Carnegie Mellon University

18 Timestamp Ordering Concurrency Control

Intro to Database Systems
15-445/15-645
Fall 2020

Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Project #3 is due Sun Nov 22nd @ 11:59pm.

Homework #4 is due Sun Nov 8th @ 11:59pm.

UPCOMING DATABASE TALKS

EraDB "Magical Indexes"
→ Monday Nov 9th @ 5pm ET

FaunaDB Serverless DBMS
→ Monday Nov 16th @ 5pm ET

Confluent ksqlDB (Kafka)
→ Monday Nov 16th @ 5pm ET

CONCURRENCY CONTROL APPROACHES

Two-Phase Locking (2PL)
→ Determine serializability order of conflicting operations at runtime while txns execute.

Pessimistic

Timestamp Ordering (T/O)
→ Determine serializability order of txns before they execute.

Optimistic

T/O 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 a serial schedule where T_i appears before T_j.

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 Clock.
→ Logical Counter.
→ Hybrid.

TODAY'S AGENDA

Basic Timestamp Ordering (T/O) Protocol

Optimistic Concurrency Control

Isolation Levels

BASIC T/O

Txns read and write objects without locks.

Every object X is tagged with timestamp of the last txn that successfully did read/write:
→ W-TS(X) – Write timestamp on X
→ R-TS(X) – Read timestamp on X

Check timestamps for every operation:
→ If txn tries to access an object "from the future", it aborts and restarts.

BASIC T/O – READS

If $TS(T_i) < W\text{-}TS(X)$, this violates timestamp order of T_i with regard to the writer of **X**.

→ Abort T_i and restart it with a new TS.

Else:

→ Allow T_i to read **X**.

→ Update $R\text{-}TS(X)$ to $\max(R\text{-}TS(X),\ TS(T_i))$

→ Make a local copy of **X** to ensure repeatable reads for T_i.

BASIC T/O – WRITES

If $TS(T_i) < R\text{-}TS(X)$ or $TS(T_i) < W\text{-}TS(X)$

→ Abort and restart T_i.

Else:

→ Allow T_i to write X and update W-TS(X)

→ Also make a local copy of X to ensure repeatable reads.

BASIC T/O – EXAMPLE #1

Schedule

TIME

$TS(T_1)=1$ $TS(T_2)=2$

T_1	T_2
BEGIN	
R(B)	
	BEGIN
	R(B)
	W(B)
R(A)	
	R(A)
R(A)	
	W(A)
COMMIT	**COMMIT**

Database

Object	R-TS	W-TS
A	0	0
B	0	0

BASIC T/O – EXAMPLE #1

Schedule

$TS(T_1)=1$ $TS(T_2)=2$

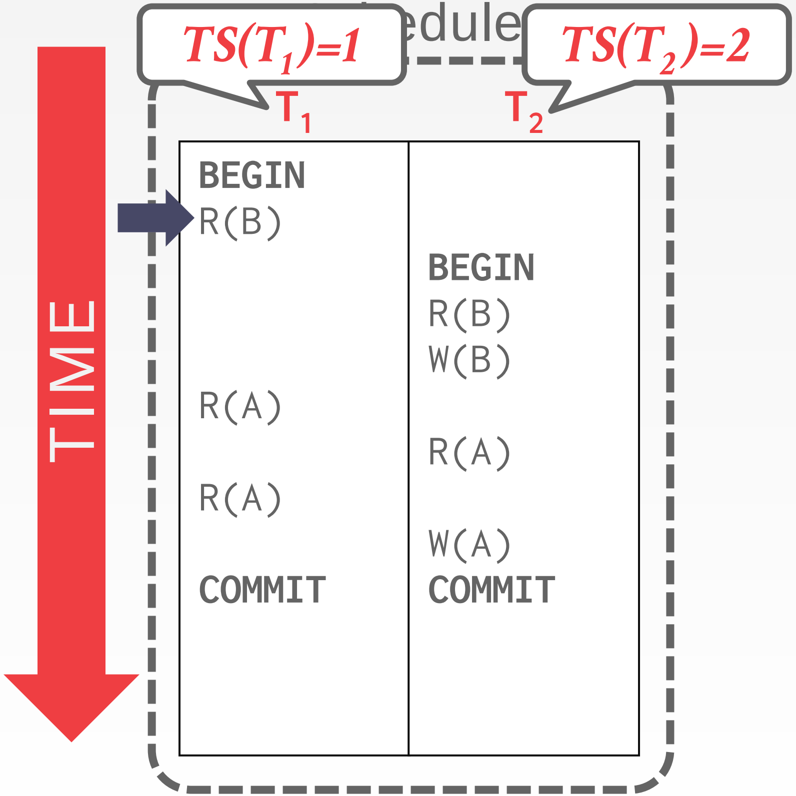
TIME

T_1	T_2
BEGIN	
R(B)	
	BEGIN
	R(B)
	W(B)
R(A)	
	R(A)
R(A)	
	W(A)
COMMIT	COMMIT

Database

Object	R-TS	W-TS
A	0	0
B	0	0

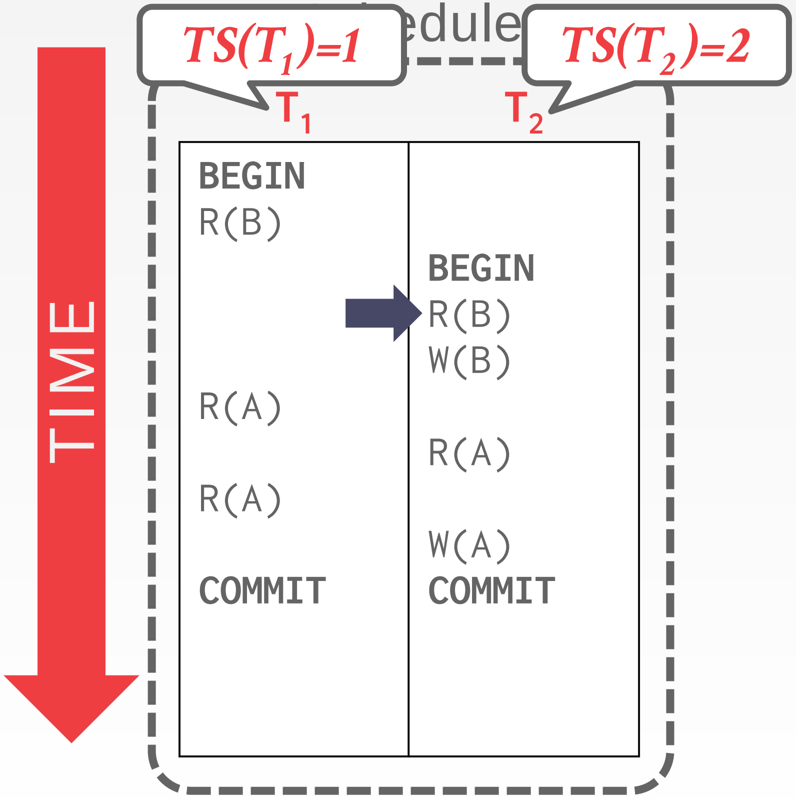
BASIC T/O – EXAMPLE #1



Database

Object	R-TS	W-TS
A	0	0
B	1	0

BASIC T/O – EXAMPLE #1



Database

Object	R-TS	W-TS
A	0	0
B	2	0

BASIC T/O – EXAMPLE #1

Schedule

TIME

$TS(T_1)=1$ $TS(T_2)=2$

T_1	T_2
BEGIN	
R(B)	
	BEGIN
	R(B)
	W(B)
R(A)	
	R(A)
R(A)	
	W(A)
COMMIT	COMMIT

Database

Object	R-TS	W-TS
A	0	0
B	2	2

BASIC T/O – EXAMPLE #1

Schedule

$TS(T_1)=1$ $TS(T_2)=2$

TIME

T_1	T_2
BEGIN	
R(B)	
	BEGIN
	R(B)
	W(B)
R(A)	
	R(A)
R(A)	
	W(A)
COMMIT	COMMIT

Database

Object	R-TS	W-TS
A	1	0
B	2	2

BASIC T/O – EXAMPLE #1

Schedule

$TS(T_1)=1$ $TS(T_2)=2$

TIME

T_1	T_2
BEGIN	
R(B)	
	BEGIN
	R(B)
	W(B)
R(A)	
	R(A)
R(A)	
	W(A)
COMMIT	COMMIT

Database

Object	R-TS	W-TS
A	2	0
B	2	2

BASIC T/O – EXAMPLE #1

Schedule

TS(T_1)=1

TS(T_2)=2

TIME

T_1	T_2
BEGIN	
R(B)	
	BEGIN
	R(B)
	W(B)
R(A)	
	R(A)
R(A)	
	W(A)
COMMIT	COMMIT

Database

TS(T_1) < TS(T_2)

Object	R-TS	W-TS
A	2	0
B	2	2

BASIC T/O – EXAMPLE #1

TIME

Schedule

$TS(T_1)=1$ $TS(T_2)=2$

T_1	T_2
BEGIN	
R(B)	
	BEGIN
	R(B)
	W(B)
R(A)	
	R(A)
R(A)	
	W(A)
COMMIT	**COMMIT**

Database

Object	R-TS	W-TS
A	2	2
B	2	2

BASIC T/O – EXAMPLE #1

Schedule

TS(T_1)=1 *TS(T_2)=2*

TIME

T_1	T_2
BEGIN	
R(B)	
	BEGIN
	R(B)
	W(B)
R(A)	
	R(A)
R(A)	
	W(A)
COMMIT	COMMIT

Database

Object	R-TS	W-TS
A	2	2
B	2	2

No violations so both txns are safe to commit.

