15-721 DATABASE SYSTEMS

Lecture #04 – Optimistic Concurrency Control

@Andy_Pavlo // Carnegie Mellon University // Spring 2017

ADMINISTRATIVE

Project #1 is due Tuesday Jan 31st @ 11:59pm

Project #2 will be released on Tuesday too.
→ You need a group of three people.
→ I will send out a sign-up sheet.

TODAY'S AGENDA

Isolation Levels

Stored Procedures

Optimistic Concurrency Control

OBSERVATION

Serializability is useful because it allows programmers to ignore concurrency issues but enforcing it may allow too little parallelism 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 Read Anomaly
→ Unrepeatable Reads Anomaly
→ Phantom Reads Anomaly

ANSI 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 LEVEL HIERARCHY

SERIALIZABLE

↑

REPEATABLE READS

↑

READ COMMITTED

↑

READ UNCOMMITTED

ANSI ISOLATION LEVELS

	Default	*Maximum*
Actian Ingres	SERIALIZABLE	SERIALIZABLE
Greenplum	READ COMMITTED	SERIALIZABLE
IBM DB2	CURSOR STABILITY	SERIALIZABLE
MySQL	REPEATABLE READS	SERIALIZABLE
MemSQL	READ COMMITTED	READ COMMITTED
MS SQL Server	READ COMMITTED	SERIALIZABLE
Oracle	READ COMMITTED	SNAPSHOT ISOLATION
Postgres	READ COMMITTED	SERIALIZABLE
SAP HANA	READ COMMITTED	SERIALIZABLE
VoltDB	SERIALIZABLE	SERIALIZABLE

Source: Peter Bailis

CRITICISM OF ISOLATION LEVELS

The isolation levels defined as part of SQL-92 standard only focused on anomalies that can occur in a 2PL-based DBMS.

Two additional isolation levels:

→ **CURSOR STABILITY**

→ **SNAPSHOT ISOLATION**

A CRITIQUE OF ANSI SQL ISOLATION LEVELS
SIGMOD 1995

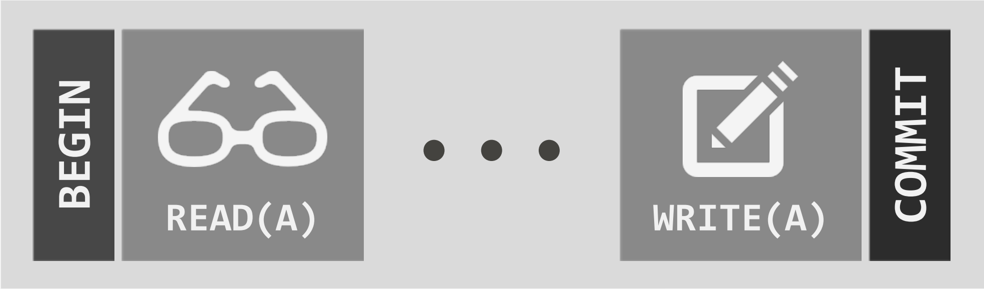
CURSOR STABILITY (CS)

The DBMS's internal cursor maintains a lock on a item in the database until it moves on to the next item.

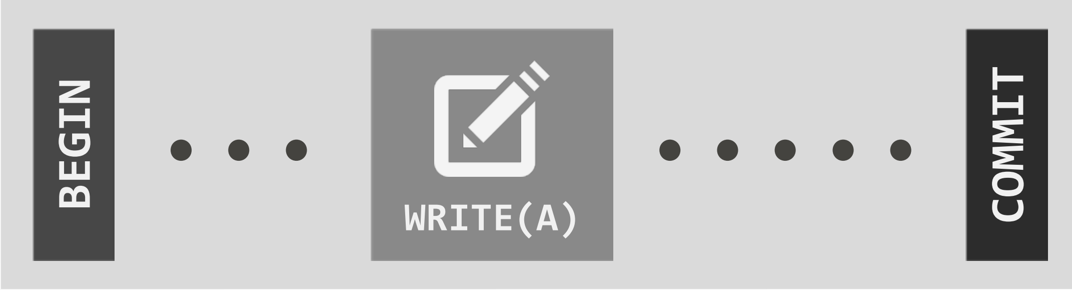
CS is a stronger isolation level in between **REPEATABLE READS** and **READ COMMITTED** that can (sometimes) prevent the **Lost Update Anomaly**.

LOST UPDATE ANOMALY

Txn #1



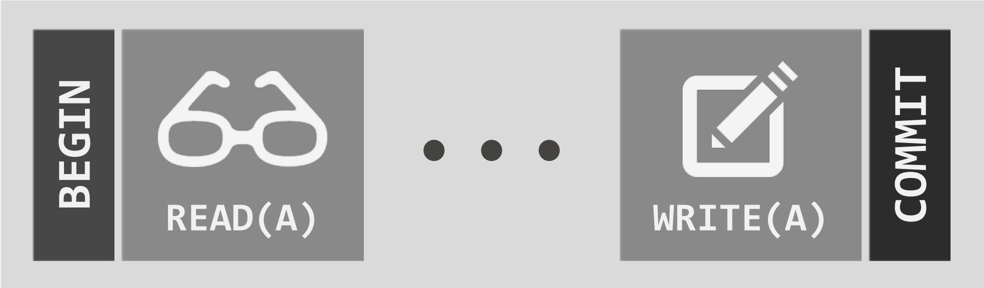
Txn #2



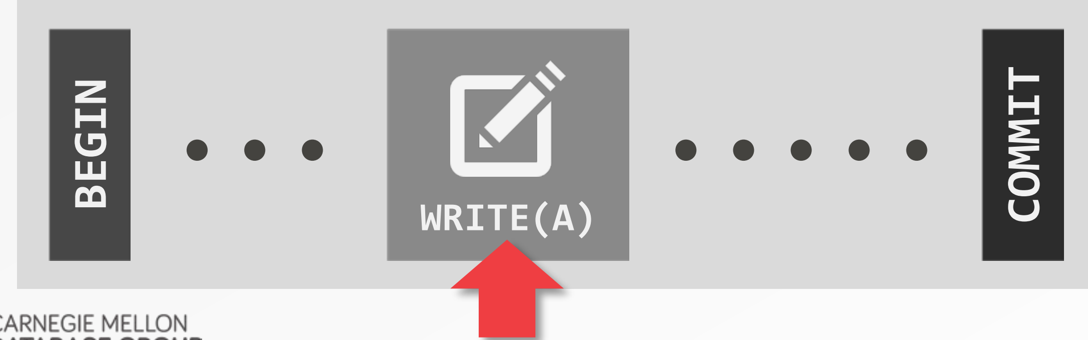
LOST UPDATE ANOMALY

LOST UPDATE ANOMALY

Txn #1

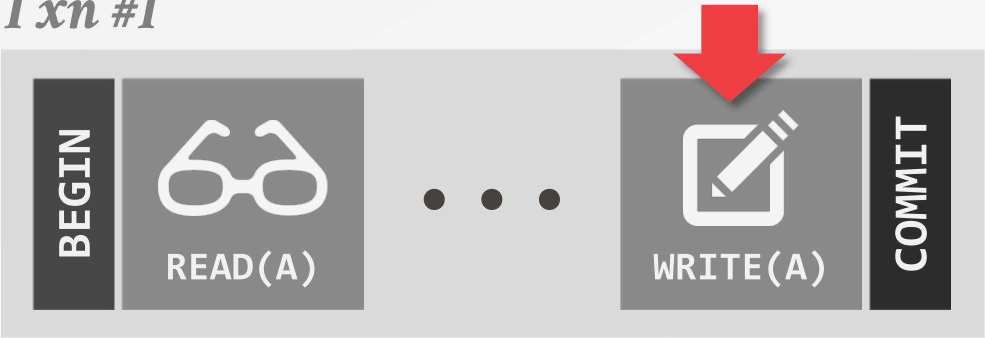


Txn #2

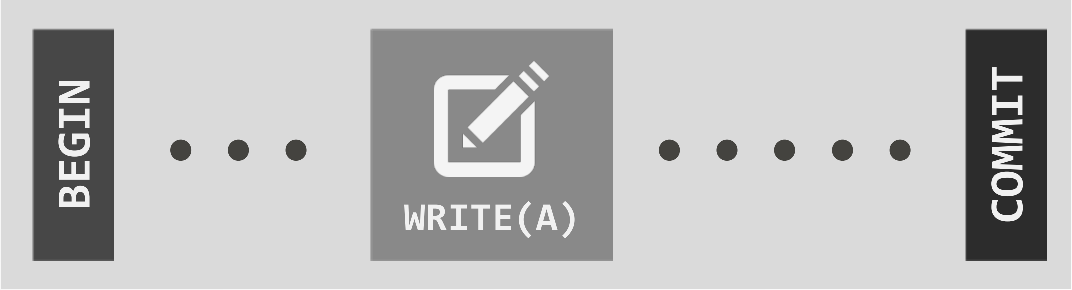


LOST UPDATE ANOMALY

Txn #1

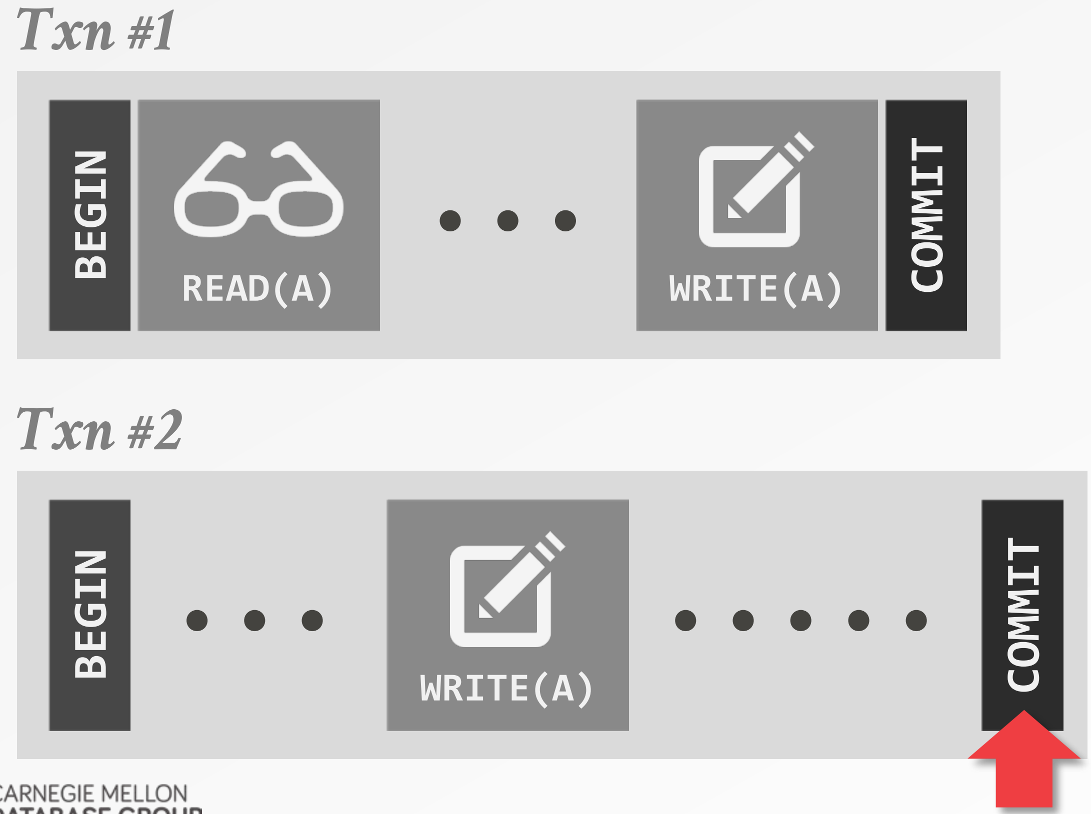


Txn #2

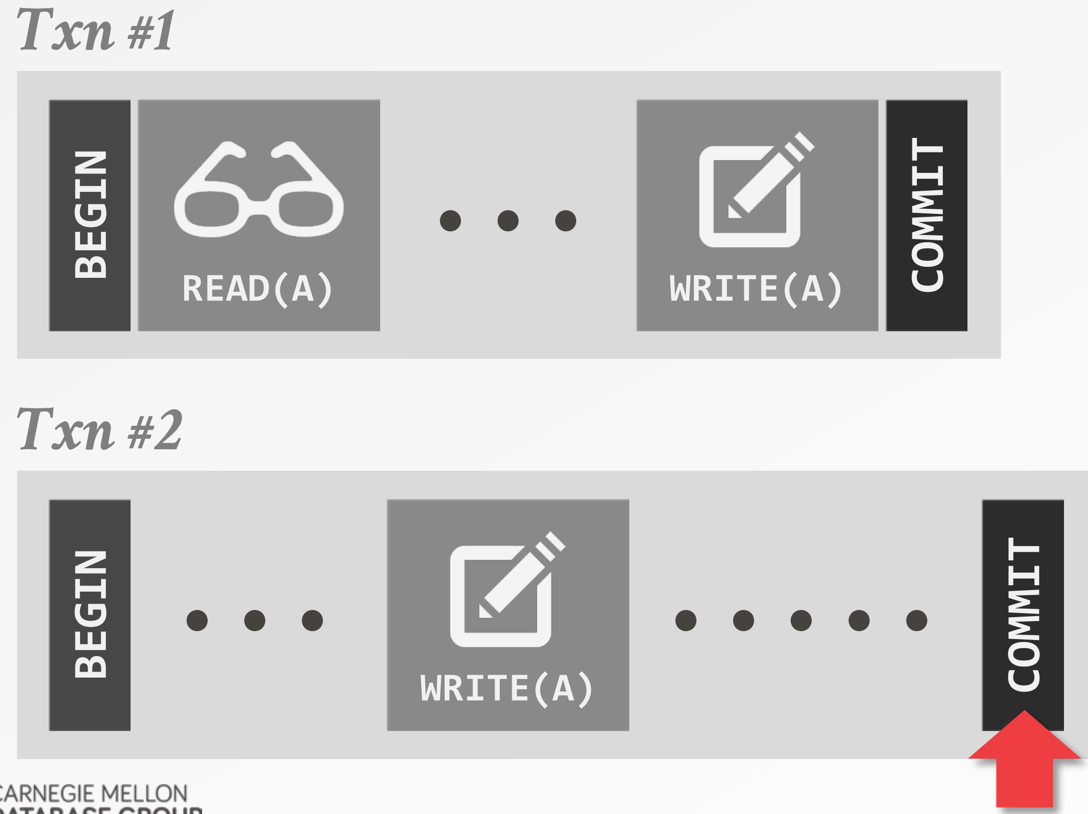


LOST UPDATE ANOMALY

LOST UPDATE ANOMALY



LOST UPDATE ANOMALY



Txn #2's write to **A** will be lost even though it commits after Txn #1.

