Carnegie Mellon University

Database Systems

15-445/645 FALL 2026

JIGNESH PATEL



Lecture #20

Concurrency Control: Multi-Versioning

ADMINISTRIVIA

Project #4 is due Sunday Dec 06th @ 11:59pm

Homework #5 is due Sunday Nov 22nd @ 11:59pm

UPCOMING DATABASE TALKS

TURSO	Nov 16	Mikaël Francoeur	Turso: An SQLite-compatible, Open-Contribution RDBMS
DB9	Nov 23	Ed Huang	DB9: A Database for Agents, Built by Agents
TERADATA	Nov 30	Artur Borycki	Building Distributed Database Engine

And may be a mystery new DB Talk on December 14

LAST CLASS

Optimistic Concurrency Control (OCC) uses timestamps, assigned during the validation phase, to ensure serializability instead of locks.

→ Txns first write changes into private workspaces and then attempt to install them into the database upon commit.

Phantom Reads occur when a txns re-reads a range of data and finds new rows or missing rows.

MULTI-VERSION CONCURRENCY CONTROL

The DBMS maintains multiple **physical** versions of a single **logical** object in the database:

→ When a txn writes to an object, the DBMS creates a new version of that object.
→ When a txn reads an object, it reads the newest version that existed when the txn started.
→ Use timestamps to determine visibility.
→ Multi-versioning without garbage collection allows the DBMS to support time-travel queries.

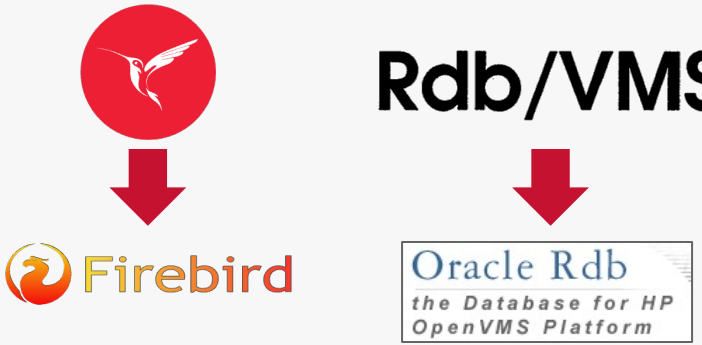
Writers do not block readers.
Readers do not block writers.

MVCC HISTORY

Protocol was first proposed in 1978 MIT PhD dissertation.

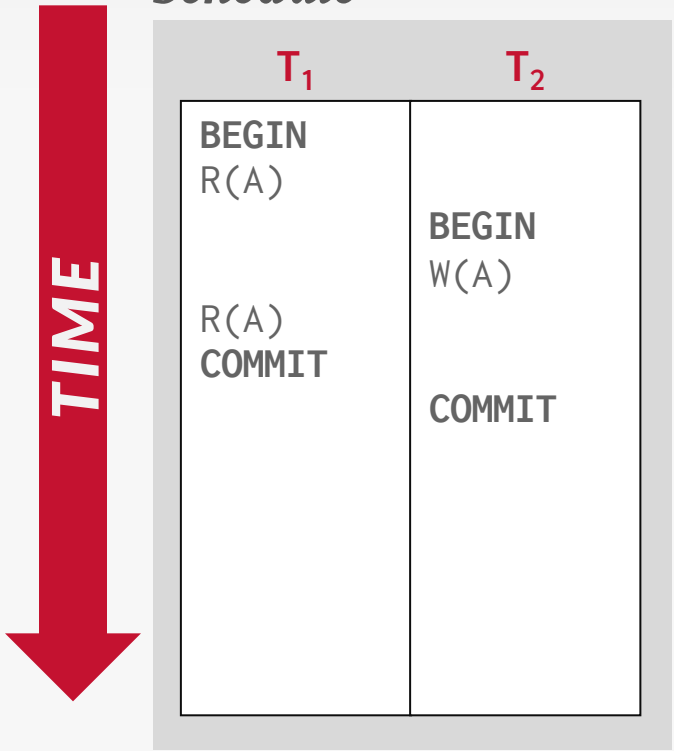
First implementations was Rdb/VMS and InterBase at DEC in early 1980s.

→ Both were by Jim Starkey, co-founder of NuoDB.
→ DEC Rdb/VMS is now "Oracle Rdb".
→ InterBase was open-sourced as Firebird.



