Carnegie Mellon University

Timestamp Ordering Concurrency Control

Lecture #18

Database Systems
15-445/15-645
Fall 2018

AP
Andy Pavlo
Computer Science
Carnegie Mellon Univ.

ADMINISTRIVIA

Homework #4: Monday Nov 12th @ 11:59pm

Project #3: Monday Nov 19th @ 11:59am

CONCURRENCY CONTROL APPROACHES

Two-Phase Locking (2PL)

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

Timestamp Ordering (T/O)

→ Determine serializability order of txns before they execute.

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 Protocol

Optimistic Concurrency Control

Partition-based Timestamp Ordering

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 same TS.

Else:

→ Allow T_i to read **X**.

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

→ Have to 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 have to make a local copy of X to ensure repeatable reads for T_i.

BASIC T/O – EXAMPLE #1

…edule

$TS(T_1)=1$　　*$TS(T_2)=2$*

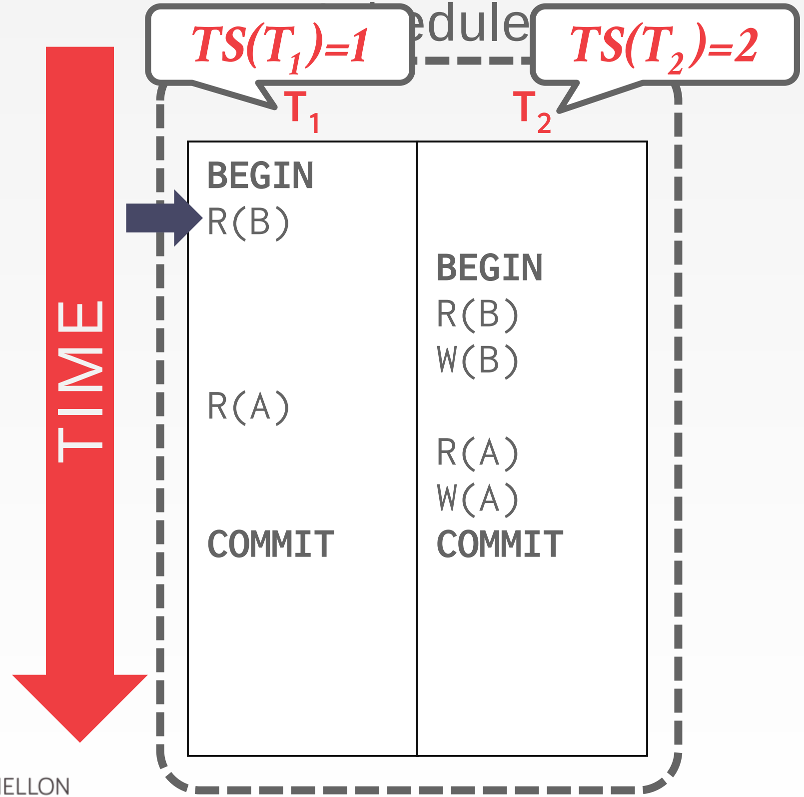
TIME

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

Database

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

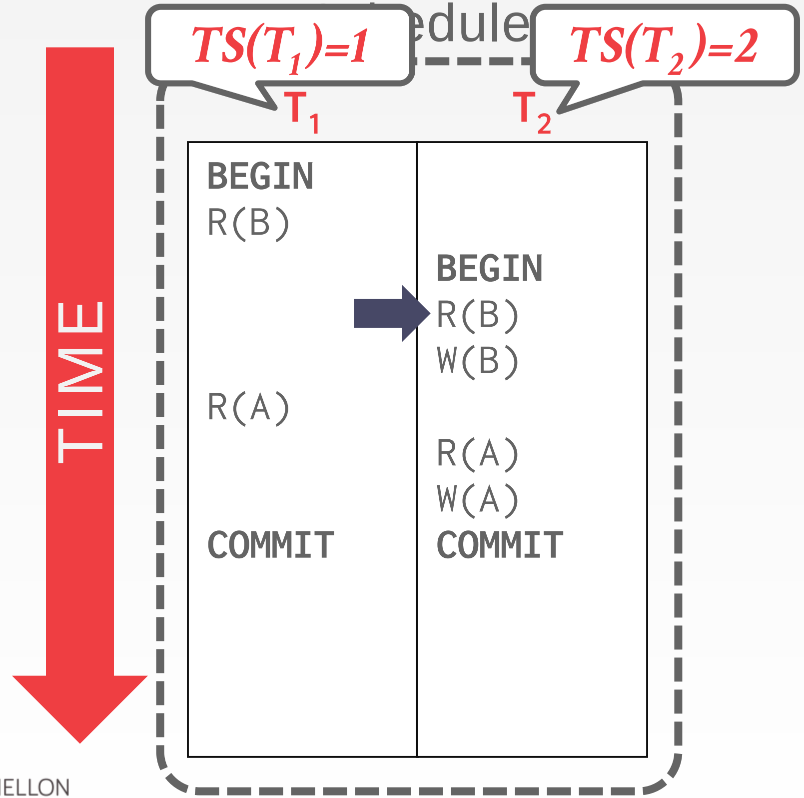
BASIC T/O – EXAMPLE #1



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

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

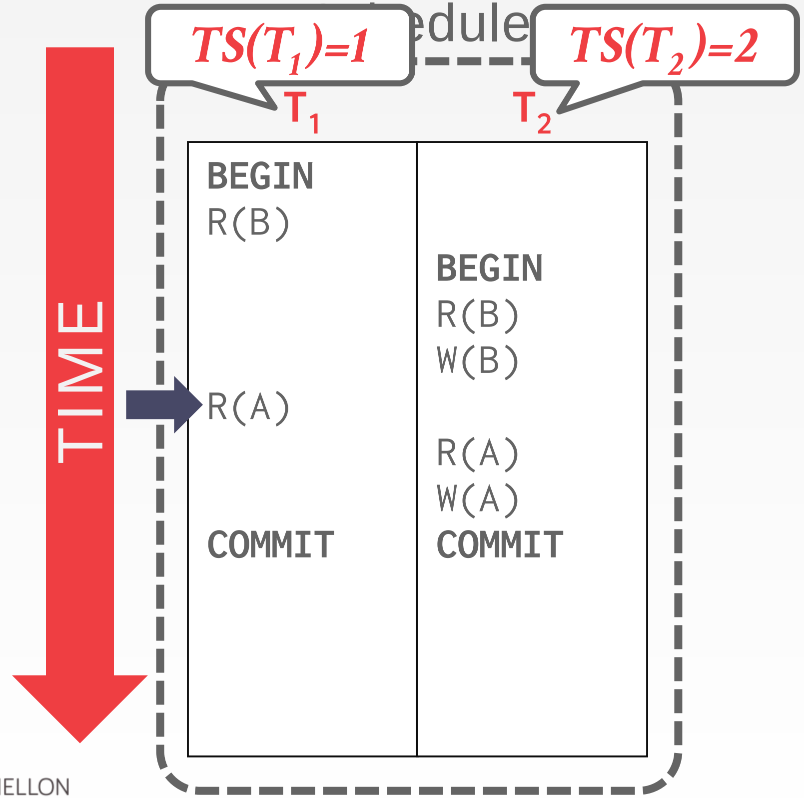
TIME

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

Database

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

BASIC T/O – EXAMPLE #1



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)
	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)
	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

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

T_1 cannot overwrite update by T_2, so the DBMS has to abort it!

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 and allow the txn to continue.

→ 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

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

TIME

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	

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.

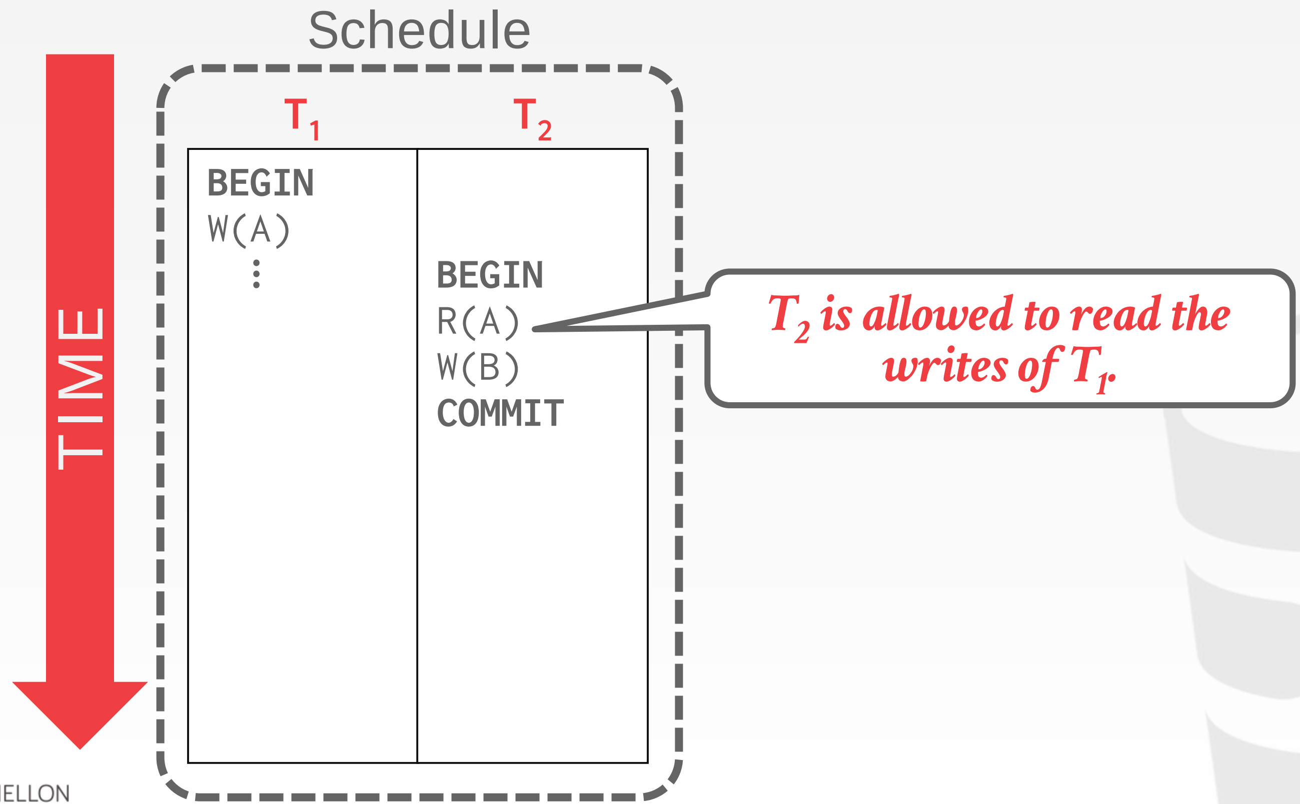
Permits schedules that are not **recoverable**.