BASIC T/O – EXAMPLE #2

Schedule

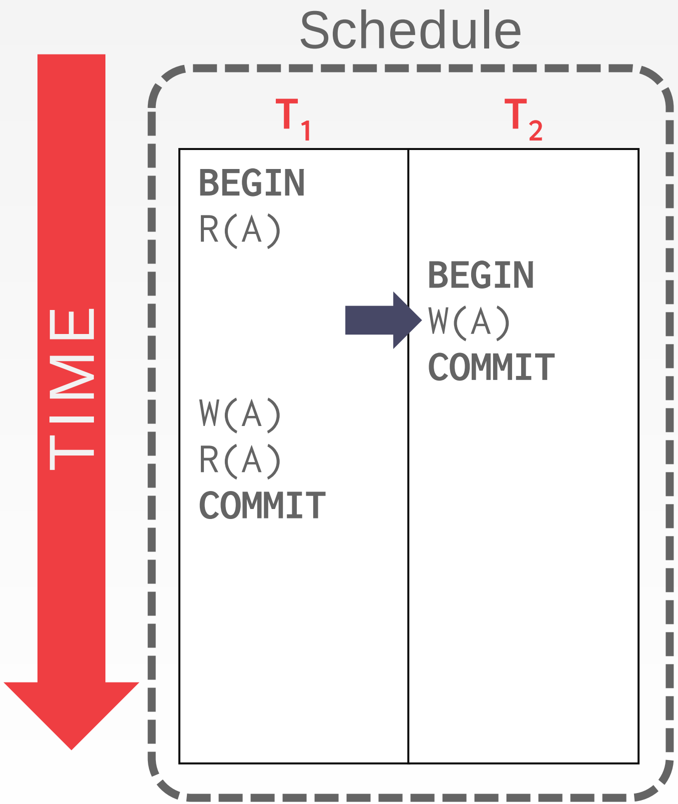
TIME

T_1	T_2
BEGIN	
R(A)	
	BEGIN
	W(A)
	COMMIT
W(A)	
R(A)	
COMMIT	

Database

Object	R-TS	W-TS
A	1	0
B	0	0

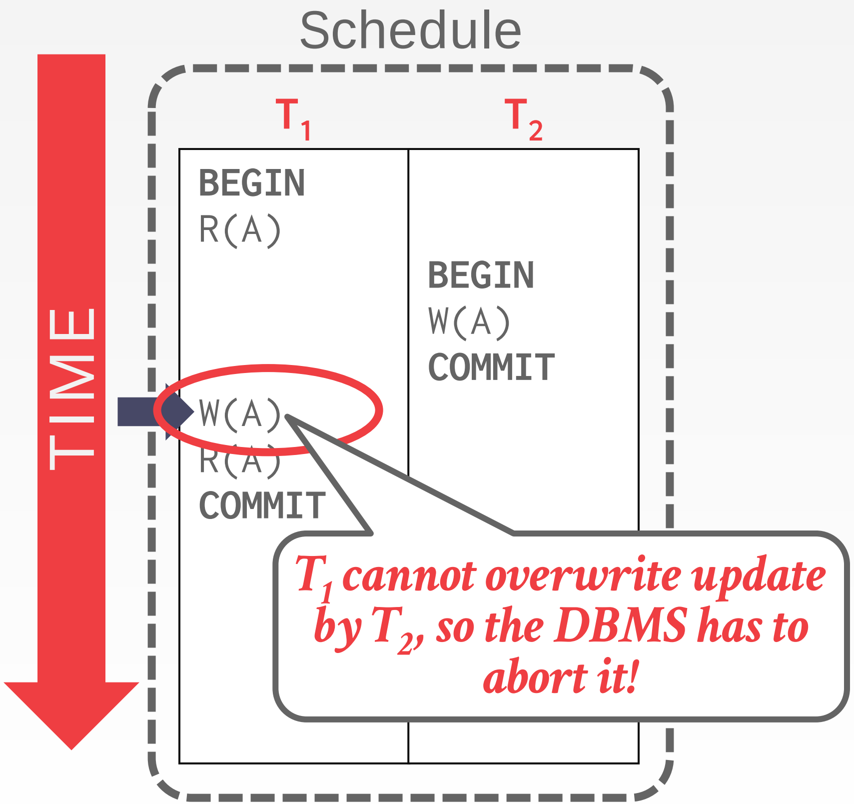
BASIC T/O – EXAMPLE #2



Database

Object	R-TS	W-TS
A	1	2
B	0	0

BASIC T/O – EXAMPLE #2



Database

Object	R-TS	W-TS
A	1	2
B	0	0

Violation:
$TS(T_1) < W\text{-}TS(A)$

THOMAS WRITE RULE

If $TS(T_i) < R\text{-}TS(X)$:
→ Abort and restart T_i.

If $TS(T_i) < W\text{-}TS(X)$:
→ **Thomas Write Rule**: Ignore the write to allow the txn to continue executing without aborting.
→ This violates timestamp order of T_i.

Else:
→ Allow T_i to write X and update $W\text{-}TS(X)$

BASIC T/O – EXAMPLE #2

Schedule

TIME →

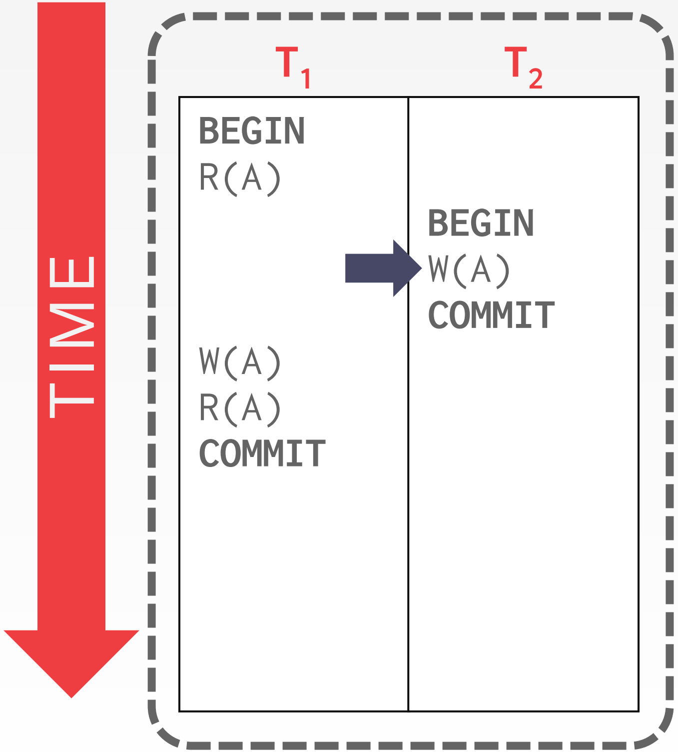
T_1	T_2
BEGIN	
R(A)	
	BEGIN
	W(A)
	COMMIT
W(A)	
R(A)	
COMMIT	

Database

Object	R-TS	W-TS
A	1	0
B	0	0

BASIC T/O – EXAMPLE #2

Schedule



Database

Object	R-TS	W-TS
A	1	2
B	0	0

BASIC T/O – EXAMPLE #2

Schedule

TIME

T_1	T_2
BEGIN	
R(A)	
	BEGIN
	W(A)
	COMMIT
W(A)	
R(A)	
COMMIT	

Ignore the write and allow T_1 to commit.

Database

Object	R-TS	W-TS
A	1	2
B	0	0

We do not update W-TS(A)

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.

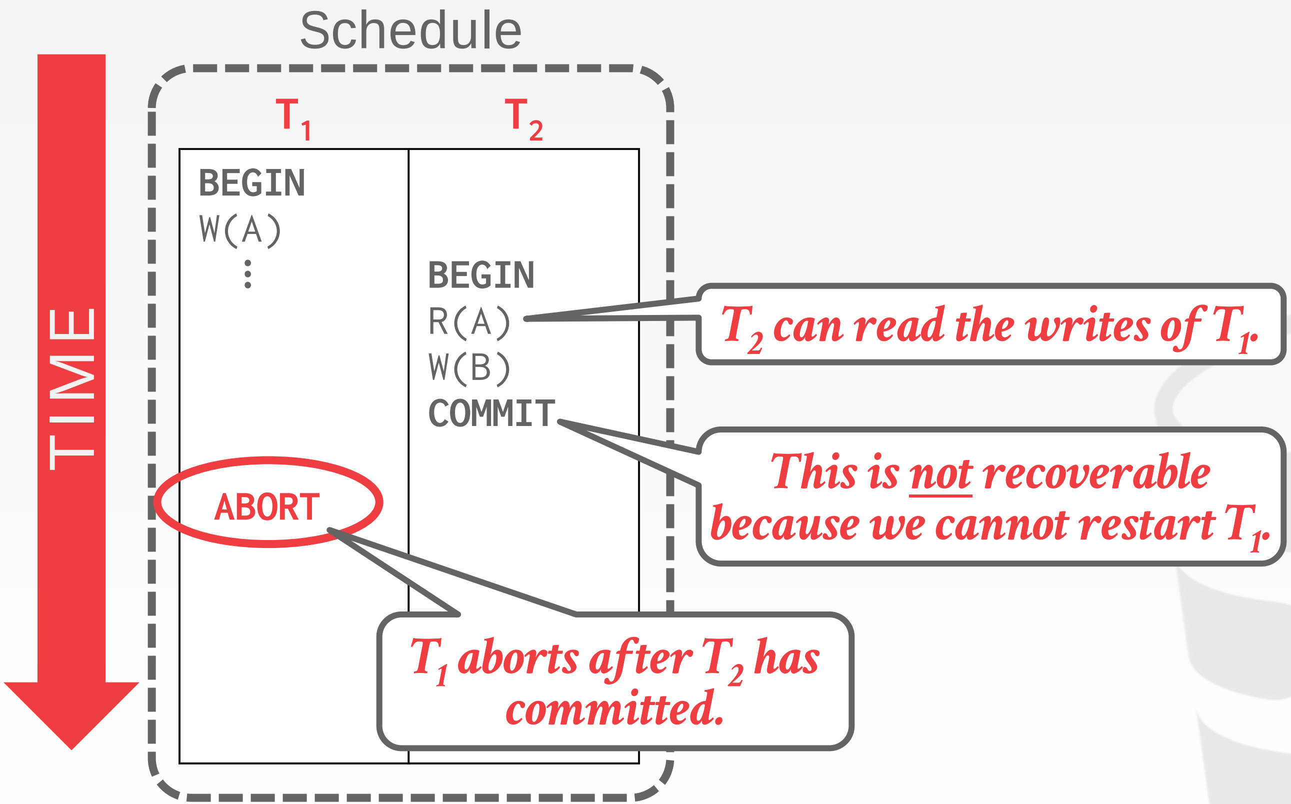
Permits schedules that are not recoverable…

RECOVERABLE SCHEDULES

A schedule is **recoverable** if txns commit only after all txns whose changes they read, commit.

Otherwise, the DBMS cannot guarantee that txns read data that will be restored after recovering from a crash.

RECOVERABLE SCHEDULES



BASIC T/O – PERFORMANCE ISSUES

High overhead from copying data to txn's workspace and from updating timestamps.

