Carnegie Mellon University

Intro to Database Systems (15-445/645)

Lecture #17

Timestamp Ordering Concurrency Control

FALL 2023 » Prof. Andy Pavlo • Prof. Jignesh Patel

CONCURRENCY CONTROL APPROACHES

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

Timestamp Ordering
→ A serialization mechanism using timestamps.

Pessimistic

Optimistic Concurrency Control
→ Run then check for serialization violations.

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 the 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/Wall 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

Don't read stuff from the "future."

Action: Transaction T_i wants to read object X.

If $TS(T_i) < W\text{-}TS(X)$, this violates the 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

Can't write if a future transaction has read or written to the object.

Action: Transaction T_i wants to write object X.

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

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

TIME

$TS(T_1)=1$

	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

TIME

$TS(T_1)=1$

	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	1	0

BASIC T/O – EXAMPLE #1

Schedule

$TS(T_1)=1$

$TS(T_2)=2$

TIME

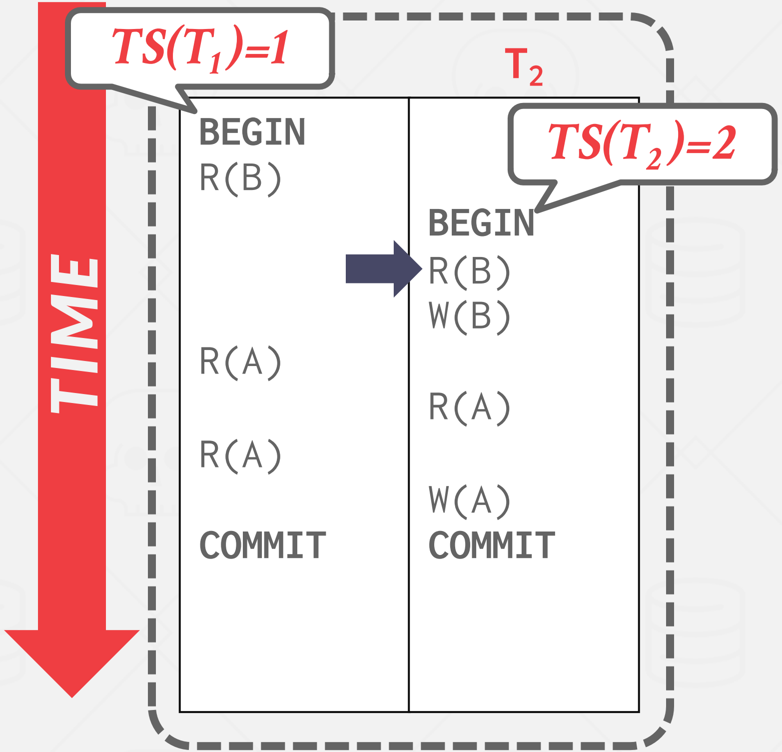
T_1	T_2
BEGIN	
R(B)	
	BEGIN
	R(B)
	W(B)
R(A)	
	R(A)
R(A)	
	W(A)
COMMIT	COMMIT

Database

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

BASIC T/O – EXAMPLE #1

Schedule

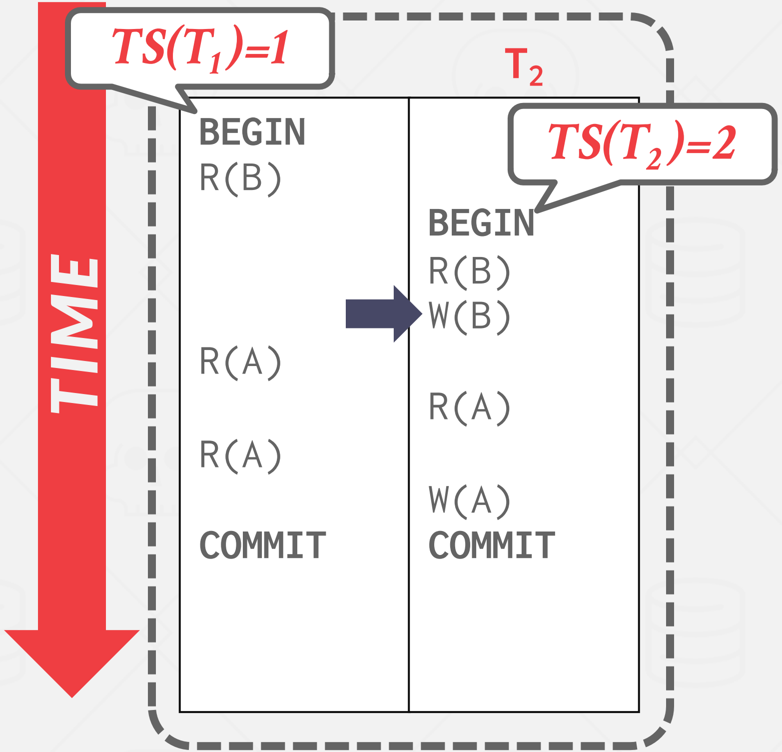


Database

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

BASIC T/O – EXAMPLE #1

Schedule



Database

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

BASIC T/O – EXAMPLE #1

Schedule

$TS(T_1)=1$

$TS(T_2)=2$

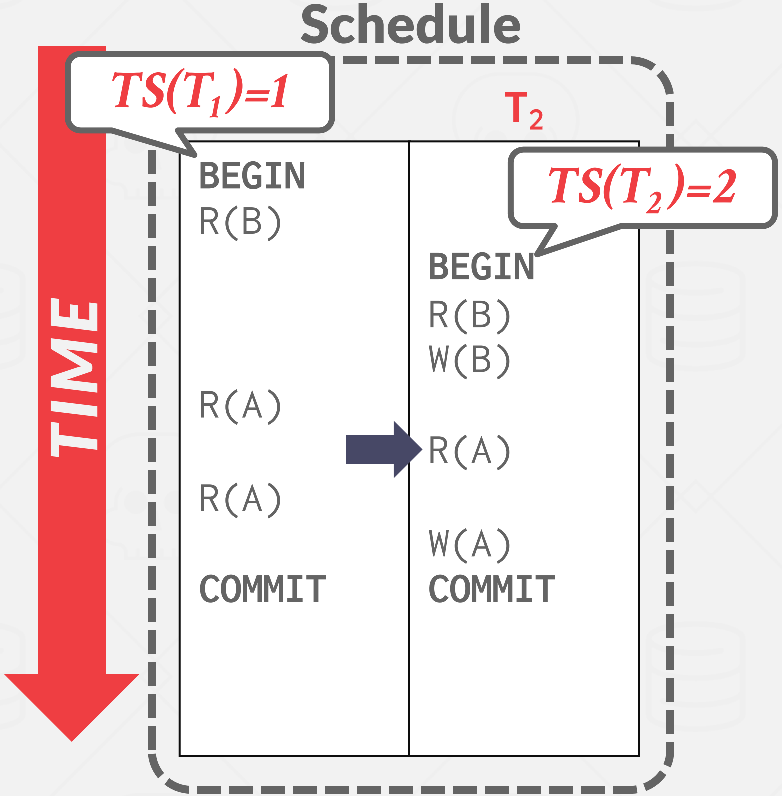
TIME

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



Database

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

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

$TS(T_1) < TS(T_2)$

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

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

BASIC T/O – EXAMPLE #1

Schedule

$TS(T_1)=1$

$TS(T_2)=2$

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

No violations so both txns are safe to commit.

Database

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

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	0	0
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	