RECOVERABLE SCHEDULES

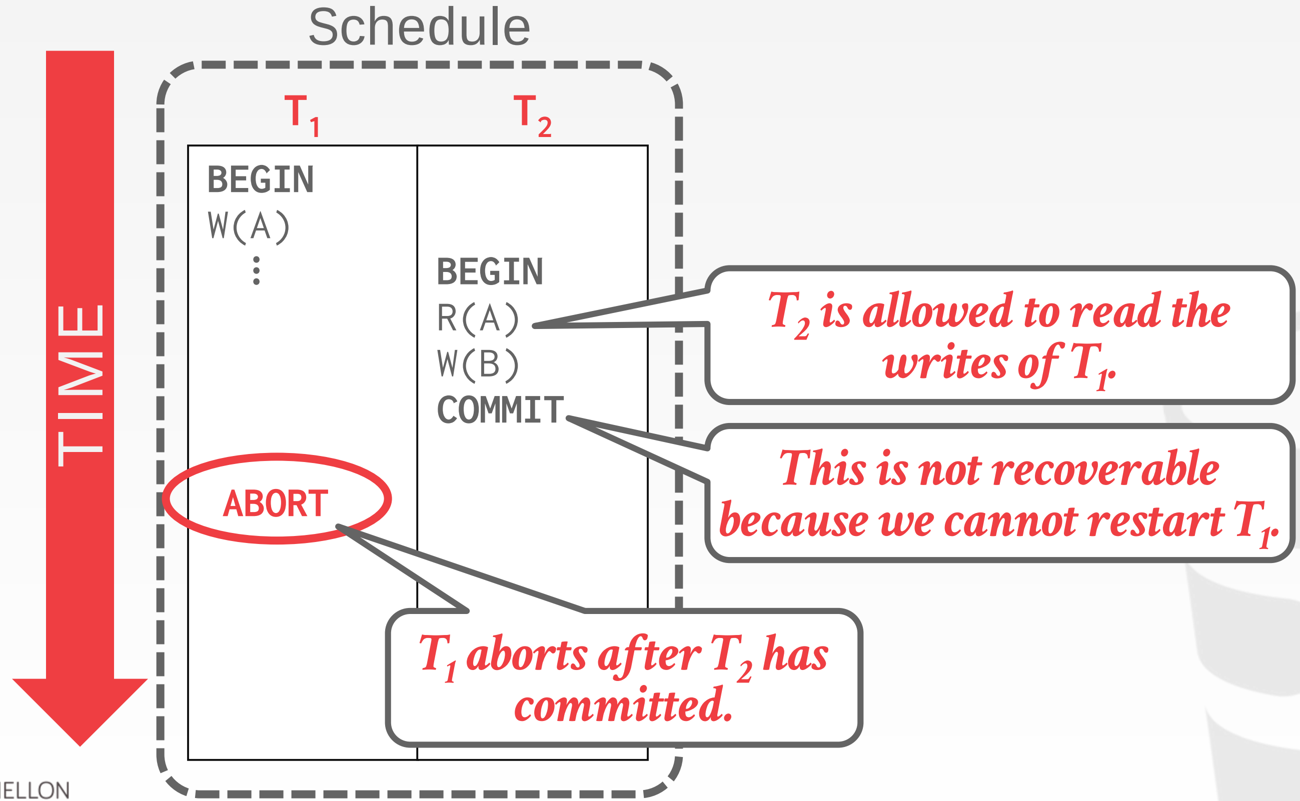
A schedule is **recoverable** if txns commit only after all txns whose changes they read, commit.

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

RECOVERABLE SCHEDULES



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.

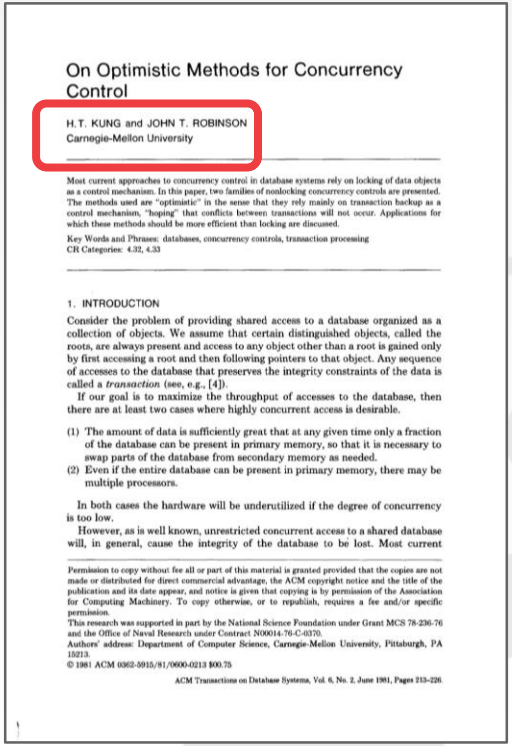
A better approach is to optimize for the no-conflict case.

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:

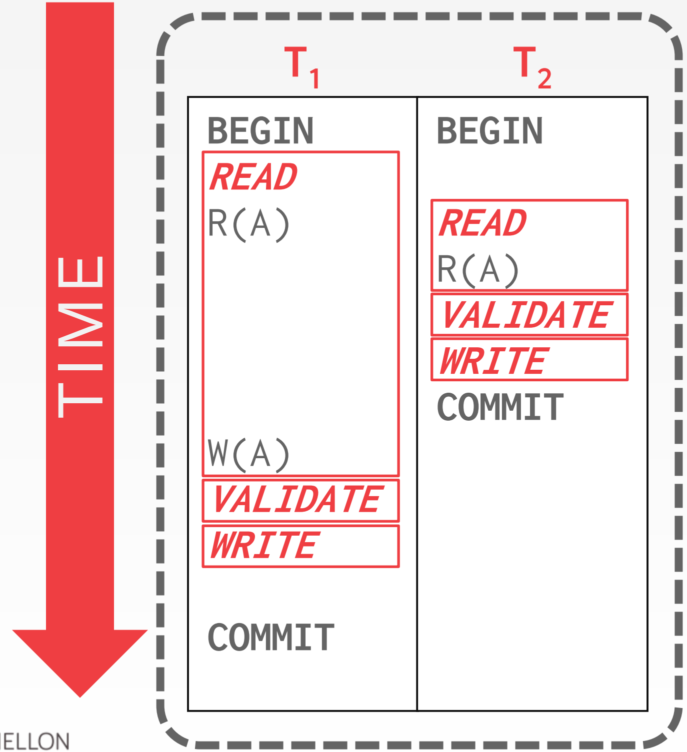
→ When a txn commits, check whether it conflicts with other txns.

#3 – Write Phase:

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

OCC – EXAMPLE

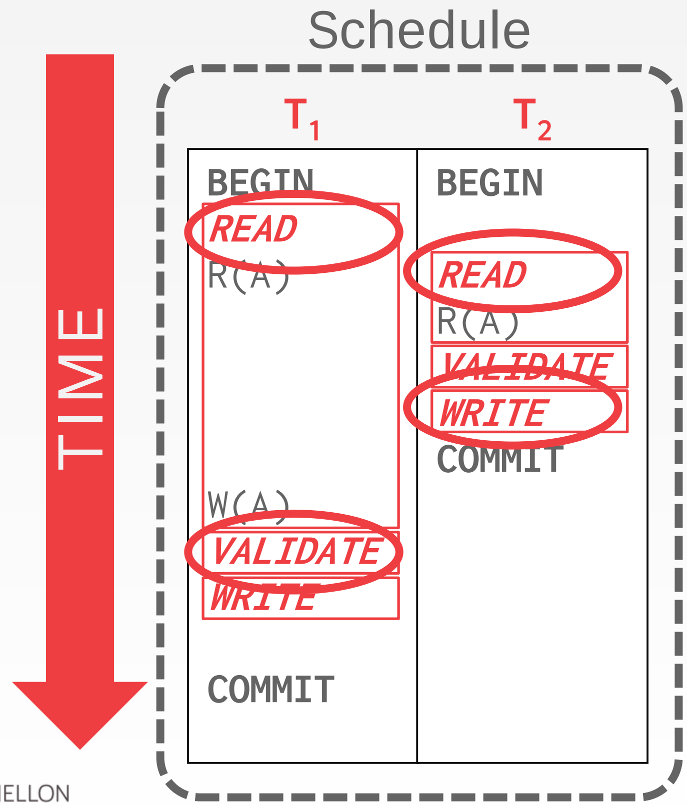
Schedule



Database

Object	Value	W-TS
A	123	0
–	–	–

OCC – EXAMPLE

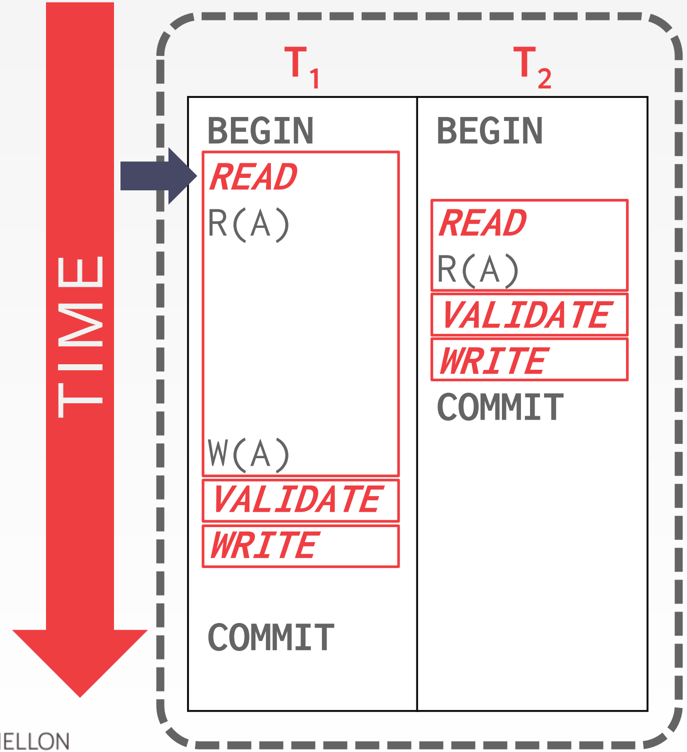


Database

Object	Value	W-TS
A	123	0
–	–	–

OCC – EXAMPLE

Schedule



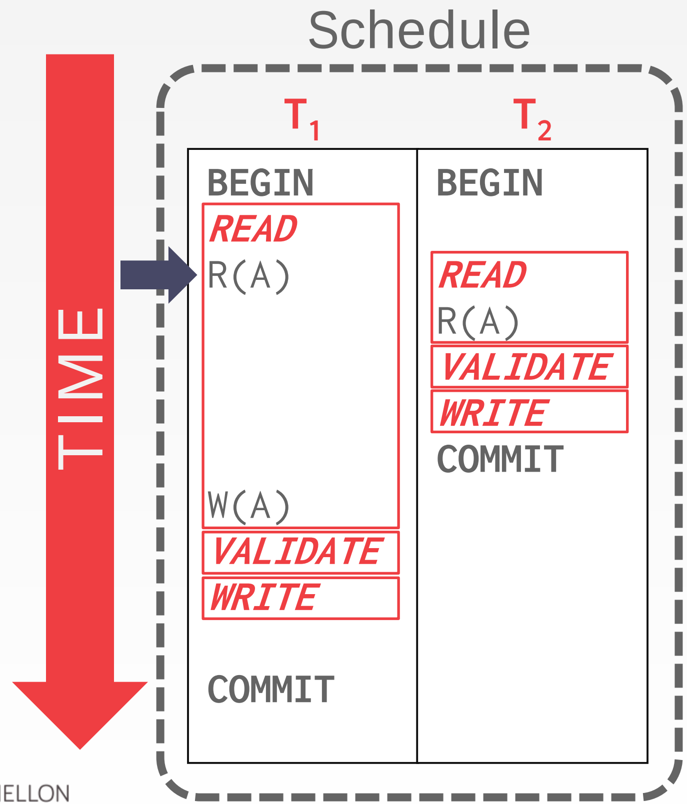
Database

Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

Object	Value	W-TS
–	–	–
–	–	–

OCC – EXAMPLE



Database

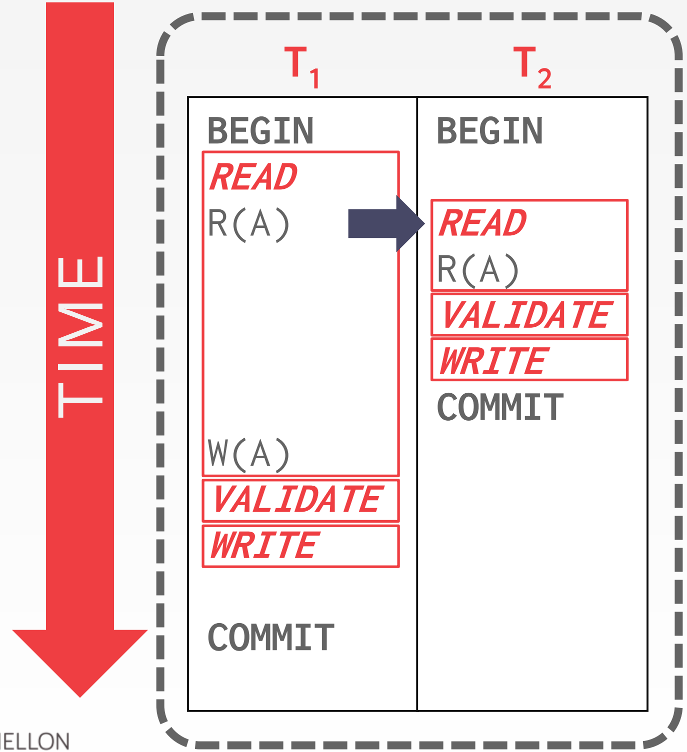
Object	Value	W-TS
A	123	0
–	–	–

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_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

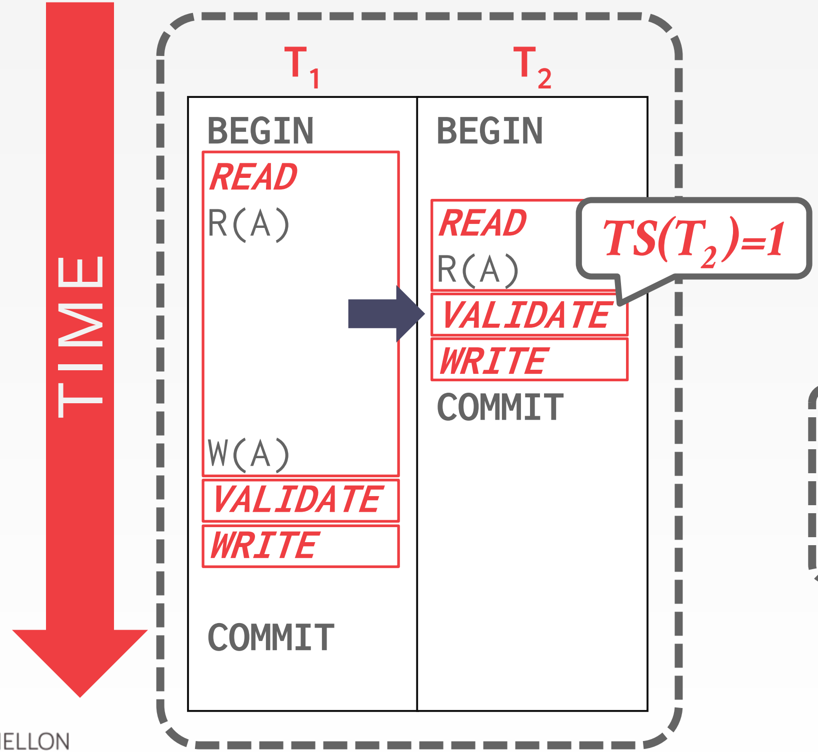
Object	Value	W-TS
A	123	0
–	–	–

T_2 Workspace

Object	Value	W-TS
A	123	0
–	–	–

OCC – EXAMPLE

Schedule



Database

Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

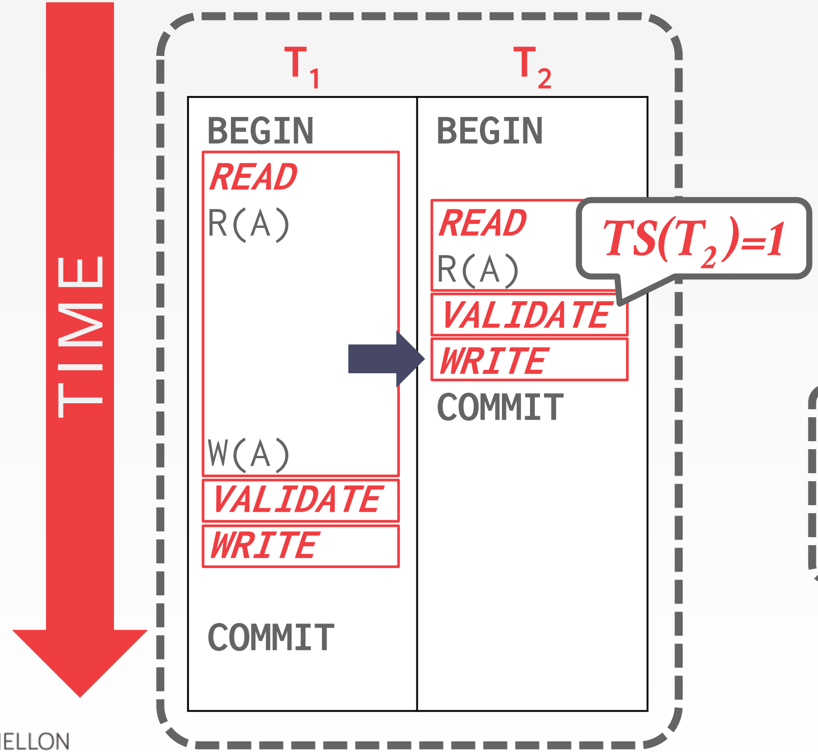
Object	Value	W-TS
A	123	0
–	–	–

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

TIME

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

$TS(T_2)=1$

Database

Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

Object	Value	W-TS
A	123	0
–	–	–

OCC – EXAMPLE

Schedule

TIME

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

$TS(T_2)=1$

Database

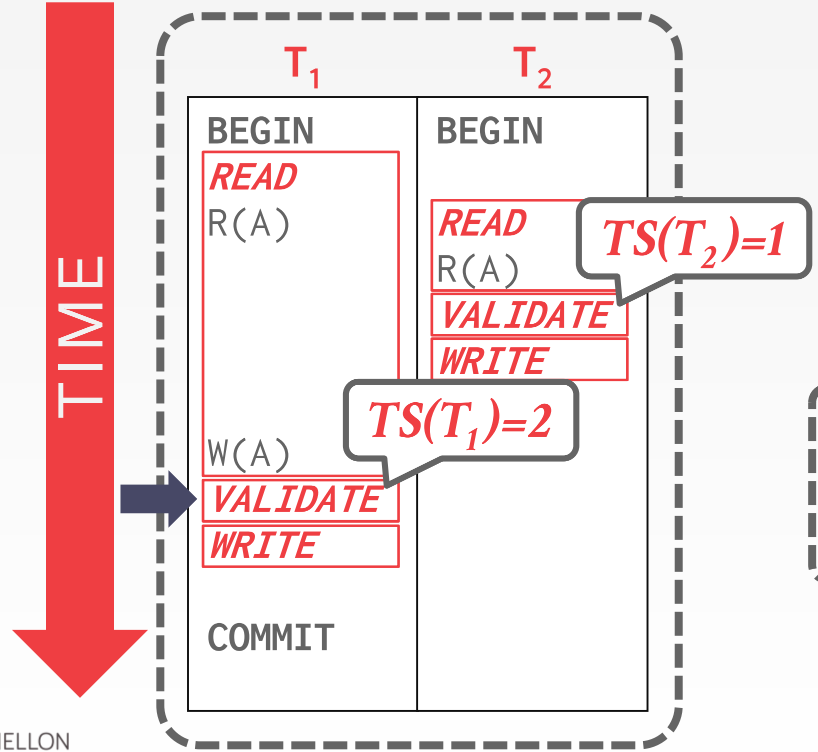
Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

