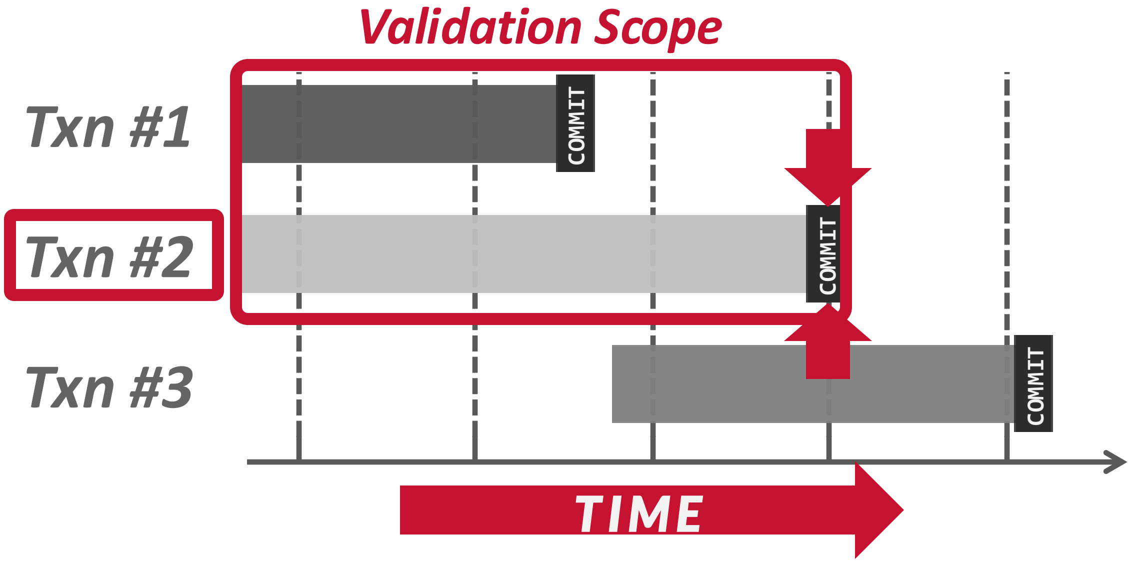
OCC: Forward Validation

Check whether the committing txn intersects its read/write sets with any active txns that have **not** yet committed.



OCC: Backward Validation

Check whether the committing txn intersects its read/write sets with those of any txns that have **already** committed.



OCC: Write Phase

Propagate changes in the txn's write set to database to make them visible to other txns.

Serial Commits:

→ Use a global latch to limit a single txn to be in the **Validation/Write** phases at a time.

Parallel Commits:

→ Use fine-grained write latches to support parallel **Validation/Write** phases.

→ Txns acquire latches in a sequential key order to avoid deadlocks.

OCC: Observations

OCC works well when the # of conflicts is low:

→ All txns are read-only (ideal).
→ Txns access disjoint subsets of data.

But OCC has its own problems:

→ High overhead for copying data locally.
→ **Validation/Write** phase bottlenecks.
→ Aborts are more wasteful than in 2PL because they only occur <u>after</u> a txn has already executed.

Observation

We have only dealt with transactions that read and update existing objects in the database.

But now if txns perform insertions, updates, and deletions, we have new problems…

The Phantom Problem

Schedule

TIME

T_1	T_2
BEGIN	BEGIN
`SELECT COUNT(*) AS cnt FROM people WHERE status='lit'`	
cnt=99	
	`INSERT INTO people VALUES (101,'Andy','lit');`
	COMMIT
`SELECT COUNT(*) AS cnt FROM people WHERE status='lit'`	
cnt=100	
COMMIT	

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

Oops?

How did this happen?

→ Because T_1 locked only existing records and not ones that other txns are adding to the database!

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

This is known as a **phantom read**.

→ A txn scans a range more than once and another txn inserts/removes tuples that fall within that range in between the scans.

Solutions To The Phantom Problem

Approach #1: Lock Everything! ⬅ ***Less Common***

→ Entire table or every page.

Approach #2: Re-Execute Scans ⬅ ***Rare***

→ Run queries again at commit to see whether they produce a different result to identify missed changes.

Approach #3: Predicate Locking ⬅ ***Very Rare***

→ Logically determine the overlap of predicates before queries start running.