Database

Object	R-TS	W-TS
A	1	0
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	

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	

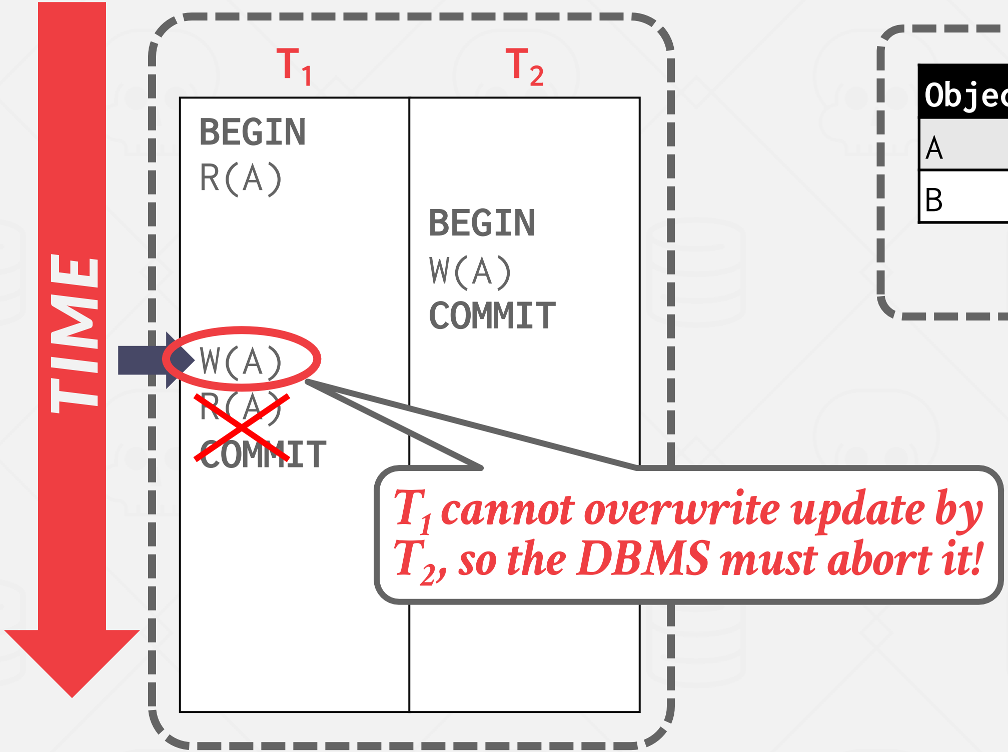
Database

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

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

BASIC T/O – EXAMPLE #2

Schedule



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 $W\text{-}TS(X)$

If **TS(T**

→ Abo

If **TS(T**

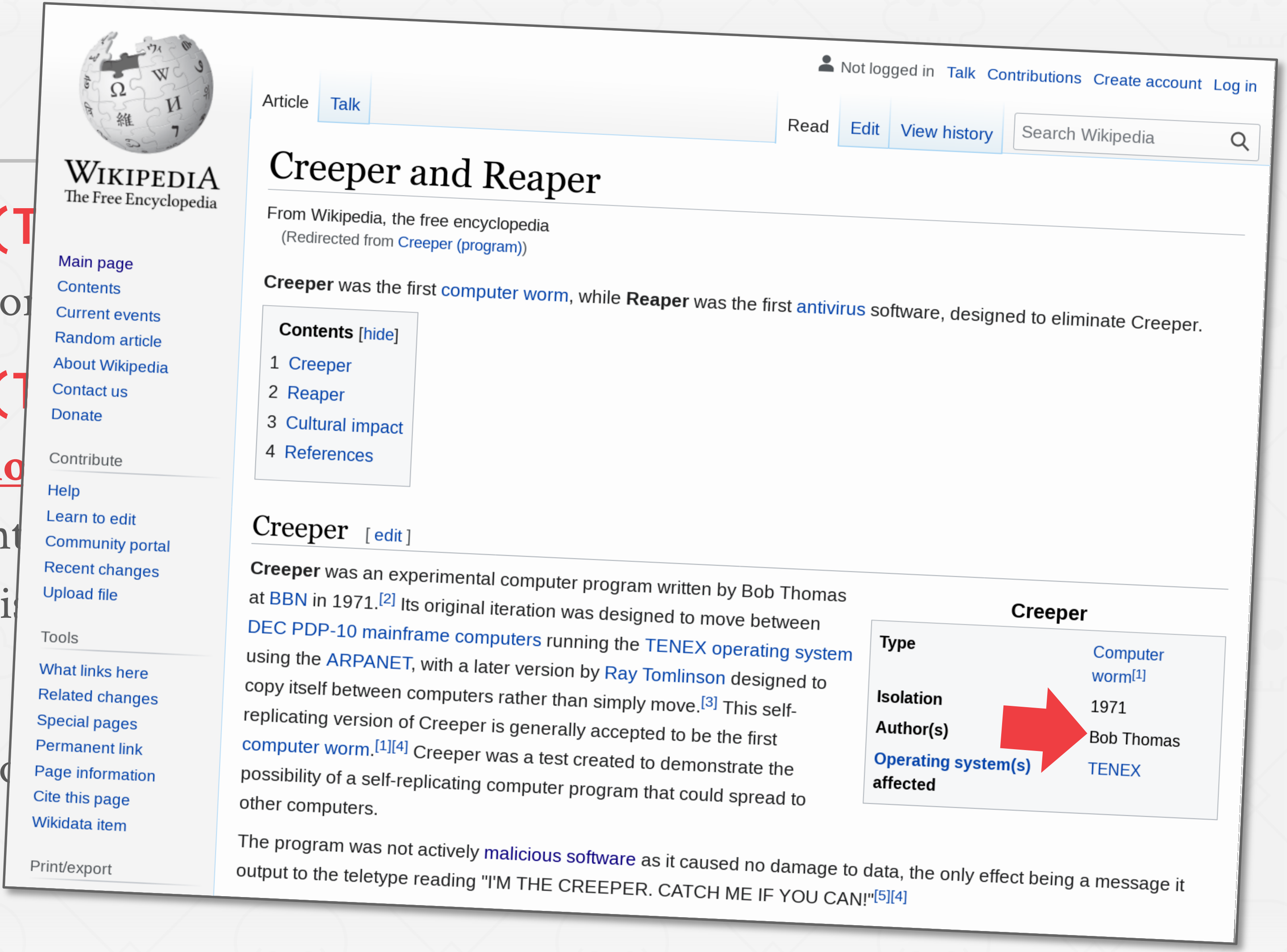
→ **Tho**

cont

→ Thi

Else:

→ All



WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate

Contribute
Help
Learn to edit
Community portal
Recent changes
Upload file

Tools
What links here
Related changes
Special pages
Permanent link
Page information
Cite this page
Wikidata item

Print/export

Not logged in Talk Contributions Create account Log in

Article Talk Read Edit View history Search Wikipedia

Creeper and Reaper

From Wikipedia, the free encyclopedia
(Redirected from Creeper (program))

Creeper was the first computer worm, while **Reaper** was the first antivirus software, designed to eliminate Creeper.

Contents [hide]

Creeper [edit]

Creeper was an experimental computer program written by Bob Thomas at BBN in 1971.[2] Its original iteration was designed to move between DEC PDP-10 mainframe computers running the TENEX operating system using the ARPANET, with a later version by Ray Tomlinson designed to copy itself between computers rather than simply move.[3] This self-replicating version of Creeper is generally accepted to be the first computer worm.[1][4] Creeper was a test created to demonstrate the possibility of a self-replicating computer program that could spread to other computers.

Creeper

Type	Computer worm[1]
Isolation	1971
Author(s)	Bob Thomas
Operating system(s) affected	TENEX

The program was not actively malicious software as it caused no damage to data, the only effect being a message it output to the teletype reading "I'M THE CREEPER. CATCH ME IF YOU CAN!"[5][4]

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

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	2
B	0	0

BASIC T/O – EXAMPLE #2

Schedule

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

TIME

Skip doing the actual write and allow T_1 to commit. (Do write to the local copy if repeatable read is required.)

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.

Not aware of any DBMS that uses the basic T/O protocol described here.

→ It provides the building blocks for OCC / MVCC.

BASIC T/O – PERFORMANCE ISSUES

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

→ Every read requires the txn to write to the database.

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 acquire locks or update timestamps adds unnecessary 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:

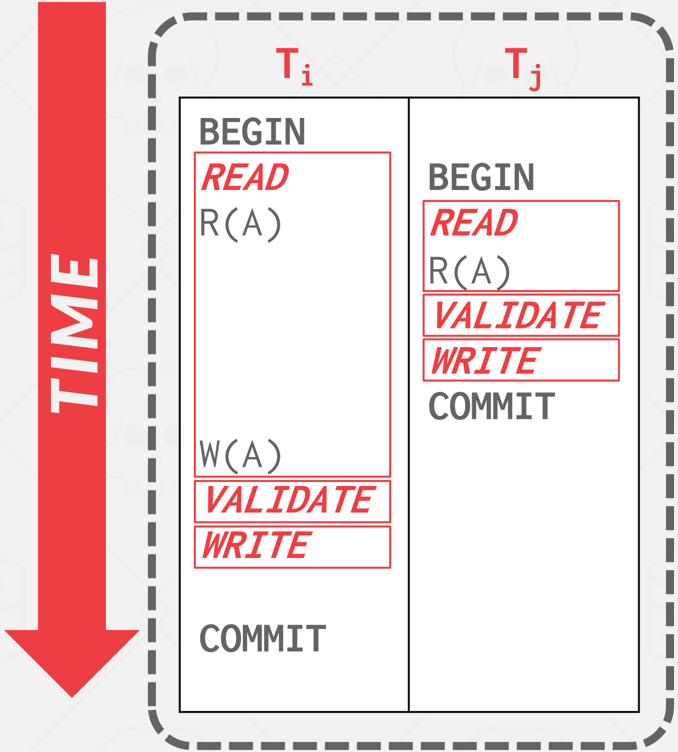
→ When a txn commits, check whether it conflicts with other txns.