MVCC EXAMPLE

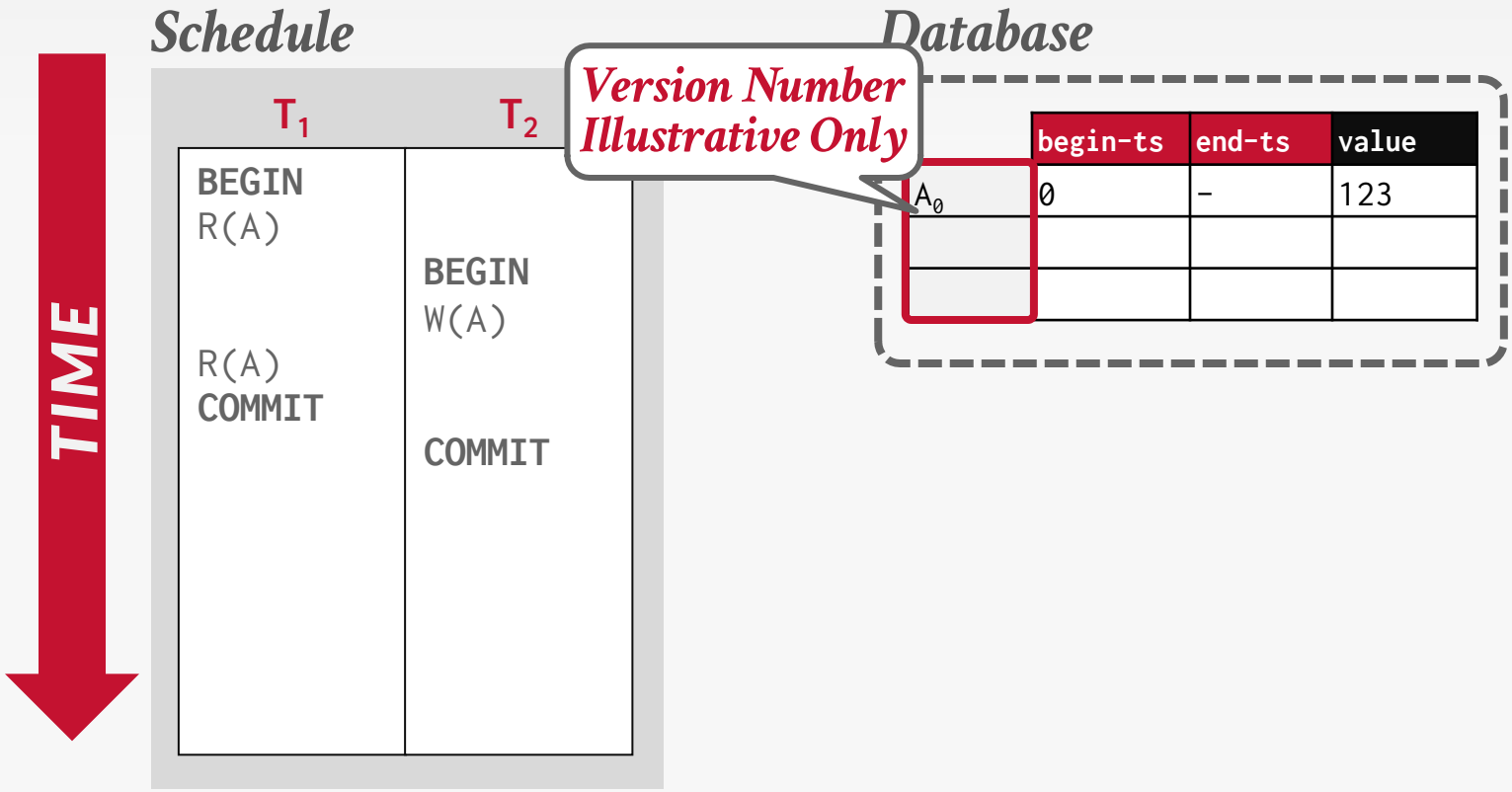
Schedule



Database

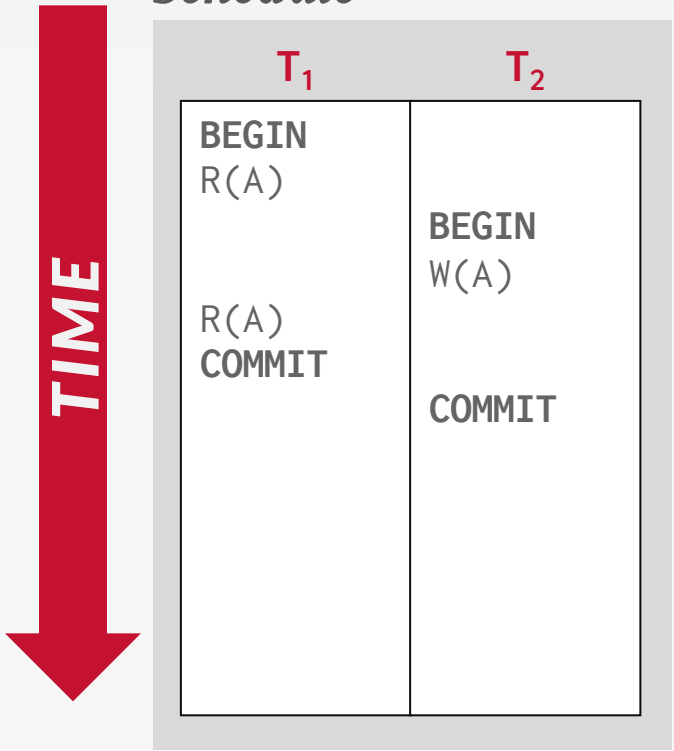
	begin-ts	end-ts	value
A_0	0	-	123

MVCC EXAMPLE



MVCC EXAMPLE

Schedule

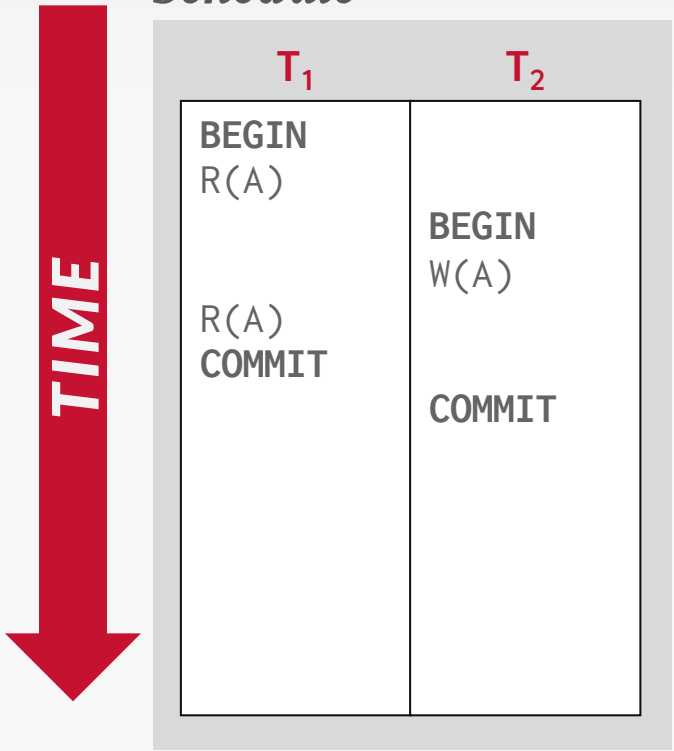


Database

	begin-ts	end-ts	value
A_0	0	-	123

MVCC EXAMPLE

Schedule

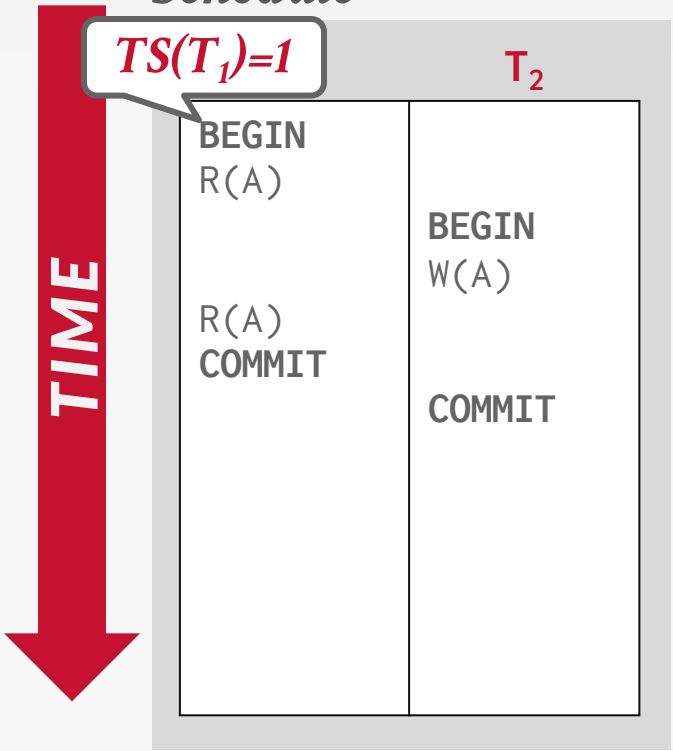


Database

	begin-ts	end-ts	value
A_0	0	-	123

MVCC EXAMPLE

Schedule

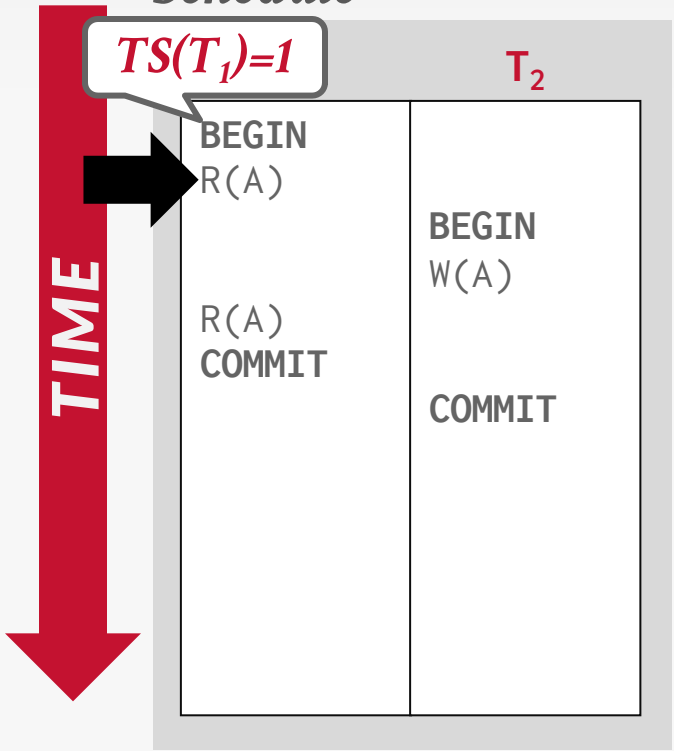


Database

	begin-ts	end-ts	value
A_0	0	-	123

MVCC EXAMPLE

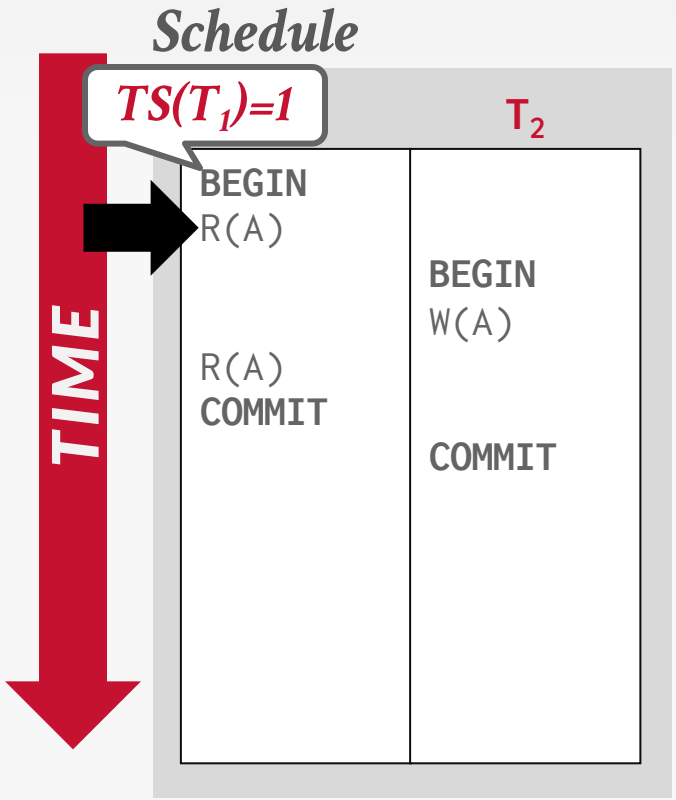
Schedule



Database

	begin-ts	end-ts	value
A_0	0	-	123

MVCC EXAMPLE

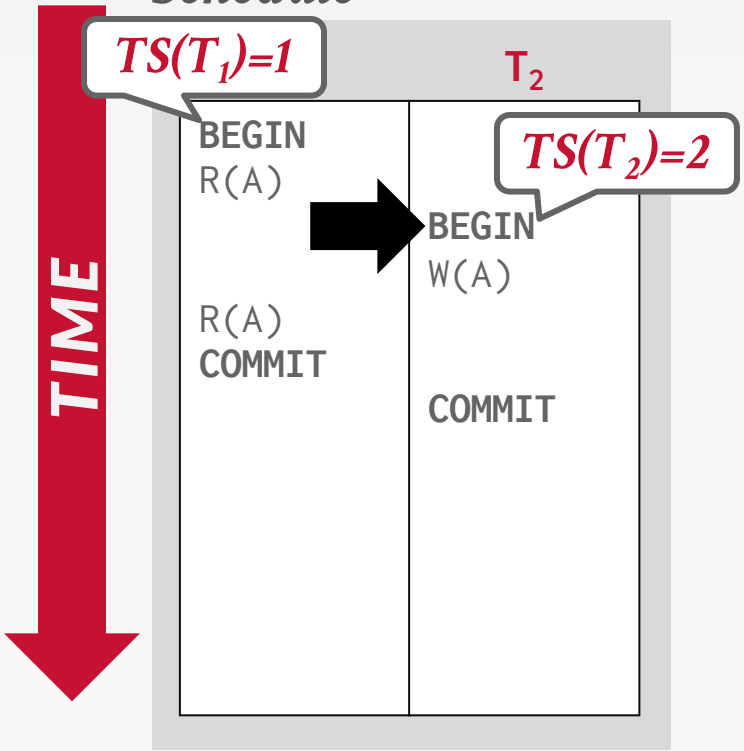