Approach #4: Index Locking ⬅ ***Common***

→ Use keys in indexes to protect ranges.

Re-execute Scans

The DBMS tracks the WHERE clause for all queries that the txn executes.

→ Retain the scan set for every range query in a txn.

Upon commit, re-execute just the scan portion of each query and check whether it generates the same result.

→ Example: Run the scan for an UPDATE query but do not modify matching tuples.

→ If changed, abort.

Predicate Locking

Proposed locking scheme from System R.

→ Shared lock on the predicate in a WHERE clause of a SELECT query.

→ Exclusive lock on the predicate in a WHERE clause of any UPDATE, INSERT, or DELETE query.

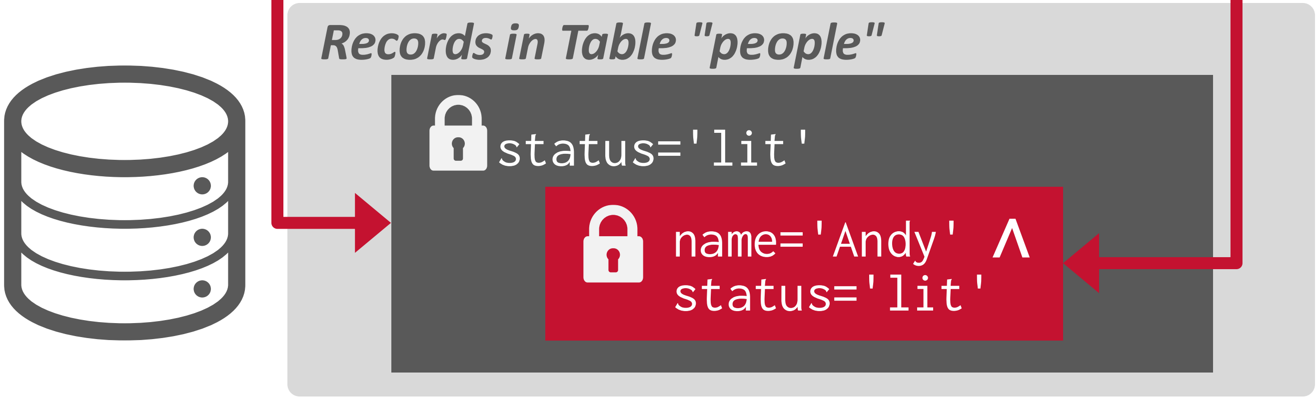
This is difficult to implement efficiently. Some systems approximate it via precision locking.

HyPer DuckDB UMBRA CedarDB

Predicate Locking

```
SELECT COUNT(*) AS cnt
  FROM people
 WHERE status='lit'
```

```
INSERT INTO people VALUES
 (101, 'Andy', 'lit')
```



Index Locking Schemes

Key-Value Locks

Gap Locks

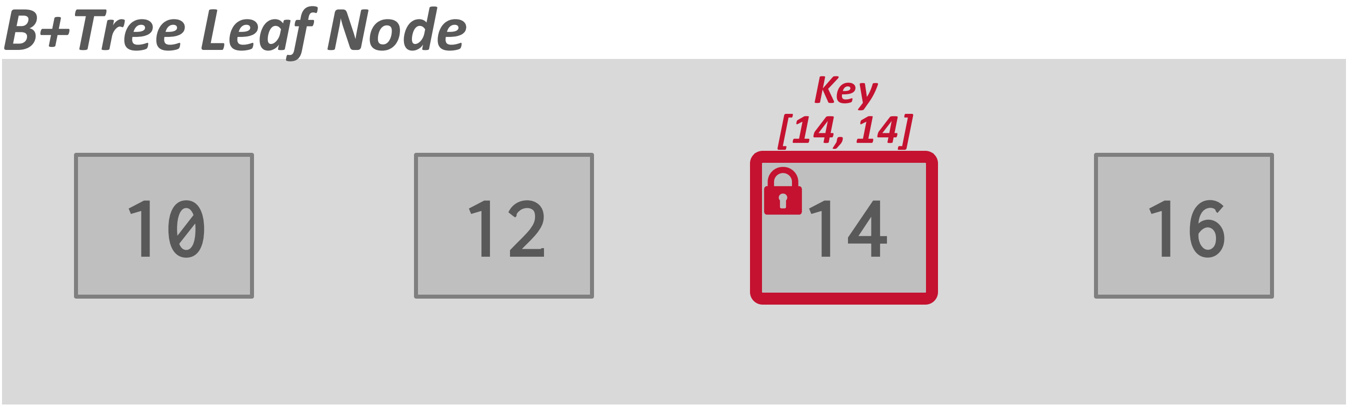
Key-Range Locks

Hierarchical Locking

Key-value Locks

Locks that cover a single key-value in an index.