A **cursor lock** on **A** would prevent this problem (but not always).

SNAPSHOT ISOLATION (SI)

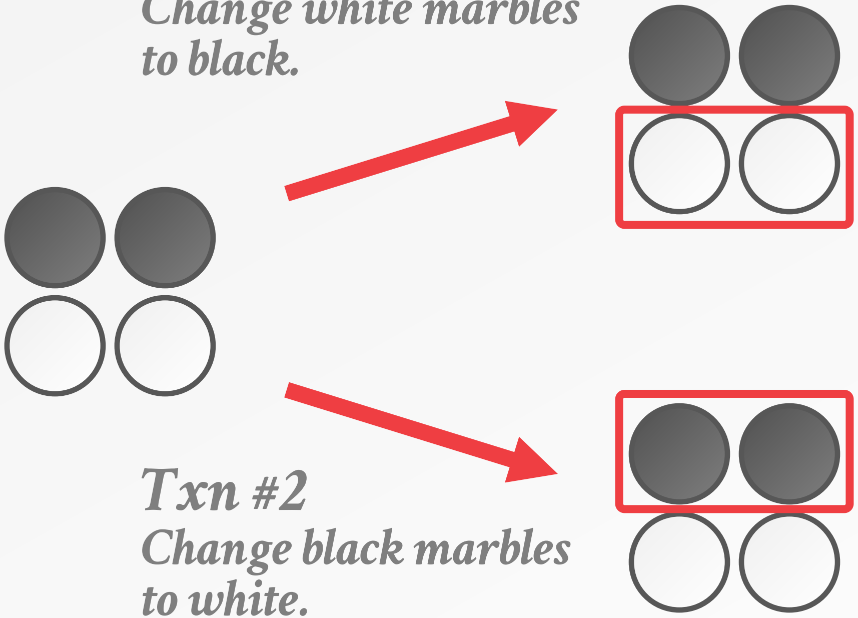
Guarantees that all reads made in a txn see a consistent snapshot of the database that existed at the time the txn started.

→ A txn will commit under SI only if its writes do not conflict with any concurrent updates made since that snapshot.

SI is susceptible to the **Write Skew Anomaly**

WRITE SKEW ANOMALY

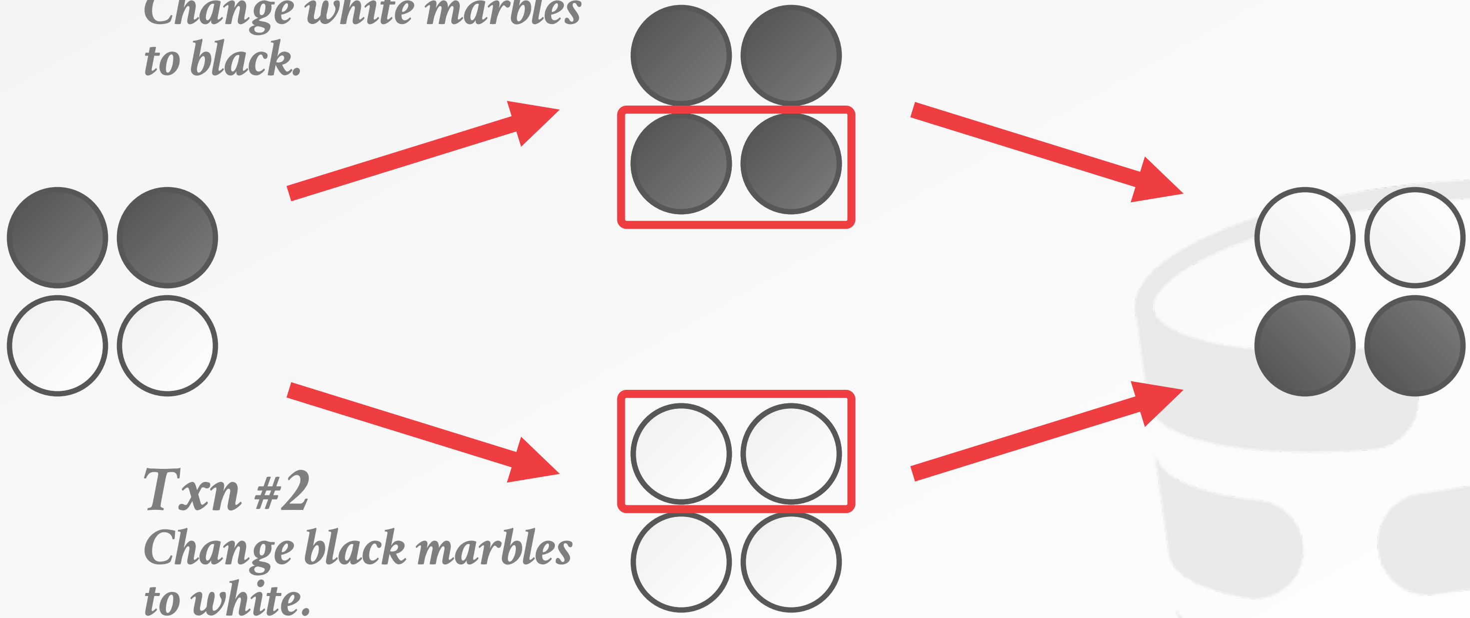
Txn #1
Change white marbles to black.



Txn #2
Change black marbles to white.

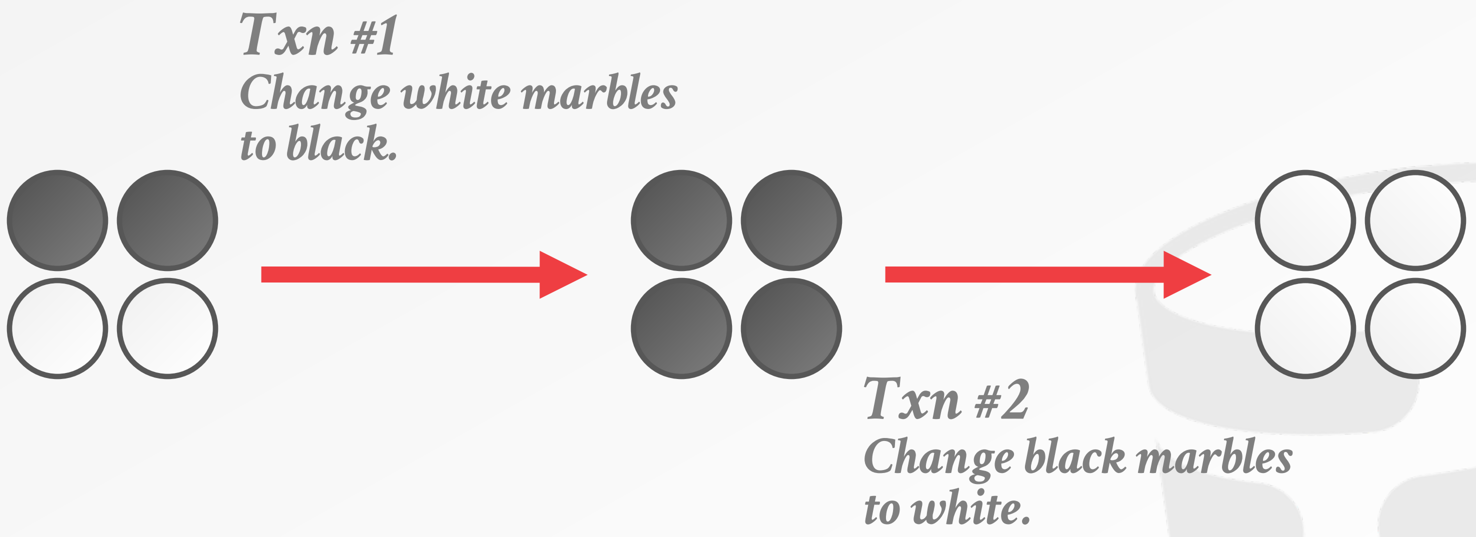
WRITE SKEW ANOMALY

Txn #1
Change white marbles to black.

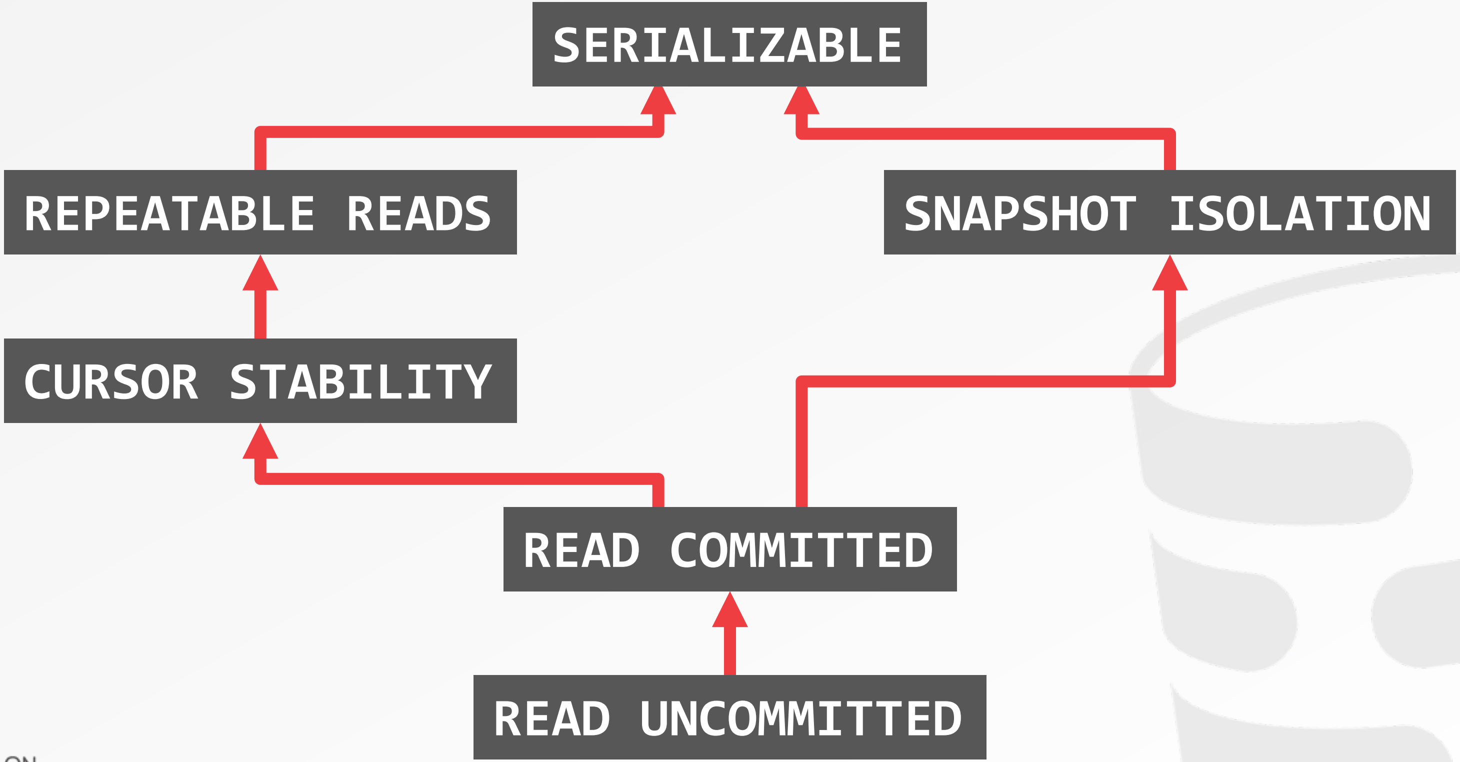


Txn #2
Change black marbles to white.