Long running txns can get starved.

→ The likelihood that a txn will read something from a newer txn increases.

OBSERVATION

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

A better approach is to optimize for the no-conflict case.

OPTIMISTIC CONCURRENCY CONTROL

The 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.

ACM Transactions on Database Systems, Vol. 6, No. 2, June 1981, Pages 213-226.

OCC PHASES

#1 – Read Phase:

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

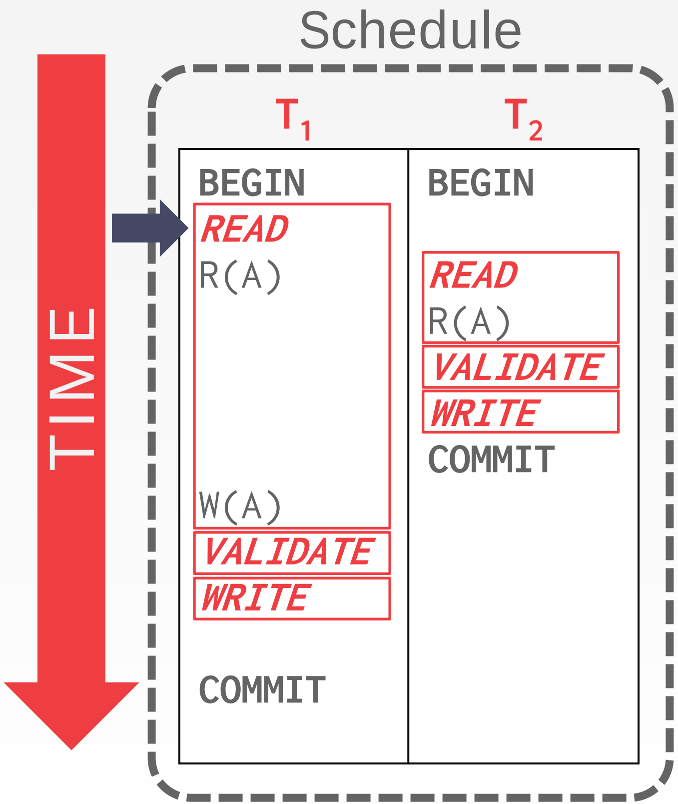
#2 – Validation Phase:

→ When a txn commits, check whether it conflicts with other txns.

#3 – Write Phase:

→ If validation succeeds, apply private changes to database. Otherwise abort and restart the txn.

OCC – EXAMPLE



Database

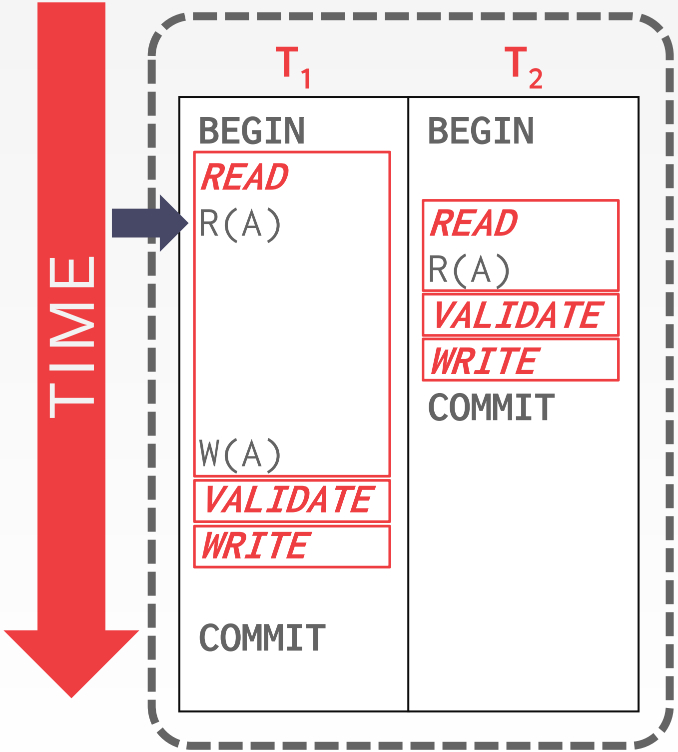
Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

Object	Value	W-TS
–	–	–
–	–	–

OCC – EXAMPLE

Schedule



Database

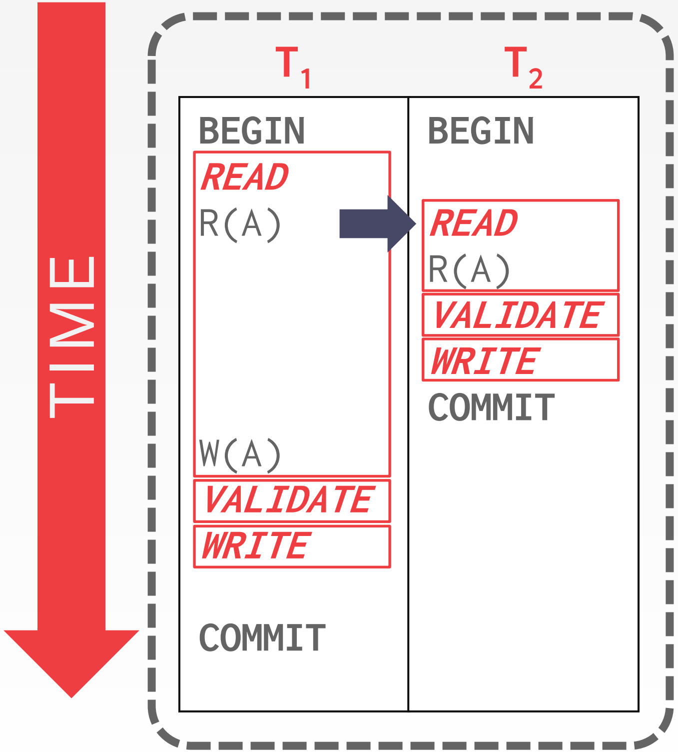
Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