#3 – Write Phase:

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

OCC – EXAMPLE

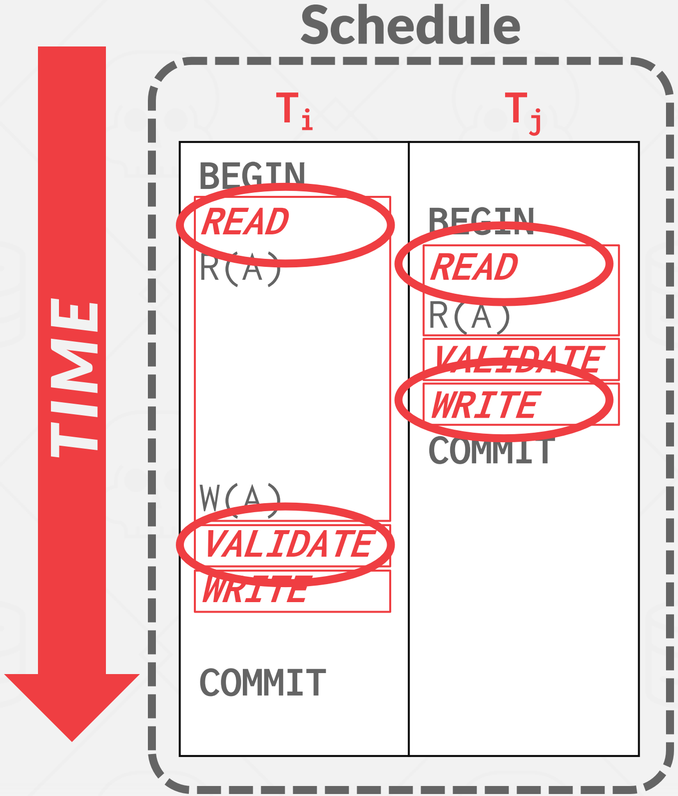
Schedule



Database

Object	Value	W-TS
A	123	0
–	–	–

OCC – EXAMPLE

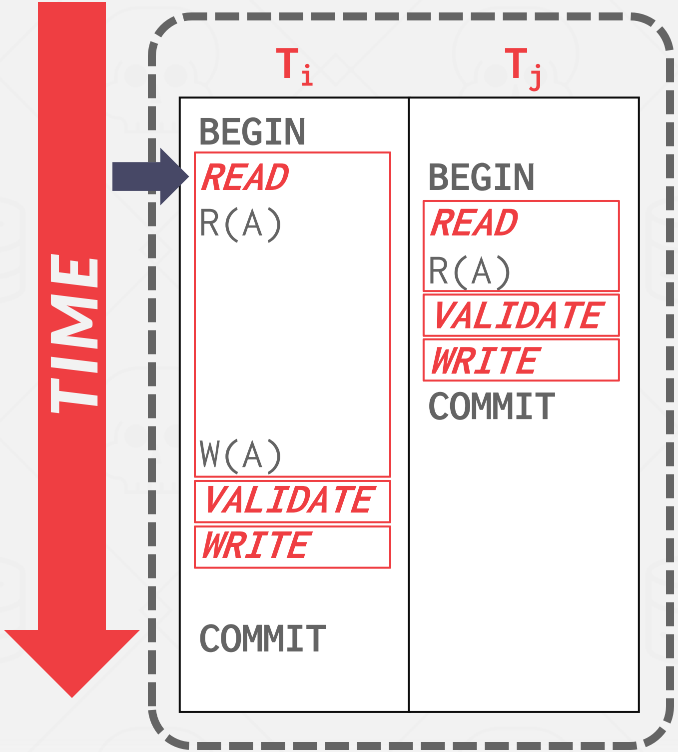


Database

Object	Value	W-TS
A	123	0
–	–	–

OCC – EXAMPLE

Schedule



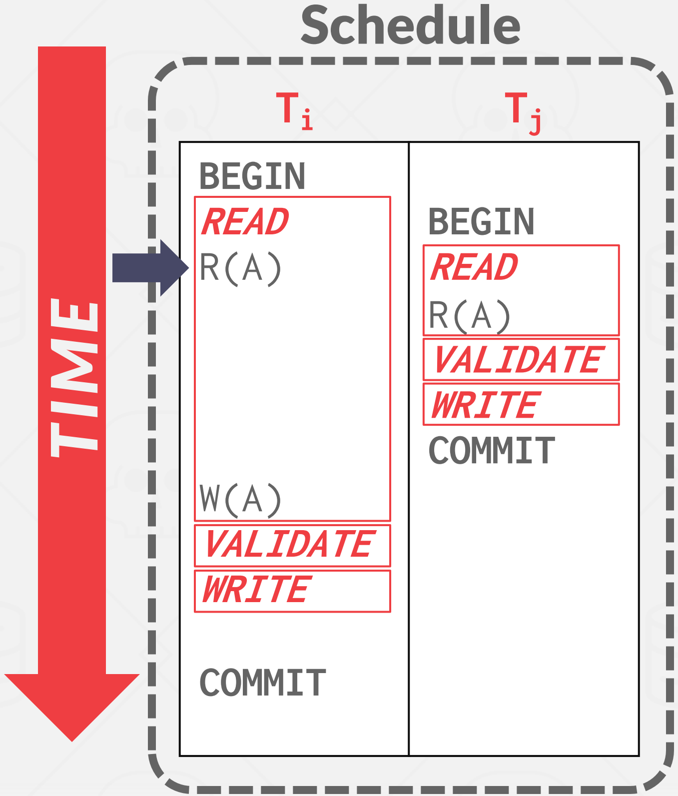
Database

Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

Object	Value	W-TS
–	–	–
–	–	–

OCC – EXAMPLE



Database

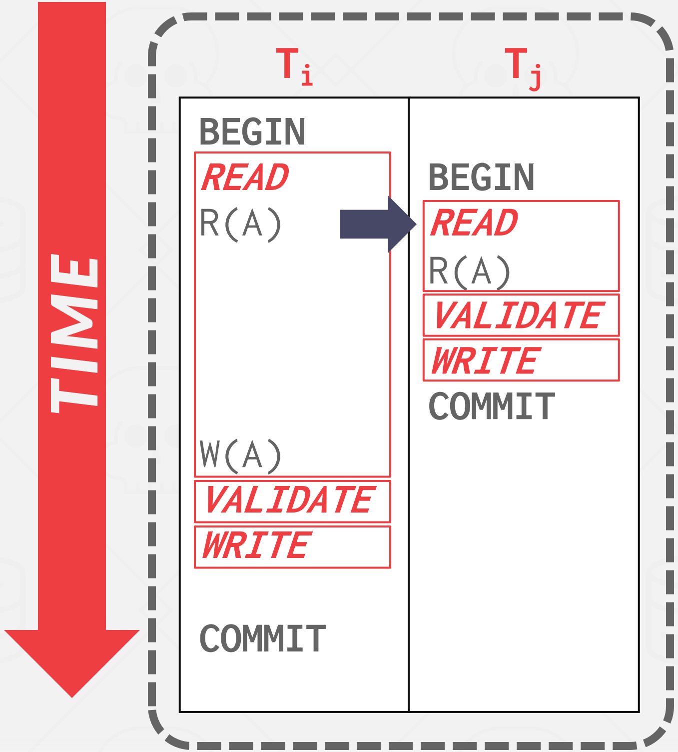
Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

Object	Value	W-TS
A	123	0
–	–	–

OCC – EXAMPLE

Schedule



Database

Object	Value	W-TS
A	123	0
–	–	–

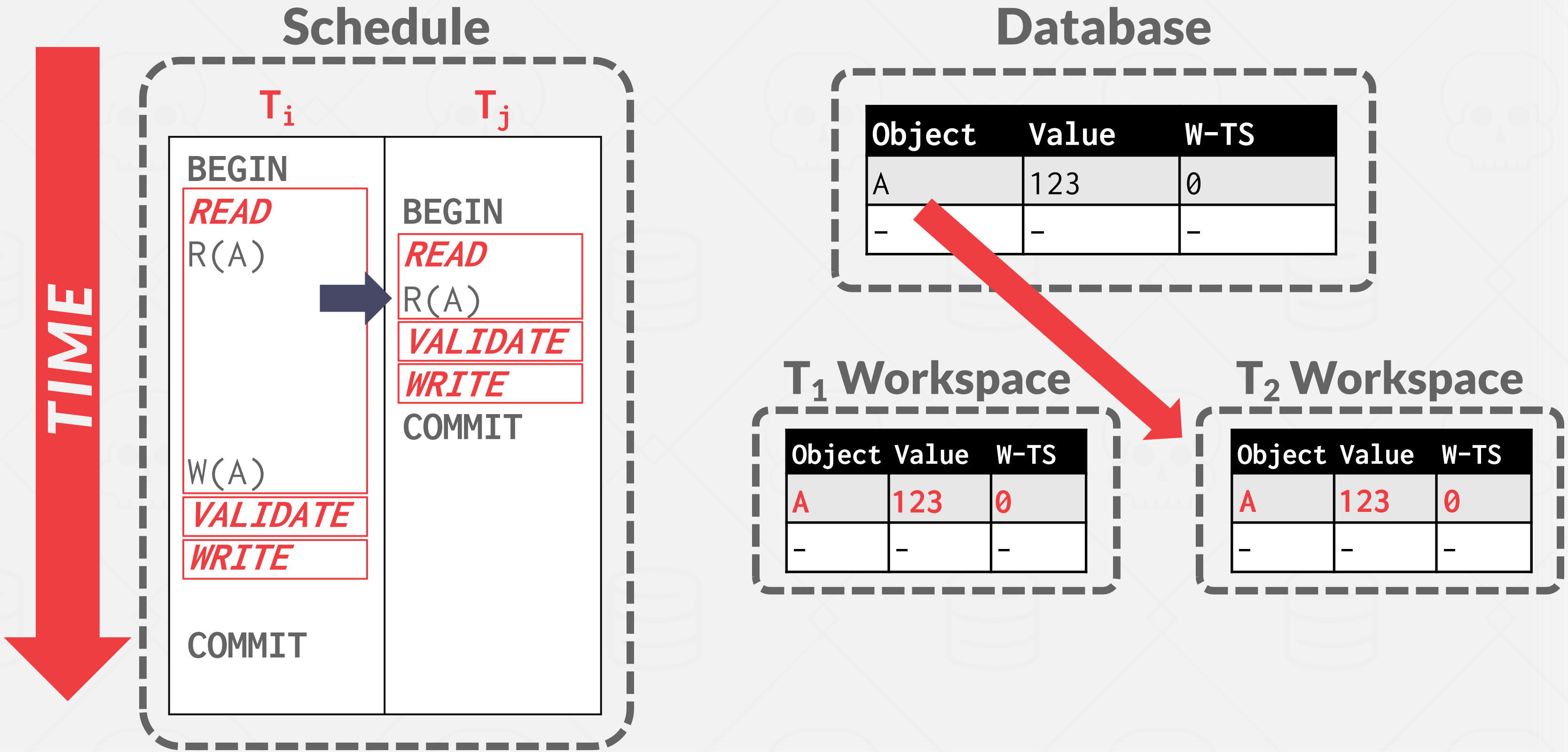
T_1 Workspace

Object	Value	W-TS
A	123	0
–	–	–

T_2 Workspace

Object	Value	W-TS
–	–	–
–	–	–

OCC – EXAMPLE


Schedule
TIME
T_i
BEGIN
READ
R(A)
W(A)
VALIDATE
WRITE
COMMIT
T_j
BEGIN
READ
R(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