WRITE SKEW ANOMALY



ISOLATION LEVEL HIERARCHY



ISO … HY

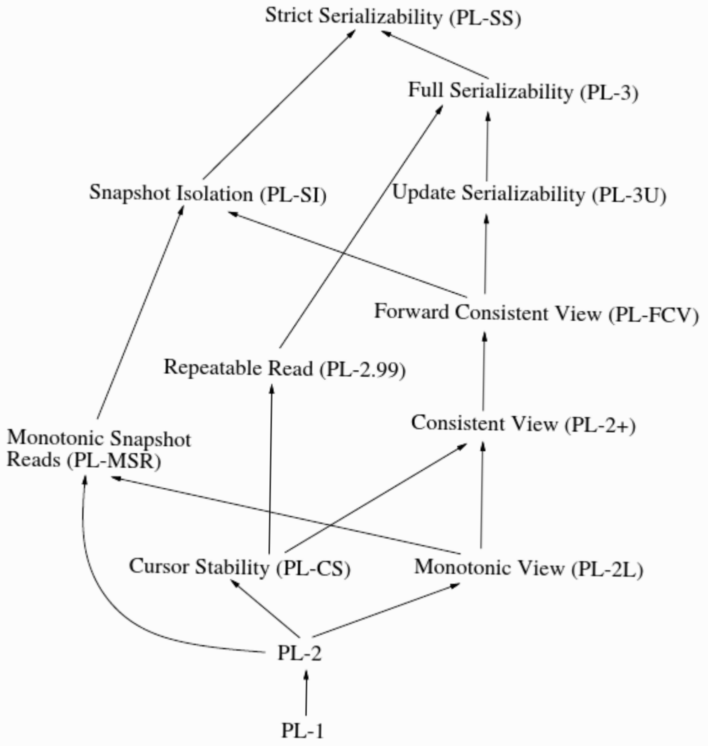


Figure 4-1: A partial order to relate various isolation levels.

Source: Atul Adya

REPEATAB… **…OLATION**

CURSOR S…

OBSERVATION

Disk stalls are (almost) gone when executing txns in an in-memory DBMS.

There are still other stalls when an app uses **conversational** API to execute queries on DBMS
→ ODBC/JDBC
→ DBMS-specific wire protocols

CONVERSATIONAL DATABASE API

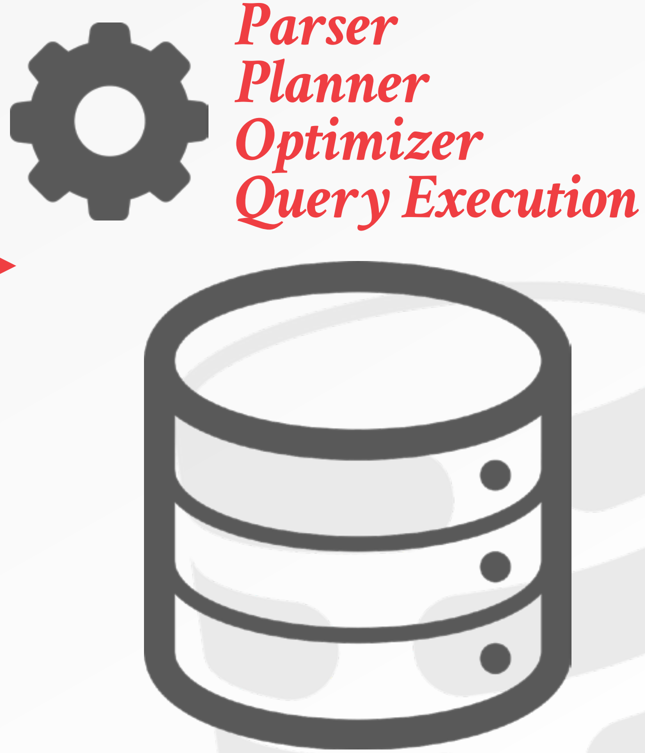
Application

```
BEGIN
  SQL
  Program Logic
  SQL
  Program Logic
  ⋮
COMMIT
```

CONVERSATIONAL DATABASE API

Application

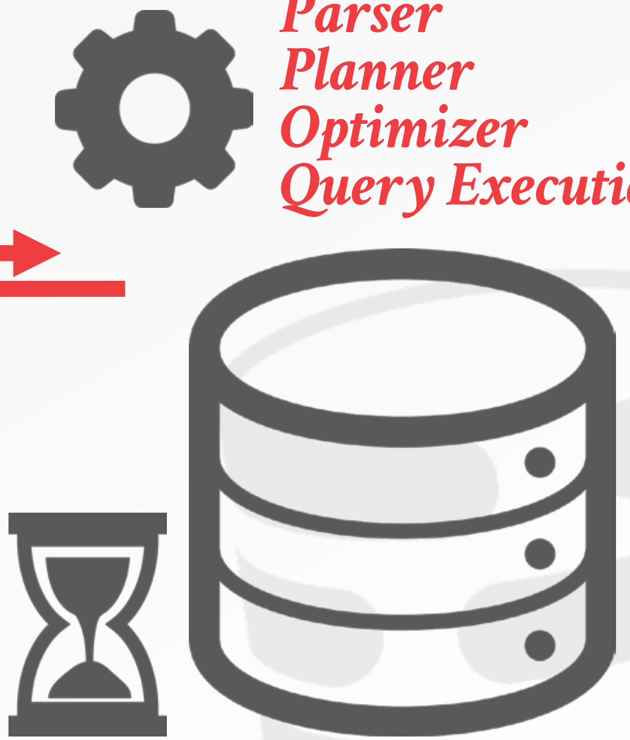
```
BEGIN
  SQL
  Program Logic
  SQL
  Program Logic
  ⋮
COMMIT
```



CONVERSATIONAL DATABASE API

Application

```
BEGIN
  SQL
  Program Logic
  SQL
  Program Logic
  ⋮
COMMIT
```

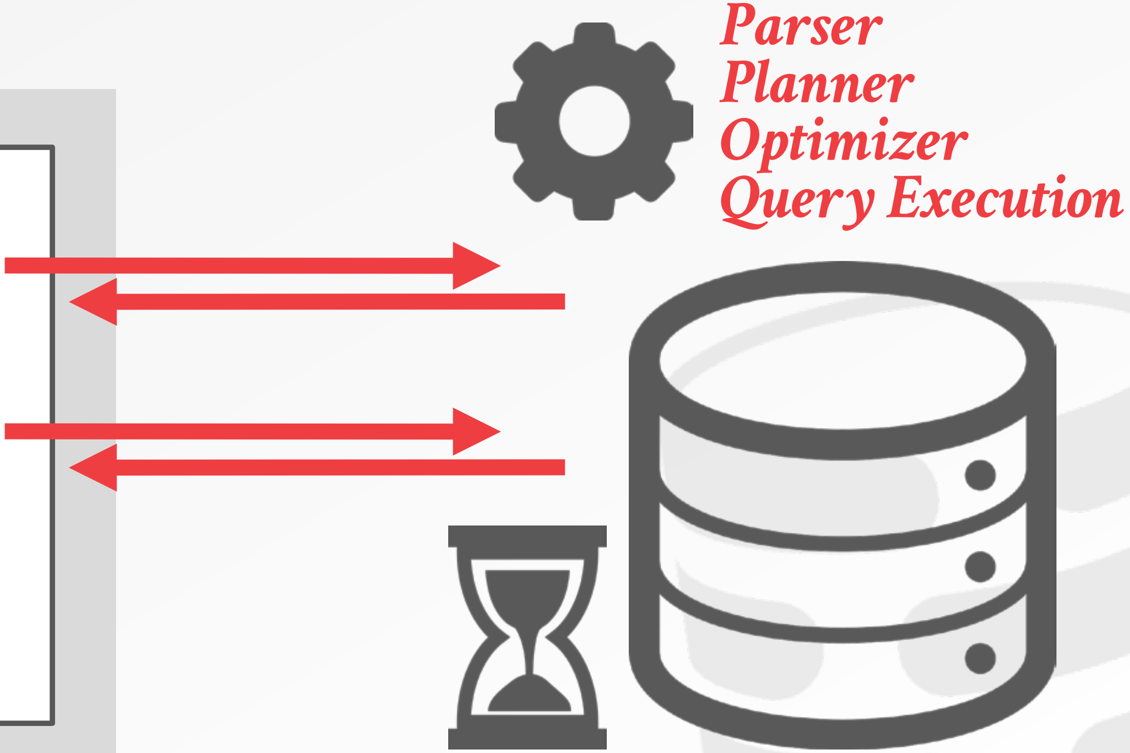


Parser
Planner
Optimizer
Query Execution

CONVERSATIONAL DATABASE API

Application

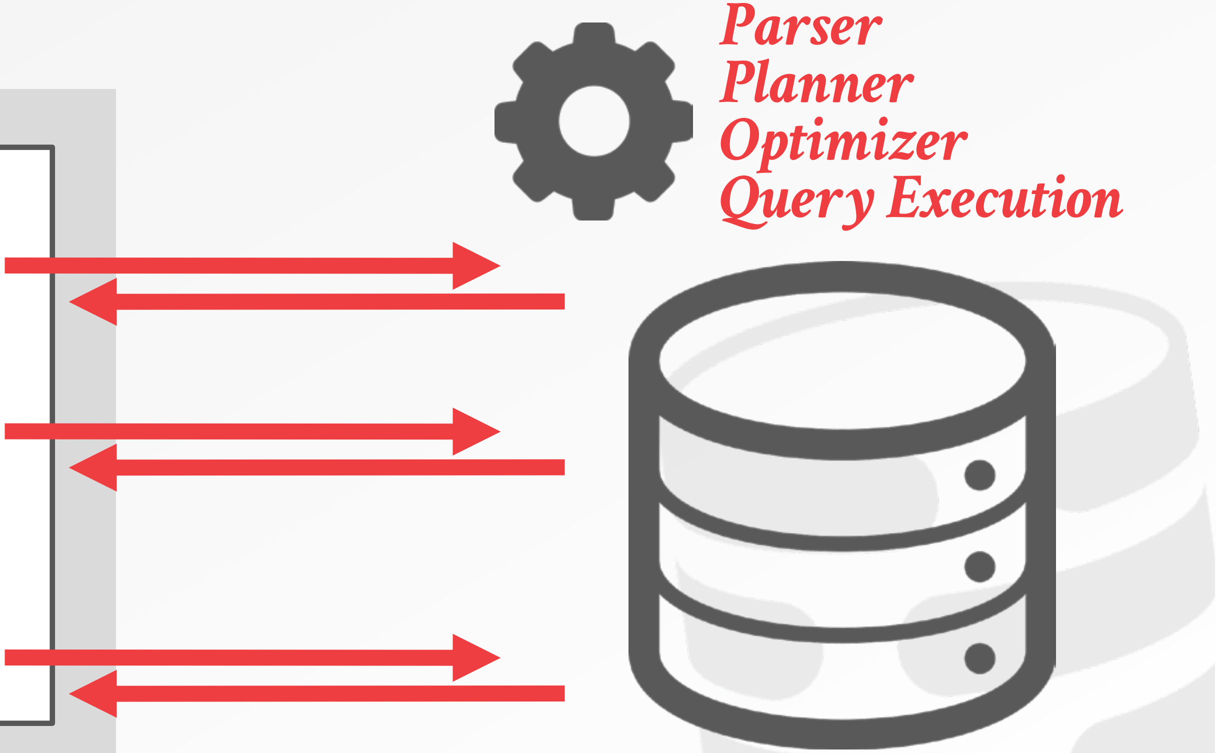
```
BEGIN
  SQL
  Program Logic
  SQL
  Program Logic
  ⋮
COMMIT
```



CONVERSATIONAL DATABASE API

Application

```
BEGIN
  SQL
  Program Logic
  SQL
  Program Logic
  ⋮
COMMIT
```



SOLUTIONS

Prepared Statements

→ Removes query preparation overhead.

Query Batches

→ Reduces the number of network roundtrips.

Stored Procedures

→ Removes both preparation and network stalls.

STORED PROCEDURES

A **stored procedure** is a group of queries that form a logical unit and perform a particular task on behalf of an application directly inside of the DBMS.