Object	Value	W-TS
A	456	∞
–	–	–

OCC – EXAMPLE

Schedule



Database

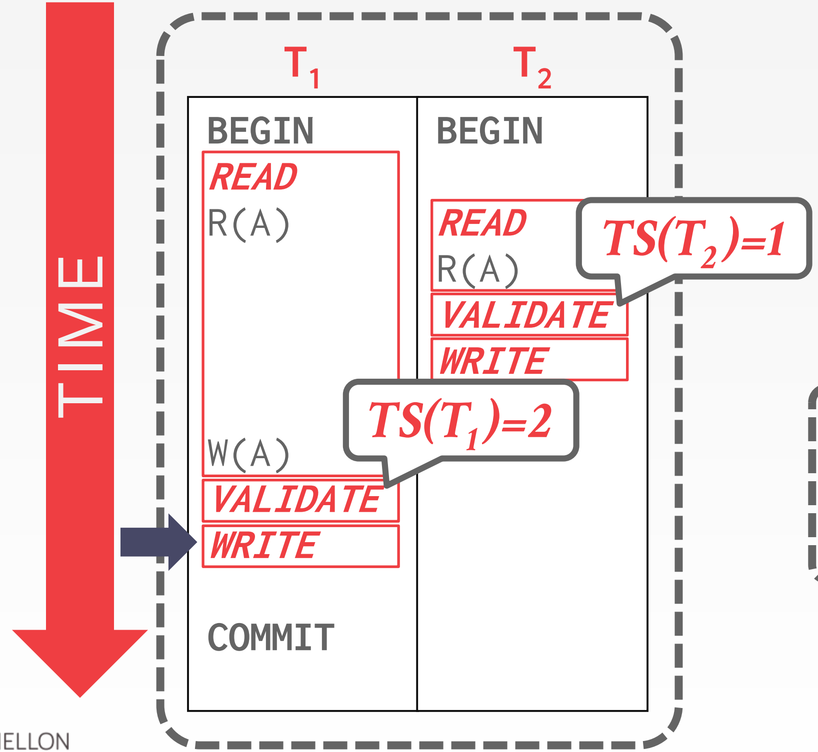
Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

Object	Value	W-TS
A	456	∞
–	–	–

OCC – EXAMPLE

Schedule



Database

Object	Value	W-TS
A	456	2
–	–	–

T_1 Workspace

Object	Value	W-TS
A	456	∞
–	–	–

OCC – VALIDATION PHASE

The DBMS needs to guarantee only serializable schedules are permitted.

T_i checks other txns for RW and WW conflicts and makes sure that all conflicts go one way (from older txns to younger txns).

OCC – SERIAL VALIDATION

Maintain global view of all active txns.

Record read set and write set while txns are running and write into private workspace.

Execute **Validation** and **Write** phase inside a protected critical section.

OCC – VALIDATION PHASE

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

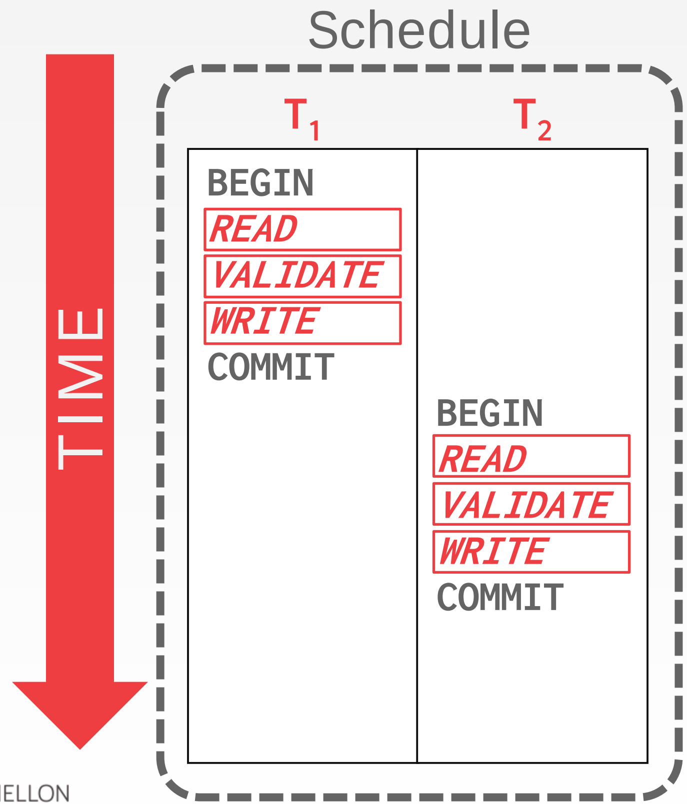
Check the timestamp ordering of the committing txn with all other running txns.

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

OCC – VALIDATION STEP #1

T_i completes all three phases before T_j begins.

OCC – VALIDATION STEP #1



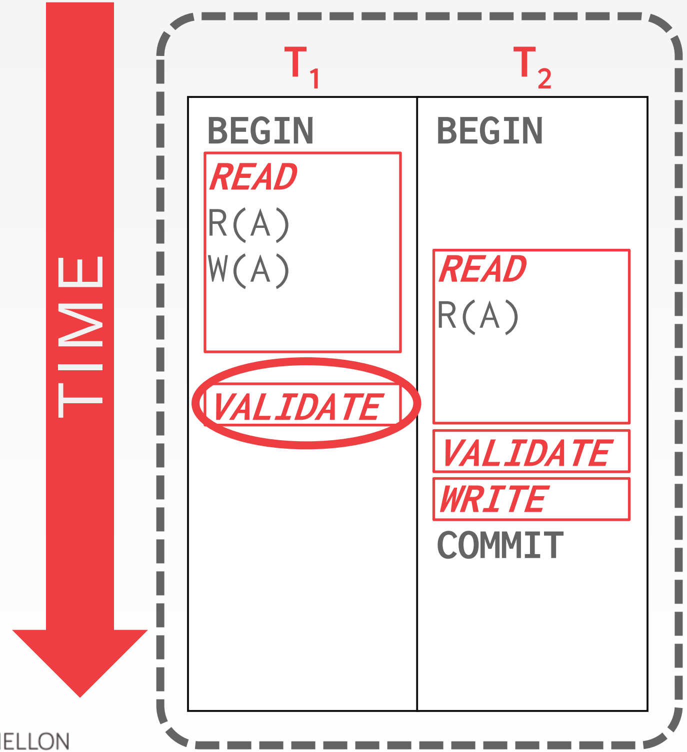
OCC – VALIDATION STEP #2

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

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

OCC – VALIDATION STEP #2

Schedule



Database

Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

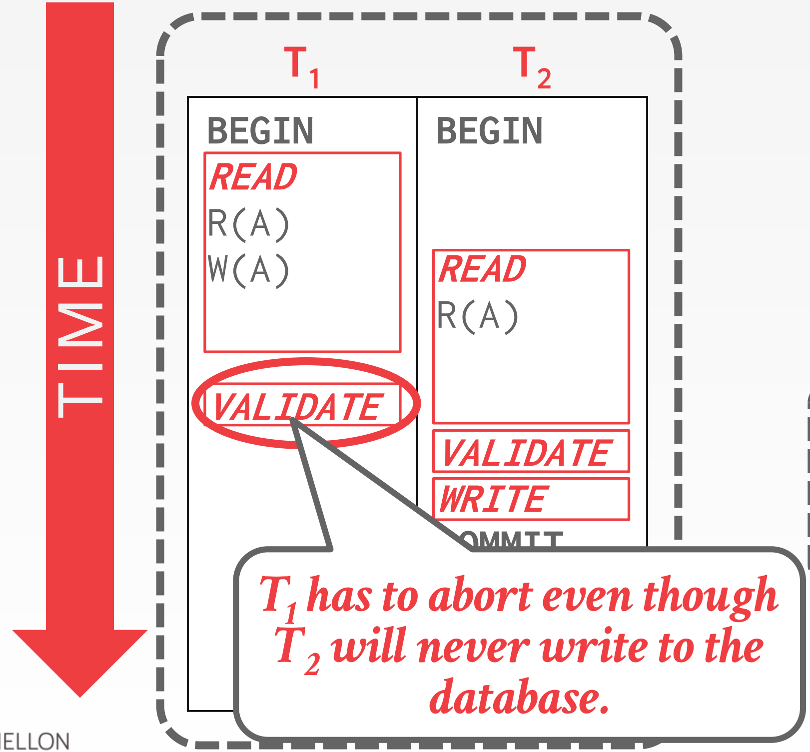
Object	Value	W-TS
A	123	∞
–	–	–

T_2 Workspace

Object	Value	W-TS
A	123	0
–	–	–

OCC – 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 – 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