Need "virtual keys" for non-existent values.

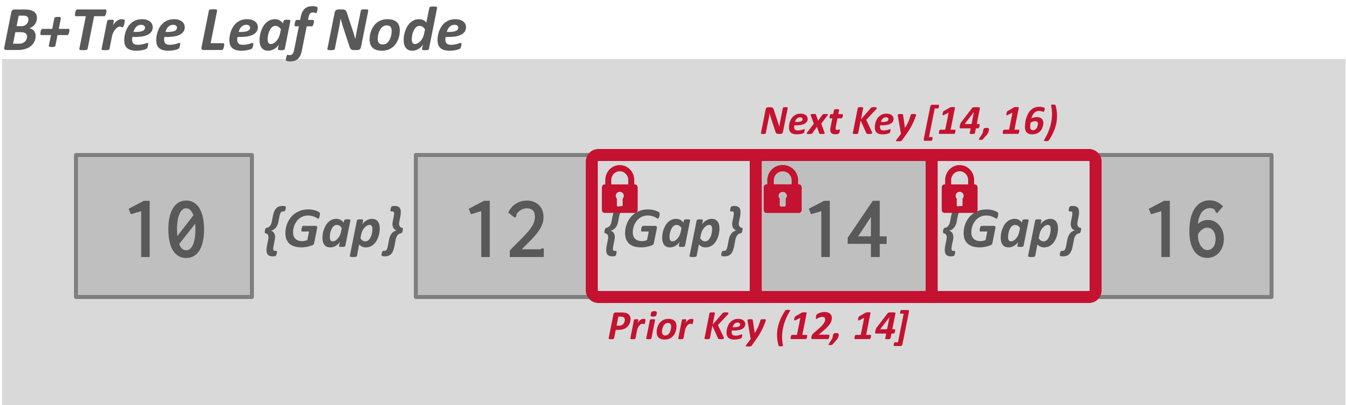


Gap Locks

Each txn acquires a key-value lock on the single key that it wants to access. Then get a gap lock on the next key gap.

Key-Range Locks

Locks that cover a key value and the gap to the next key value in a single index.
→ Need "virtual keys" for artificial values (infinity)



Weaker Levels Of Isolation

Serializability is useful because it allows programmers to ignore concurrency issues.

But enforcing it may allow too little concurrency and limit performance.

We may want to use a weaker level of consistency to improve scalability.

Isolation Levels

Controls the extent that a txn is exposed to the actions of other concurrent txns.

Provides for greater concurrency at the cost of exposing txns to uncommitted changes:

→ Dirty Reads
→ Unrepeatable Reads
→ Lost Updates
→ Phantom Reads

Isolation Levels

Isolation (High→Low)

SERIALIZABLE: No phantoms, all reads repeatable, no dirty reads.

REPEATABLE READS: Phantoms <u>may</u> happen.

READ COMMITTED: Phantoms, unrepeatable reads, and lost updates <u>may</u> happen.

READ UNCOMMITTED: All anomalies <u>may</u> happen.

Isolation Levels

	Dirty Read	*Unrepeatable Read*	*Lost Updates*	*Phantom*
SERIALIZABLE	No	No	No	No
REPEATABLE READ	No	No	No	Maybe
READ COMMITTED	No	Maybe	Maybe	Maybe
READ UNCOMMITTED	Maybe	Maybe	Maybe	Maybe

Isolation Levels

SERIALIZABLE: Strong Strict 2PL with phantom protection (e.g., index locks).

REPEATABLE READS: Same as above, but without phantom protection.

READ COMMITTED: Same as above, but S locks are released immediately. (No repeatable reads)

READ UNCOMMITTED: Same as above but allows dirty reads (no S locks).