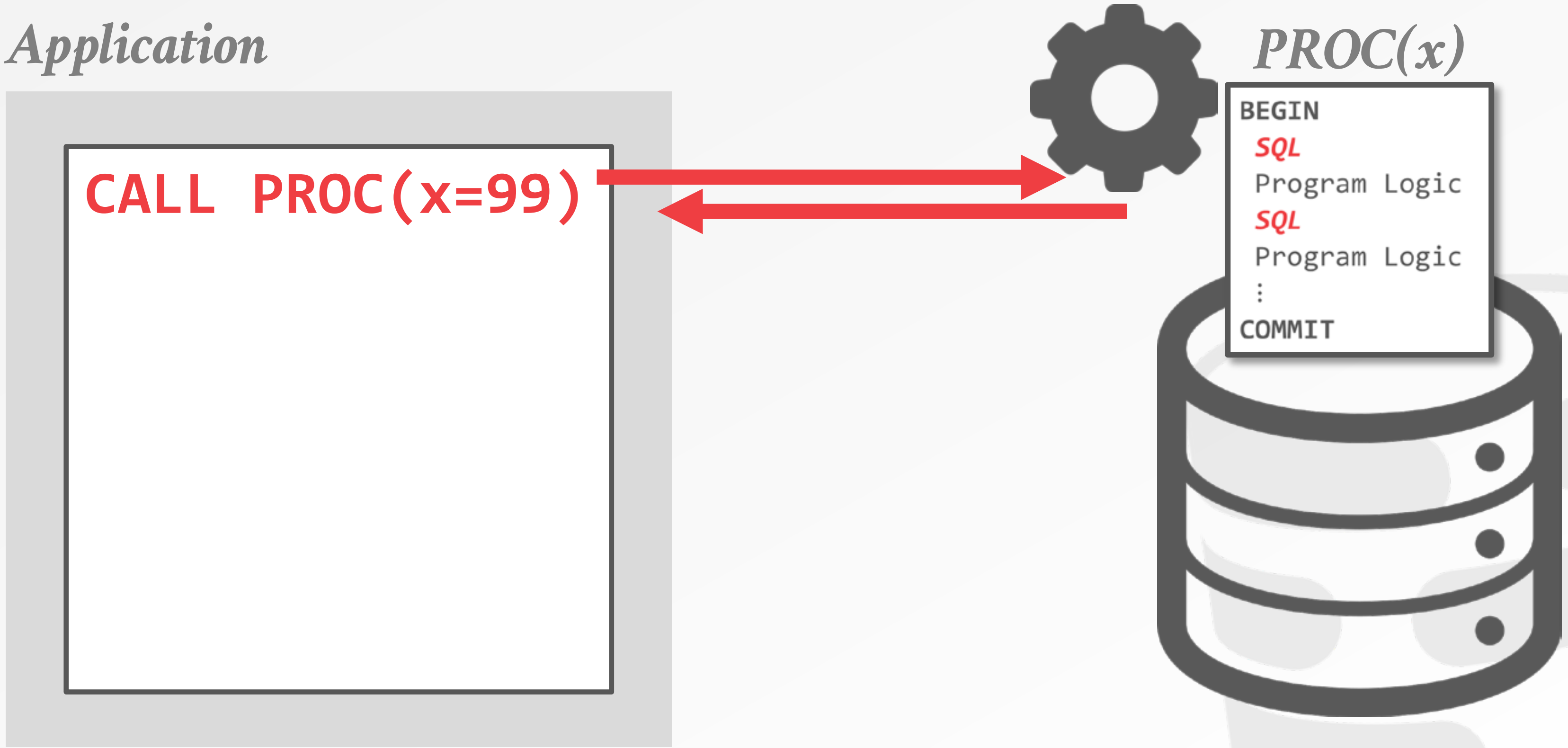
Programming languages:
→ **SQL/PSM** (standard)
→ **PL/SQL** (Oracle / IBM / MySQL)
→ **PL/pgSQL** (Postgres)
→ **Transact-SQL** (Microsoft / Sybase)

STORED PROCEDURES

Application

```
BEGIN
  SQL
  Program Logic
  SQL
  Program Logic
  ⋮
COMMIT
```

STORED PROCEDURES



STORED PROCEDURE EXAMPLE

```
CREATE PROCEDURE testProc
  (num INT, name VARCHAR) RETURNS INT
  BEGIN
    DECLARE cnt INT DEFAULT 0;
    LOOP
      INSERT INTO student VALUES (cnt, name);
      SET cnt := cnt + 1;
      IF (cnt > 15) THEN
        RETURN cnt;
      END IF;
    END LOOP;
  END;
```

ADVANTAGES

Reduce the number of round trips between application and database servers.

Increased performance because queris are pre-compiled and stored in DBMS.

Procedure reuse across applications.

Server-side txn restarts on conflicts.

DISADVANTAGES

Not as many developers know how to write SQL/PSM code.
→ Safe Languages vs. Sandbox Languages

Outside the scope of the application so it is difficult to manage versions and hard to debug.

Probably not be portable to other DBMSs.

DBAs usually don't give permissions out freely…

OPTIMISTIC CONCURRENCY CONTROL

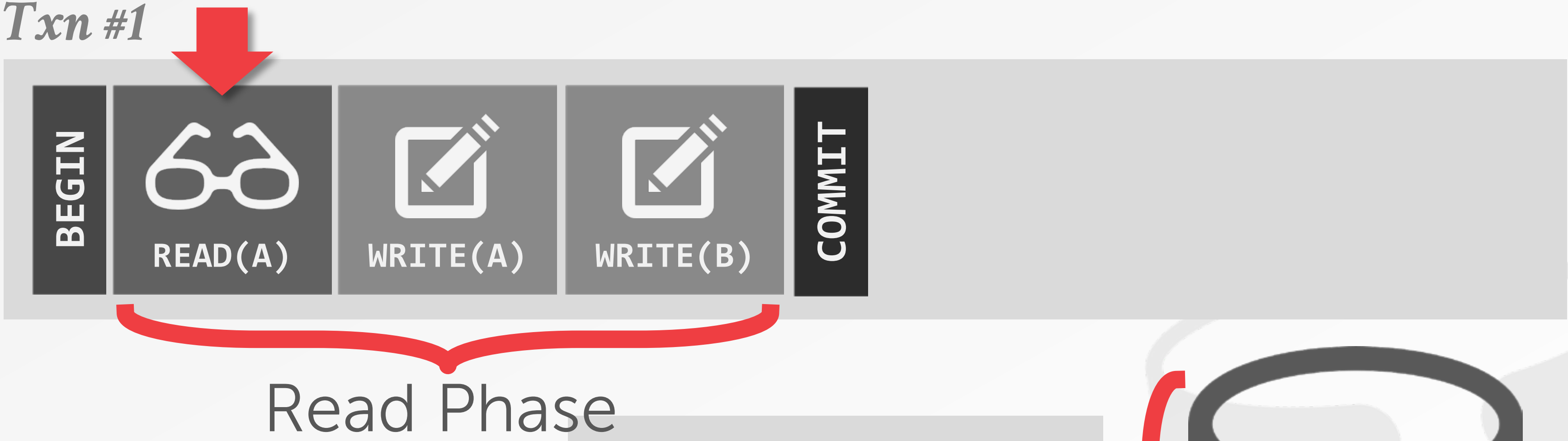
Timestamp-ordering scheme where txns copy data read/write into a private workspace that is not visible to other active txns.

When a txn commits, the DBMS verifies that there are no conflicts.

First proposed in 1981 at CMU by H.T. Kung.