TIME

T_i	T_j
BEGIN	
READ	
R(A)	
	BEGIN
	READ
	R(A)
	VALIDATE
	WRITE
	COMMIT
W(A)	
VALIDATE	
WRITE	
COMMIT	

$TS(T_j)=1$

Database

Object	Value	W-TS
A	123	0
–	–	–

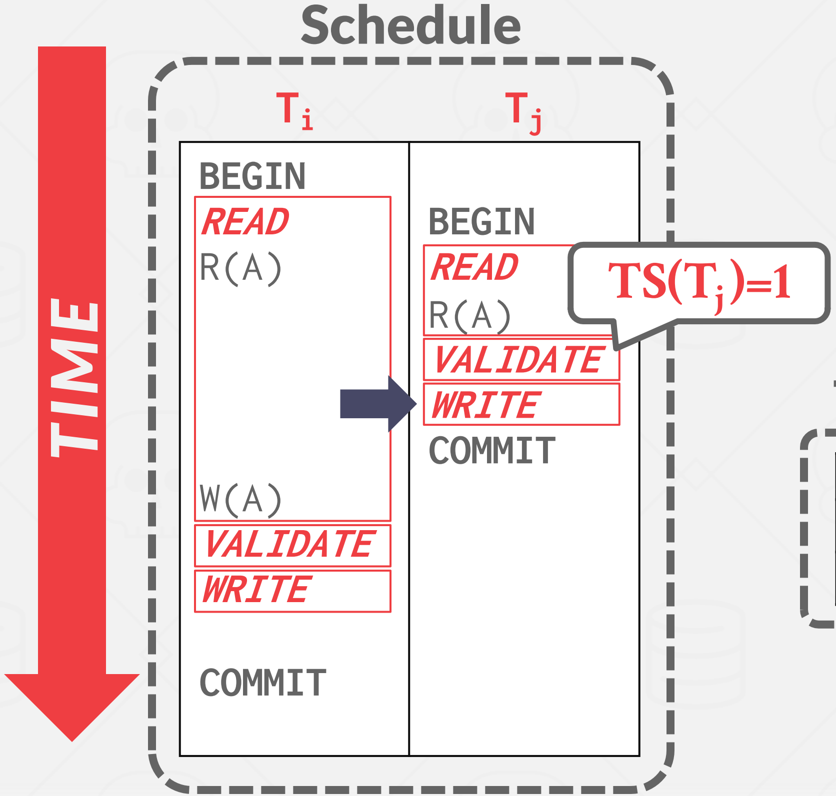
T_1 Workspace

Object	Value	W-TS
A	123	0
–	–	–

T_2 Workspace

Object	Value	W-TS
A	123	0
–	–	–

OCC – EXAMPLE



Database

Object	Value	W-TS
A	123	0
–	–	–

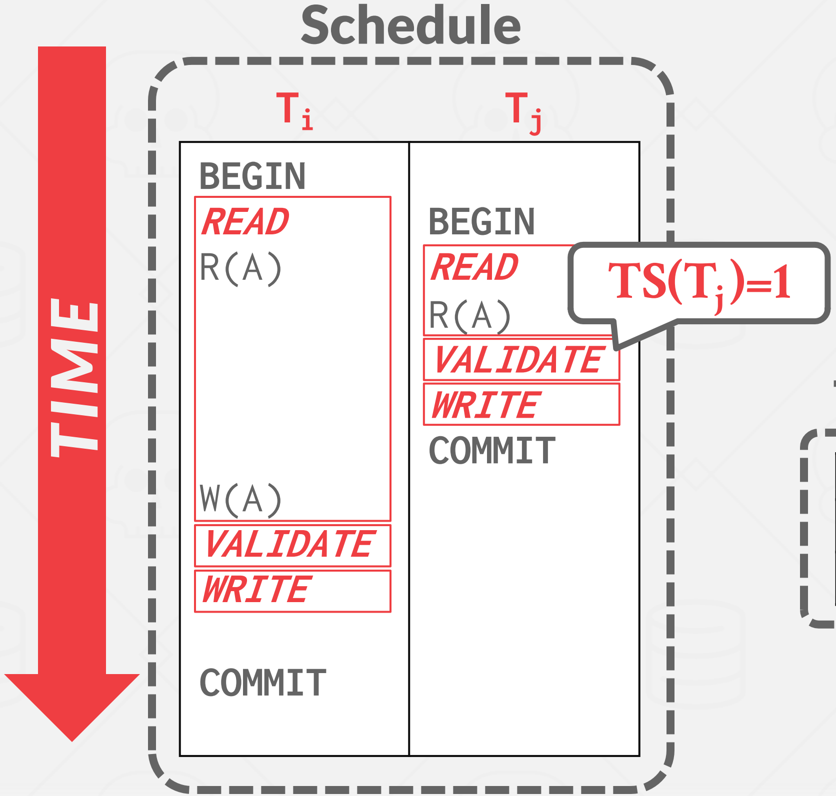
T_1 Workspace

Object	Value	W-TS
A	123	0
–	–	–

T_2 Workspace

Object	Value	W-TS
A	123	0
–	–	–

OCC – EXAMPLE



Database

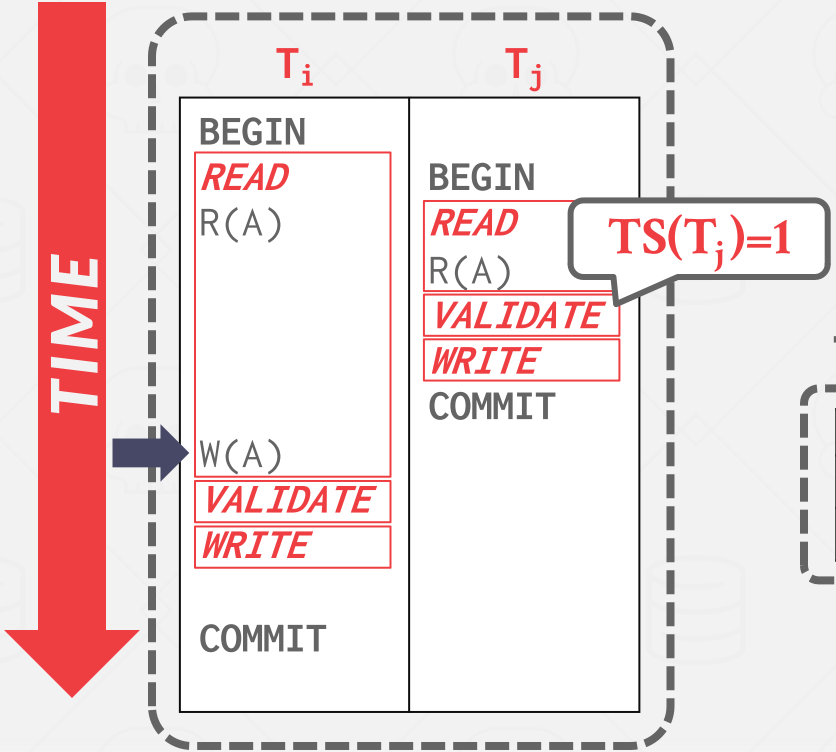
Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