Object	Value	W-TS
A	123	0
–	–	–

OCC – EXAMPLE

Schedule



Database

Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

Object	Value	W-TS
A	123	0
–	–	–

T_2 Workspace

Object	Value	W-TS
–	–	–
–	–	–

OCC – EXAMPLE

Schedule

TIME →

T_1	T_2
BEGIN	BEGIN
READ	
R(A)	
	READ
	R(A)
	VALIDATE
	WRITE
	COMMIT
W(A)	
VALIDATE	
WRITE	
COMMIT	

Database

Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

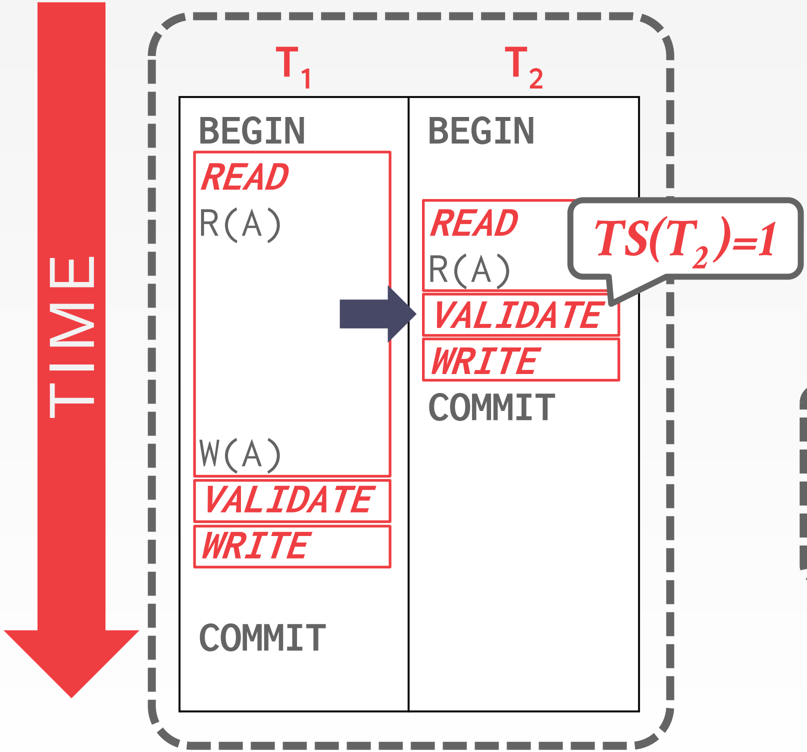
Object	Value	W-TS
A	123	0
–	–	–

T_2 Workspace

Object	Value	W-TS
A	123	0
–	–	–

OCC – EXAMPLE

Schedule



Database

Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

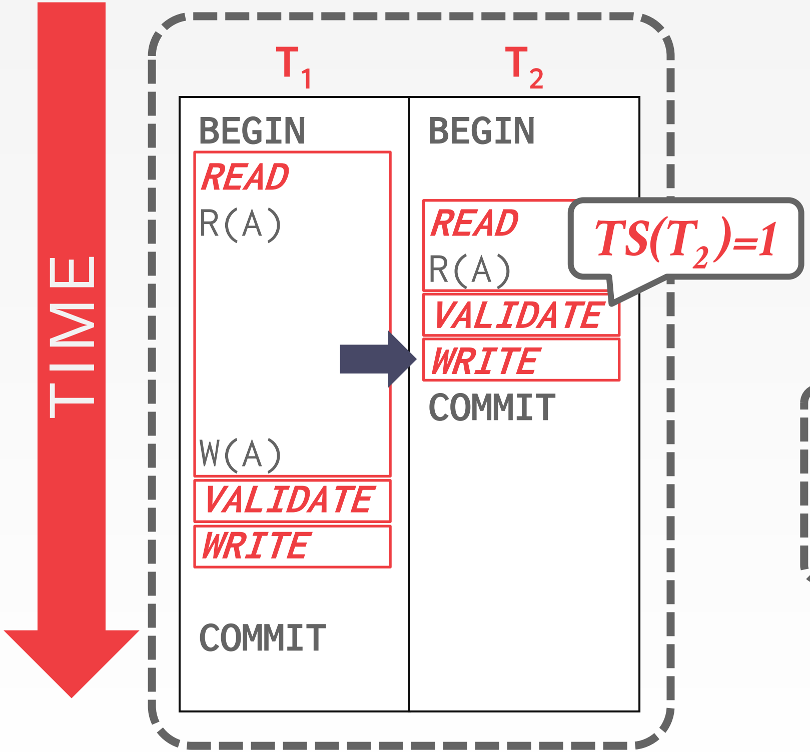
Object	Value	W-TS
A	123	0
–	–	–

T_2 Workspace

Object	Value	W-TS
A	123	0
–	–	–

OCC – EXAMPLE

Schedule



Database

Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

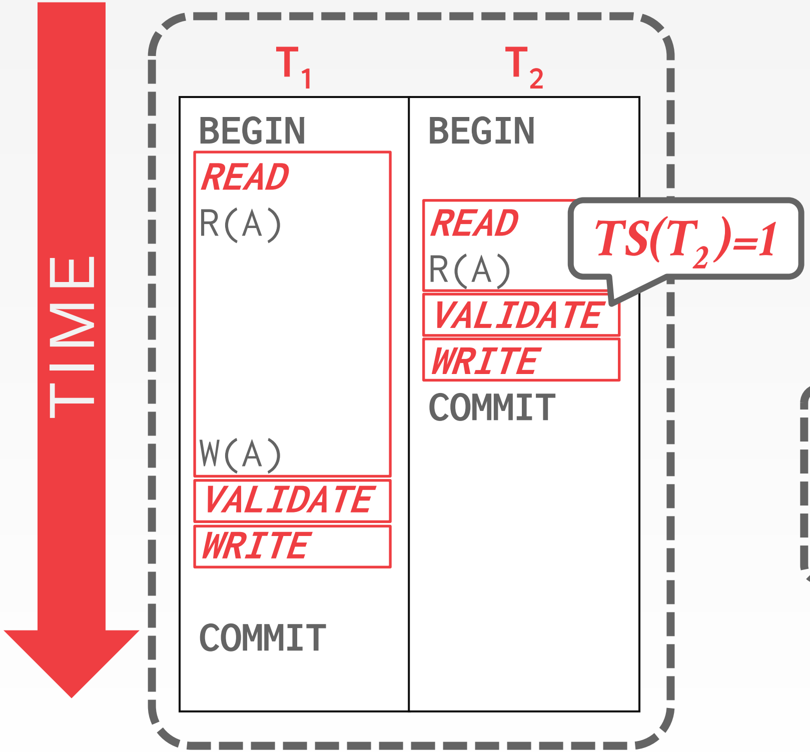
Object	Value	W-TS
A	123	0
–	–	–

T_2 Workspace

Object	Value	W-TS
A	123	0
–	–	–

OCC – EXAMPLE

Schedule



Database

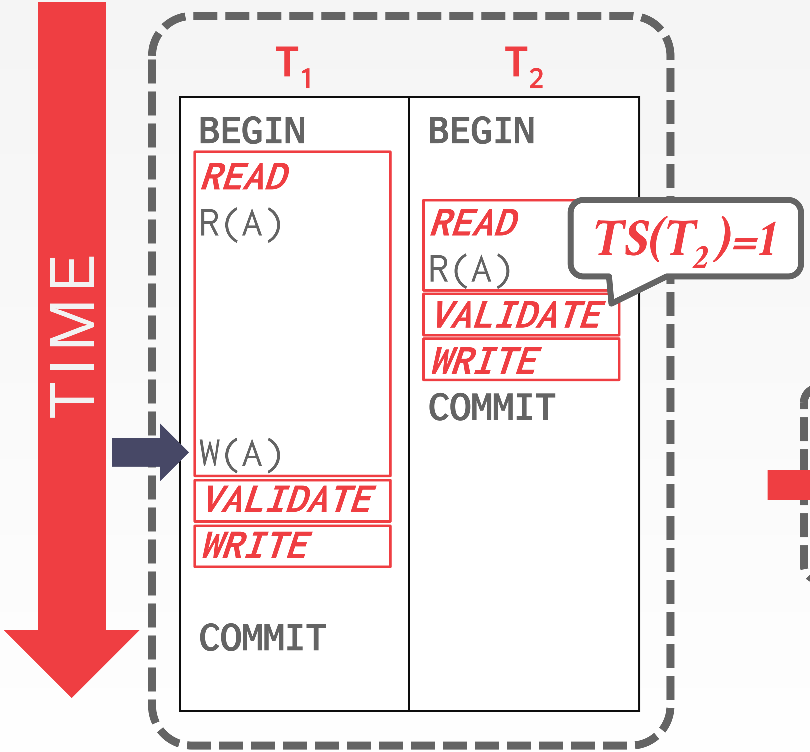
Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

Object	Value	W-TS
A	123	0
–	–	–

OCC – EXAMPLE

Schedule



Database

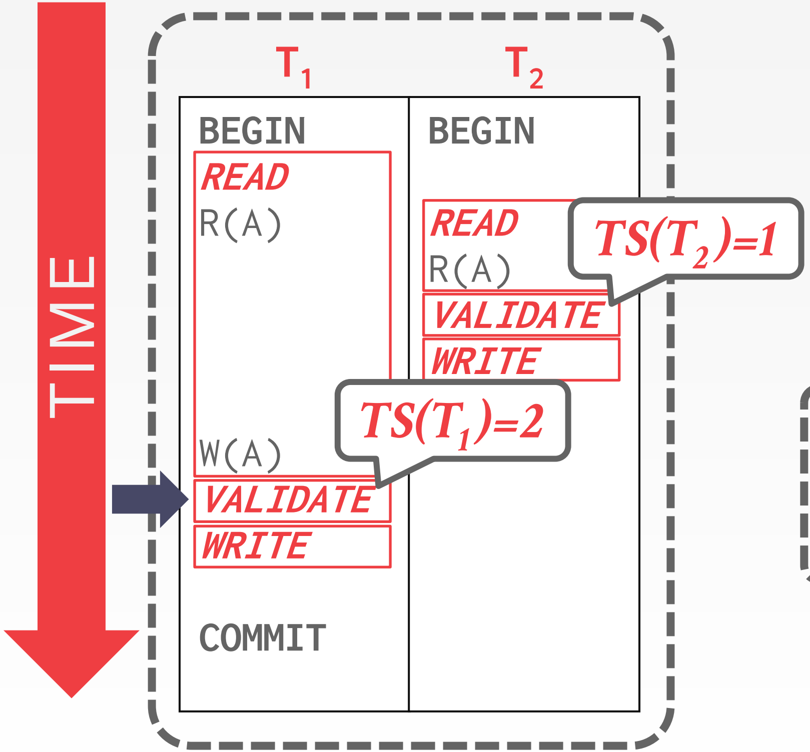
Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

Object	Value	W-TS
A	456	∞
–	–	–

OCC – EXAMPLE

Schedule



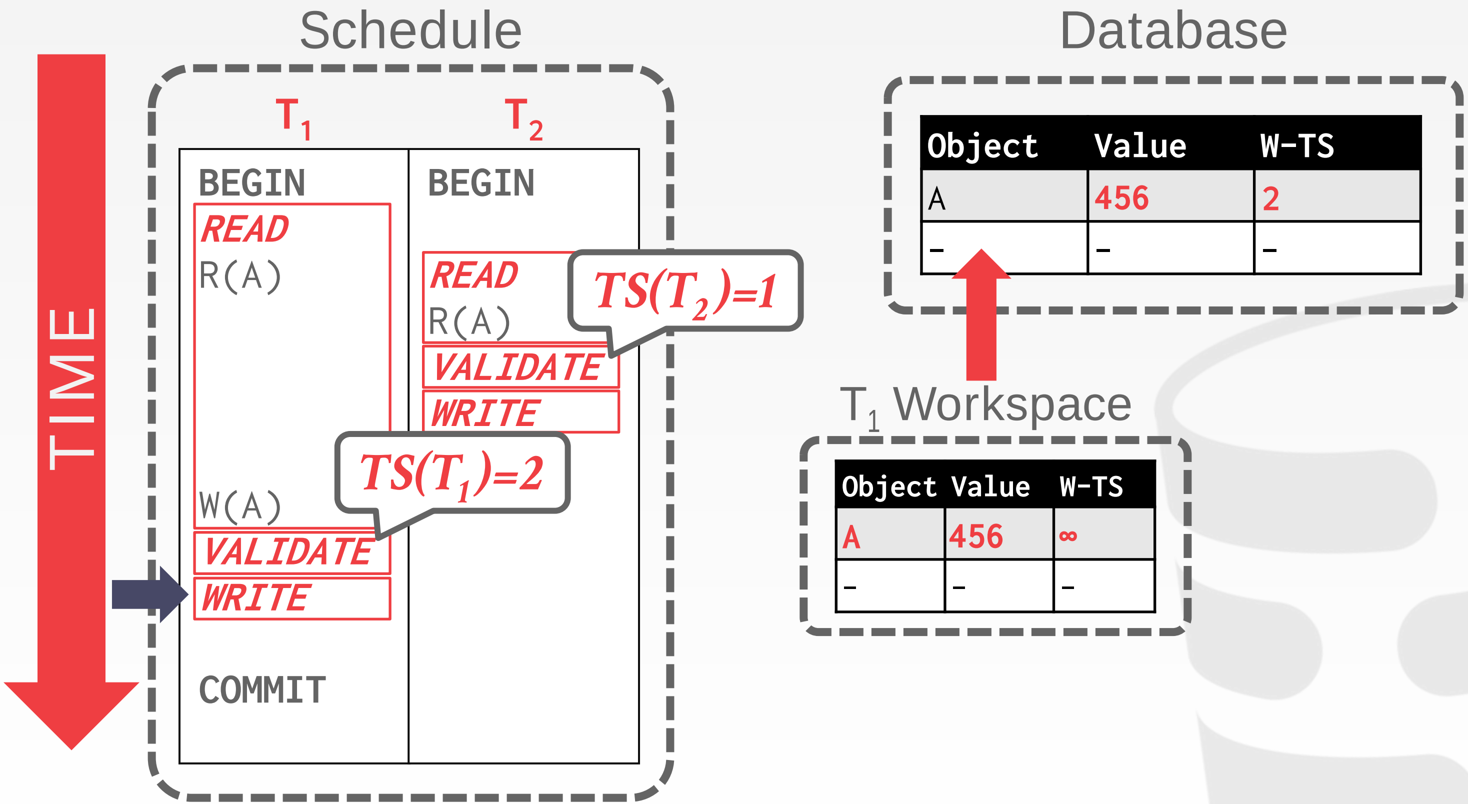
Database

Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

Object	Value	W-TS
A	456	∞
–	–	–

OCC – EXAMPLE


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

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.