OPTIMISTIC CONCURRENCY CONTROL

Txn #1



Record	Value	Write Timestamp
A	123	10000
B	456	10000

OPTIMISTIC CONCURRENCY CONTROL

Txn #1

BEGIN | READ(A) | WRITE(A) | WRITE(B) | COMMIT

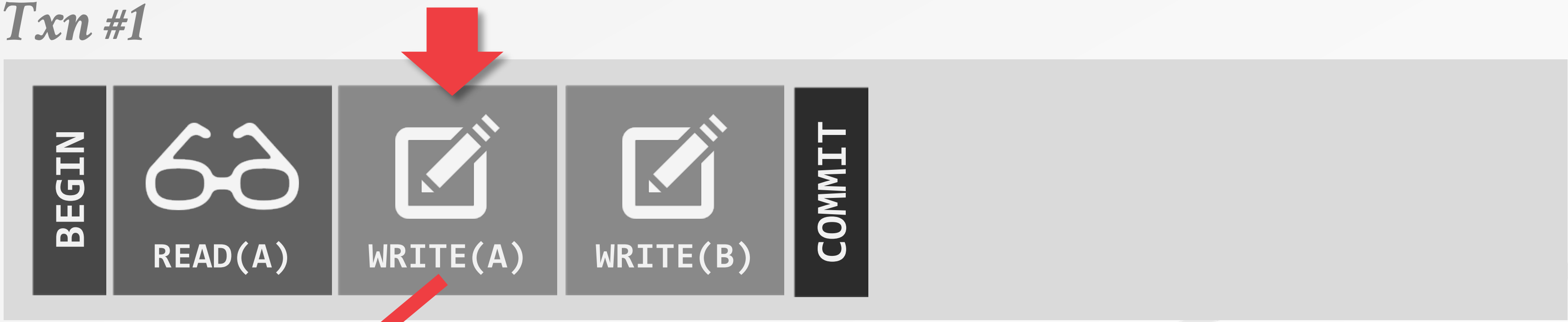
Workspace

Record	Value	Write Timestamp
A	123	10000

Record	Value	Write Timestamp
A	123	10000
B	456	10000

OPTIMISTIC CONCURRENCY CONTROL

Txn #1



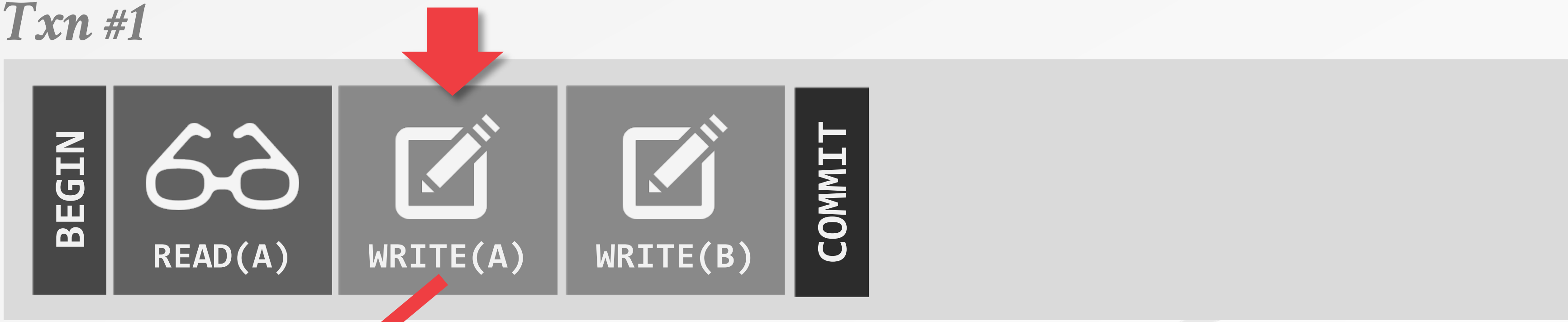
Workspace

Record	Value	Write Timestamp
A	123	10000

Record	Value	Write Timestamp
A	123	10000
B	456	10000

OPTIMISTIC CONCURRENCY CONTROL

Txn #1



Workspace

Record	Value	Write Timestamp
A	888	∞

Record	Value	Write Timestamp
A	123	10000
B	456	10000

OPTIMISTIC CONCURRENCY CONTROL

Txn #1

BEGIN | READ(A) | WRITE(A) | WRITE(B) | COMMIT

Workspace

Record	Value	Write Timestamp
A	888	∞
B	456	10000

Record	Value	Write Timestamp
A	123	10000
B	456	10000

OPTIMISTIC CONCURRENCY CONTROL

Txn #1

BEGIN | READ(A) | WRITE(A) | WRITE(B) | COMMIT

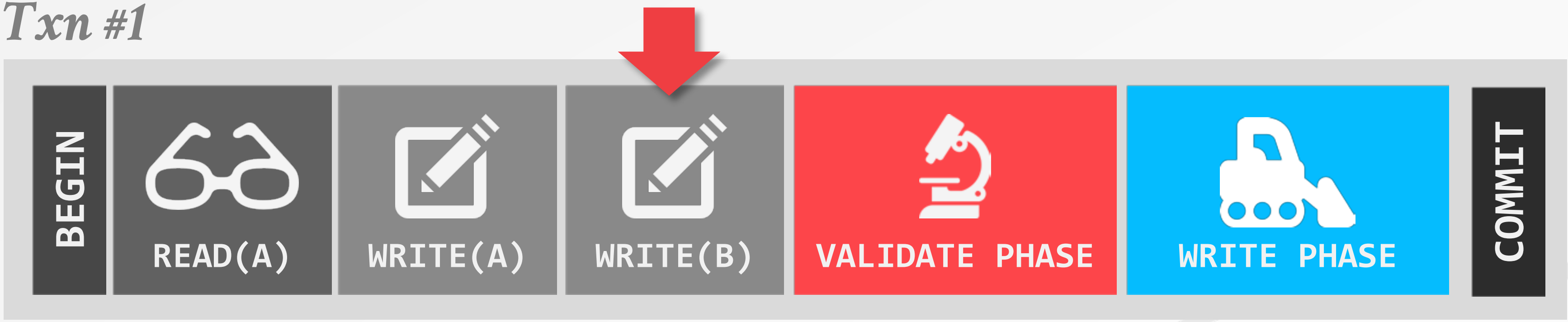
Workspace

Record	Value	Write Timestamp
A	888	∞
B	999	∞

Record	Value	Write Timestamp
A	123	10000
B	456	10000

OPTIMISTIC CONCURRENCY CONTROL

Txn #1



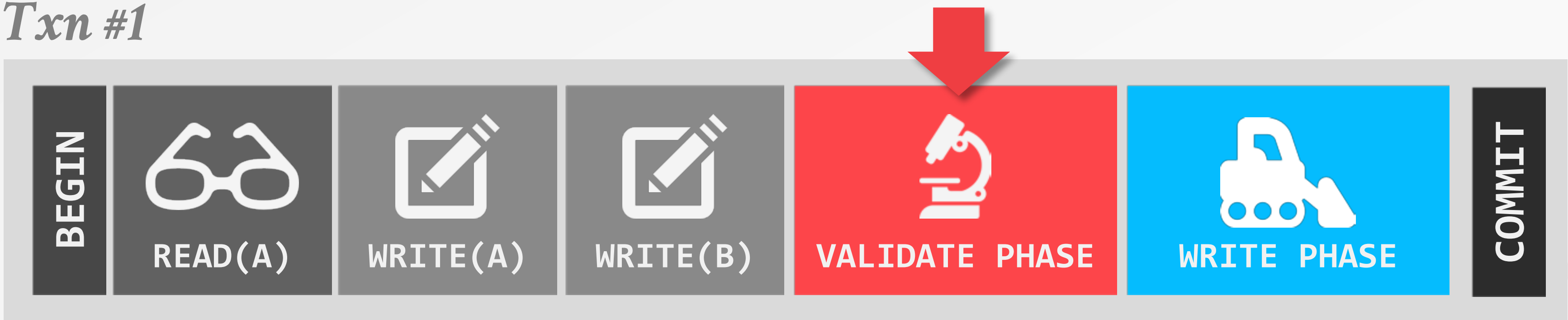
Workspace

Record	Value	Write Timestamp
A	888	∞
B	999	∞

Record	Value	Write Timestamp
A	123	10000
B	456	10000

OPTIMISTIC CONCURRENCY CONTROL

Txn #1



Workspace

Record	Value	Write Timestamp
A	888	∞
B	999	∞

Record	Value	Write Timestamp
A	123	10000
B	456	10000

OPTIMISTIC CONCURRENCY CONTROL

Txn #1

BEGIN | READ(A) | WRITE(A) | WRITE(B) | VALIDATE PHASE | WRITE PHASE | COMMIT

10001

Workspace

Record	Value	Write Timestamp
A	888	∞
B	999	∞

Record	Value	Write Timestamp
A	888	10001
B	999	10001

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.

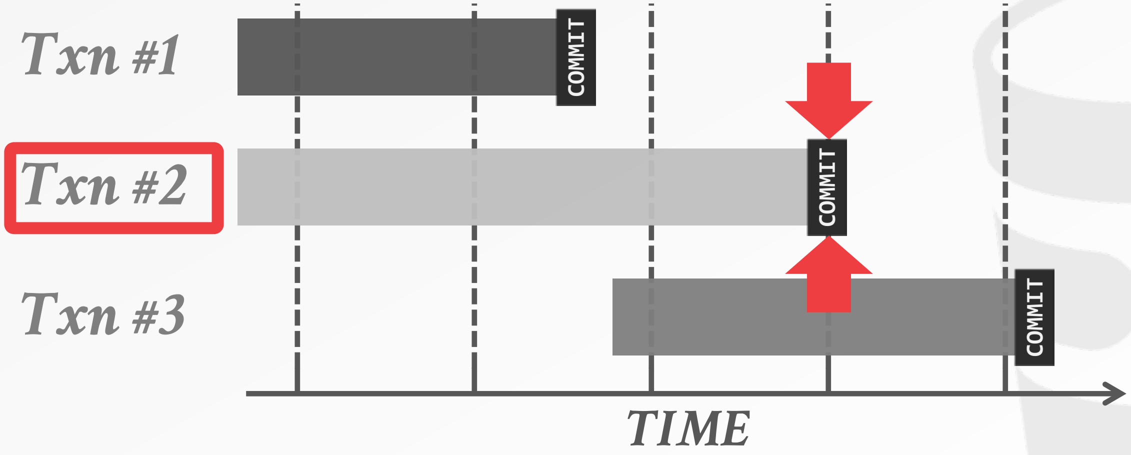
VALIDATION PHASE

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

Two methods for this phase:
→ Backward Validation
→ Forward Validation

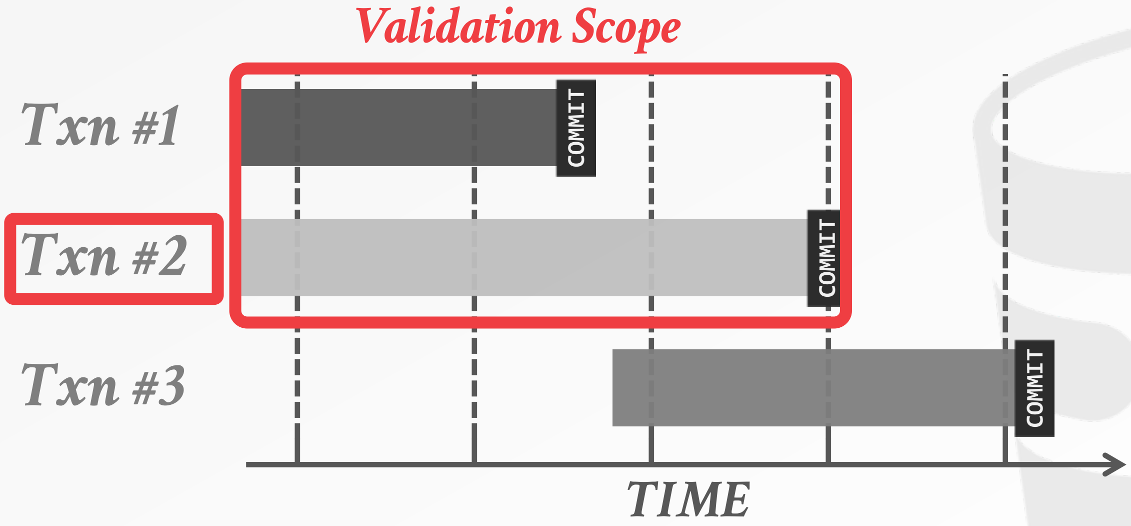
BACKWARD VALIDATION

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



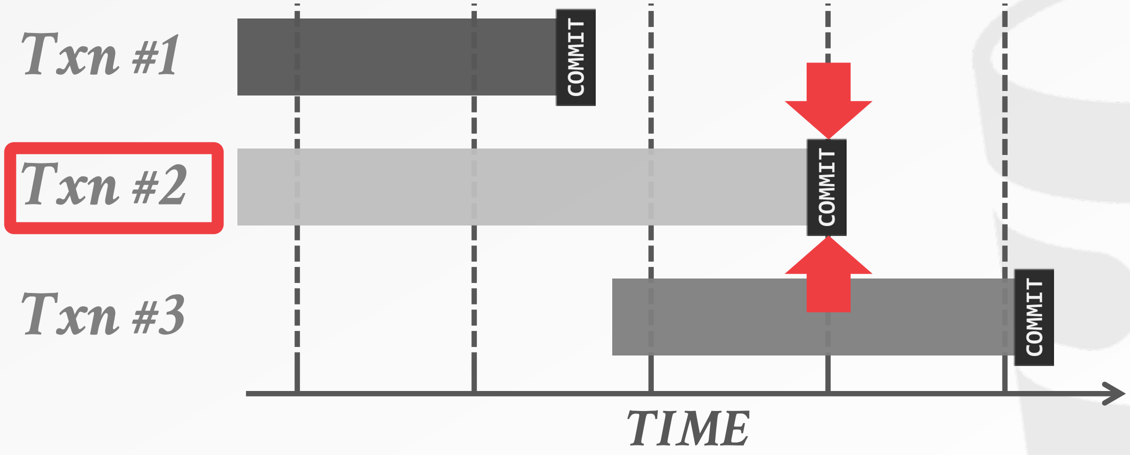
BACKWARD VALIDATION

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



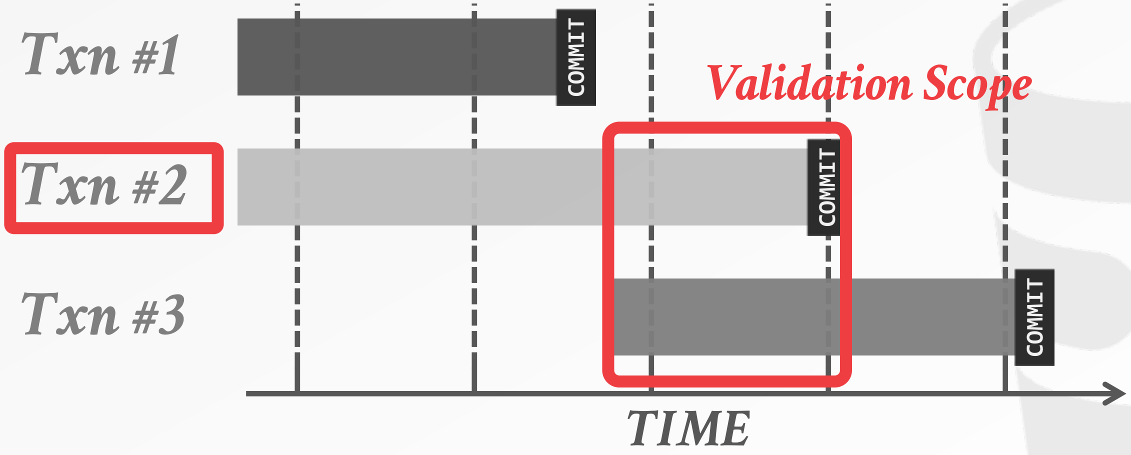
FORWARD VALIDATION

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



FORWARD VALIDATION

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



VALIDATION PHASE

Original OCC uses serial validation.

Parallel validation means that each txn must check the read/write sets of other txns that are trying to validate at the same time.

→ Each txn has to acquire locks for its write set records in some **global order**.

→ The txn does not need locks for read set records.

WRITE PHASE

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

As each record is updated, the txn releases the lock acquired during the Validation Phase.

MODERN OCC

Harvard/MIT Silo

MIT/CMU TicToc

SILO

Single-node, in-memory OLTP DBMS.
→ Serializable OCC with parallel backward validation.
→ Stored procedure-only API.

No writes to shared-memory for read txns.

Batched timestamp allocation using **epochs**.

Pure awesomeness from Eddie Kohler.

SPEEDY TRANSACTIONS IN MULTICORE IN-MEMORY DATABASES
SOSP 2013

SILO: EPOCHS

Time is sliced into fixed-length epochs (40ms).

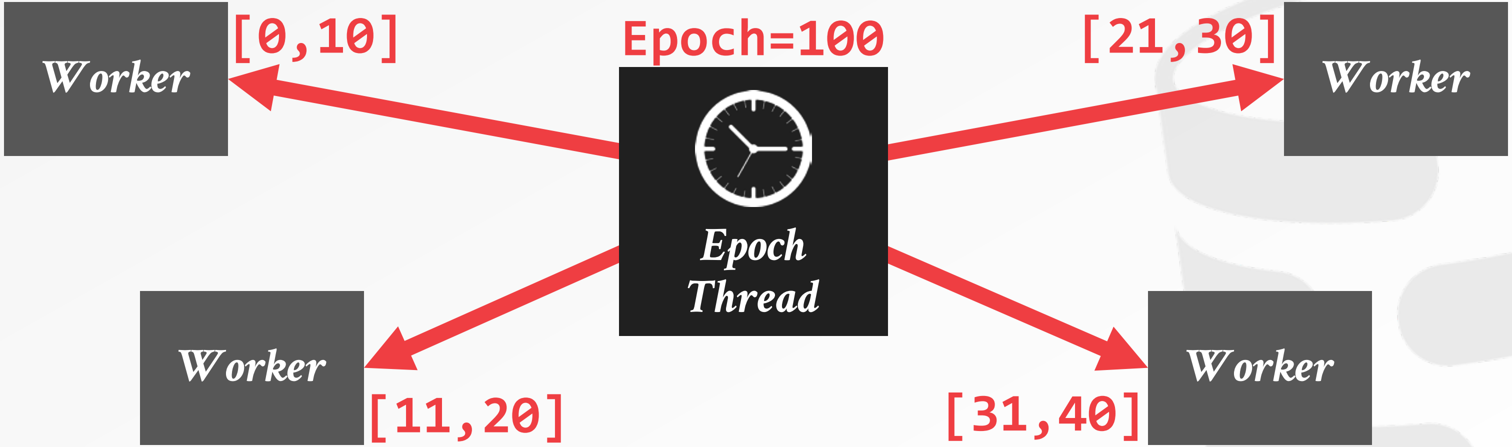
All txns that start in the same epoch will be committed together at the end of the epoch.

→ Txns that span an epoch have to refresh themselves to be carried over into the next epoch.

Worker threads only need to synchronize at the beginning of each epoch.

SILO: TRANSACTION IDS

Each worker thread generates a unique txn id based on the current epoch number and the next value in its assigned batch.