Object	Value	W-TS
A	123	0
–	–	–

OCC – EXAMPLE

Schedule



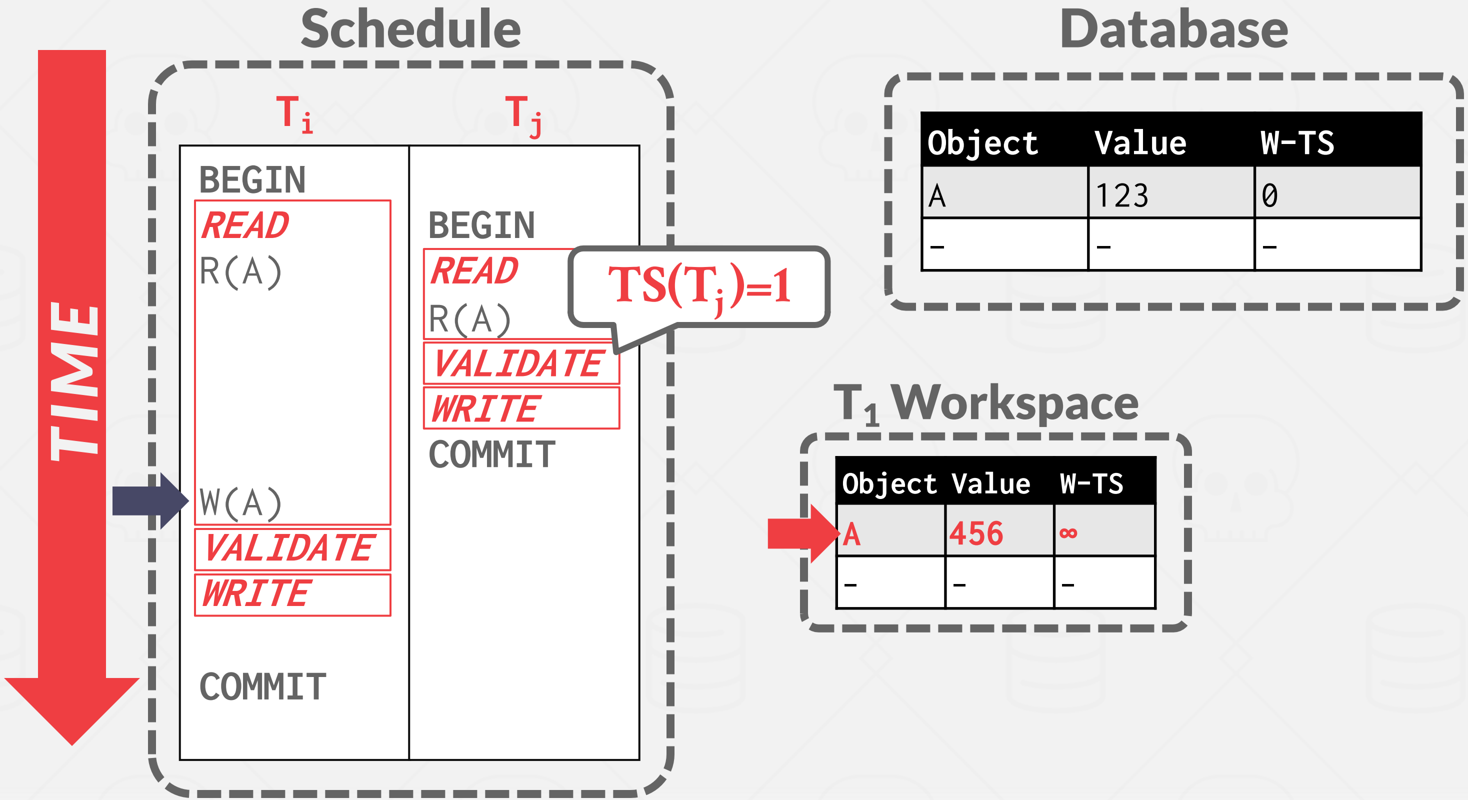
Database

Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

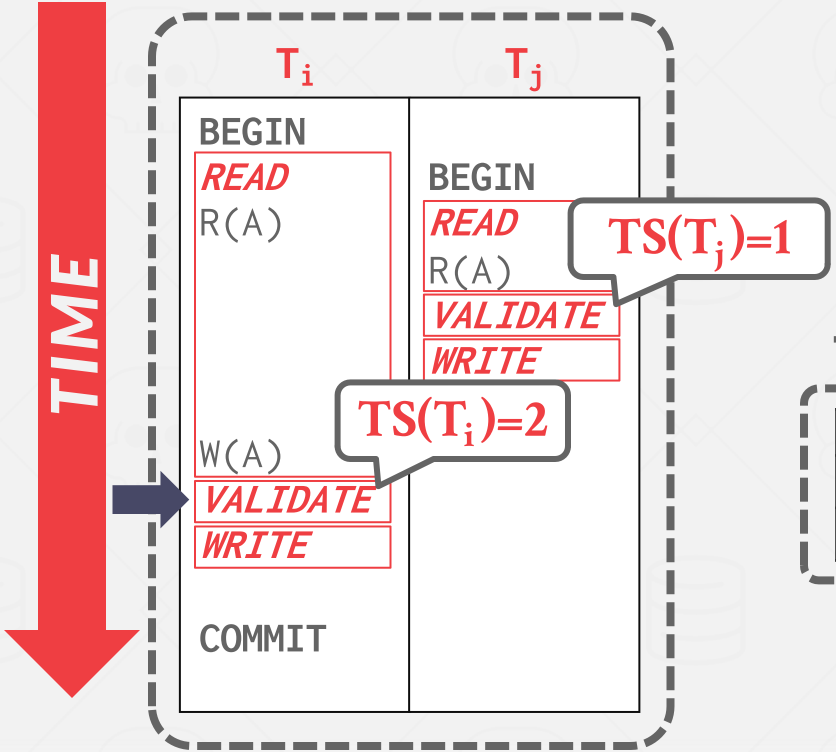
Object	Value	W-TS
A	123	0
–	–	–

OCC – EXAMPLE


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

OCC – EXAMPLE

Schedule



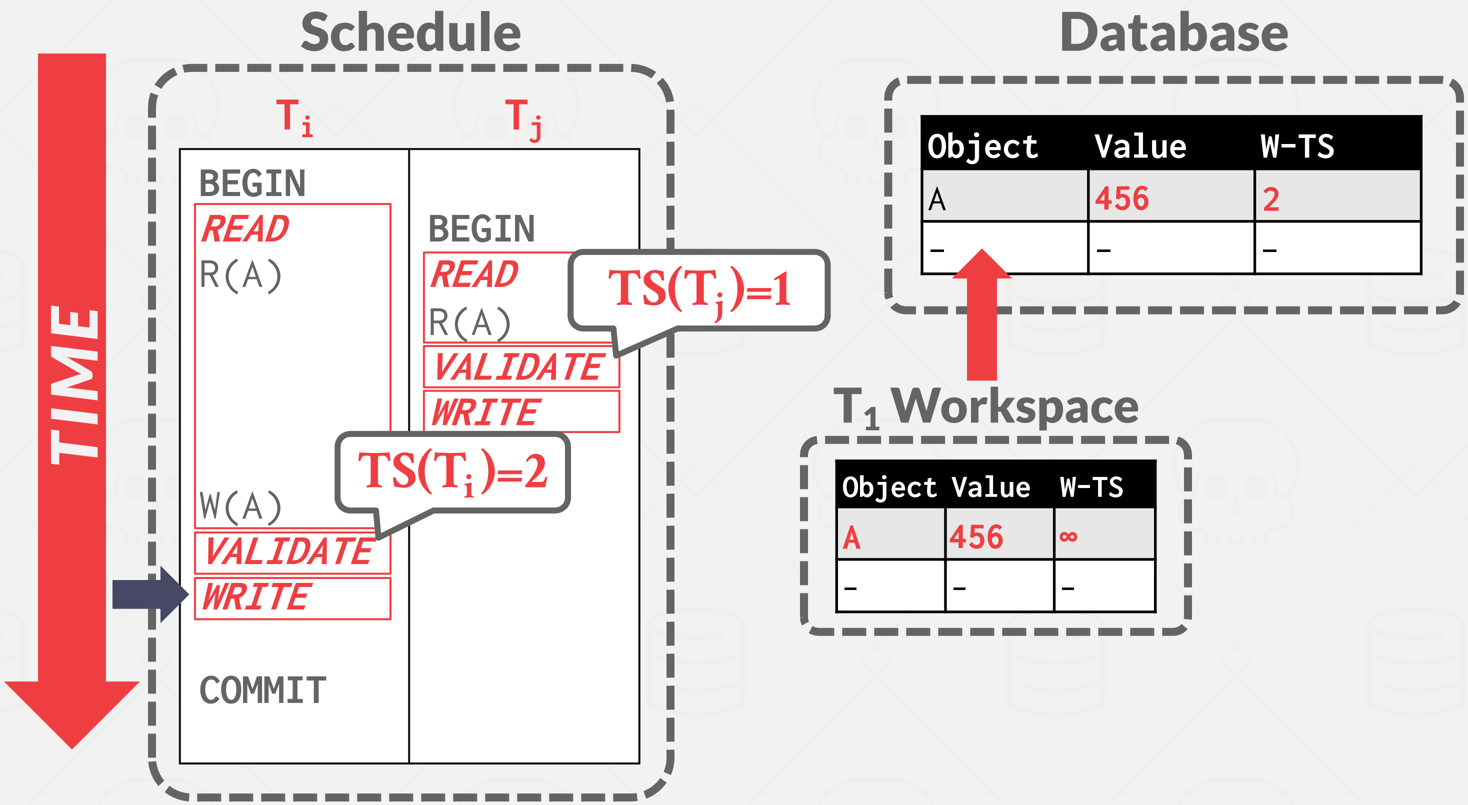
Database

Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

Object	Value	W-TS
A	456	∞
–	–	–

OCC – EXAMPLE



Schedule
TIME
T_i
T_j
BEGIN
READ
R(A)
W(A)
VALIDATE
WRITE
COMMIT
BEGIN
READ
R(A)
VALIDATE
WRITE
TS(T_j)=1
TS(T_i)=2
Database
Object Value W-TS
A 456 2
– – –
T_1 Workspace
Object Value W-TS
A 456 ∞
– – –

OCC – READ PHASE

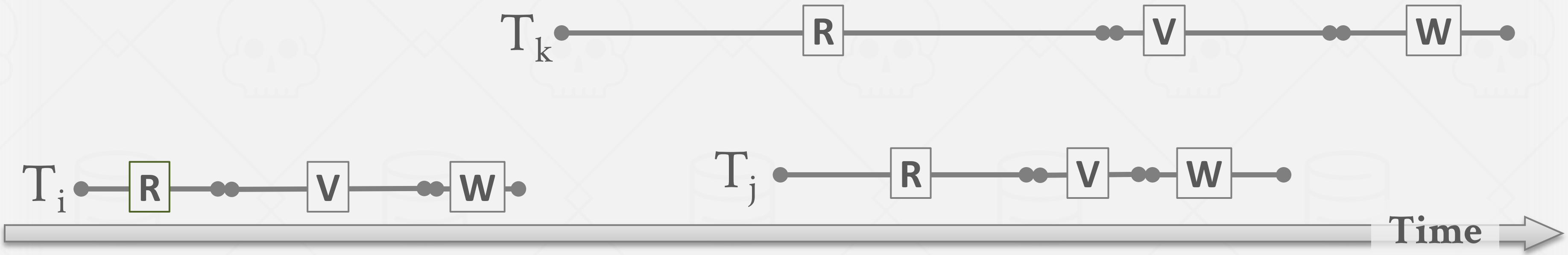
Track the read/write sets of txns and store their writes in a private workspace.