OCC – VALIDATION PHASE

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

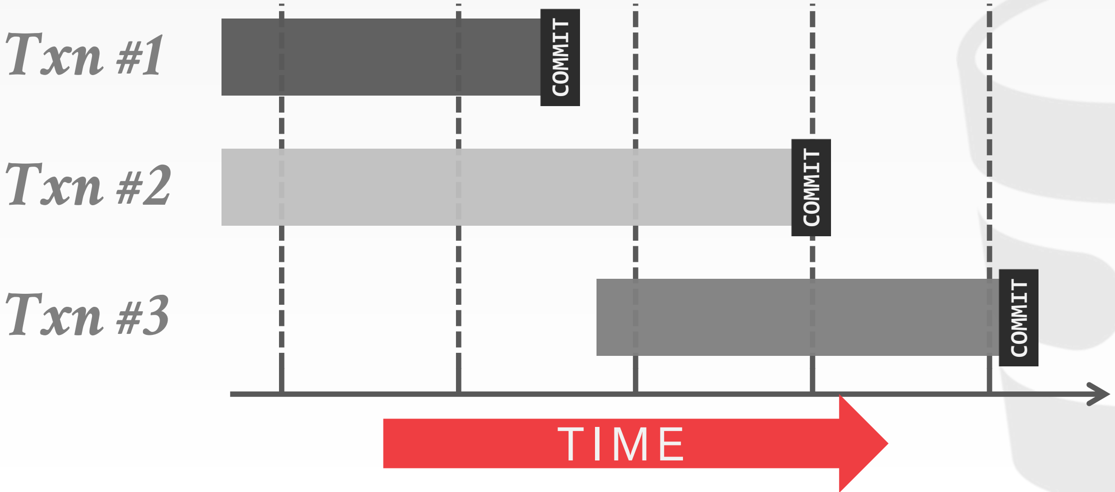
→ The DBMS needs to guarantee only serializable schedules are permitted.
→ Checks other txns for RW and WW conflicts and ensure that conflicts are in one direction (e.g., older→younger).

Two methods for this phase:

→ Backward Validation
→ Forward Validation

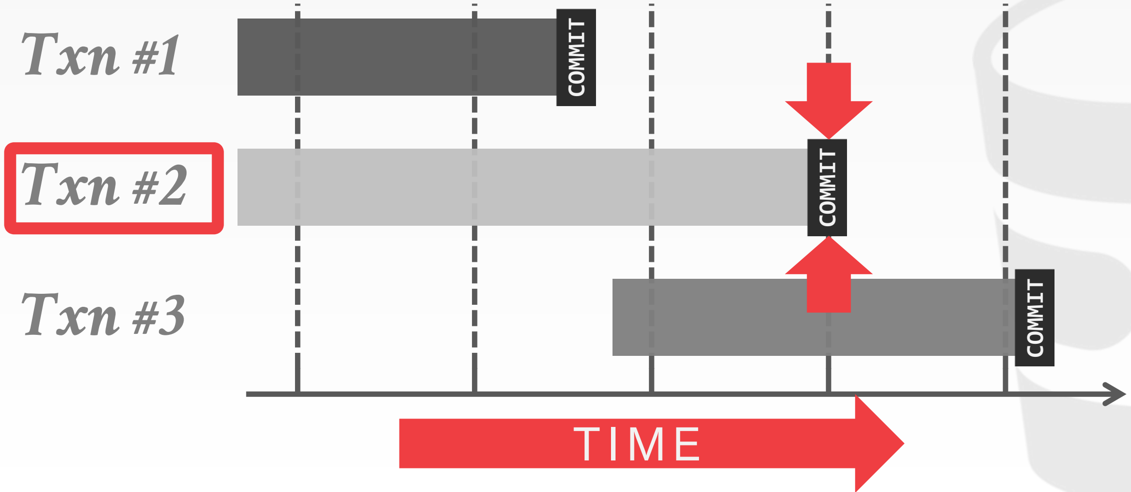
OCC – BACKWARD VALIDATION

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



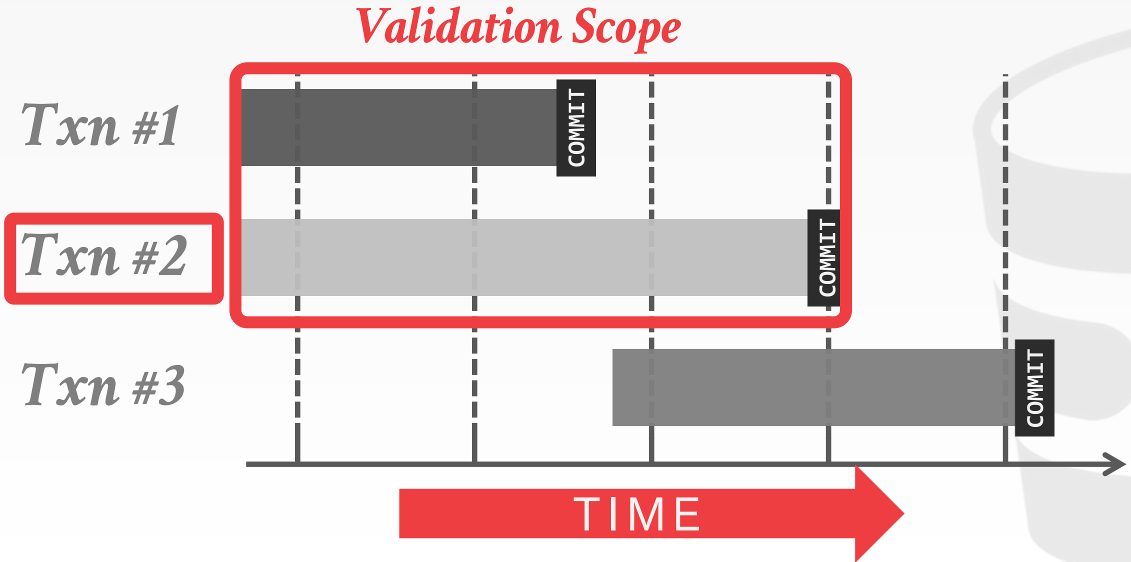
OCC – BACKWARD VALIDATION

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



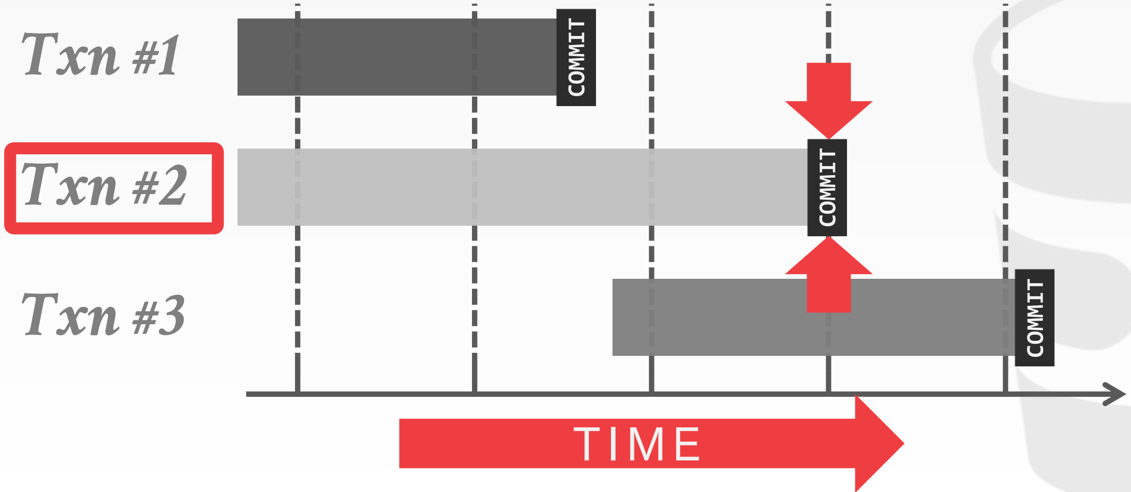
OCC – BACKWARD VALIDATION

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



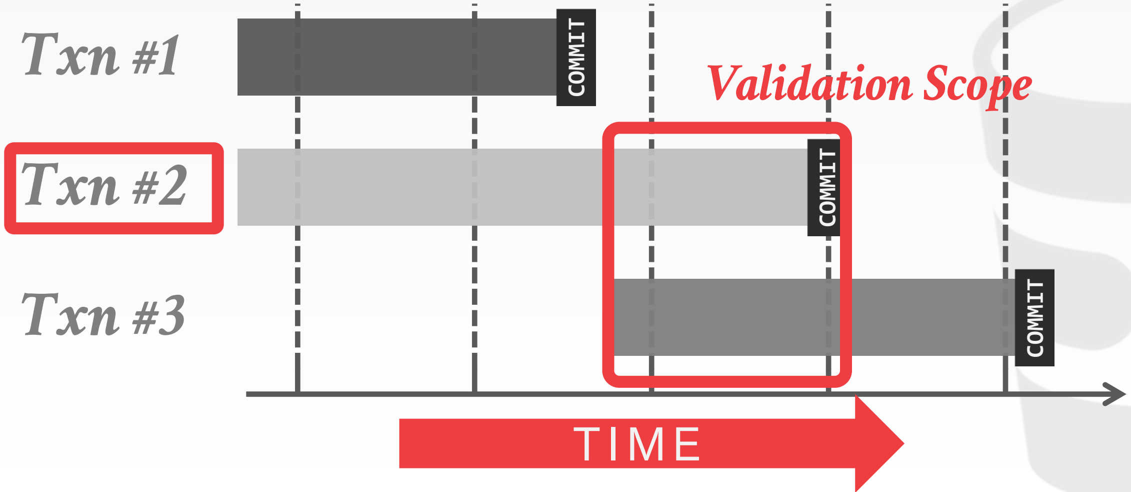
OCC – FORWARD VALIDATION

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



OCC – FORWARD VALIDATION

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



OCC – FORWARD VALIDATION

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

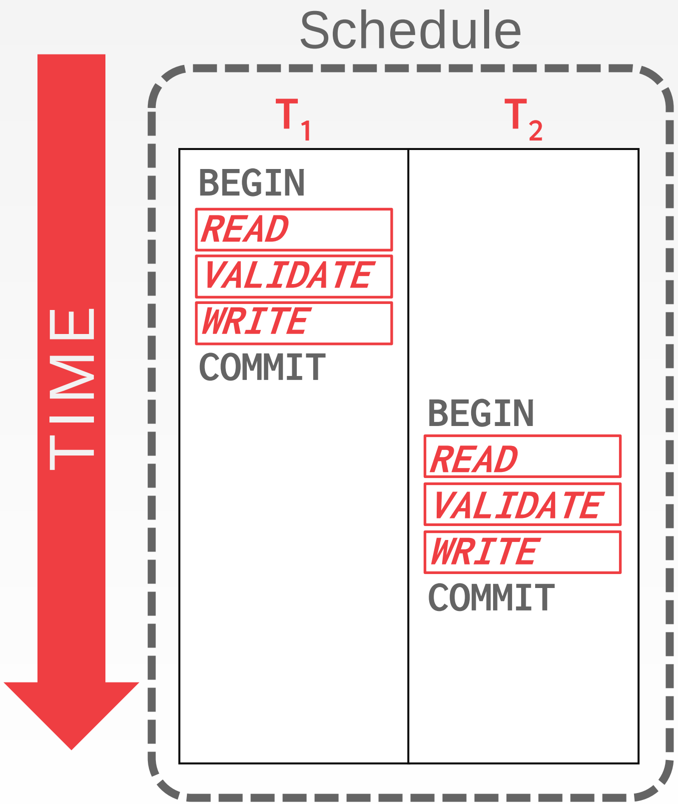
Check the timestamp ordering of the committing txn with all other running txns.