SILO: COMMIT PROTOCOL

TID Word	POINTER	ATTR1	ATTR2
#-###-#	¤	John	$100
#-###-#	¤	Tupac	$999
#-###-#	¤	Wiz	$67
#-###-#	¤	O.D.B.	$13

EPOCH	BATCH TIMESTAMP	EXTRA

Write Lock Bit
Latest Version Bit
Absent Bit

SILO: COMMIT PROTOCOL

Workspace

Read Set

#-###-#	O.D.B.	$13
#-###-#	Tupac	$999

Write Set

Tupac	$777

TID Word	POINTER	ATTR1	ATTR2
#-###-#	¤	John	$100
#-###-#	¤	Tupac	$999
#-###-#	¤	Wiz	$67
#-###-#	¤	O.D.B.	$13

Step #1: Lock Write Set

SILO: COMMIT PROTOCOL

Workspace

Read Set

#-###-#	O.D.B.	$13
#-###-#	Tupac	$999

Write Set

Tupac	$777

???

TID Word	POINTER	ATTR1	ATTR2
#-###-#	¤	John	$100
#-###-#	¤	Tupac	$999
#-###-#	¤	Wiz	$67
#-###-#	¤	O.D.B.	$13

Step #1: Lock Write Set

Step #2: Examine Read Set

SILO: COMMIT PROTOCOL

Workspace

Read Set

#-###-#	O.D.B.	$13
#-###-#	Tupac	$999

Write Set

Tupac	$777

???

TID Word	POINTER	ATTR1	ATTR2
#-###-#	¤	John	$100
#-###-#	¤	Tupac	$999
#-###-#	¤	Wiz	$67
#-###-#	¤	O.D.B.	$13

Step #1: Lock Write Set

Step #2: Examine Read Set

SILO: COMMIT PROTOCOL

Workspace

Read Set

#-###-#	O.D.B.	$13
#-###-#	Tupac	$999

Write Set

Tupac	$777

TID Word	POINTER	ATTR1	ATTR2
#-###-#	¤	John	$100
#-###-#	¤	Tupac	$999
#-###-#	¤	Wiz	$67
#-###-#	¤	O.D.B.	$13

Step #1: Lock Write Set

Step #2: Examine Read Set

SILO: COMMIT PROTOCOL

Workspace

Read Set

#-###-#	O.D.B.	$13
#-###-#	Tupac	$999

Write Set

Tupac	$777

TID Word	POINTER	ATTR1	ATTR2
#-###-#	¤	John	$100
#-###-#	¤	Tupac	$777
#-###-#	¤	Wiz	$67
#-###-#	¤	O.D.B.	$13

Step #1: Lock Write Set

Step #2: Examine Read Set

Step #3: Install Write Set

SILO: GARBAGE COLLECTION

Cooperative threads GC.

Each worker thread marks a deleted object with a **reclamation epoch**.

→ This is the epoch after which no thread could access the object again, and thus can be safely removed.

→ Object references are maintained in thread-local storage to avoid unnecessary data movement.

SILO: RANGE QUERIES

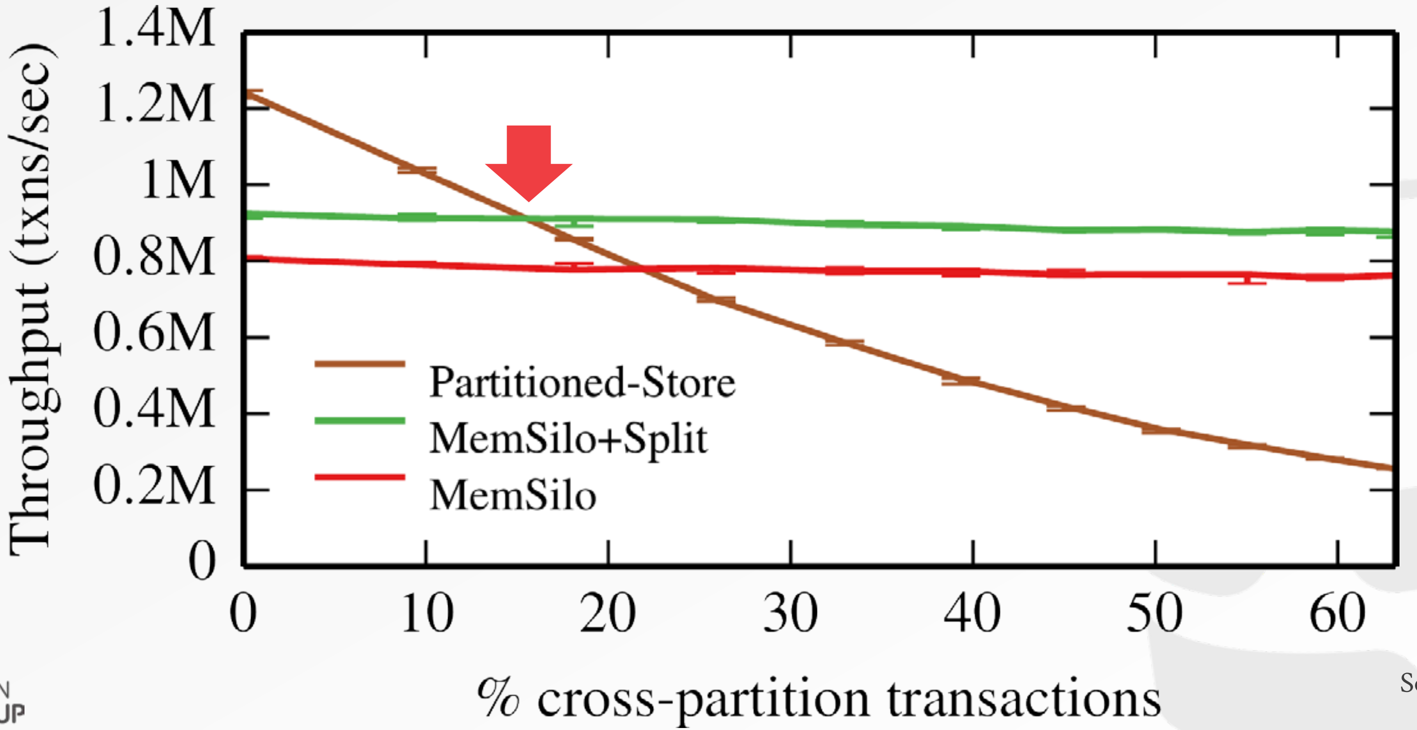
DBMS handles phantoms by tracking the txn's scan set (node set) on indexes.

→ Have to include "virtual" entries for keys that do not exist in the index.

We will discuss key-range and index gap locking next week…

SILO: PERFORMANCE

Database: TPC-C with 28 Warehouses
Processor: 4 sockets, 8 cores per socket



Source: Eddie Kohler

TICTOC

Serializable OCC implemented in DBx1000.
→ Parallel backward validation
→ Stored procedure-only API

No global timestamp allocation.
Txn timestamps are derived from records.

TICTOC: TIME-TRAVELING OPTIMISTIC CONCURRENCY CONTROL
SIGMOD 2016

TICTOC: RECORD TIMESTAMPS

Write Timestamp (W-TS):

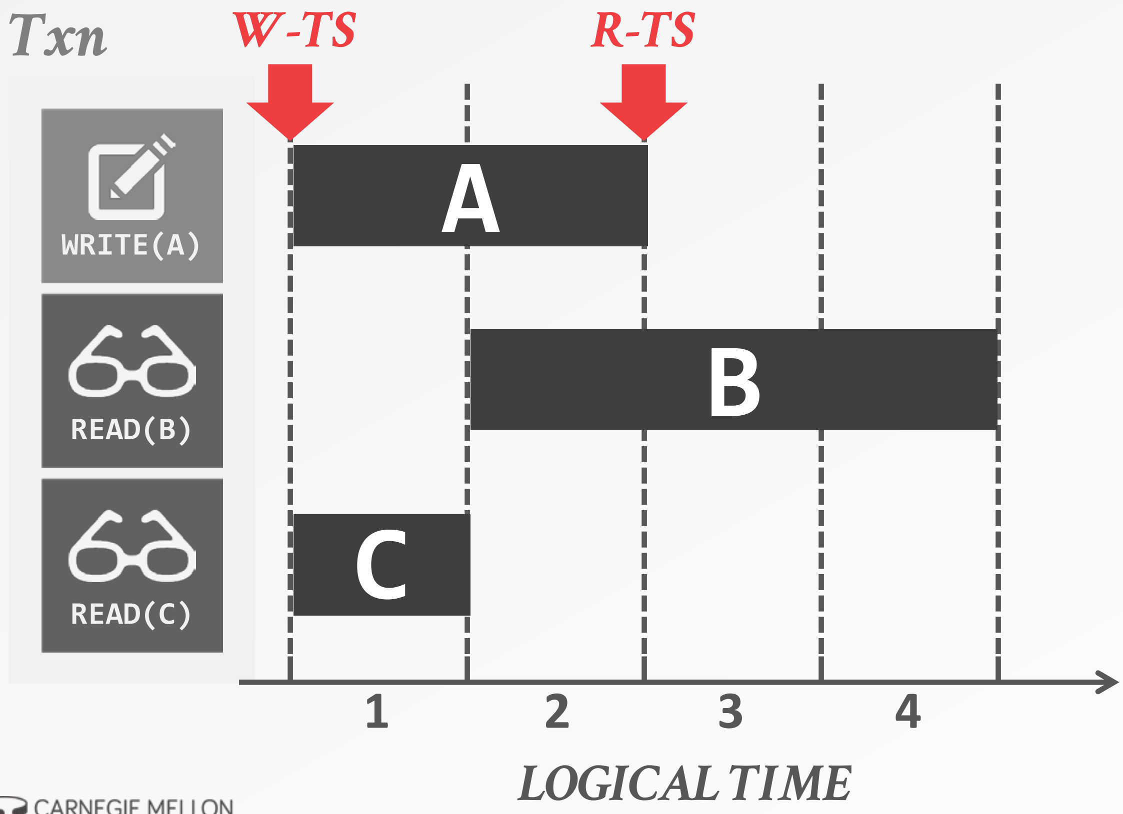
→ The logical timestamp of the last committed txn that wrote to the record.

Read Timestamp (R-TS):

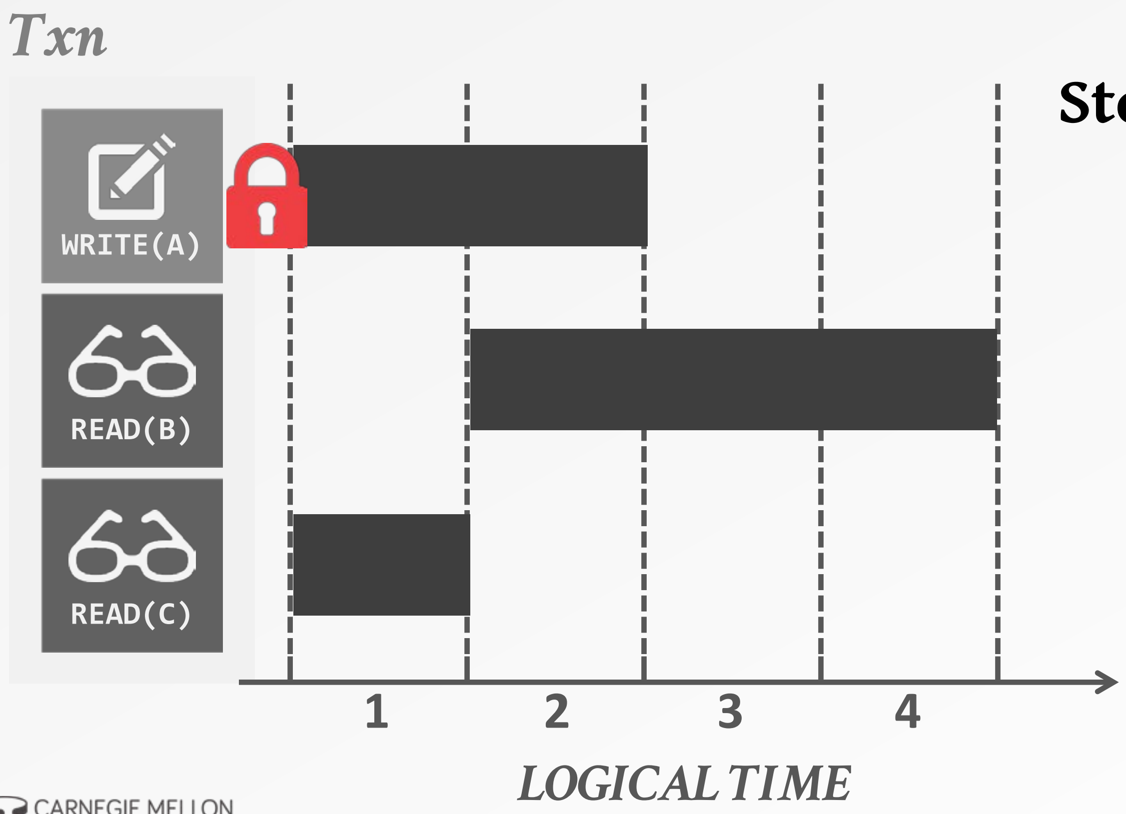
→ The logical timestamp of the last txn that read the record.