Database

	begin-ts	end-ts	value
A_0	0	-	123

MVCC EXAMPLE

Schedule

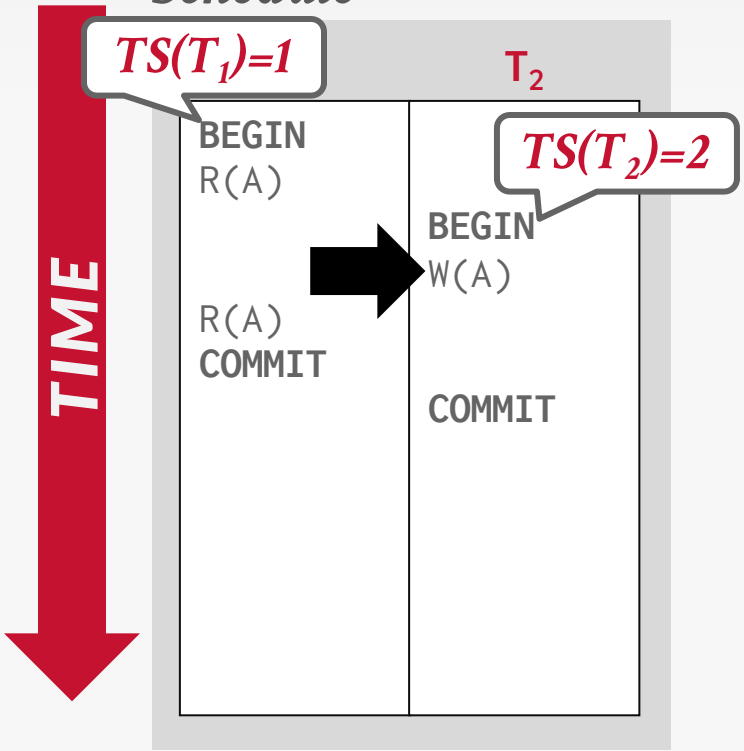


Database

	begin-ts	end-ts	value
A_0	0	-	123

MVCC EXAMPLE

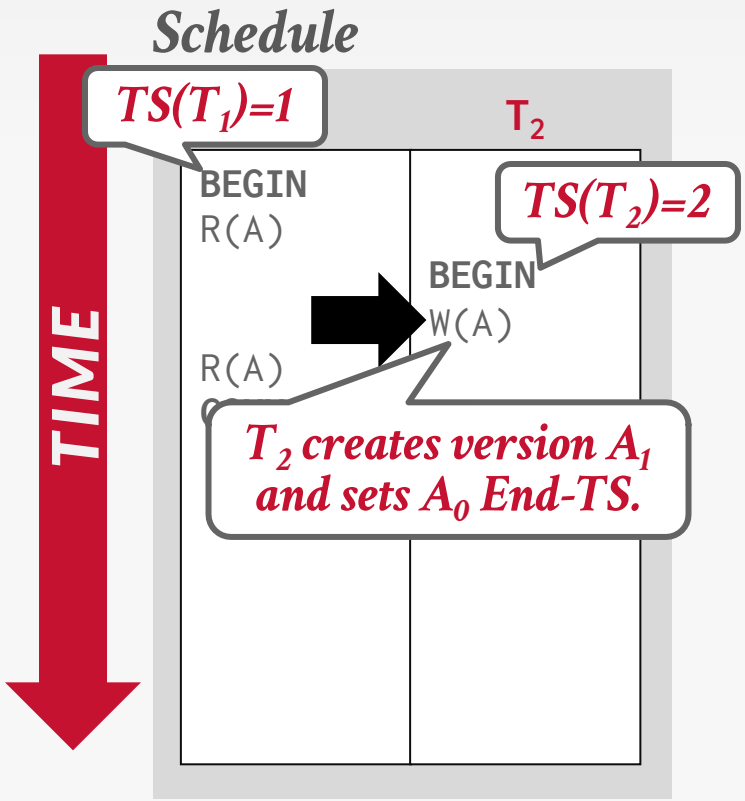
Schedule



Database

	begin-ts	end-ts	value
A_0	0	-	123

MVCC EXAMPLE

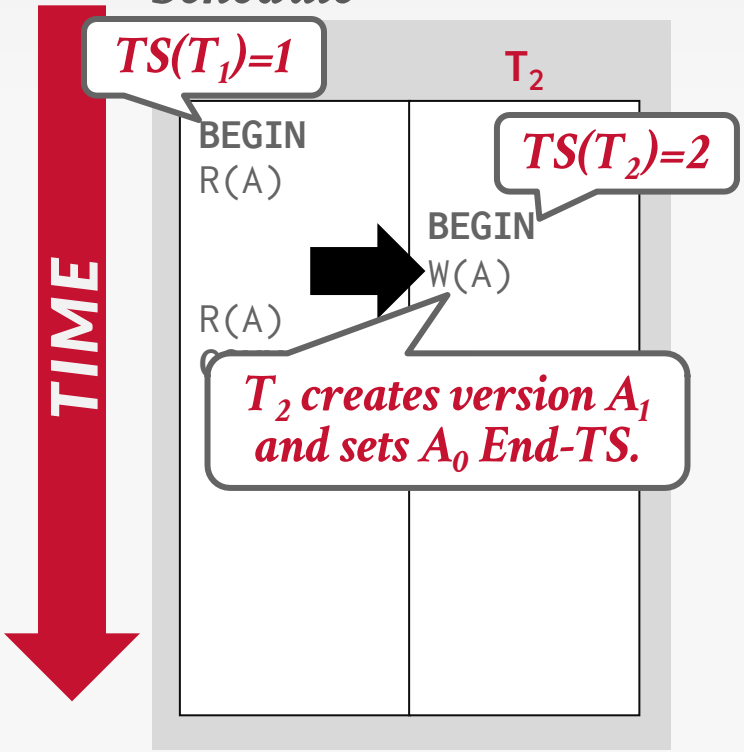


Database

	begin-ts	end-ts	value
A_0	0	-	123
A_1	2	-	456

MVCC EXAMPLE

Schedule

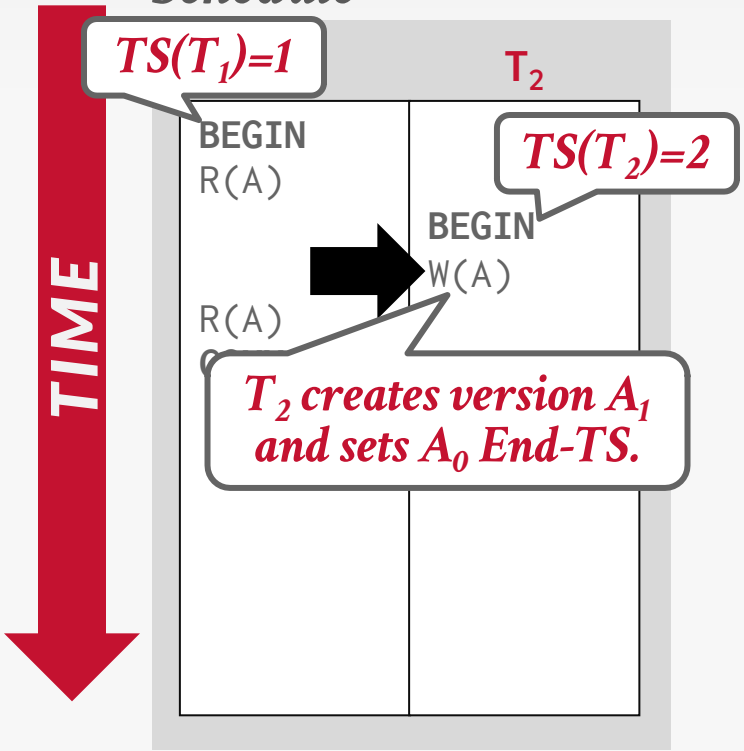


Database

	begin-ts	end-ts	value
A_0	0	2	123
A_1	2	-	456

MVCC EXAMPLE

Schedule



Database

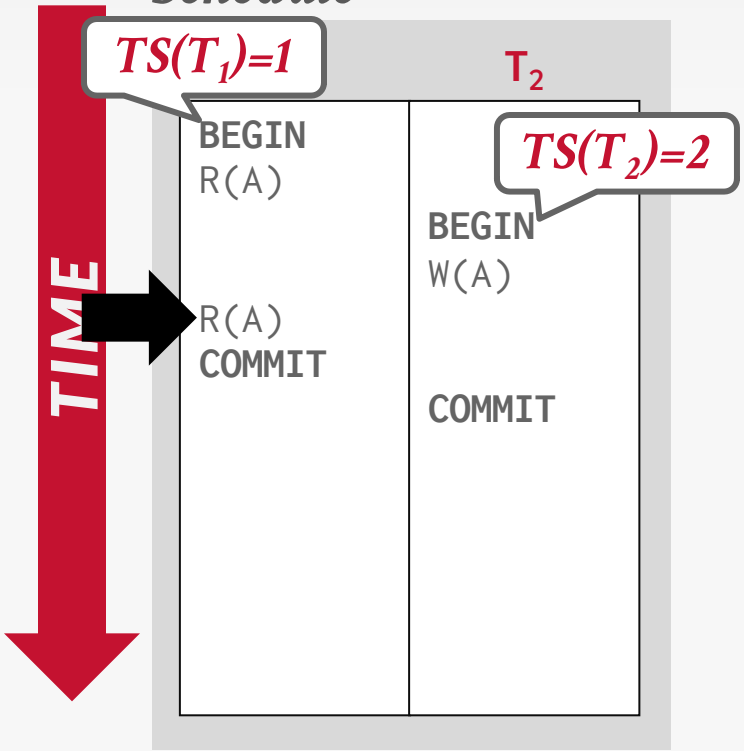
	begin-ts	end-ts	value
A_0	0	2	123
A_1	2	-	456