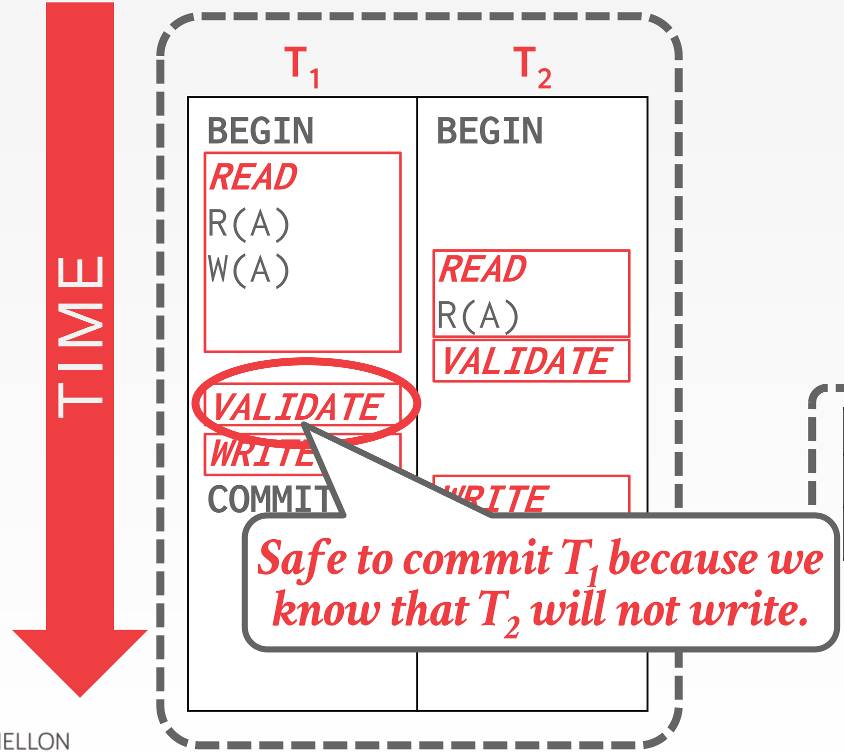
Object	Value	W-TS
A	456	∞
–	–	–

T_2 Workspace

Object	Value	W-TS
A	123	0
–	–	–

OCC – 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 – VALIDATION STEP #3

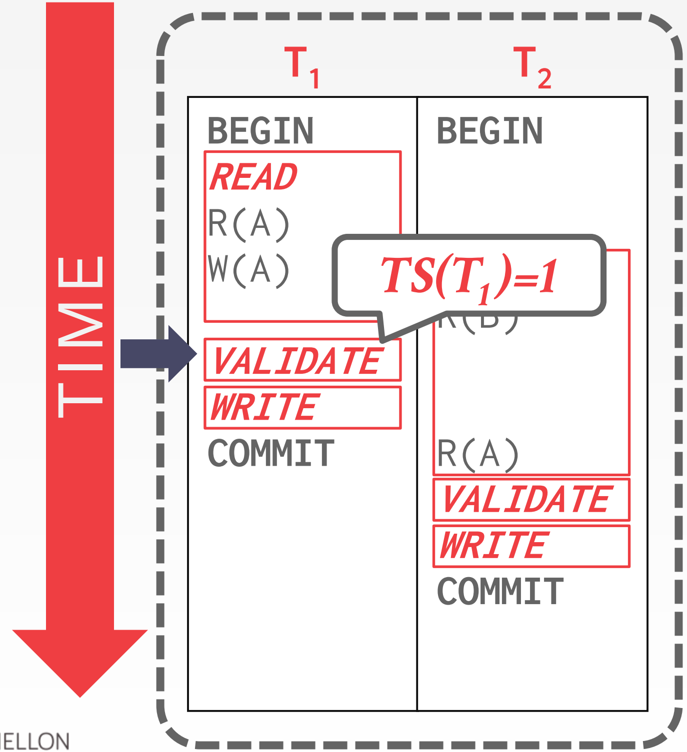
T_i completes its **Read** phase before T_j completes its **Read** phase

And T_i does not write to any object that is either read or written by T_j:

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

OCC – VALIDATION STEP #3

Schedule



Database

Object	Value	W-TS
A	123	0
B	XYZ	0

T_1 Workspace

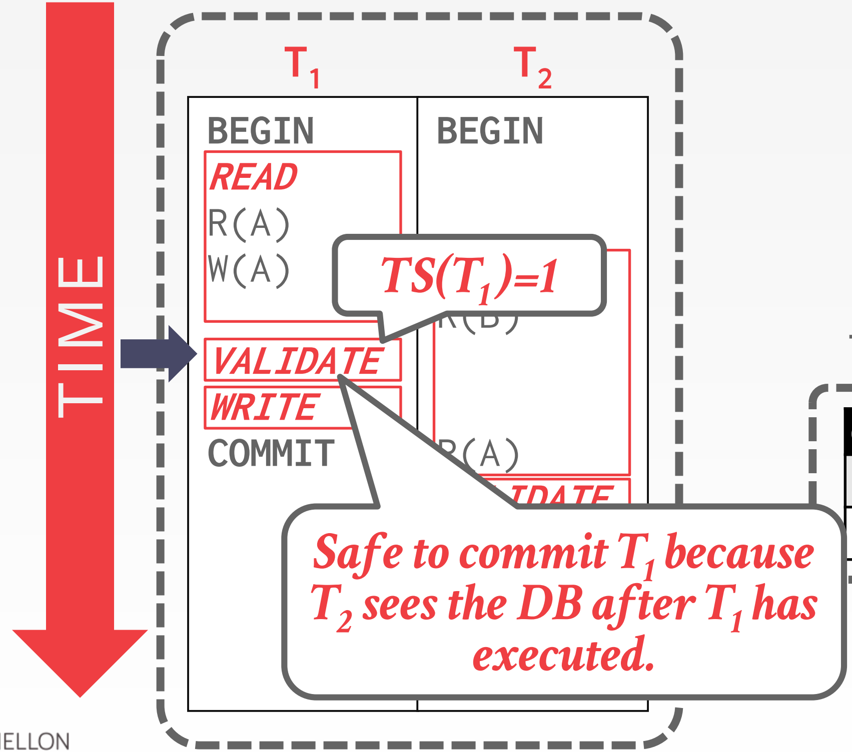
Object	Value	W-TS
A	456	∞
–	–	–

T_2 Workspace

Object	Value	W-TS
B	XYZ	0
–	–	–

OCC – 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 – VALIDATION STEP #3

Schedule

TIME

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

$TS(T_1)=1$

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 – VALIDATION STEP #3

Schedule

TIME →

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

Database

Object	Value	W-TS
A	456	1
B	XYZ	0

T_2 Workspace

Object	Value	W-TS
B	XYZ	0
–	–	–

OCC – VALIDATION STEP #3

Schedule

TIME

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

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 – 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.

OBSERVATION

When a txn commits, all previous T/O schemes check to see whether there is a conflict with concurrent txns.

→ This requires latches.

If you have a lot of concurrent txns, then this is slow even if the conflict rate is low.

PARTITION-BASED T/O

Split the database up in disjoint subsets called ***horizontal partitions*** (aka shards).

Use timestamps to order txns for serial execution at each partition.

→ Only check for conflicts between txns that are running in the same partition.

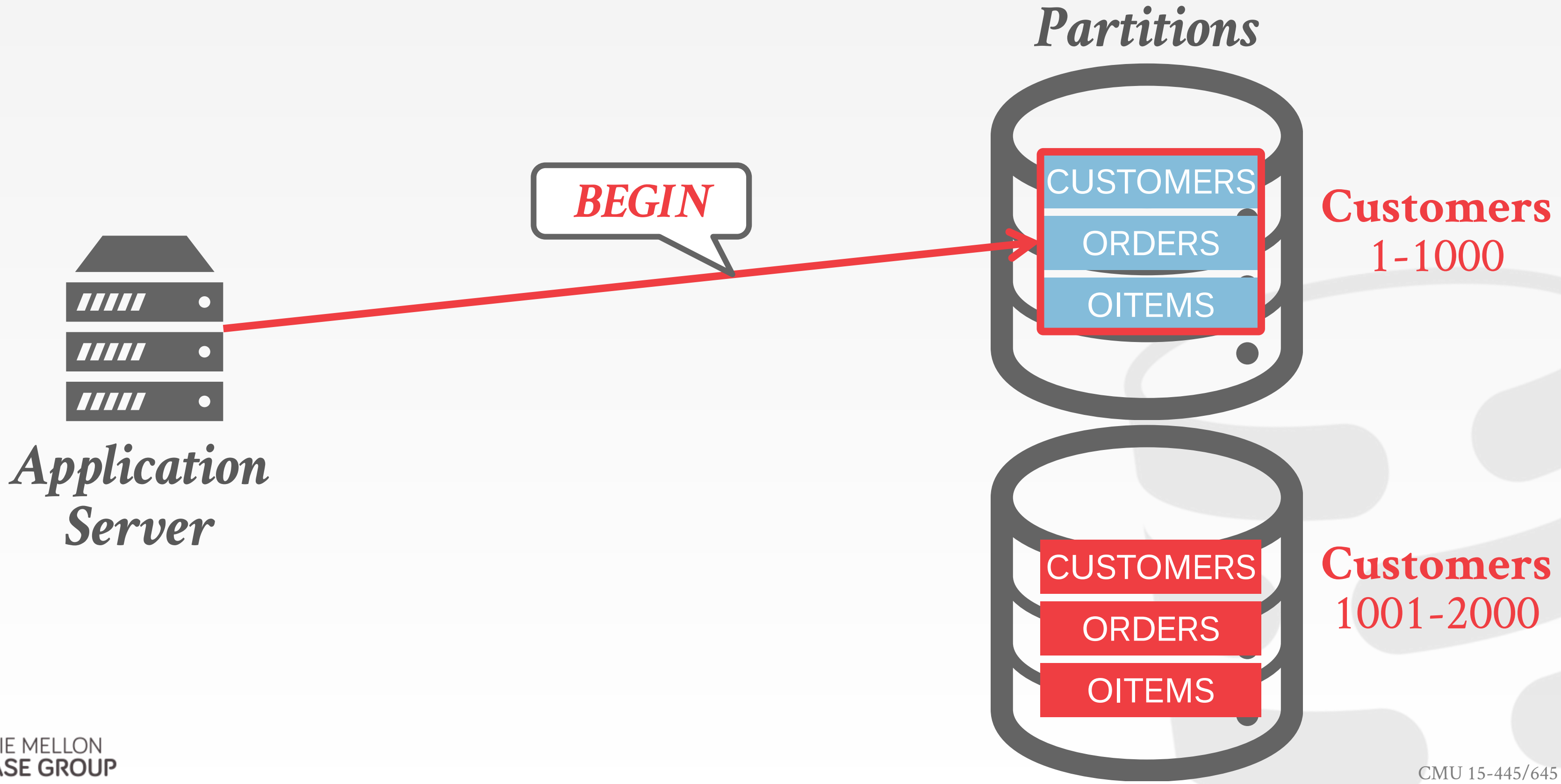
DATABASE PARTITIONING

```
CREATE TABLE customer (
  c_id INT PRIMARY KEY,
  c_email VARCHAR UNIQUE,
  ⋮
);
```