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

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

OCC: THREE PHASES

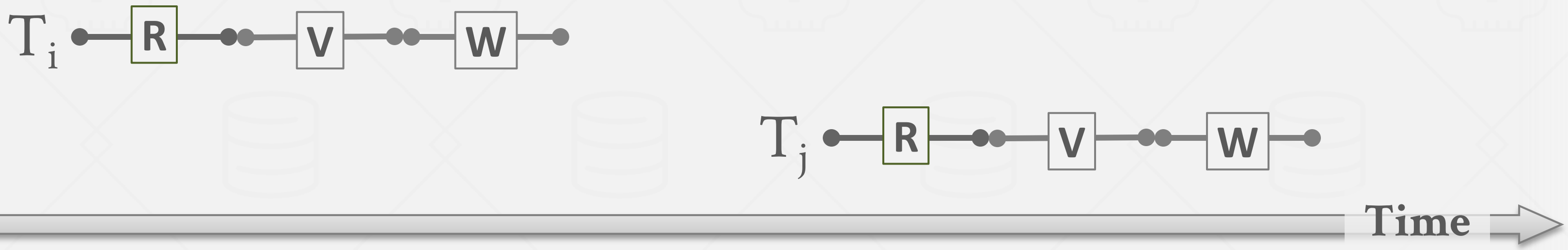
When to assign the transaction number? At the end of the read phase.



1. **READ** phase: Read and write objects, making local copies.
2. **VALIDATION** Phase: Check for serializable schedule-related anomalies.
3. **WRITE** Phase: It is safe. Write the local objects, making them permanent.

OCC: VALIDATION (T_i < T_j)

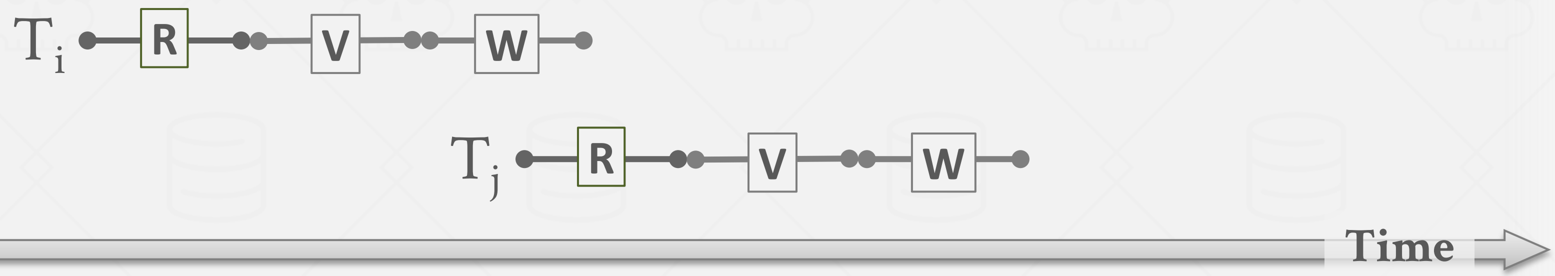
Case 1: T_i completes its write phase before T_j starts its read phase.



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

OCC: VALIDATION (T_i < T_j)

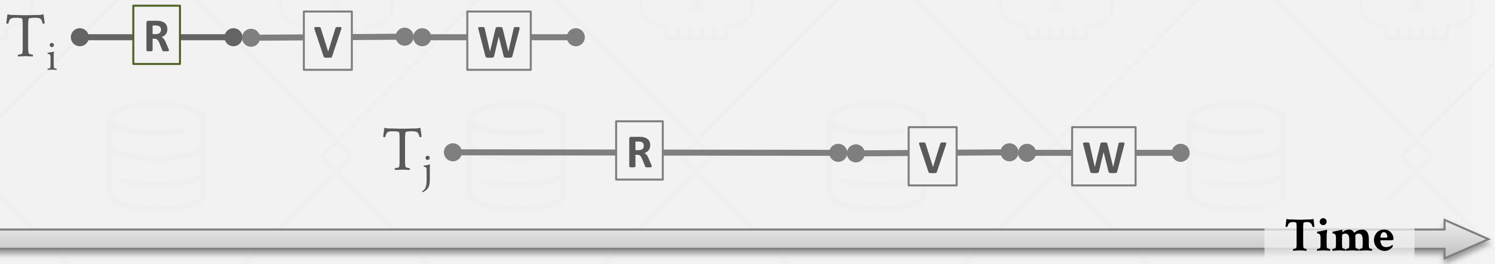
Case 2: T_i completes its write phase before T_j starts its write phase.



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

OCC: VALIDATION (T_i < T_j)

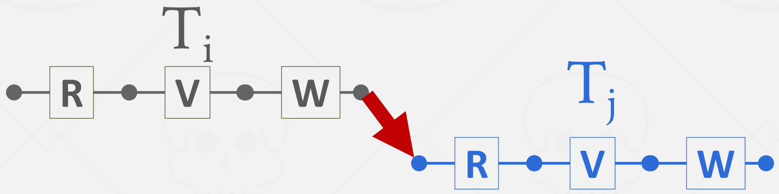
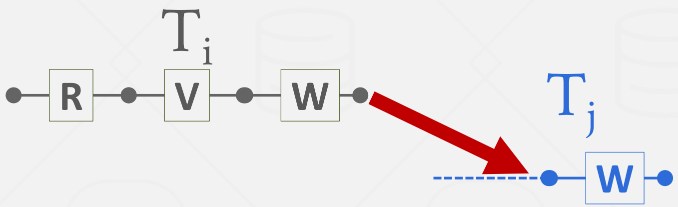
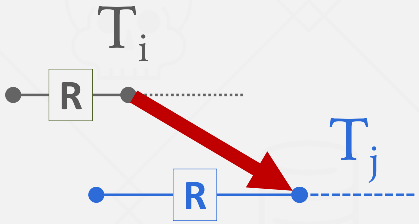
Case 3: T_i completes its **read** phase before T_j completes its **read** phase.



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

AND $\texttt{WriteSet}(T_i) \cap \texttt{WriteSet}(T_j) = \emptyset$

OCC: VALIDATION ($T_i < T_j$)

	$R \rightarrow W$	$W \rightarrow R$	$W \rightarrow W$
Case 1 	✓	✓	✓
Case 2 	✓	$WriteSet(T_i) \cap ReadSet(T_j) = \emptyset$	✓
Case 3 	✓	$WriteSet(T_i) \cap ReadSet(T_j) = \emptyset$	$WriteSet(T_i) \cap WriteSet(T_j) = \emptyset$

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 primary 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.

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 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(age) FROM people WHERE status='lit'	→ 99
	INSERT INTO people (age=30, status='lit')
	COMMIT
SELECT COUNT(age) FROM people WHERE status='lit'	→ 100
COMMIT	

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

OOPS?

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

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

Approach #2: Predicate Locking

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

Approach #3: Index Locking

→ 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.

PREDICATE LOCKING

Proposed locking scheme from System R.

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

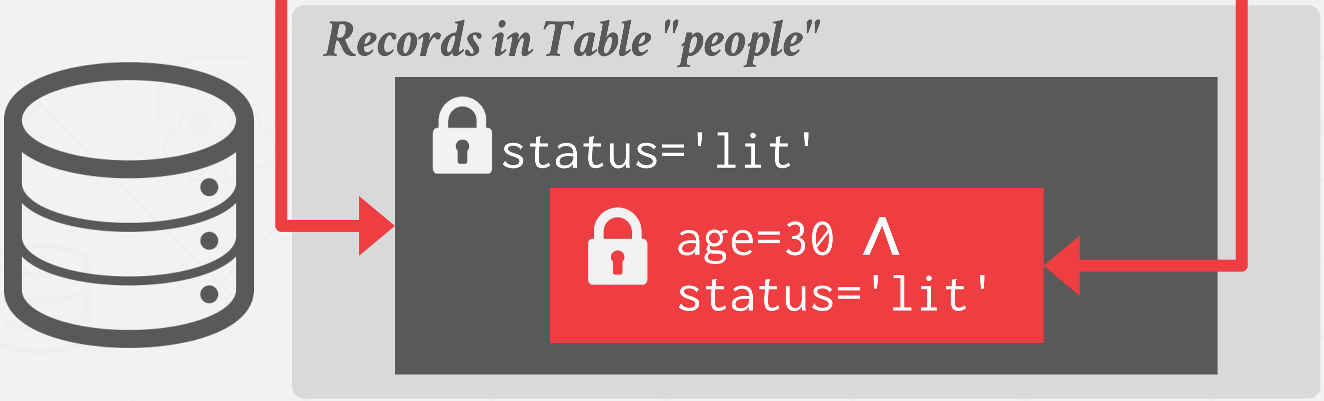
→ Exclusive lock on the predicate in a **WHERE** clause of any **UPDATE**, **INSERT**, or **DELETE** query.