Txn Status Table

txnid	timestamp	status
T_1	1	Active
T_2	2	Active

MVCC EXAMPLE

Schedule



Database

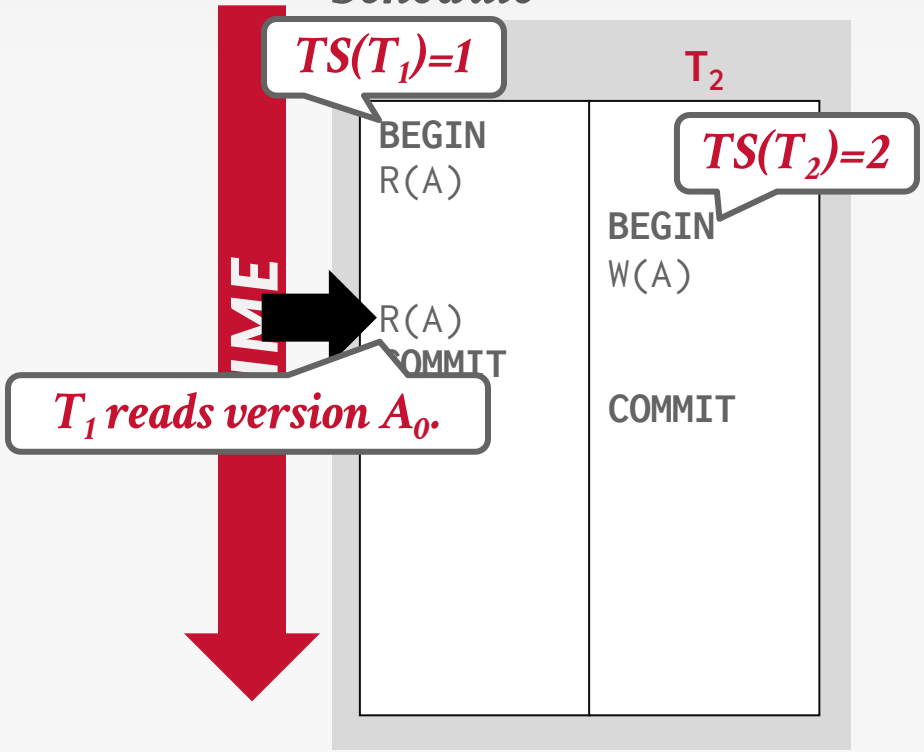
	begin-ts	end-ts	value
A_0	0	2	123
A_1	2	-	456

Txn Status Table

txnid	timestamp	status
T_1	1	Active
T_2	2	Active

MVCC EXAMPLE

Schedule



Database

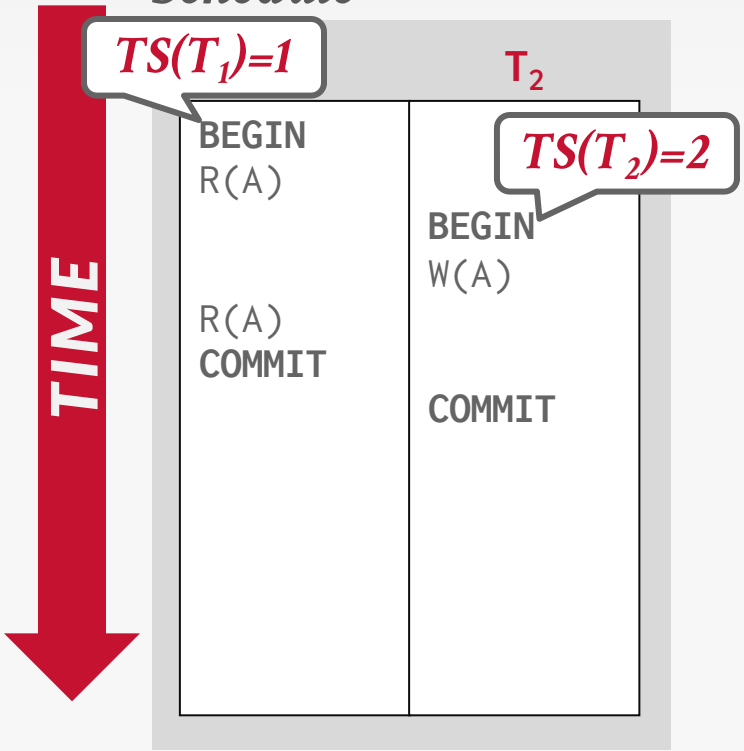
	begin-ts	end-ts	value
A_0	0	2	123
A_1	2	-	456

Txn Status Table

txnid	timestamp	status
T_1	1	Active
T_2	2	Active

MVCC EXAMPLE

Schedule



Database

	begin-ts	end-ts	value
A_0	0	2	123
A_1	2	-	456

Txn Status Table

txnid	timestamp	status
T_1	1	Active
T_2	2	Active

MULTI-VERSION CONCURRENCY CONTROL

MVCC is more than just a concurrency control protocol. It completely affects how the DBMS manages transactions and the database.

MULTI-VERSION CONCURRENCY CONTROL

MVCC is more than just a concurrency control protocol. It completely affects how the DBMS manages transactions and the database.



TODAY'S AGENDA

Concurrency Control Protocol

Version Storage

Garbage Collection

Index Management

Deletes

⚡ **DB Flash Talk: SpiralDB**

CONCURRENCY CONTROL PROTOCOL

Two-Phase Locking

→ Txns acquire exclusive locks on logical objects before writing.
→ Txns do not acquire shared locks on objects before reading.

Optimistic Concurrency Control

→ Three-phase protocol from last class.
→ Use private workspace for new versions.

CONCURRENCY CONTROL PROTOCOL

Two-Phase Locking

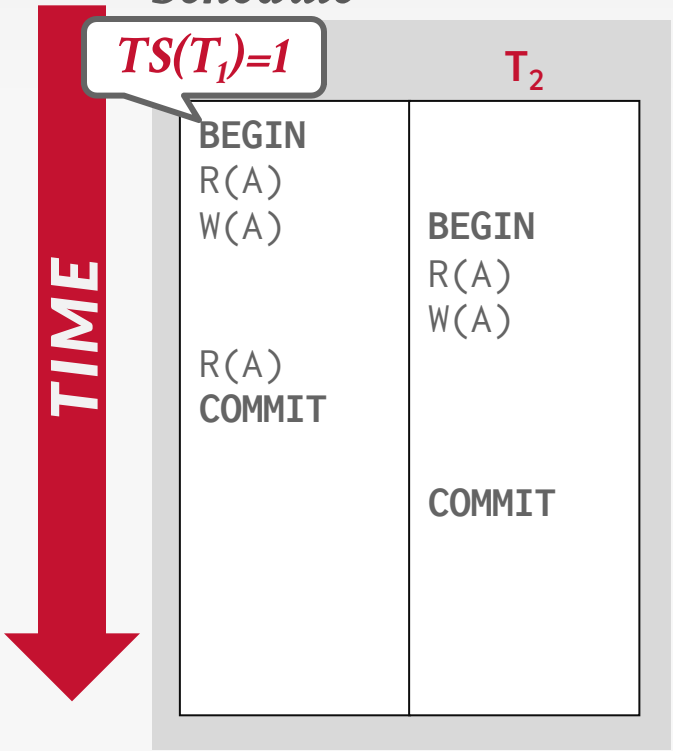
→ Txns acquire exclusive locks on logical objects before writing.
→ Txns do not acquire shared locks on objects before reading.

Optimistic Concurrency Control

→ Three-phase protocol from last class.
→ Use private workspace for new versions.