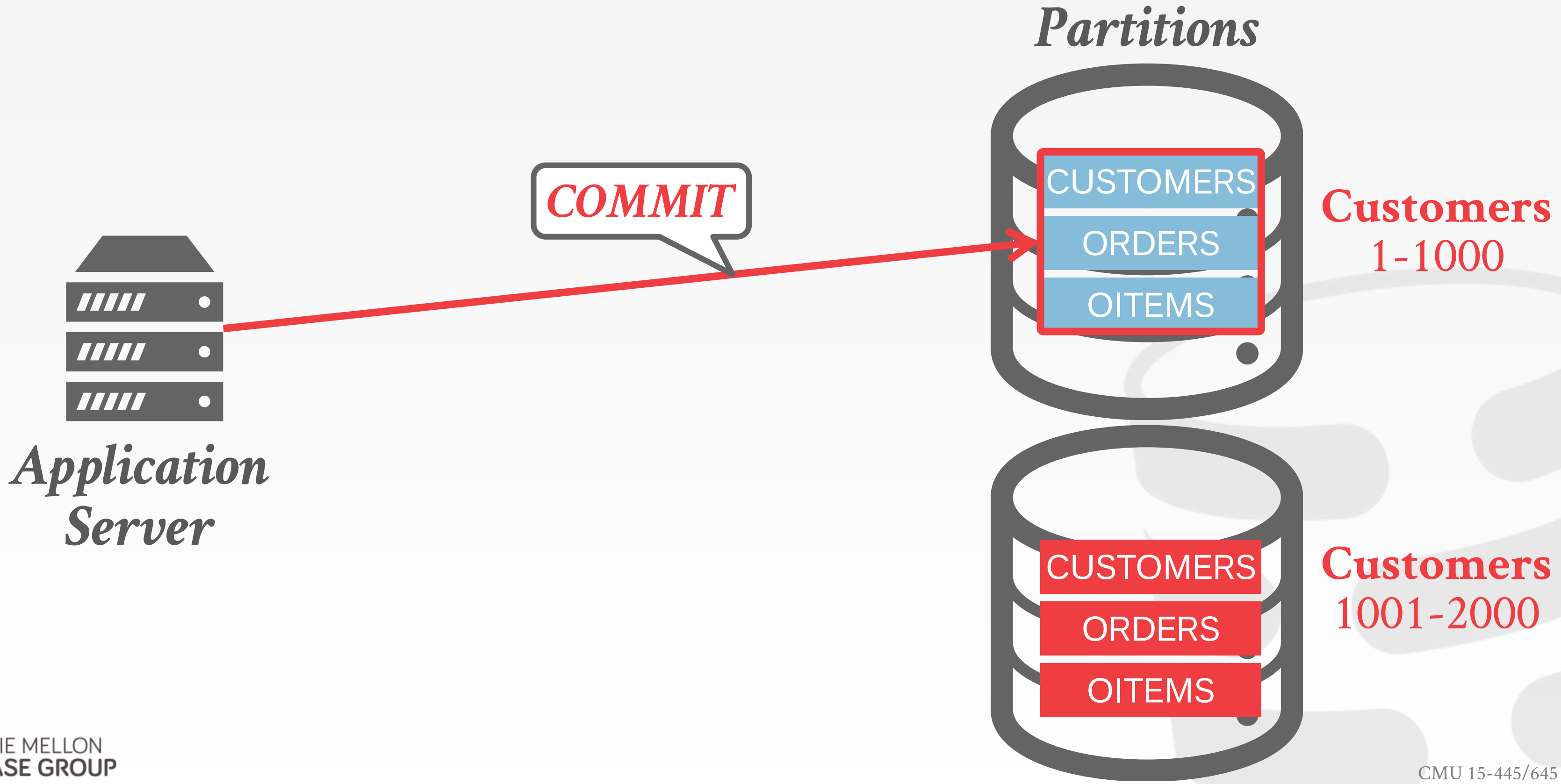
```
CREATE TABLE orders (
  o_id INT PRIMARY KEY,
  o_c_id INT REFERENCES
      ↳customer (c_id),
  ⋮
);
```

```
CREATE TABLE oitems (
  oi_id INT PRIMARY KEY,
  oi_o_id INT REFERENCES
      ↳orders (o_id),
  oi_c_id INT REFERENCES
      ↳orders (o_c_id),
  ⋮
);
```

HORIZONTAL PARTITIONING



HORIZONTAL PARTITIONING



PARTITION-BASED T/O

Txns are assigned timestamps based on when they arrive at the DBMS.

Partitions are protected by a single lock:
→ Each txn is queued at the partitions it needs.
→ The txn acquires a partition's lock if it has the lowest timestamp in that partition's queue.
→ The txn starts when it has all of the locks for all the partitions that it will read/write.

PARTITION-BASED T/O – READS

Txns can read anything that they want at the partitions that they have locked.

If a txn tries to access a partition that it does not have the lock, it is aborted + restarted.

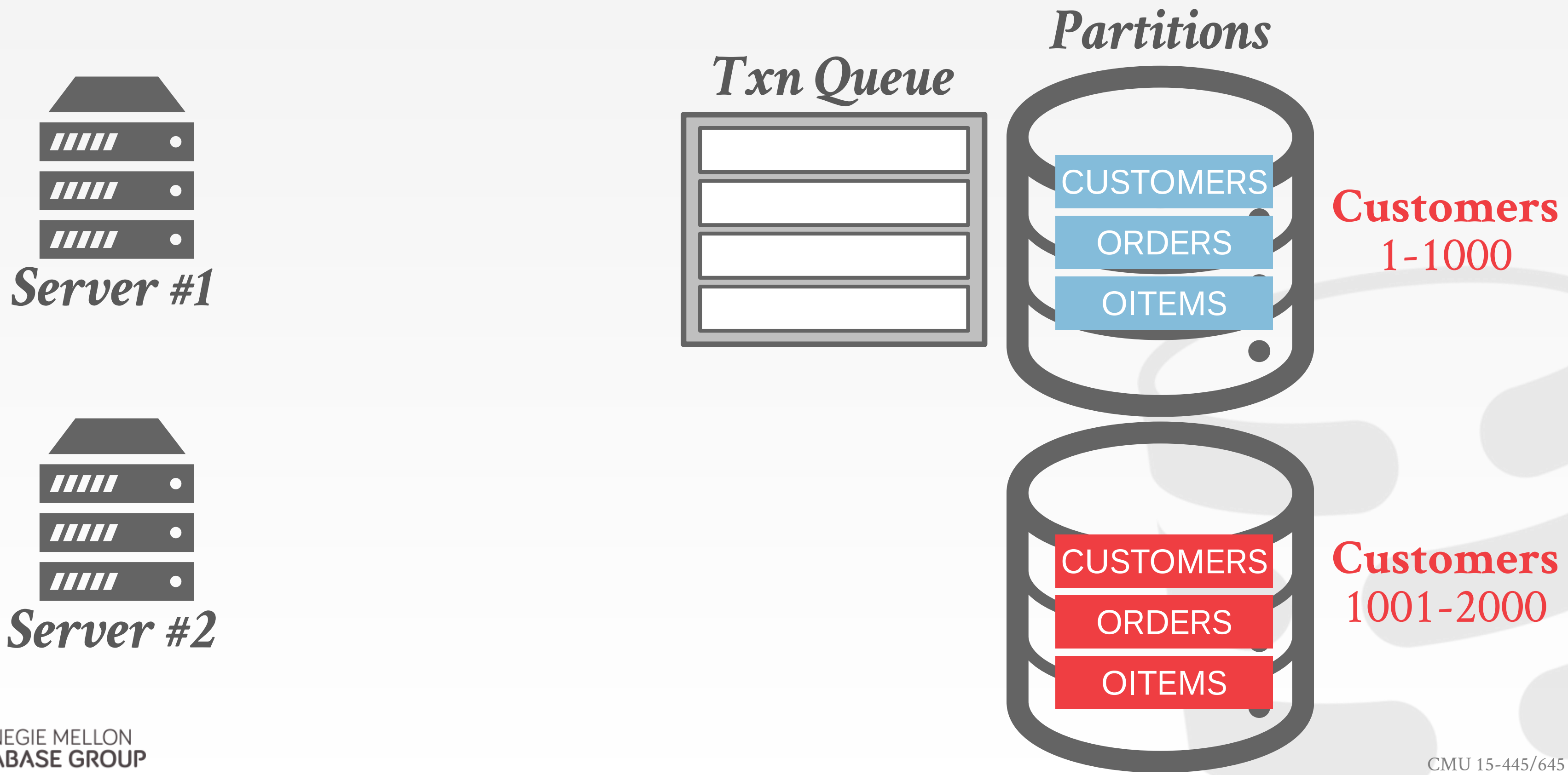
PARTITION-BASED T/O – WRITES

All updates occur in place.

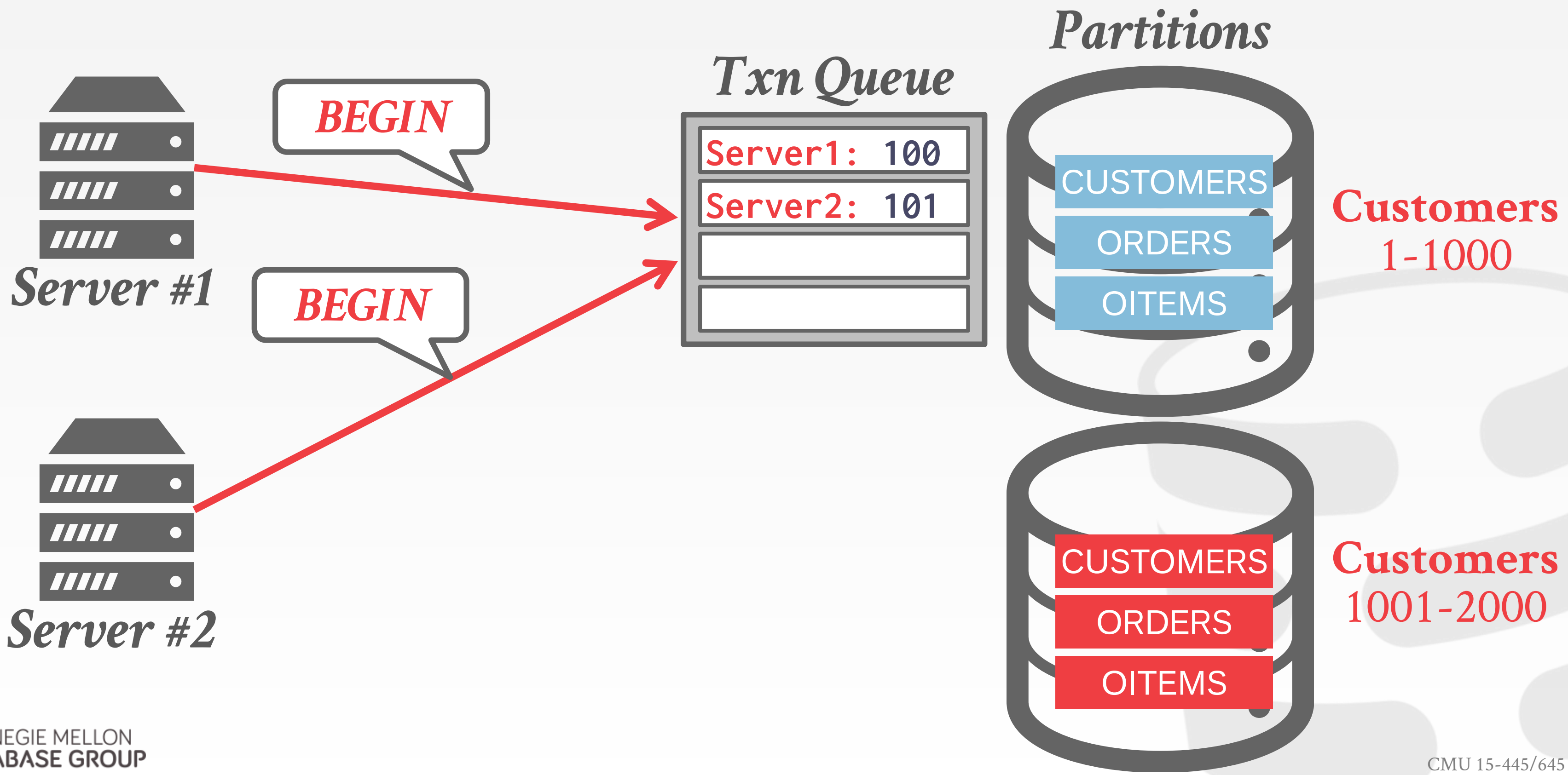
→ Maintain a separate in-memory buffer to undo changes if the txn aborts.