SQL-92 Isolation Levels

The application can set a txn's isolation level before it executes any queries in that txn.

```
SET TRANSACTION ISOLATION LEVEL
  <isolation-level>;
```

Not all DBMS support all isolation levels in all execution scenarios

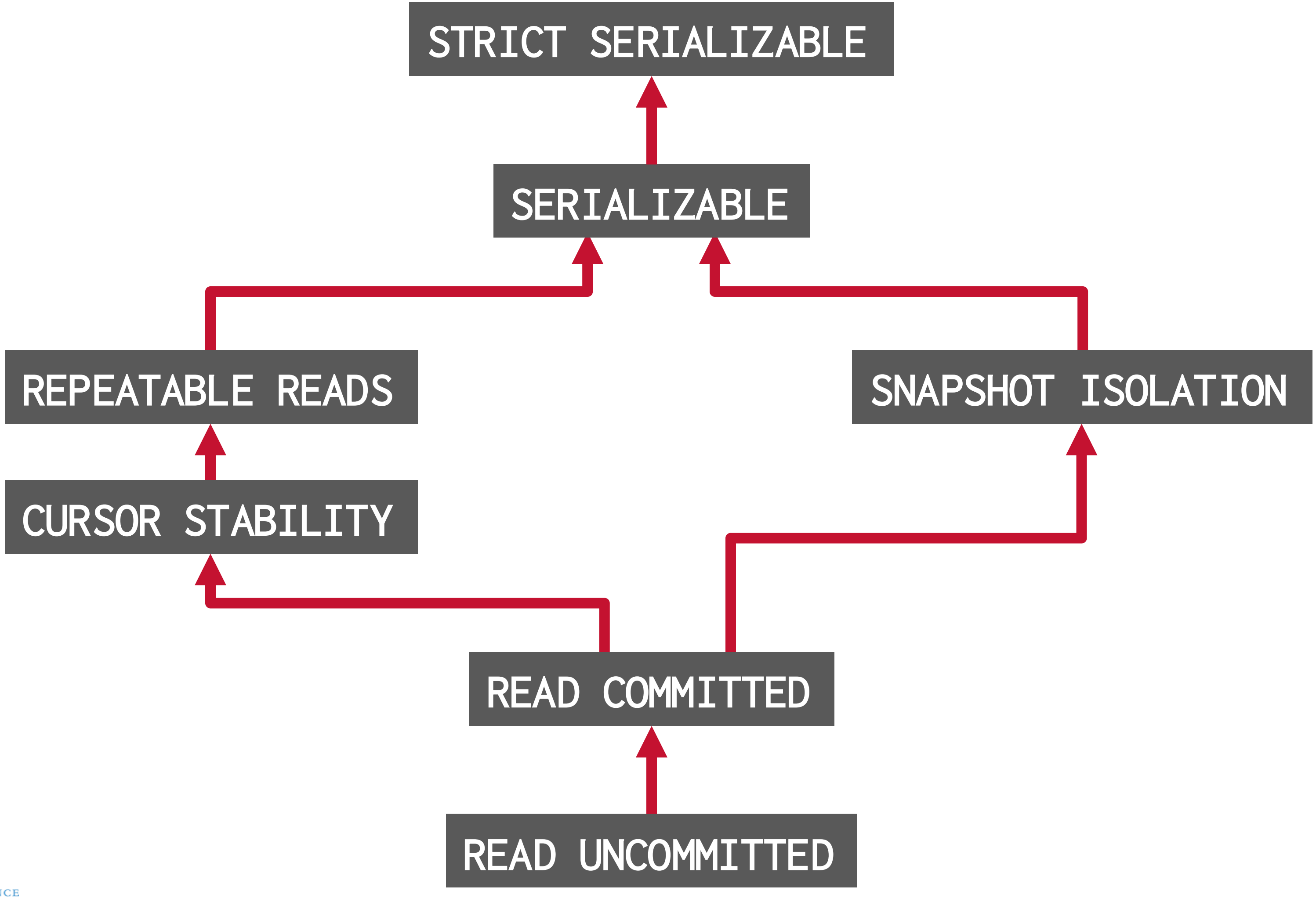
→ Replicated Environments

```
BEGIN TRANSACTION ISOLATION LEVEL
  <isolation-level>;
```