MVCC WITH 2PL

Schedule



Database

	begin-ts	end-ts	value
A_0	0	-	123

Txn Status Table

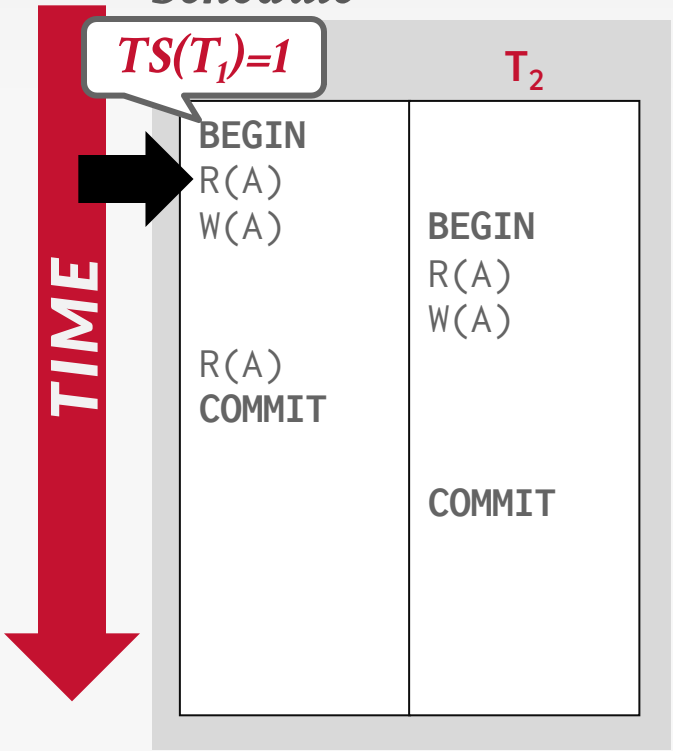
txnid	timestamp	status
T_1	1	Active

Lock Table

obj	owner	waiting

MVCC WITH 2PL

Schedule



Database

	begin-ts	end-ts	value
A_0	0	-	123

Txn Status Table

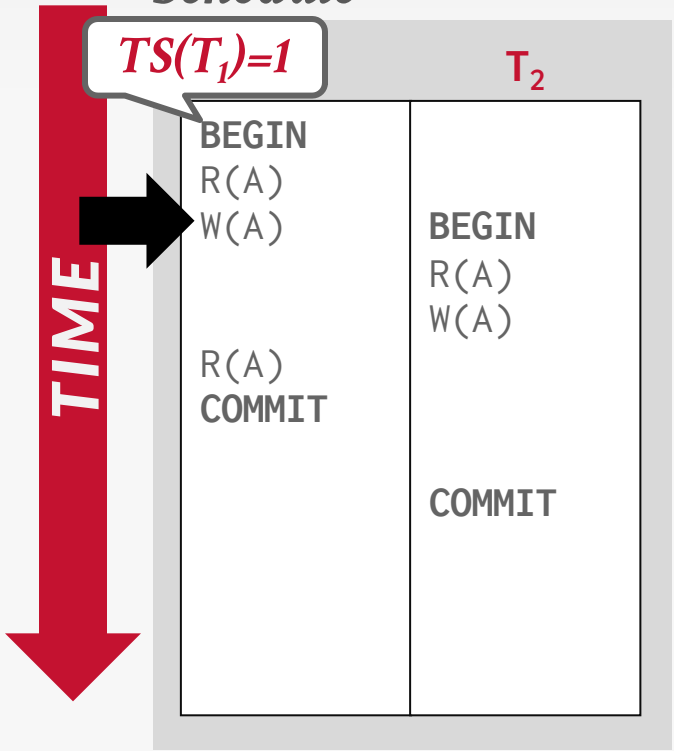
txnid	timestamp	status
T_1	1	Active

Lock Table

obj	owner	waiting

MVCC WITH 2PL

Schedule



Database

	begin-ts	end-ts	value
A_0	0	-	123

Txn Status Table

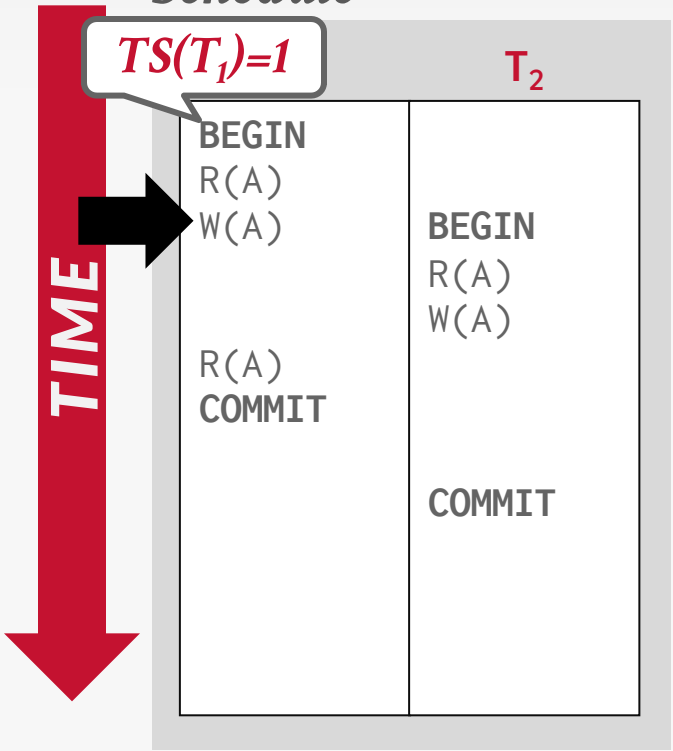
txnid	timestamp	status
T_1	1	Active

Lock Table

obj	owner	waiting

MVCC WITH 2PL

Schedule



Database

	begin-ts	end-ts	value
A_0	0	-	123

Txn Status Table

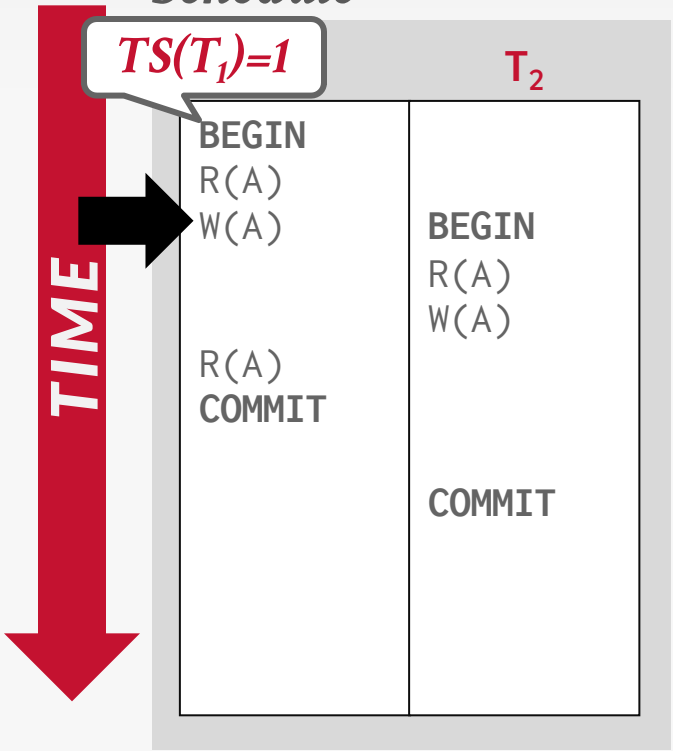
txnid	timestamp	status
T_1	1	Active

Lock Table

obj	owner	waiting
A	T_1 ▸ X	

MVCC WITH 2PL

Schedule



Database

	begin-ts	end-ts	value
A_0	0	-	123
A_1	1	-	456

Txn Status Table

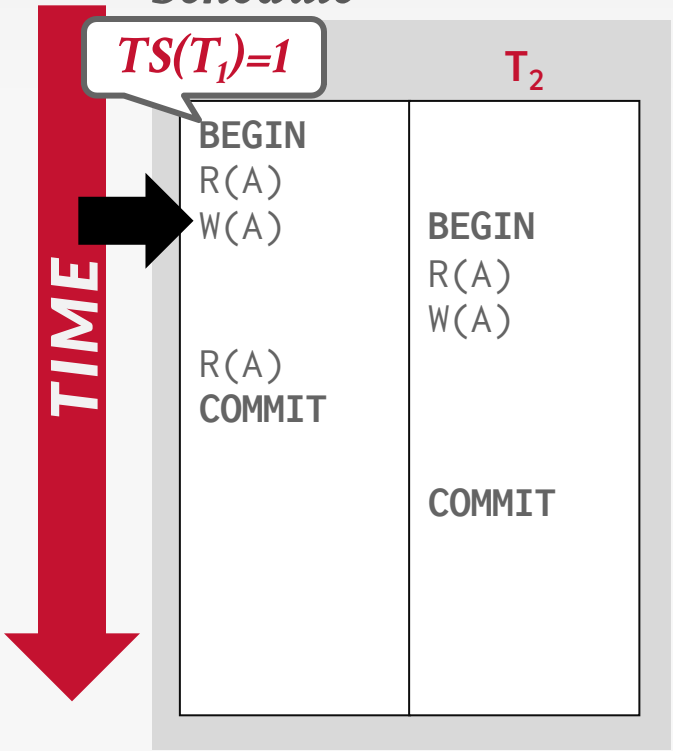
txnid	timestamp	status
T_1	1	Active

Lock Table

obj	owner	waiting
A	T_1 ▸ X	

MVCC WITH 2PL

Schedule