It is rarely implemented in systems; an example of a system that uses it is HyPer (precision locking).

PREDICATE LOCKING

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

```
INSERT INTO people VALUES
(age=30, status='lit')
```



INDEX LOCKING SCHEMES

Key-Value Locks

Gap Locks

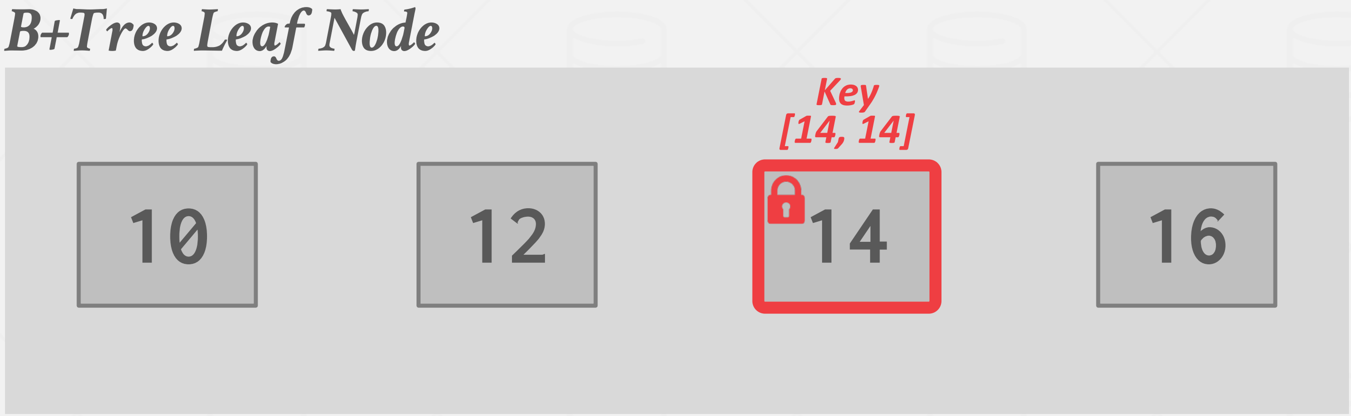
Key-Range Locks

Hierarchical Locking

KEY-VALUE LOCKS

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

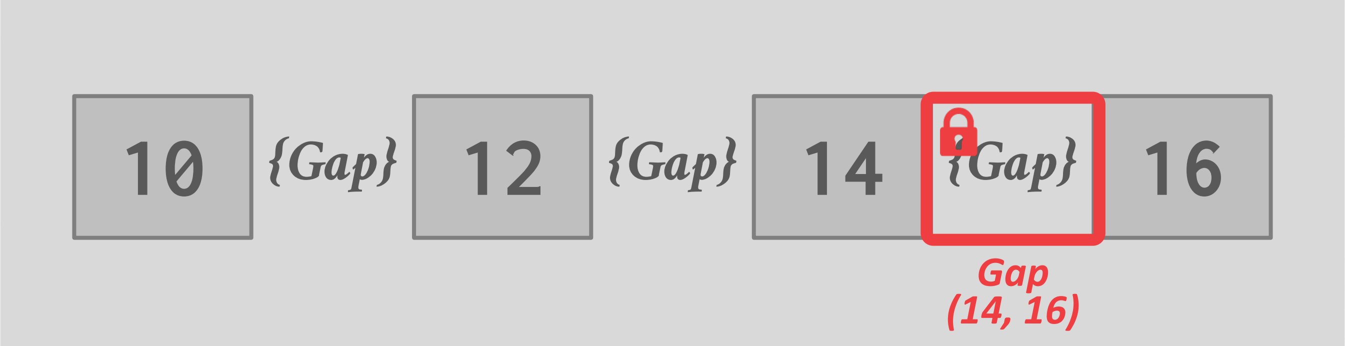
Need "virtual keys" for non-existent values.



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.

B+Tree Leaf Node



KEY-RANGE LOCKS

A txn takes locks on ranges in the key space.

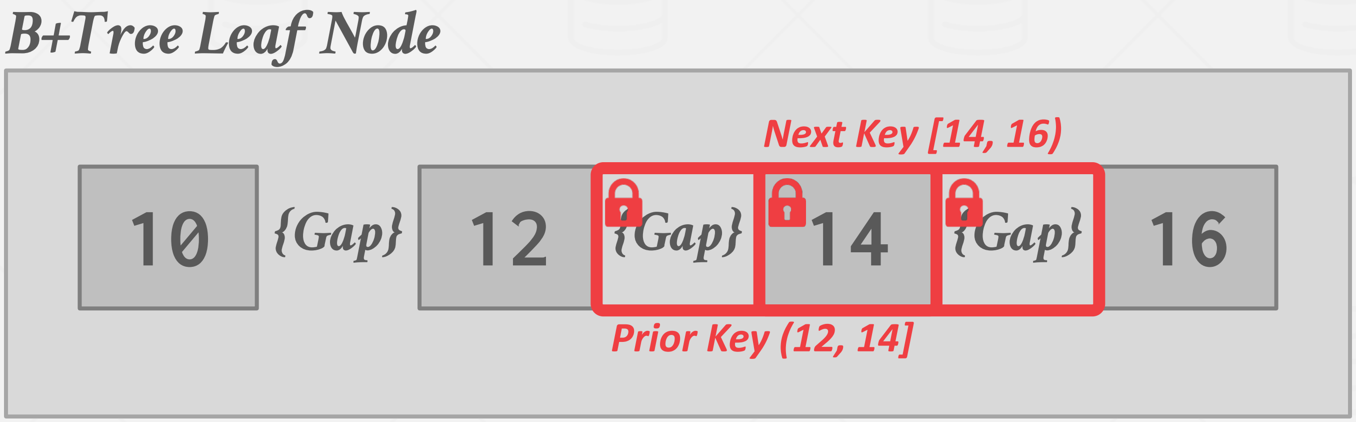
→ Each range is from one key that appears in the relation, to the next that appears.

→ Define lock modes so conflict table will capture commutativity of the operations available.

KEY-RANGE LOCKS

Locks that cover a key value and the gap to the next key value in a single index.

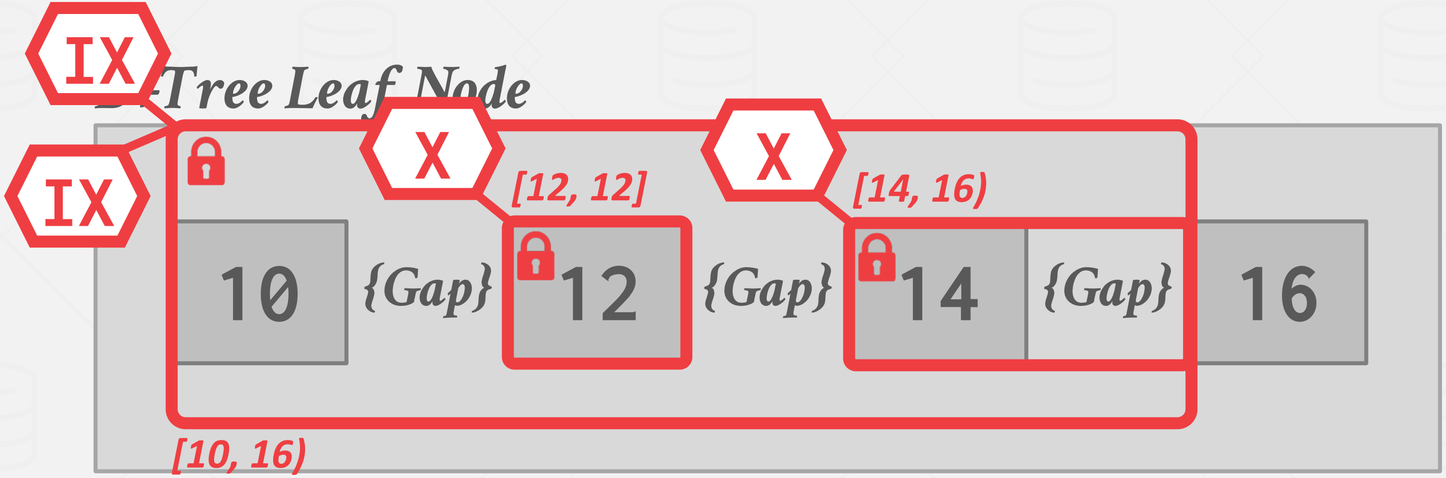
→ Need "virtual keys" for artificial values (infinity)



HIERARCHICAL LOCKING

Allow for a txn to hold wider key-range locks with different locking modes.

→ Reduces the number of visits to lock manager.



LOCKING WITHOUT AN INDEX

If there is no suitable index, then to avoid phantoms 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

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 (Low→High)

SERIALIZAB

dirty reads.

REPEATABLE

READ COMMIT

may happen.