The default depends on implementation…

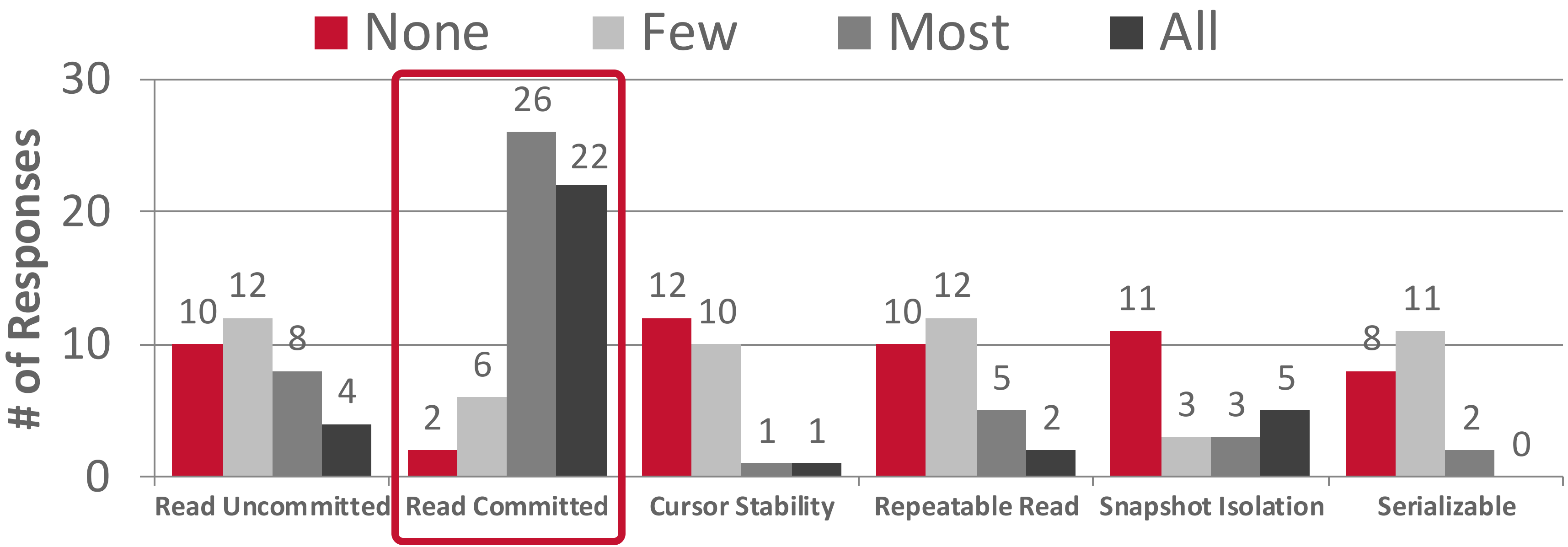
Isolation Levels

	Default	*Maximum*
Actian Ingres	SERIALIZABLE	SERIALIZABLE
IBM DB2	CURSOR STABILITY	SERIALIZABLE
CockroachDB	SERIALIZABLE	SERIALIZABLE
Google Spanner	STRICT SERIALIZABLE	STRICT SERIALIZABLE
MSFT SQL Server	READ COMMITTED	SERIALIZABLE
MySQL	REPEATABLE READS	SERIALIZABLE
Oracle	READ COMMITTED	SNAPSHOT ISOLATION
PostgreSQL	READ COMMITTED	SERIALIZABLE
SAP HANA	READ COMMITTED	SERIALIZABLE
VoltDB	SERIALIZABLE	SERIALIZABLE
YugaByte	SNAPSHOT ISOLATION	SERIALIZABLE


STRICT SERIALIZABLE
SERIALIZABLE
REPEATABLE READS
SNAPSHOT ISOLATION
CURSOR STABILITY
READ COMMITTED
READ UNCOMMITTED

Database Admin Survey

What isolation level do transactions execute at on this DBMS?



Conclusion

Every concurrency control protocol can be broken down into the basic concepts that have been described in the last two lectures.

→ Pessimistic: Locking

→ Optimistic: Timestamps

There is no one protocol that is always better than all others…

NEXT CLASS

Multi-Version Concurrency Control