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

OCC – FORWARD VALIDATION STEP #1

T_i completes all three phases before T_j begins.

OCC – FORWARD VALIDATION STEP #1



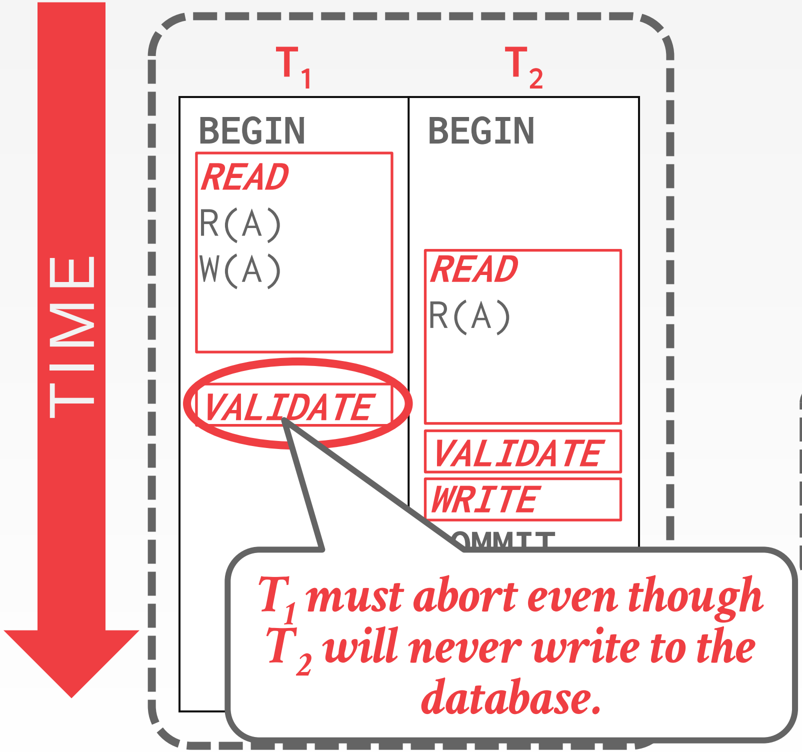
OCC – FORWARD VALIDATION STEP #2

T_i completes before T_j starts its **Write** phase, and T_i does not write to any object read by T_j.

→ $WriteSet(T_i) \cap ReadSet(T_j) = \emptyset$

OCC – FORWARD VALIDATION STEP #2

Schedule



Database

Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

Object	Value	W-TS
A	123	∞
–	–	–

T_2 Workspace

Object	Value	W-TS
A	123	0
–	–	–

OCC – FORWARD VALIDATION STEP #2

Schedule

TIME

T_1	T_2
BEGIN	BEGIN
READ R(A) W(A)	
	READ R(A)
	VALIDATE
VALIDATE	
WRITE	
COMMIT	
	WRITE
	COMMIT

Database

Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

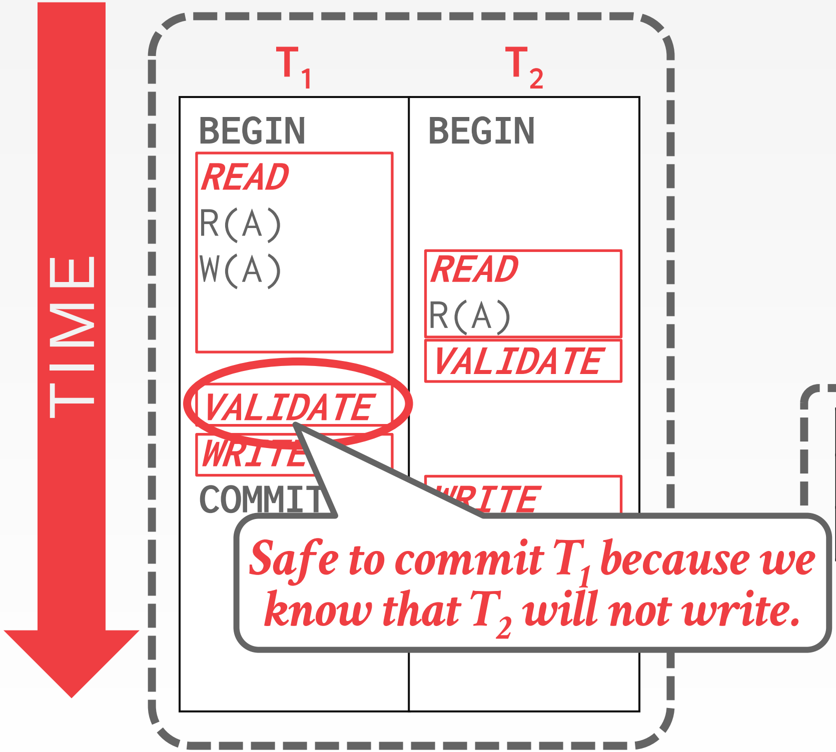
Object	Value	W-TS
A	456	∞
–	–	–

T_2 Workspace

Object	Value	W-TS
A	123	0
–	–	–

OCC – FORWARD VALIDATION STEP #2

Schedule



Database

Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

Object	Value	W-TS
A	456	∞
–	–	–

T_2 Workspace

Object	Value	W-TS
A	123	0
–	–	–

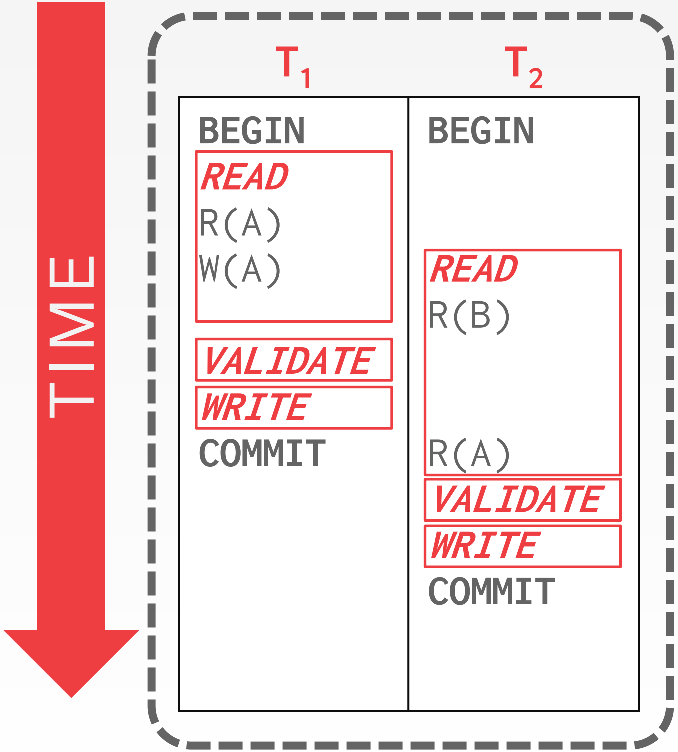
OCC – FORWARD VALIDATION STEP #3

T_i completes its **Read** phase before T_j completes its **Read** phase

And T_i does not write to any object that is either read or written by T_j:
→ $WriteSet(T_i) \cap ReadSet(T_j) = \emptyset$
→ $WriteSet(T_i) \cap WriteSet(T_j) = \emptyset$