Database

	begin-ts	end-ts	value
A_0	0	-	123
A_1	1	-	456

Txn Status Table

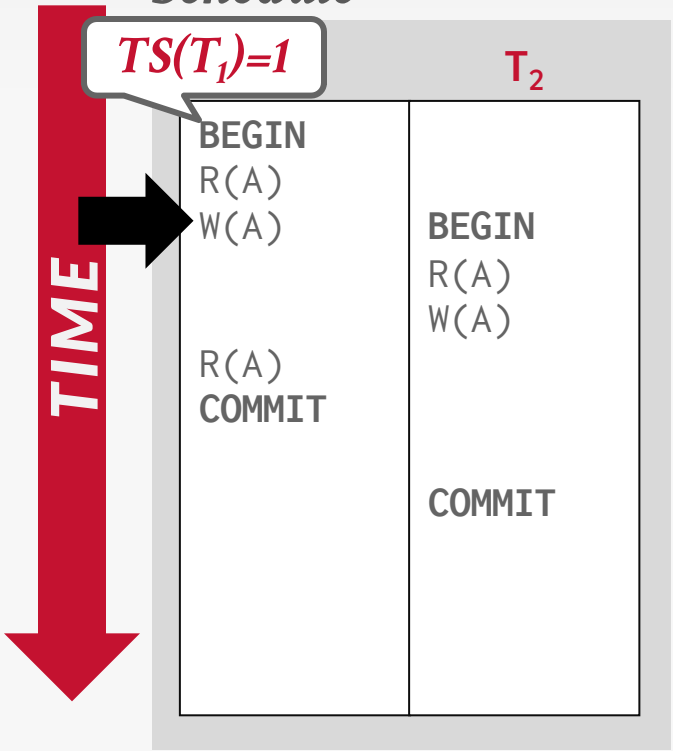
txnid	timestamp	status
T_1	1	Active

Lock Table

obj	owner	waiting
A	T_1 ▶ X	

MVCC WITH 2PL

Schedule



Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

Txn Status Table

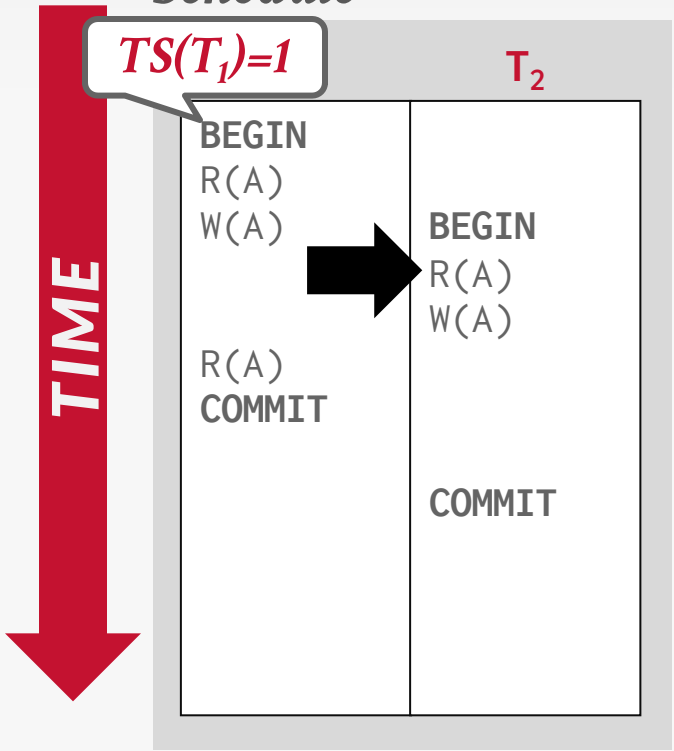
txnid	timestamp	status
T_1	1	Active

Lock Table

obj	owner	waiting
A	T_1 ▸ X	

MVCC WITH 2PL

Schedule



Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

Txn Status Table

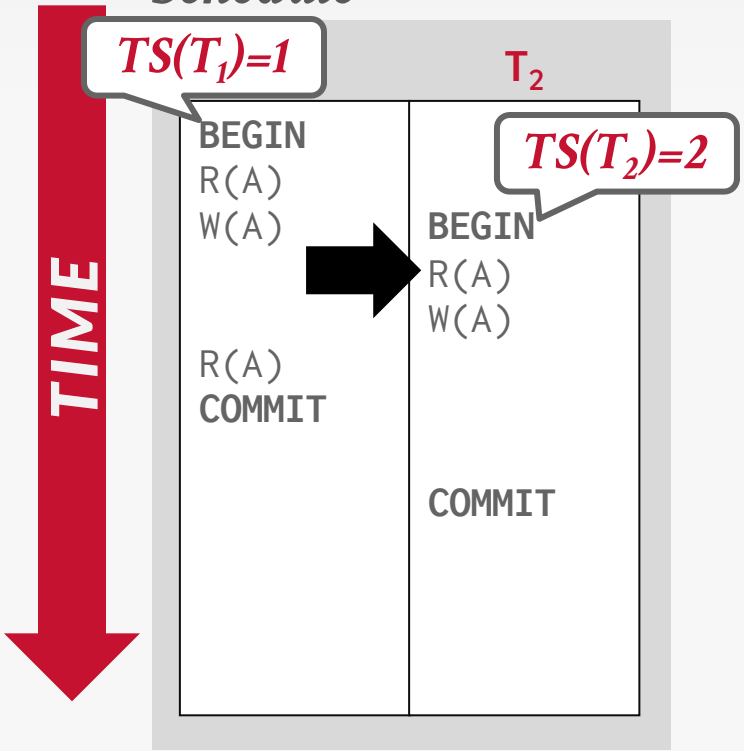
txnid	timestamp	status
T_1	1	Active

Lock Table

obj	owner	waiting
A	T_1 ▸ X	

MVCC WITH 2PL

Schedule



Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

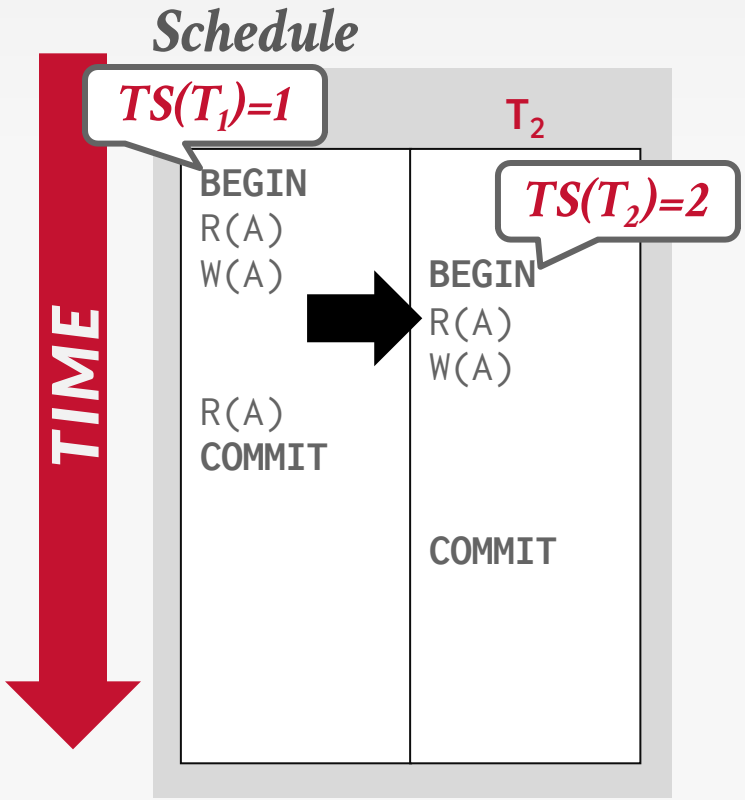
Txn Status Table

txnid	timestamp	status
T_1	1	Active
T_2	2	Active

Lock Table

obj	owner	waiting
A	T_1 ▸ X	

MVCC WITH 2PL



Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

Txn Status Table

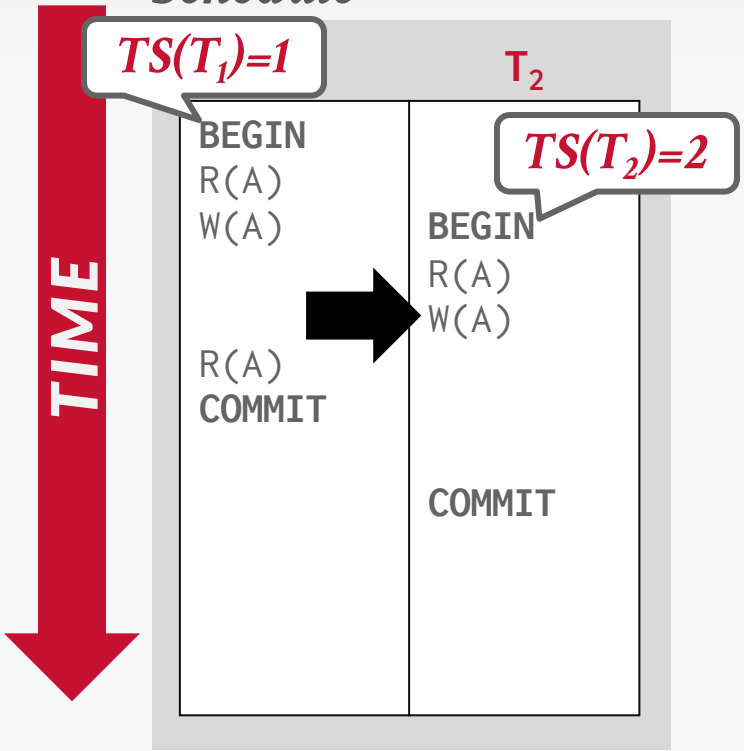
txnid	timestamp	status
T_1	1	Active
T_2	2	Active