A record is considered valid from W-TS to R-TS

TICTOC: VALIDATION PHASE

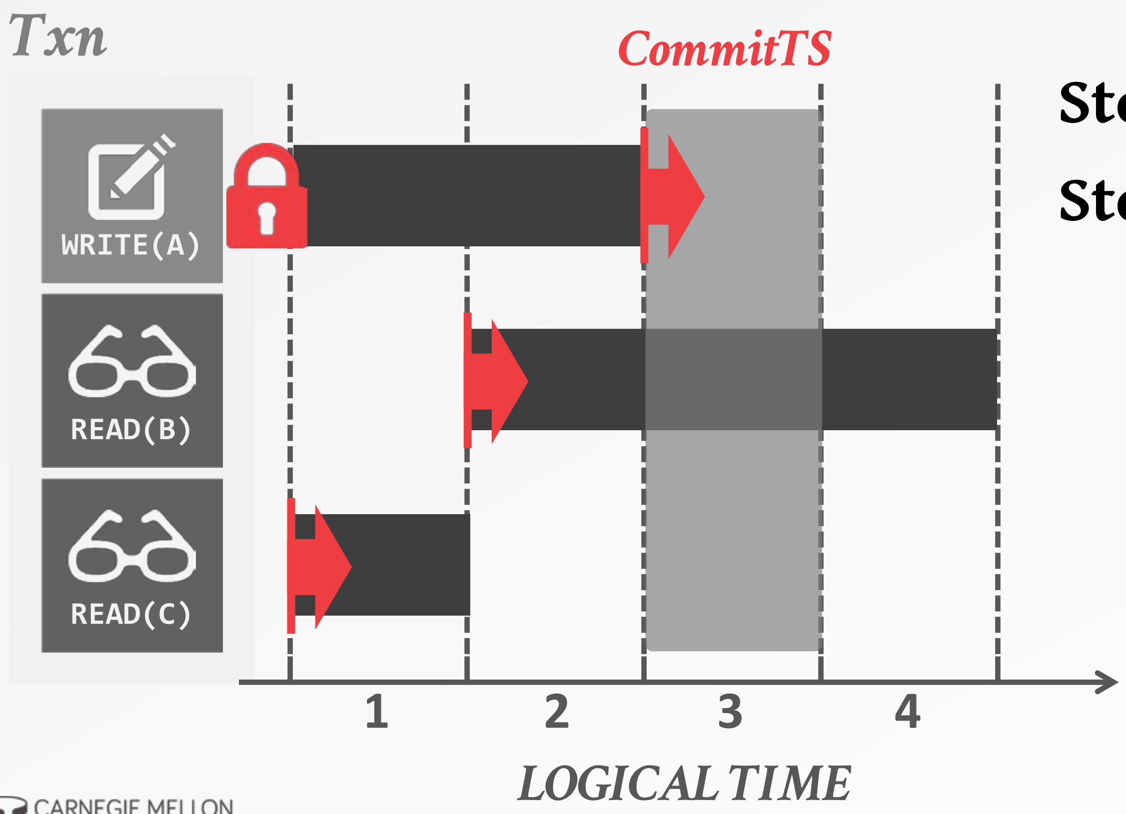


TICTOC: VALIDATION PHASE



Step #1: Lock Write Set

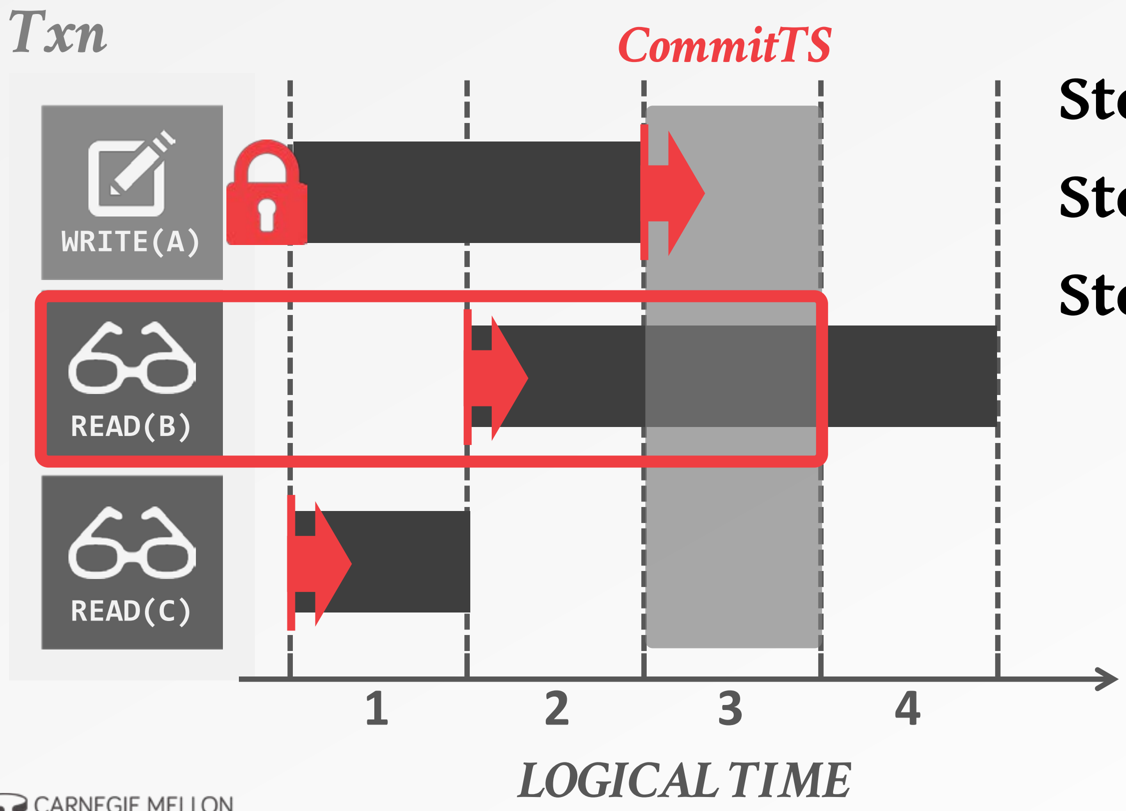
TICTOC: VALIDATION PHASE



Step #1: Lock Write Set

Step #2: Compute CommitTS

TICTOC: VALIDATION PHASE

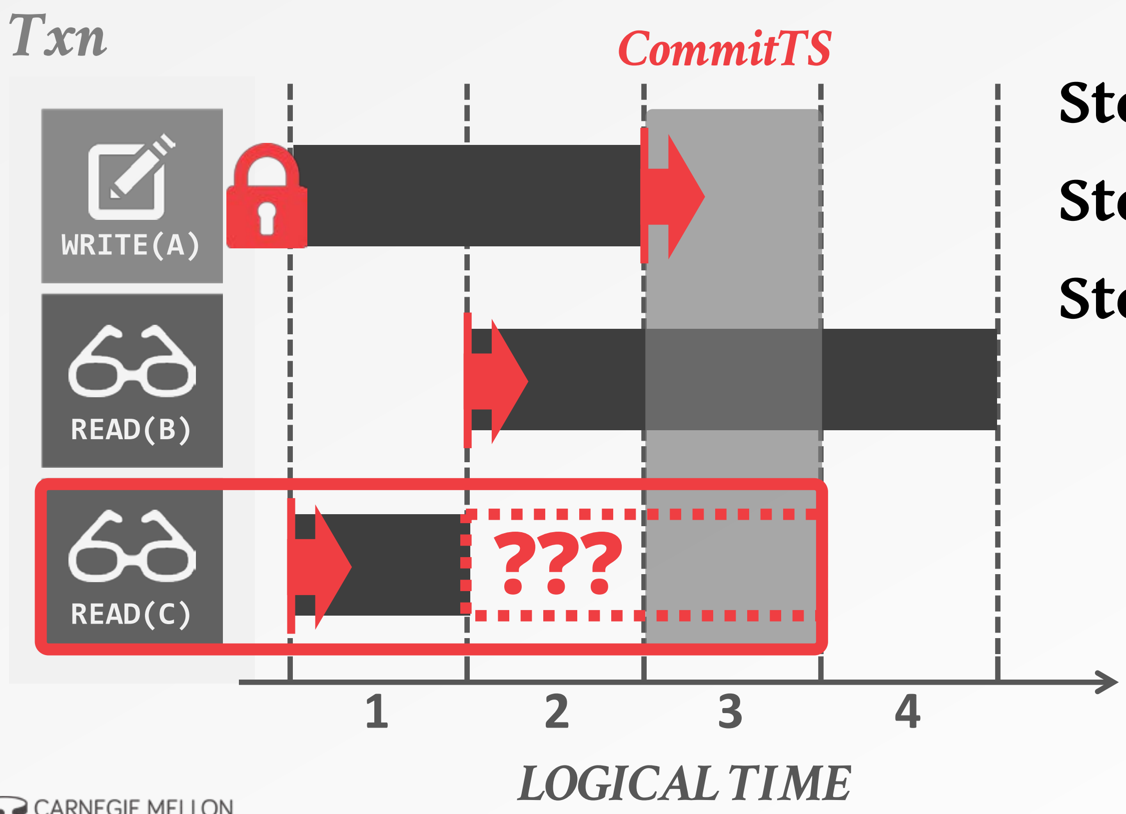


Step #1: Lock Write Set

Step #2: Compute CommitTS

Step #3: Validate Read Set

TICTOC: VALIDATION PHASE

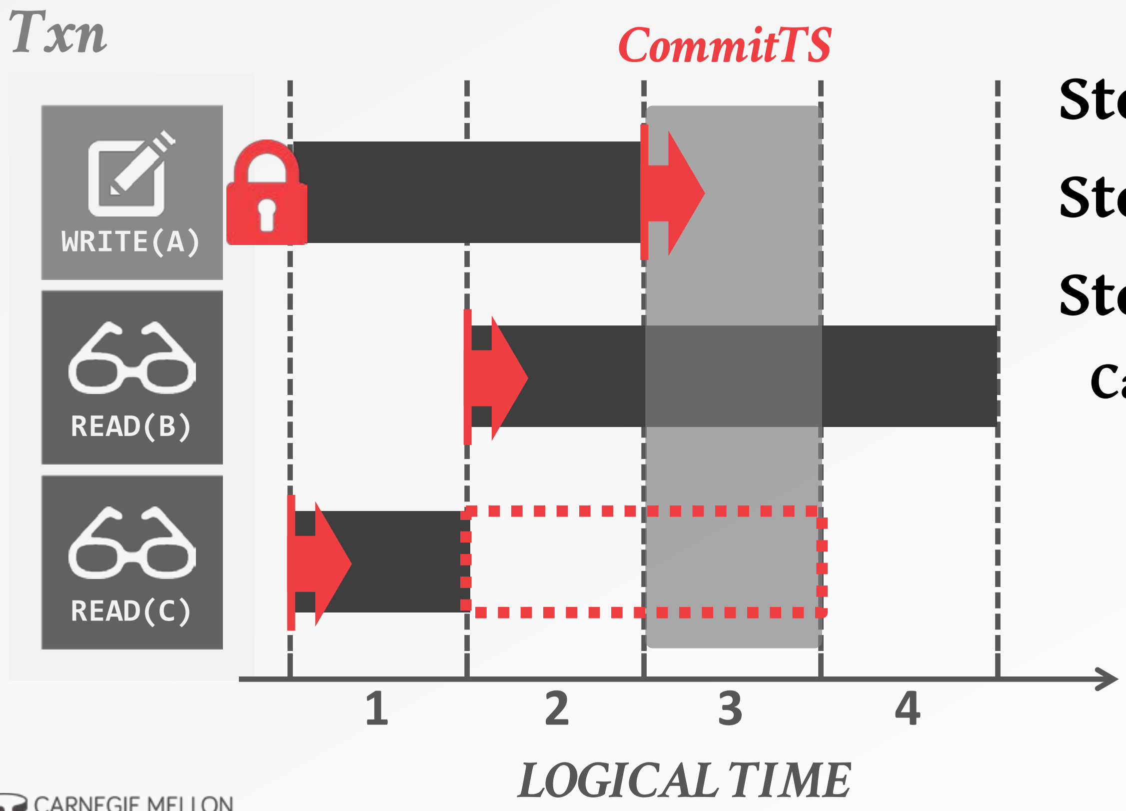


Step #1: Lock Write Set

Step #2: Compute CommitTS

Step #3: Validate Read Set

TICTOC: VALIDATION PHASE



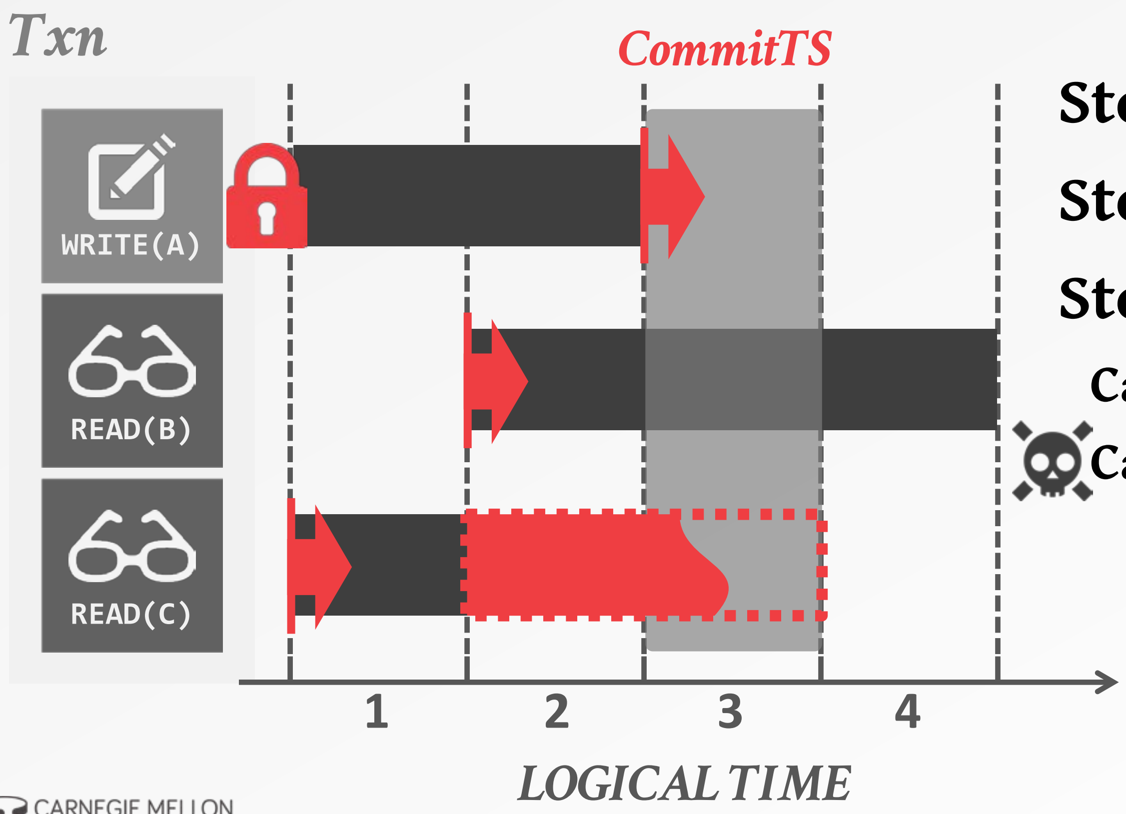
Step #1: Lock Write Set

Step #2: Compute CommitTS

Step #3: Validate Read Set

Case 1: Latest Version

TICTOC: VALIDATION PHASE



Step #1: Lock Write Set

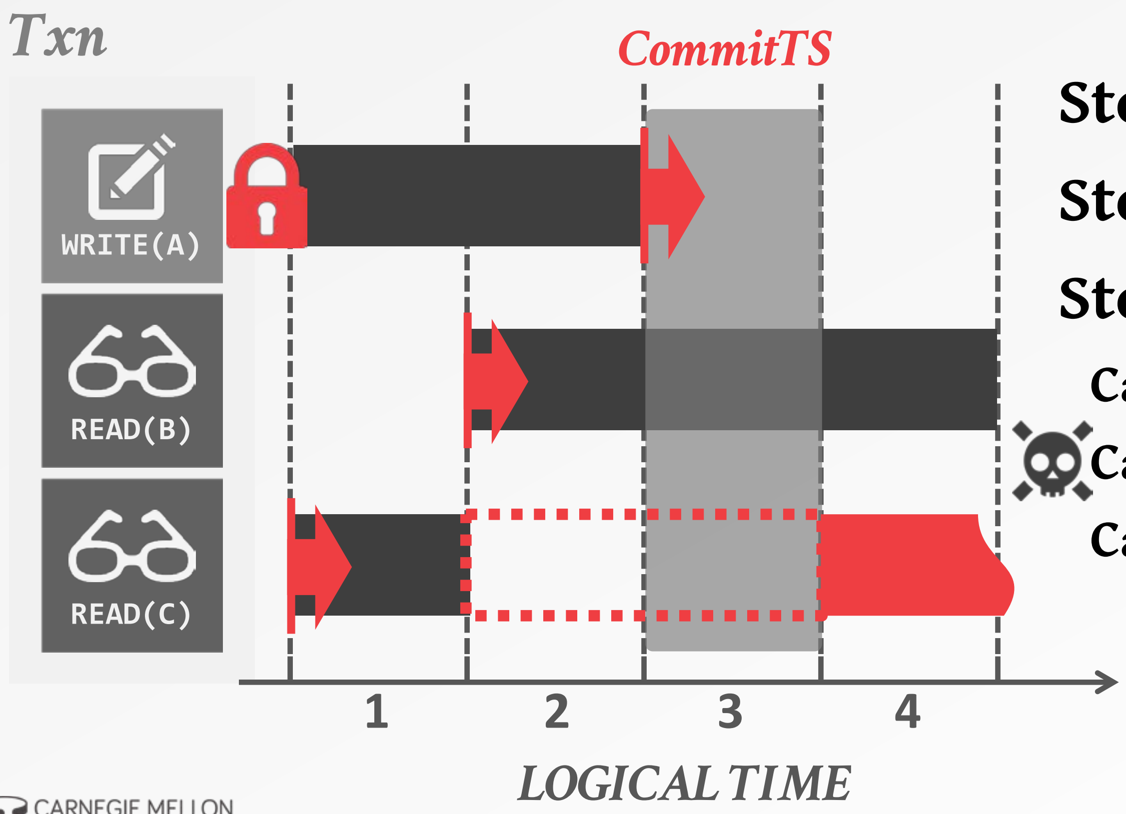
Step #2: Compute CommitTS

Step #3: Validate Read Set

Case 1: Latest Version

Case 2: Updated Before CommitTS

TICTOC: VALIDATION PHASE



Step #1: Lock Write Set

Step #2: Compute CommitTS

Step #3: Validate Read Set

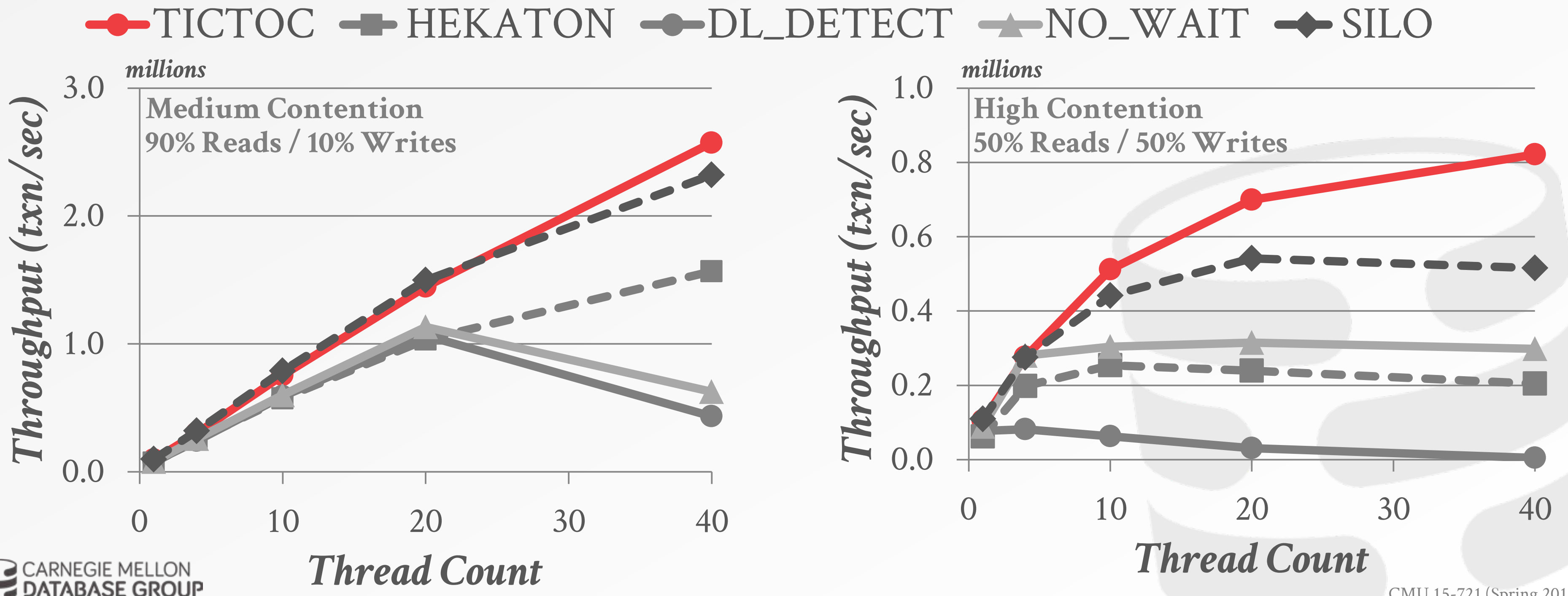
Case 1: Latest Version

Case 2: Updated Before CommitTS

Case 3: Updated After CommitTS

TICTOC: PERFORMANCE

Database: 10GB YCSB
Processor: 4 sockets, 10 cores per socket



PARTING THOUGHTS