READ UNCOMM


United States Department of Justice

Offices of the United States Attorneys

THE UNITED STATES ATTORNEY'S OFFICE

SOUTHERN DISTRICT of NEW YORK

Search

SEARCH

HOME | ABOUT | PRIORITIES | NEWS | RESOURCES | PROGRAMS | EMPLOYMENT | CONTACT

U.S. Attorneys » Southern District of New York » News » Press Releases

SHARE

Department of Justice

U.S. Attorney's Office

Southern District of New York

FOR IMMEDIATE RELEASE

Monday, November 7, 2022

U.S. Attorney Announces Historic $3.36 Billion Cryptocurrency Seizure And Conviction In Connection With Silk Road Dark Web Fraud

In November 2021, Law Enforcement Seized Over 50,676 Bitcoin Hidden in Devices in Defendant JAMES ZHONG's Home; ZHONG Has Now Pled Guilty to Unlawfully Obtaining that Bitcoin From the Silk Road Dark Web in 2012

Damian Williams, the United States Attorney for the Southern District of New York, and Tyler Hatcher, the Special Agent in Charge of the Internal Revenue Service, Criminal Investigation, Los Angeles Field Office ("IRS-CI"), announced today that JAMES ZHONG pled guilty to committing wire fraud in September 2012 when he unlawfully obtained over 50,000 Bitcoin from the Silk Road dark web internet marketplace. ZHONG pled guilty on Friday, November 4, 2022, before United States District Judge Paul G. Gardephe.

On November 9, 2021, pursuant to a judicially authorized premises search warrant of ZHONG's Gainesville, Georgia, house, law enforcement seized approximately 50,676.17851897 Bitcoin, then valued at over $3.36 billion. This seizure was then the largest cryptocurrency seizure in the history of the U.S. Department of Justice and today remains the Department's second largest financial seizure ever. The Government is seeking to forfeit, collectively: approximately 51,680.32473733 Bitcoin; ZHONG's 80% interest in RE&D Investments, LLC, a Memphis-based company with substantial real estate holdings; $661,900 in cash seized from ZHONG's home; and various metals also seized from ZHONG's home.

Click here to report information on Amazon warehouses.

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 strong 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>;
```

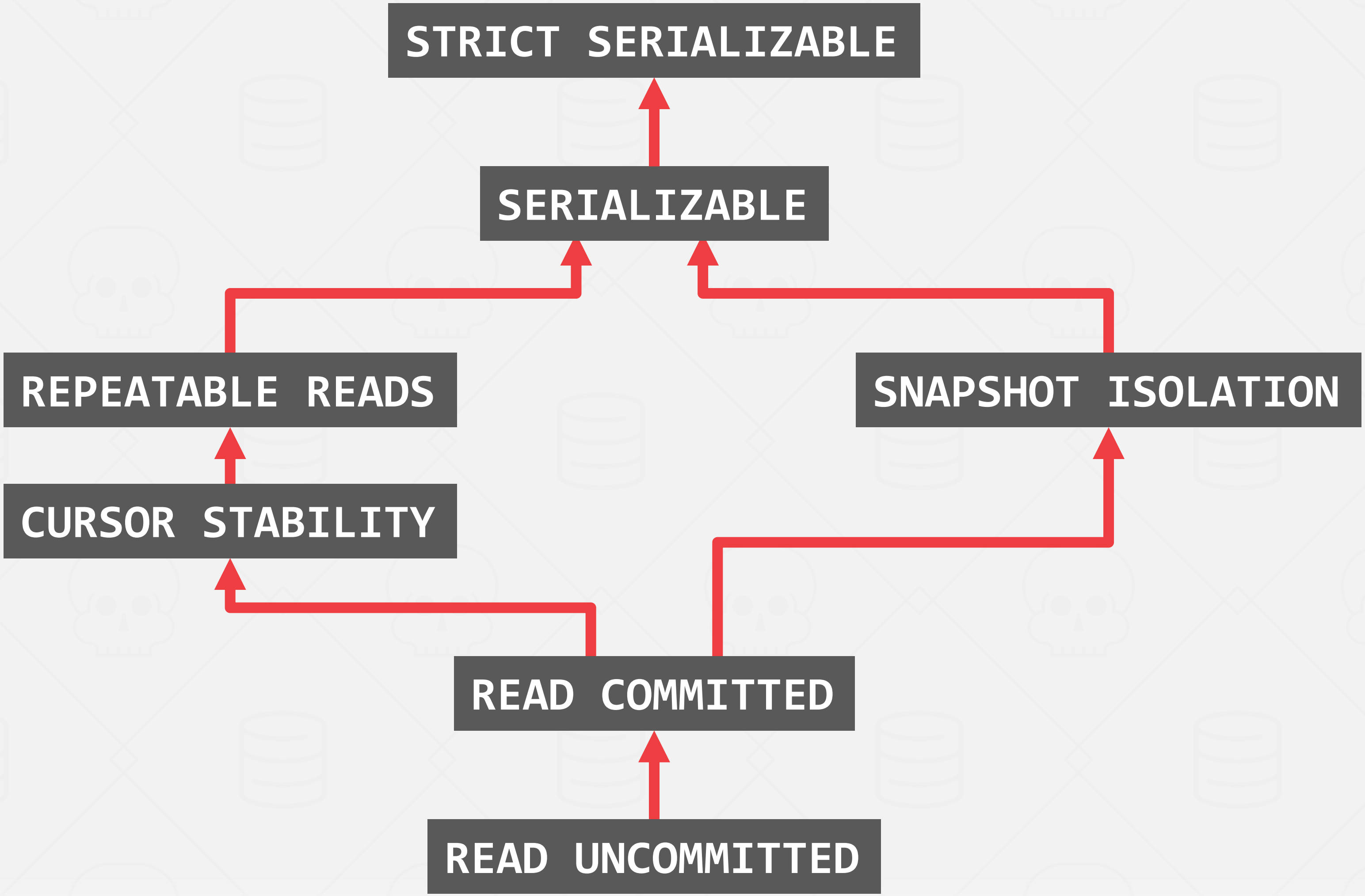
Not all DBMS support all isolation levels in all execution scenarios

→ Replicated Environments

The default depends on implementation…

ISOLATION LEVELS

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

Every protocol has pros and cons.

CRITIQUE OF SQL ISOLATION LEVELS

"ANSI SQL-92 … defines Isolation Levels in terms of phenomena: Dirty Reads, Non-Repeatable Reads, and Phantoms. … these phenomena and the ANSI SQL definitions fail to characterize several popular isolation levels, including the standard locking implementations of the levels. Investigating the ambiguities of the phenomena leads to clearer definitions; in addition new phenomena that better characterize isolation types are introduced. An important multiversion isolation type, Snapshot Isolation, is defined."

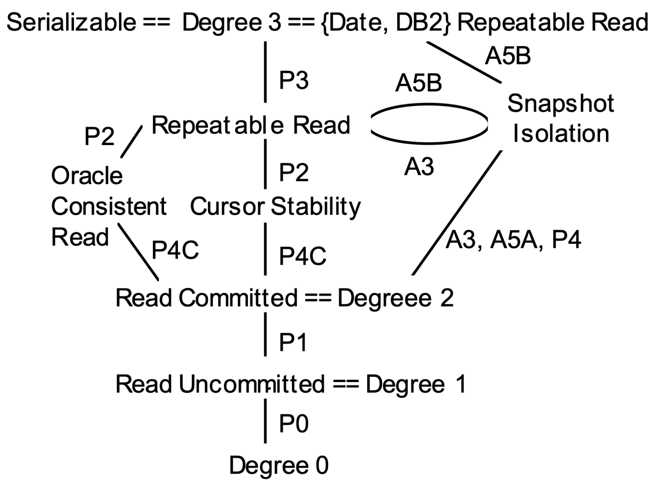
A Critique of ANSI SQL Isolation Levels

Hal Berenson, Microsoft Corp., haroldb@microsoft.com
Phil Bernstein, Microsoft Corp., philbe@microsoft.com
Jim Gray, U.C. Berkeley, gray@crl.com
Jim Melton, Sybase Corp., jim.melton@sybase.com
Elizabeth O'Neil, UMass/Boston, eoneil@cs.umb.edu
Patrick O'Neil, UMass/Boston, poneil@cs.umb.edu

Abstract: ANSI SQL-92 [MS, ANSI] defines Isolation *Levels* in terms of *phenomena:* Dirty Reads, Non-Repeatable Reads, and Phantoms. This paper shows that these phenomena and the ANSI SQL definitions fail to characterize several popular isolation levels, including the standard locking implementations of the levels. Investigating the ambiguities of the phenomena leads to

distinction is not crucial for a general understanding.

The ANSI isolation levels are related to the behavior of lock schedulers. Some lock schedulers allow transactions to vary the scope and duration of their lock requests, thus departing from pure two-phase locking. This idea was introduced by [GLPT], which defined *Degrees of Consistency* in three ways: locking, data-flow graphs, and anomalies.



Section 5 explores some new anomalies to differentiate the isolation levels introduced in Sections 3 and 4. The extended ANSI SQL phenomena proposed here lack the power to characterize Snapshot isolation and Cursor Stability. Section 6 presents a Summary and Conclusions.

"A Critique of ANSI SQL Isolation Levels," Proc. ACM SIGMOD 95, pp. 1-10, San Jose CA, June 1995, © ACM. -1-

NEXT CLASS

Multi-Version Concurrency Control