Lock Table

obj	owner	waiting
A	T_1 ▶ X	

MVCC WITH 2PL

Schedule



Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

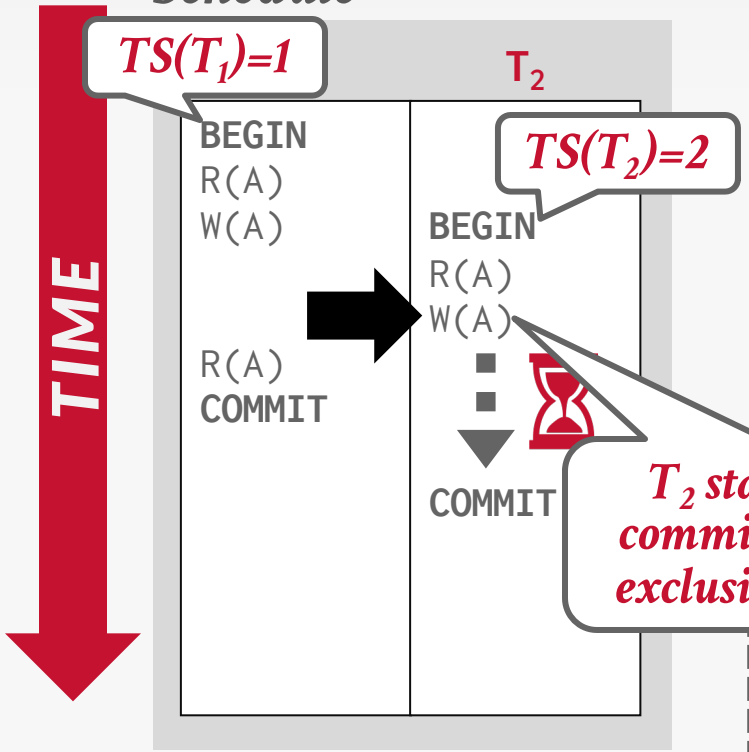
Txn Status Table

txnid	timestamp	status
T_1	1	Active
T_2	2	Active

Lock Table

obj	owner	waiting
A	T_1 ▸ X	T_2 ▸ X

MVCC WITH 2PL



Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

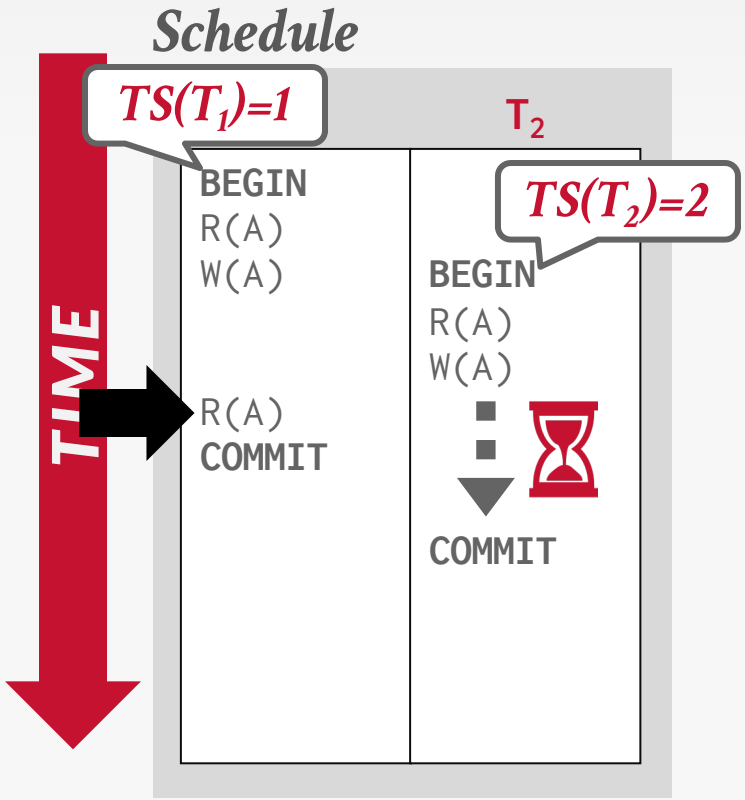
Table

		status
T_1	1	Active
T_2	2	Active

Lock Table

obj	owner	waiting
A	T_1 ▶ X	T_2 ▶ X

MVCC WITH 2PL



Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

Txn Status Table

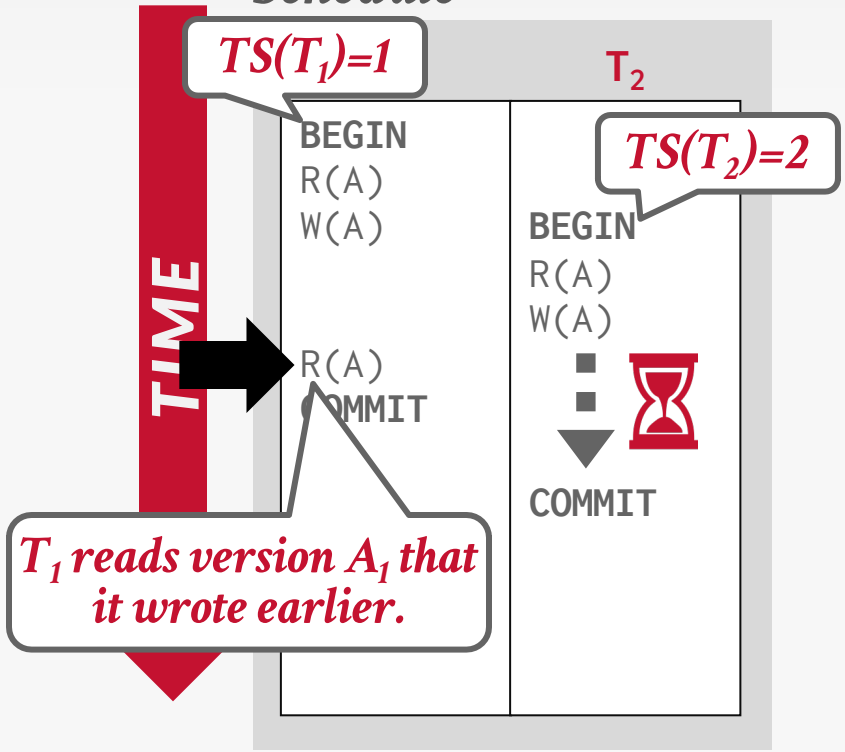
txnid	timestamp	status
T_1	1	Active
T_2	2	Active

Lock Table

obj	owner	waiting
A	T_1 ▶ X	T_2 ▶ X

MVCC WITH 2PL

Schedule



Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

Txn Status Table

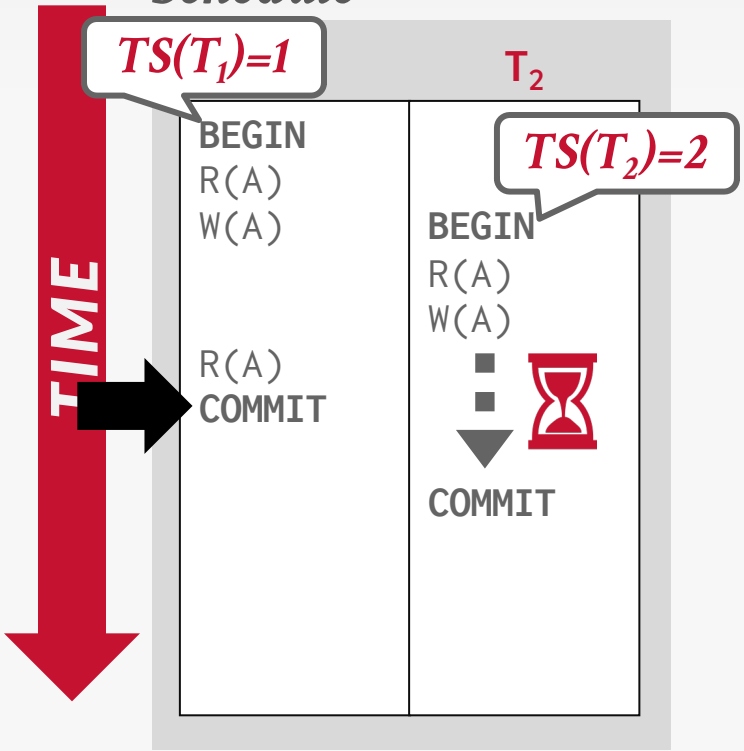
txnid	timestamp	status
T_1	1	Active
T_2	2	Active

Lock Table

obj	owner	waiting
A	T_1 ▶ X	T_2 ▶ X

MVCC WITH 2PL

Schedule



Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

Txn Status Table

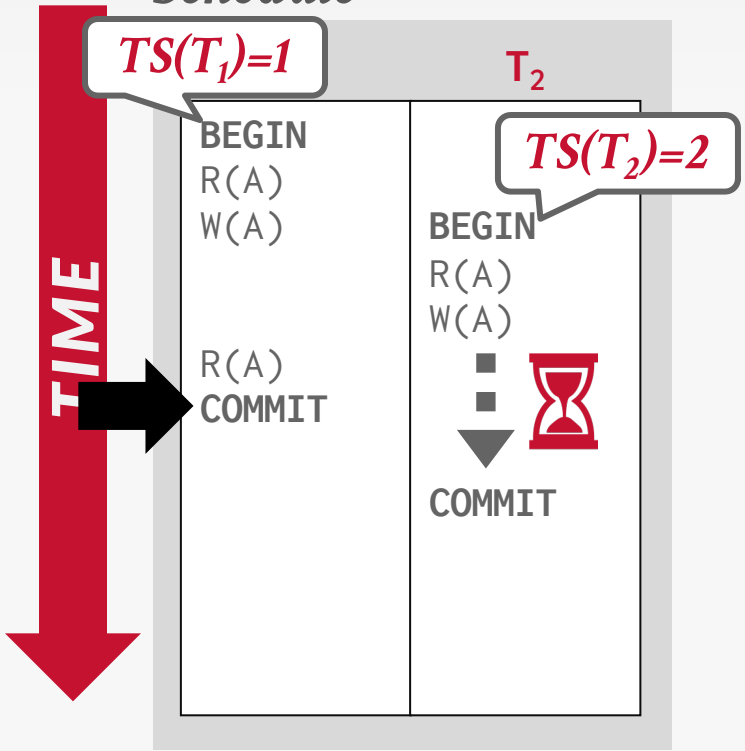
txnid	timestamp	status
T_1	1	Active
T_2	2	Active