Trade-off between aborting txns early or later.

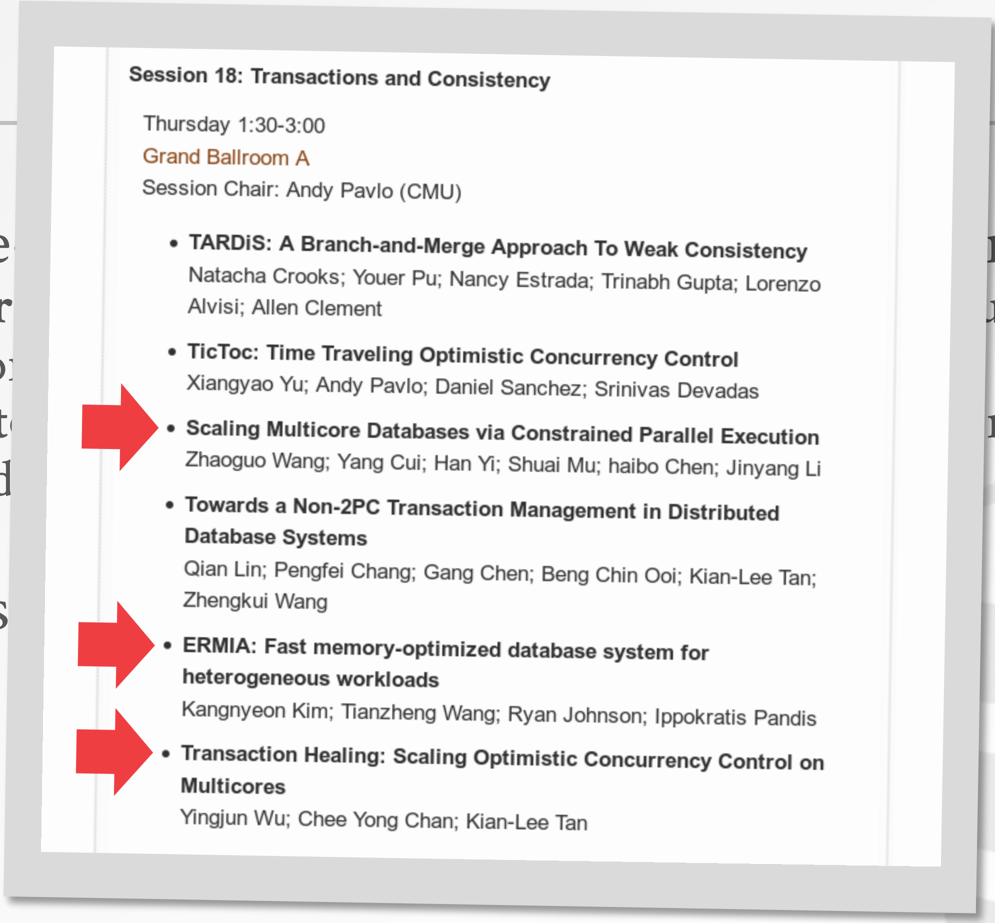
→ **Early**: Avoid wasted work for txns that will eventually abort, but has checking overhead.

→ **Later**: No runtime overhead but lots of wasted work under high contention.

Silo is a very influential system.

Trade… …r.
→ Ear… …ually
abo…
→ Lat… …rk
und…

Silo is…



Session 18: Transactions and Consistency

Thursday 1:30-3:00
Grand Ballroom A
Session Chair: Andy Pavlo (CMU)

- **TARDiS: A Branch-and-Merge Approach To Weak Consistency**
  Natacha Crooks; Youer Pu; Nancy Estrada; Trinabh Gupta; Lorenzo Alvisi; Allen Clement
- **TicToc: Time Traveling Optimistic Concurrency Control**
  Xiangyao Yu; Andy Pavlo; Daniel Sanchez; Srinivas Devadas
- **Scaling Multicore Databases via Constrained Parallel Execution**
  Zhaoguo Wang; Yang Cui; Han Yi; Shuai Mu; haibo Chen; Jinyang Li
- **Towards a Non-2PC Transaction Management in Distributed Database Systems**
  Qian Lin; Pengfei Chang; Gang Chen; Beng Chin Ooi; Kian-Lee Tan; Zhengkui Wang
- **ERMIA: Fast memory-optimized database system for heterogeneous workloads**
  Kangnyeon Kim; Tianzheng Wang; Ryan Johnson; Ippokratis Pandis
- **Transaction Healing: Scaling Optimistic Concurrency Control on Multicores**
  Yingjun Wu; Chee Yong Chan; Kian-Lee Tan

NEXT CLASS

Multi-Version Concurrency Control