OCC – FORWARD VALIDATION STEP #3

Schedule



Database

Object	Value	W-TS
A	123	0
B	XYZ	0

T_1 Workspace

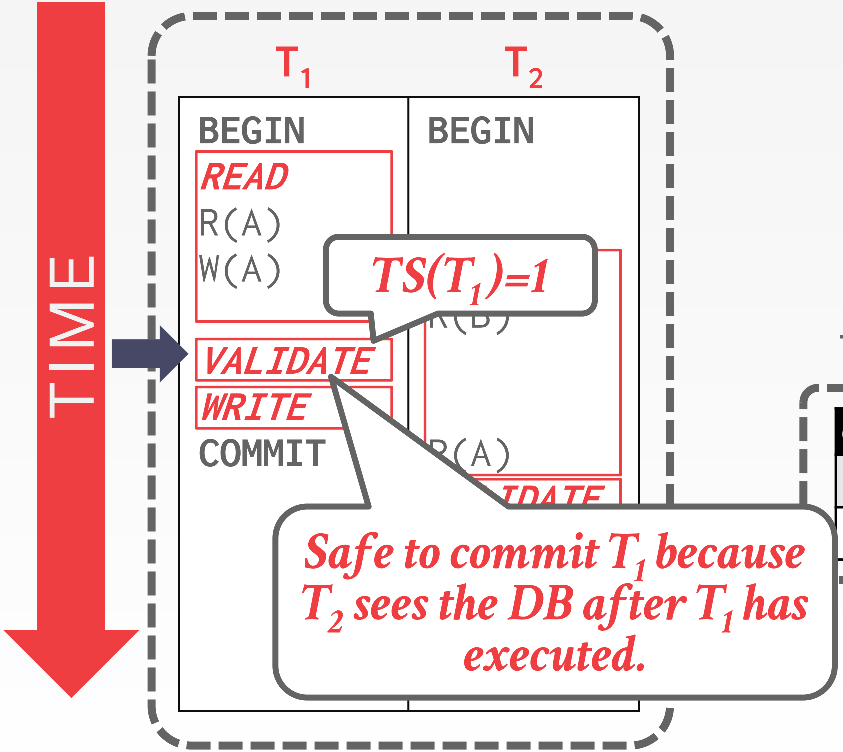
Object	Value	W-TS
A	456	∞
–	–	–

T_2 Workspace

Object	Value	W-TS
B	XYZ	0
–	–	–

OCC – FORWARD VALIDATION STEP #3

Schedule



Database

Object	Value	W-TS
A	123	0
B	XYZ	0

T_1 Workspace

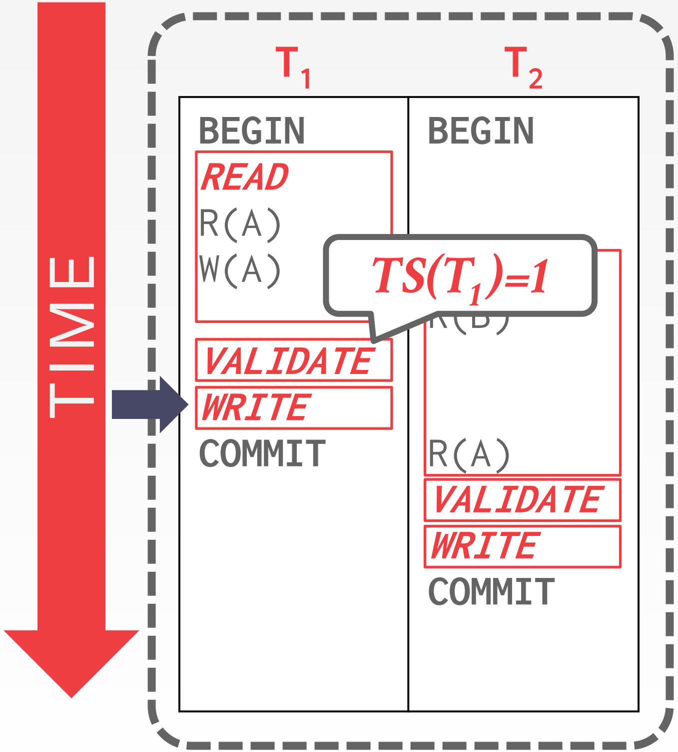
Object	Value	W-TS
A	456	∞
-	-	-

T_2 Workspace

Object	Value	W-TS
B	XYZ	0
-	-	-

OCC – FORWARD VALIDATION STEP #3

Schedule



Database

Object	Value	W-TS
A	456	1
B	XYZ	0

T_1 Workspace

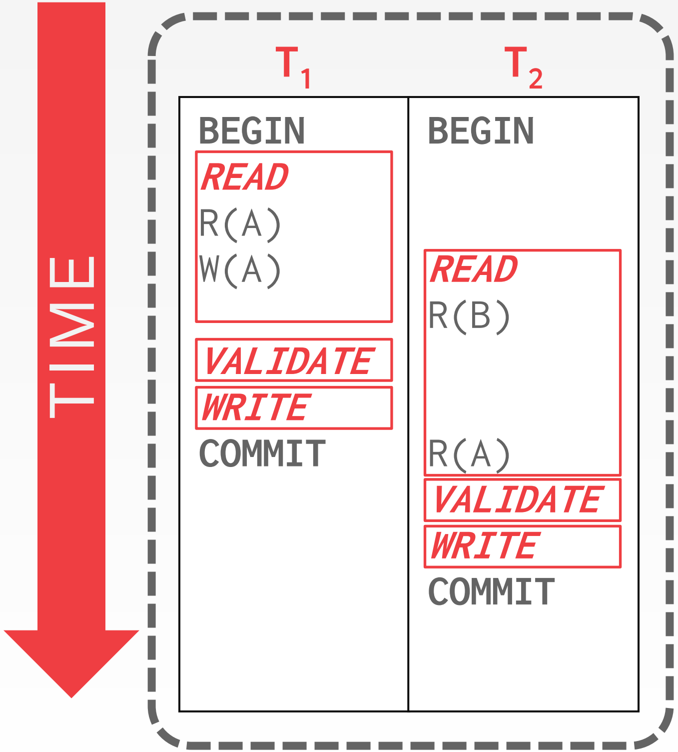
Object	Value	W-TS
A	456	∞
–	–	–

T_2 Workspace

Object	Value	W-TS
B	XYZ	0
–	–	–

OCC – FORWARD VALIDATION STEP #3

Schedule



Database

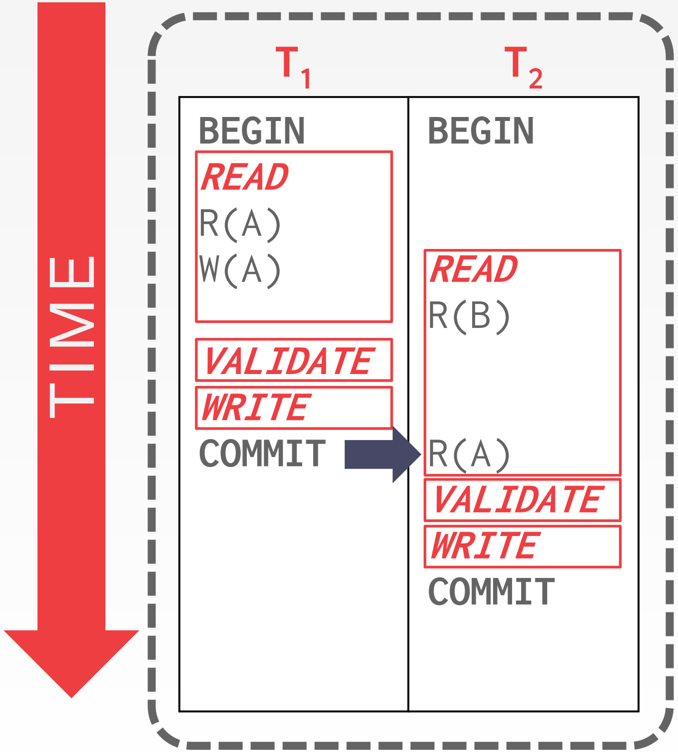
Object	Value	W-TS
A	456	1
B	XYZ	0

T_2 Workspace

Object	Value	W-TS
B	XYZ	0
–	–	–

OCC – FORWARD VALIDATION STEP #3

Schedule



Database

Object	Value	W-TS
A	456	1
B	XYZ	0

T_2 Workspace

Object	Value	W-TS
B	XYZ	0
A	456	1

OCC – WRITE PHASE

The DBMS propagates the changes in the txn's write set to the database and makes them visible to other txns.

Assume that only one txn can be in the **Write** Phase at a time.

→ Use write latches to support parallel validation/writes.

OCC – OBSERVATIONS

OCC works well when the # of conflicts is low:
→ All txns are read-only (ideal).
→ Txns access disjoint subsets of data.

If the database is large and the workload is not skewed, then there is a low probability of conflict, so again locking is wasteful.

OCC – PERFORMANCE ISSUES

High overhead for copying data locally.

Validation/Write phase bottlenecks.

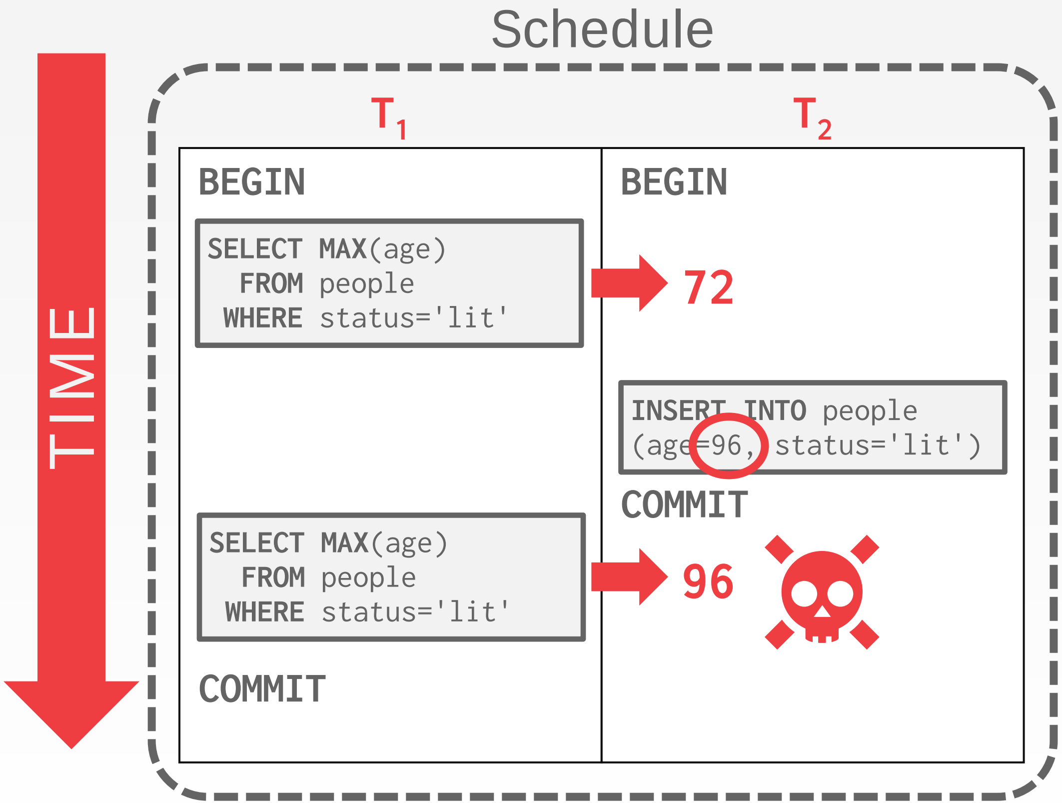
Aborts are more wasteful than in 2PL because they only occur after a txn has already executed.

DYNAMIC DATABASES

Recall that so far we have only dealing with transactions that read and update existing objects in the database.

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

THE PHANTOM PROBLEM




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

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.

THE PHANTOM PROBLEM

Approach #1: Re-Execute Scans

Approach #2: Predicate Locking

Approach #3: Index Locking

RE-EXECUTE SCANS

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

→ Have to 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.