If a txn tries to write to a partition that it does not have the lock, it is aborted + restarted.

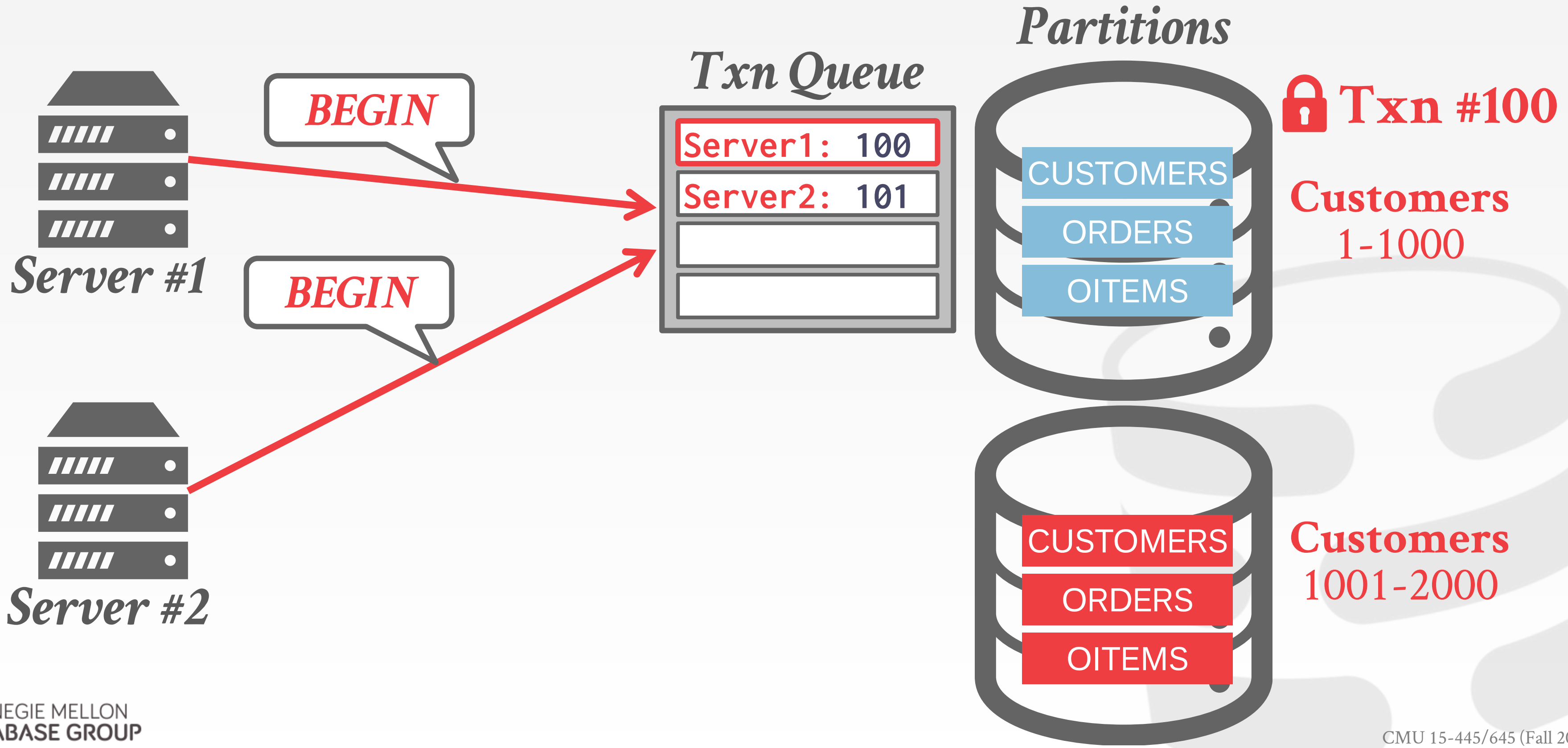
PARTITION-BASED T/O



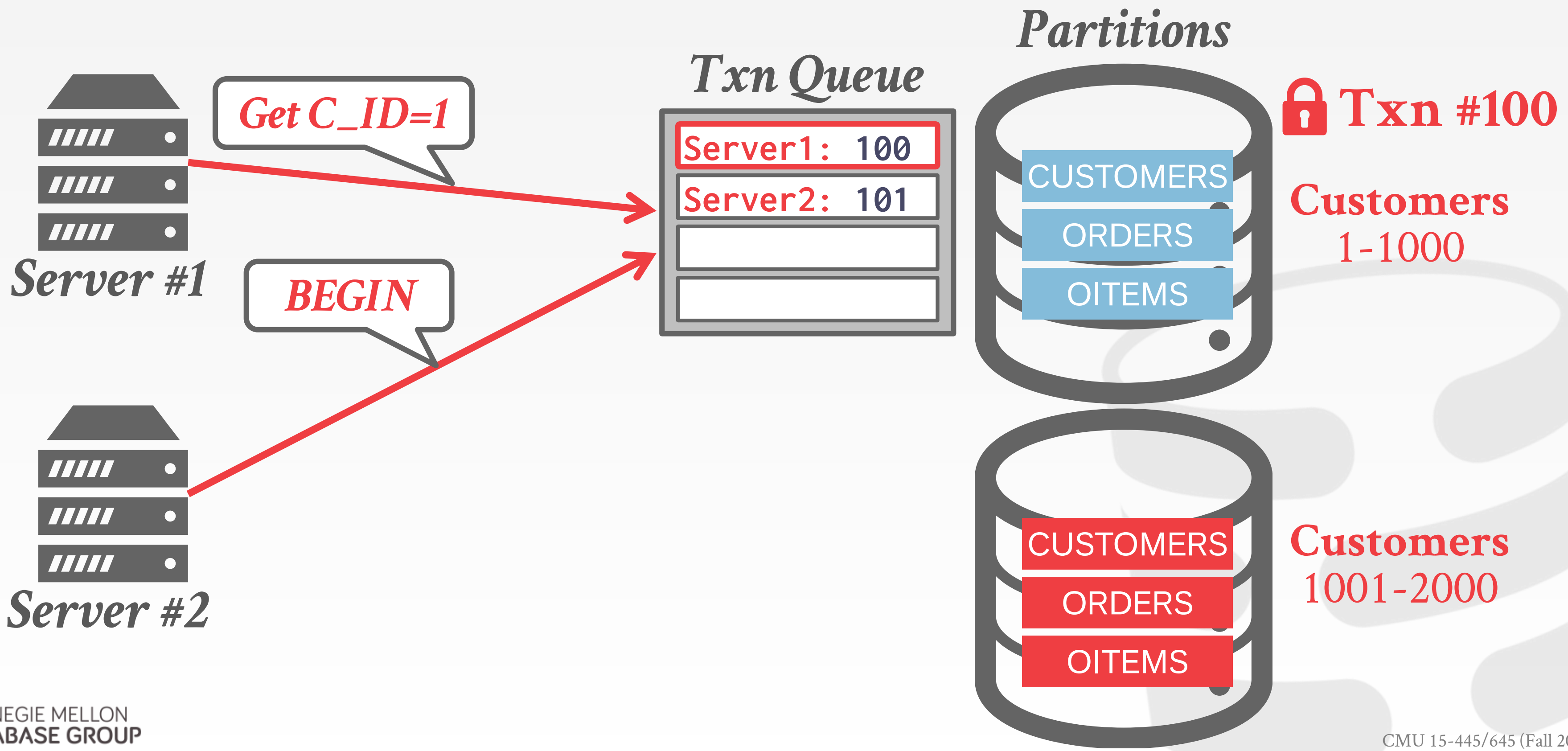
PARTITION-BASED T/O



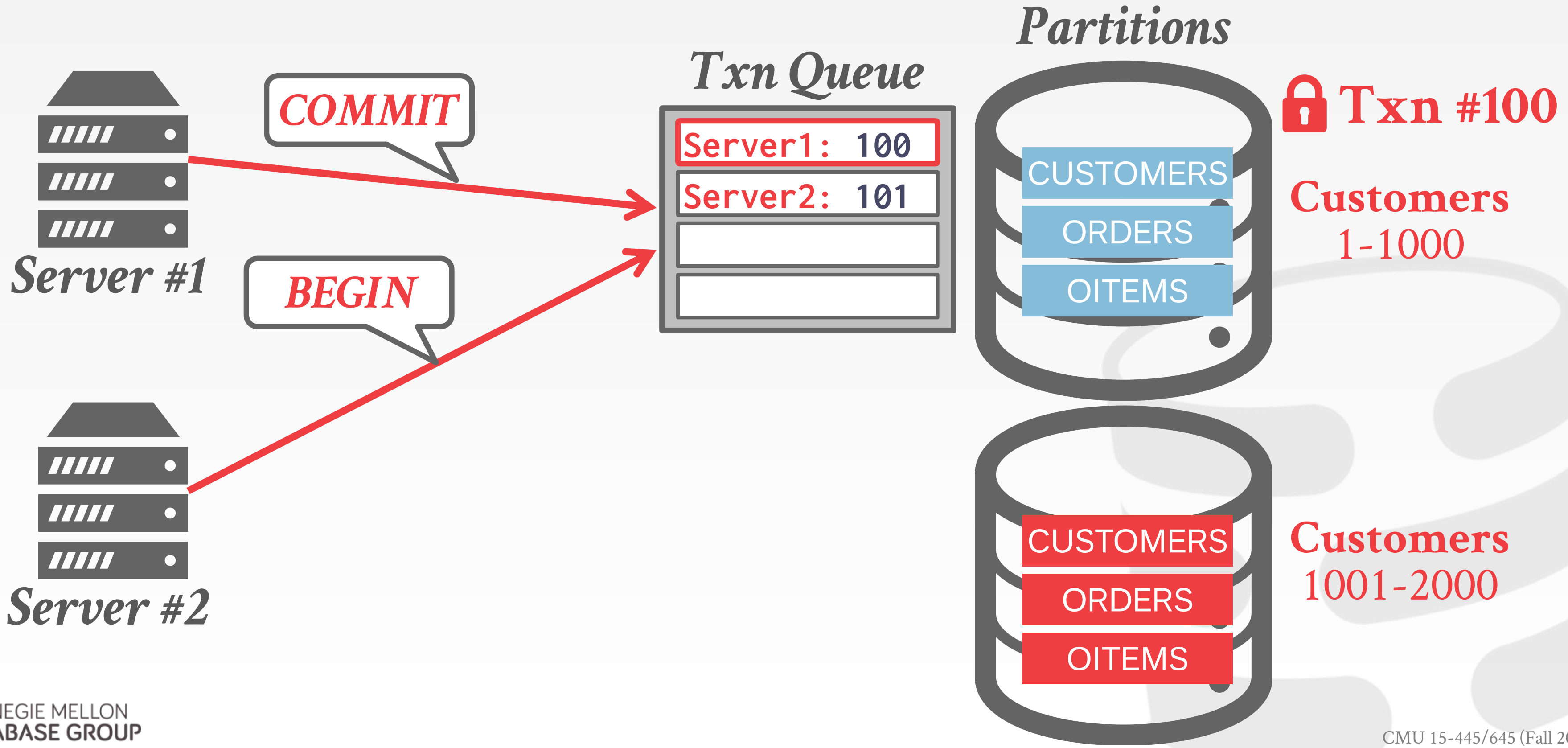
PARTITION-BASED T/O



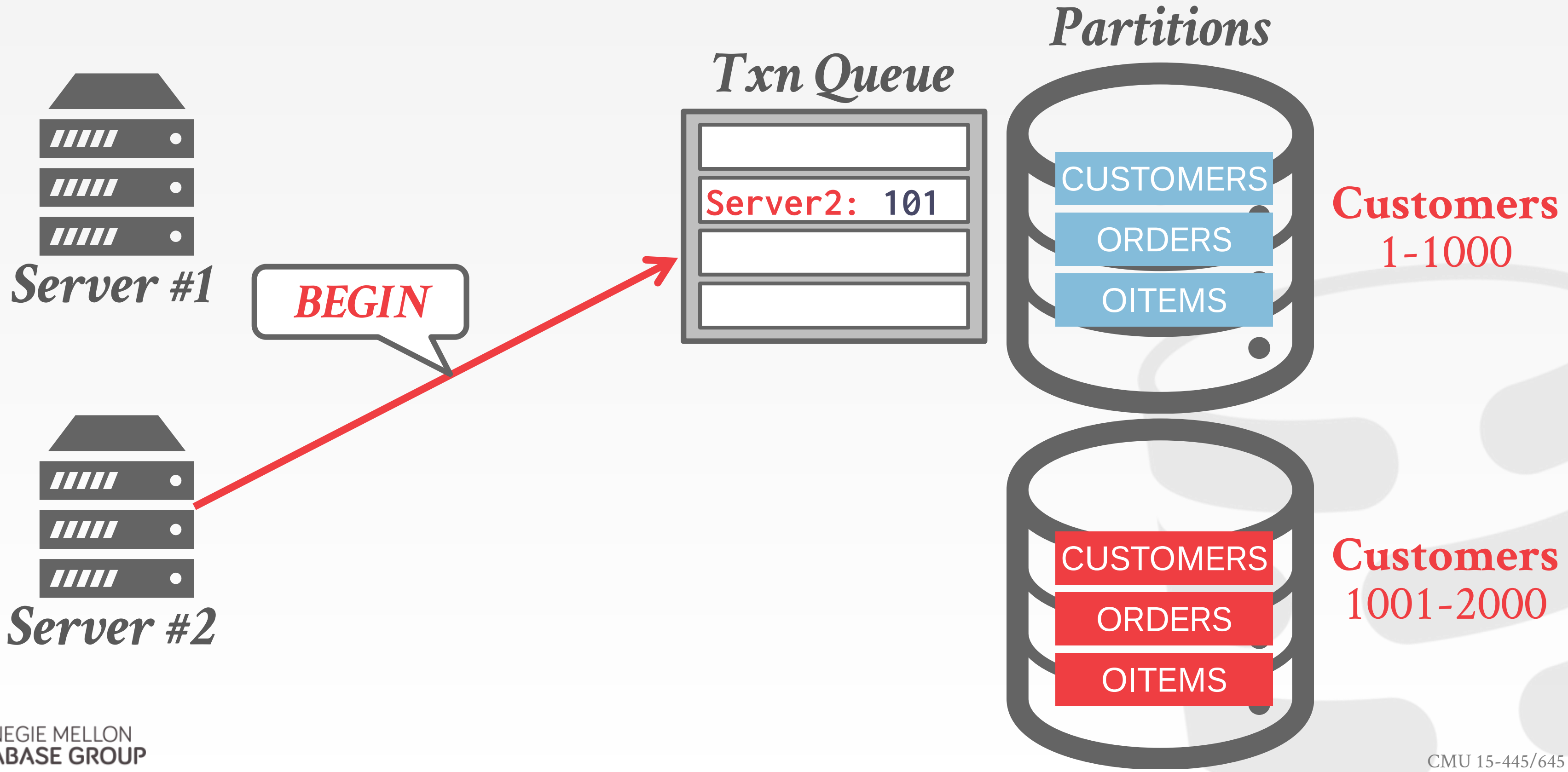
PARTITION-BASED T/O



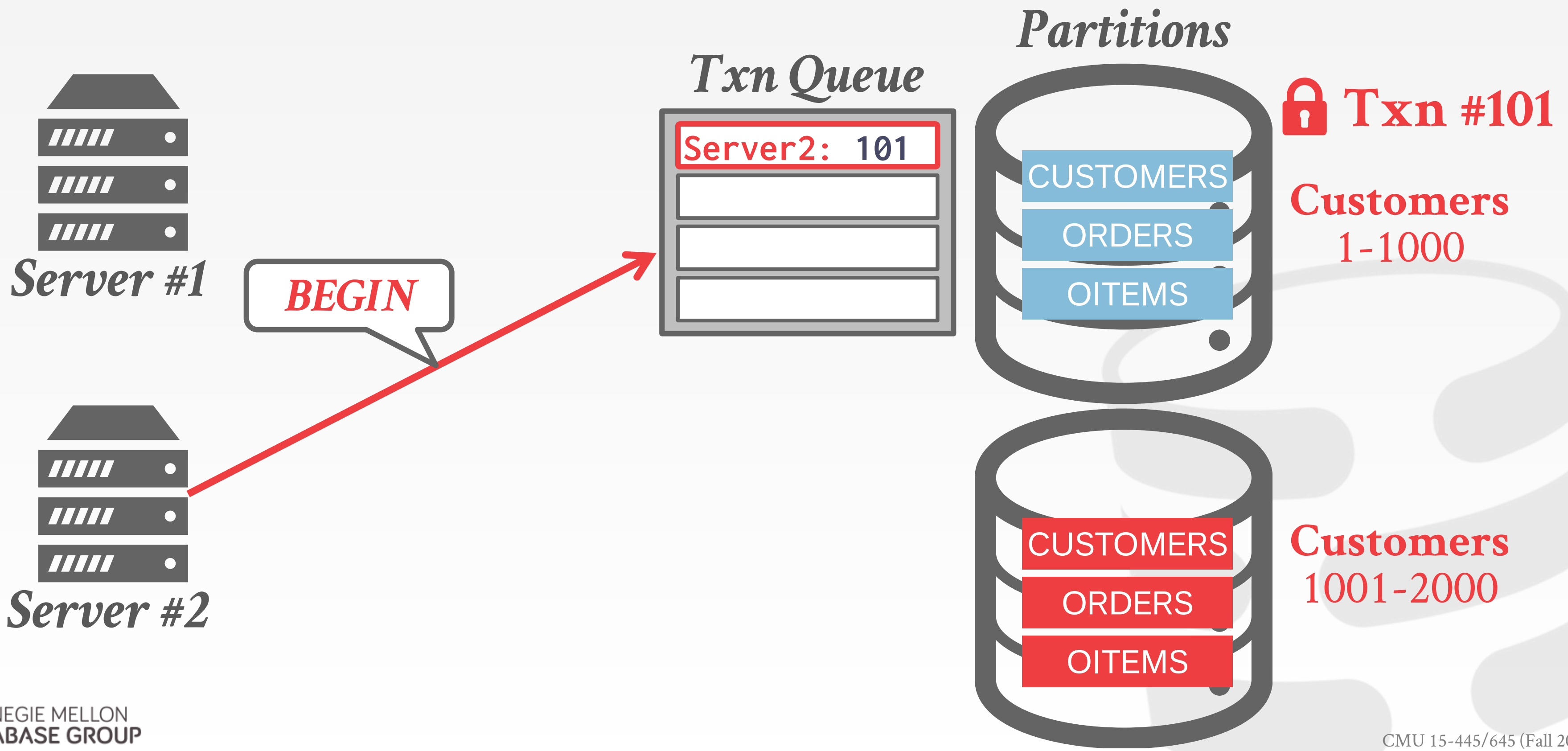
PARTITION-BASED T/O



PARTITION-BASED T/O



PARTITION-BASED T/O



PARTITIONED T/O – PERFORMANCE ISSUES

Partition-based T/O protocol is fast if:

→ The DBMS knows what partitions the txn needs before it starts.

→ Most (if not all) txns only need to access a single partition.

Multi-partition txns causes partitions to be idle while txn executes.

DYNAMIC DATABASES

Recall that so far we have only dealing with transactions that read and update data.

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

THE PHANTOM PROBLEM

Schedule

TIME

T_1	T_2
BEGIN	BEGIN
`SELECT MAX(age) FROM people WHERE status='lit'`	→ 72
	`INSERT INTO people (age=96, status='lit')`
	COMMIT
`SELECT MAX(age) FROM people WHERE status='lit'`	→ 96
COMMIT	

```
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.

PREDICATE LOCKING

Lock records that satisfy a logical predicate:

→ Example: `status='lit'`

In general, predicate locking has a lot of locking overhead.

Index locking is a special case of predicate locking that is potentially more efficient.

INDEX LOCKING

If there is a dense index on the status field 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.

REPEATING SCANS

An alternative is to just re-execute every scan again when the txn commits and check whether it gets the same result.

→ Have to retain the scan set for every range query in a txn.

→ Andy doesn't know of any commercial system that does this (only just Silo?).

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.

Not all DBMS support all isolation levels in all execution scenarios

→ Replicated Environments

The default depends on implementation…

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

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

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

SQL-92 ACCESS MODES

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

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

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

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.

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