PREDICATE LOCKING

Proposed locking scheme from System R.

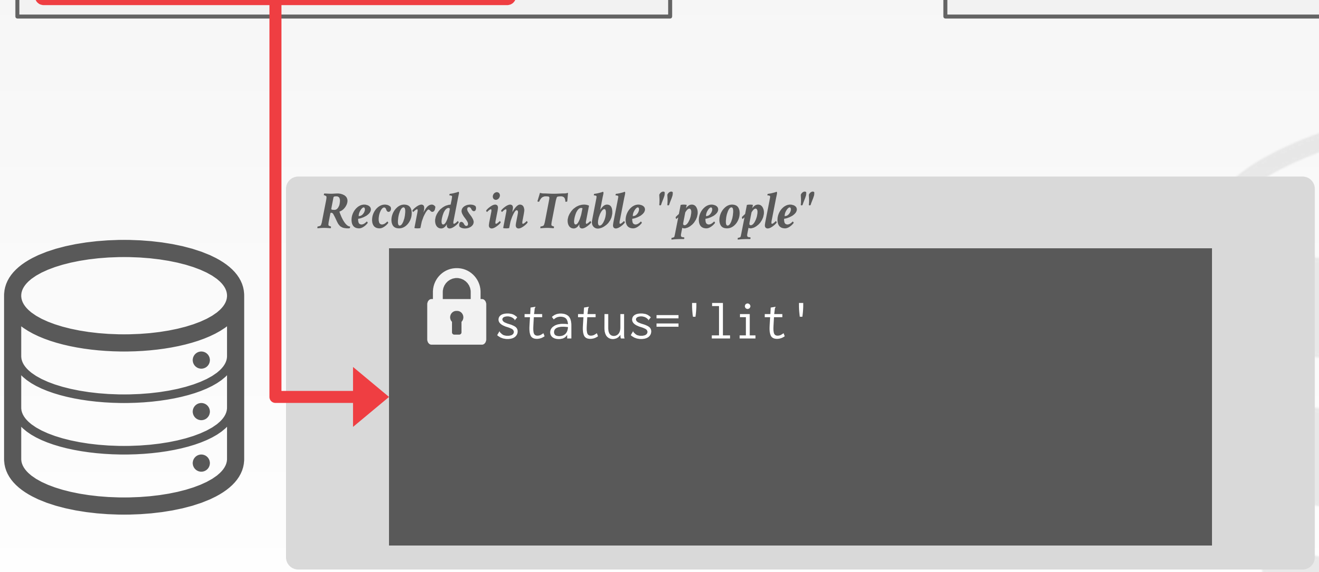
- → Shared lock on the predicate in a **WHERE** clause of a **SELECT** query.
- → Exclusive lock on the predicate in a **WHERE** clause of any **UPDATE**, **INSERT**, or **DELETE** query.

Never implemented in any system except for HyPer (precision locking).

PREDICATE LOCKING

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

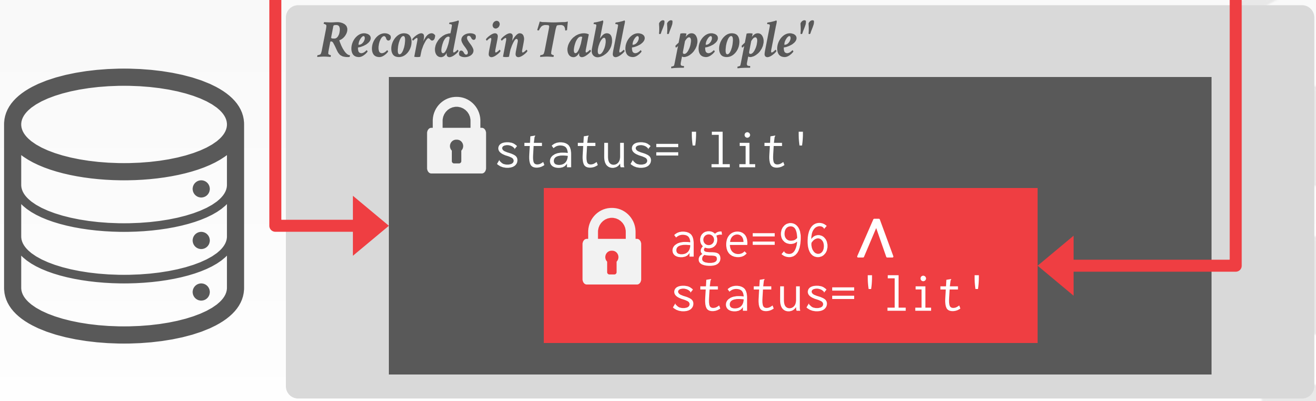
```
INSERT INTO people VALUES
(age=96, status='lit')
```



PREDICATE LOCKING

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

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



INDEX LOCKING

If there is an index on the status attribute then the txn can lock index page containing the data with **status='lit'**.

If there are no records with **status='lit'**, the txn must lock the index page where such a data entry would be, if it existed.

LOCKING WITHOUT AN INDEX

If there is no suitable index, then the txn must obtain:

→ A lock on every page in the table to prevent a record's `status='lit'` from being changed to `lit`.
→ The lock for the table itself to prevent records with `status='lit'` from being added or deleted.

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
→ Phantom Reads

ISOLATION LEVELS

Isolation (High→Low)

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

REPEATABLE READS: Phantoms may happen.

READ COMMITTED: Phantoms and unrepeatable reads may happen.

READ UNCOMMITTED: All of them may happen.

ISOLATION LEVELS

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

ISOLATION LEVELS

SERIALIZABLE: Obtain all locks first; plus index locks, plus strict 2PL.

REPEATABLE READS: Same as above, but no index locks.

READ COMMITTED: Same as above, but **S** locks are released immediately.

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

SQL-92 ISOLATION LEVELS

You set a txn's isolation level before you execute any queries in that txn.

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

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

Not all DBMS support all isolation levels in all execution scenarios

→ Replicated Environments

The default depends on implementation…

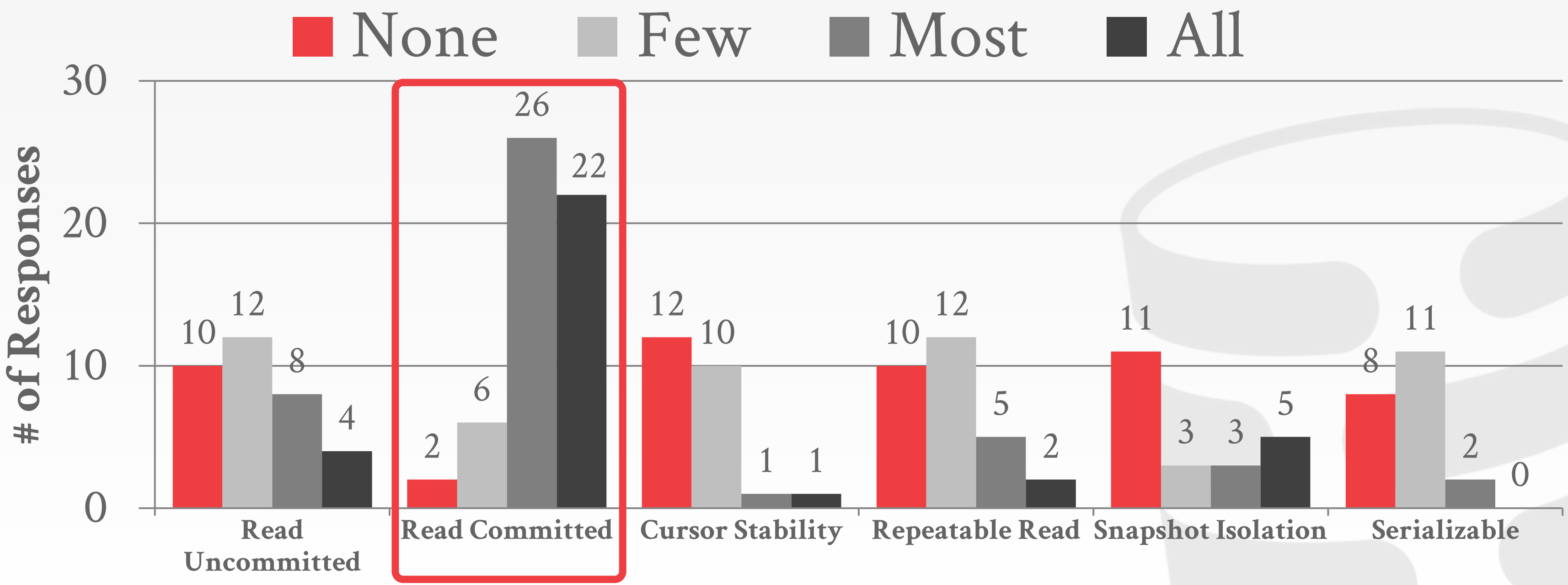
ISOLATION LEVELS (2013)

	Default	*Maximum*
Actian Ingres 10.0/10S	SERIALIZABLE	SERIALIZABLE
Aerospike	READ COMMITTED	READ COMMITTED
Greenplum 4.1	READ COMMITTED	SERIALIZABLE
MySQL 5.6	REPEATABLE READS	SERIALIZABLE
MemSQL 1b	READ COMMITTED	READ COMMITTED
MS SQL Server 2012	READ COMMITTED	SERIALIZABLE
Oracle 11g	READ COMMITTED	SNAPSHOT ISOLATION
Postgres 9.2.2	READ COMMITTED	SERIALIZABLE
SAP HANA	READ COMMITTED	SERIALIZABLE
ScaleDB 1.02	READ COMMITTED	READ COMMITTED
VoltDB	SERIALIZABLE	SERIALIZABLE

Source: Peter Bailis

DATABASE ADMIN SURVEY

What isolation level do transactions execute at on this DBMS?



SQL-92 ACCESS MODES

You can provide hints to the DBMS about whether a txn will modify the database during its lifetime.

Only two possible modes:

→ **READ WRITE** (Default)

→ **READ ONLY**

Not all DBMSs will optimize execution if you set a txn to in **READ ONLY** mode.

```
SET TRANSACTION <access-mode>;
```

```
BEGIN TRANSACTION <access-mode>;
```

CONCLUSION

Every concurrency control can be broken down into the basic concepts that I've described in the last two lectures.

I'm not showing benchmark results because I don't want you to get the wrong idea.

NEXT CLASS

Multi-Version Concurrency Control