Lock Table

obj	owner	waiting
A	T_1 ▶ X	T_2 ▶ X

MVCC WITH 2PL

Schedule



Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

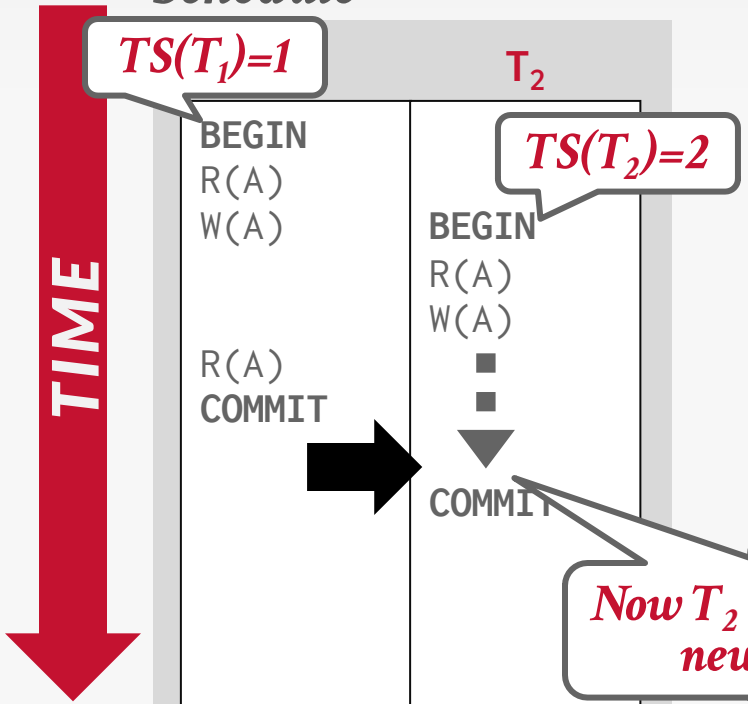
Txn Status Table

txnid	timestamp	status
T_1	1	Committed
T_2	2	Active

Lock Table

obj	owner	waiting
A	T_1 ▸ X	T_2 ▸ X

MVCC WITH 2PL



Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

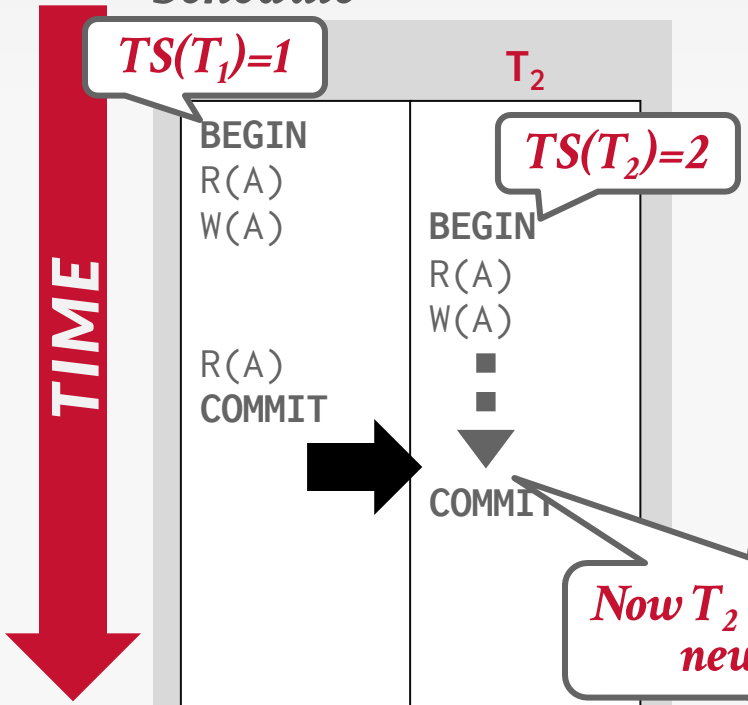
Txn Status Table

		status
		Committed
T_2	2	Active

Lock Table

obj	owner	waiting
A		T_2 ▶ X

MVCC WITH 2PL



Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	2	456
A_2	2	-	789

Now T_2 can create the new version.

Txn Status Table

		status
		Committed
T_2	2	Active

Lock Table

obj	owner	waiting
A	T_2 ▸ X	

SNAPSHOT ISOLATION (SI)

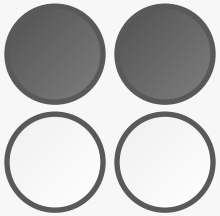
When a txn starts, it sees a consistent snapshot of the database that existed when that the txn started.

→ No torn writes from active txns.

→ If two txns update the same object, then last writer wins.

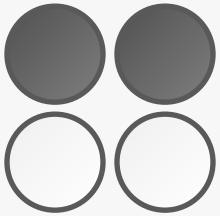
SI is susceptible to the **Write Skew Anomaly**.

WRITE SKEW ANOMALY



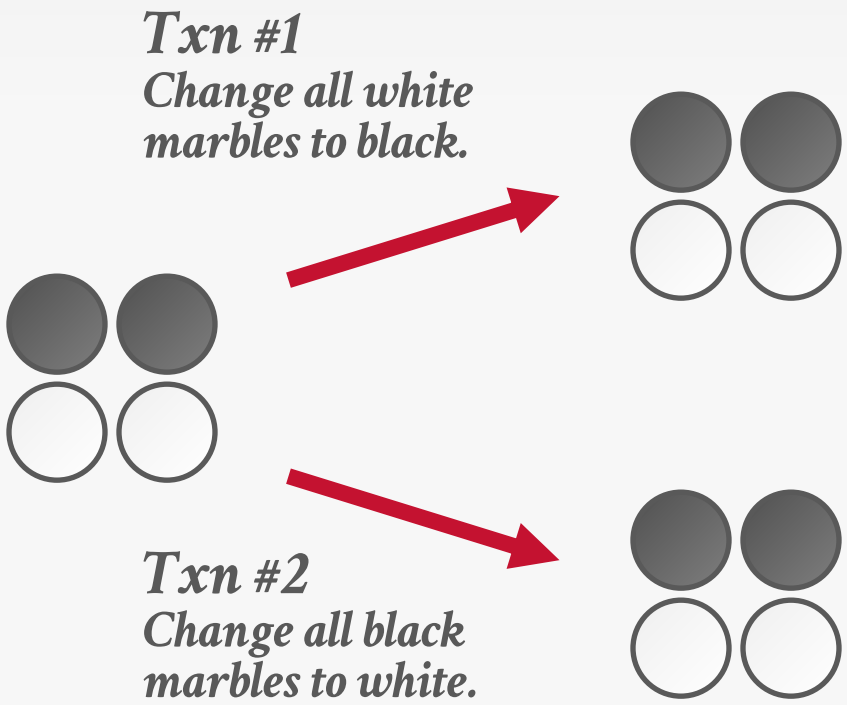
WRITE SKEW ANOMALY

Txn #1
Change all white marbles to black.

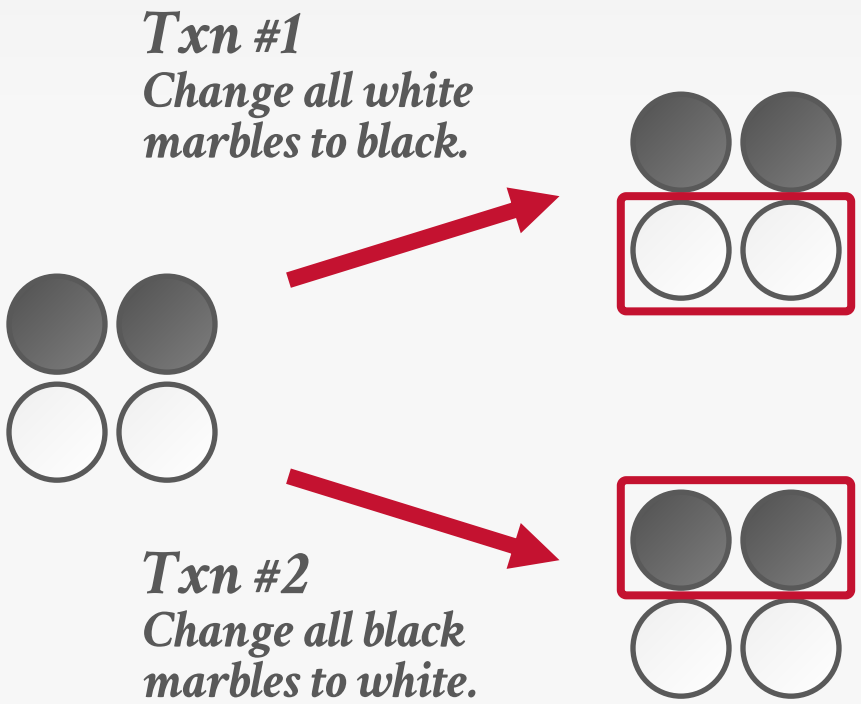


Txn #2
Change all black marbles to white.

WRITE SKEW ANOMALY

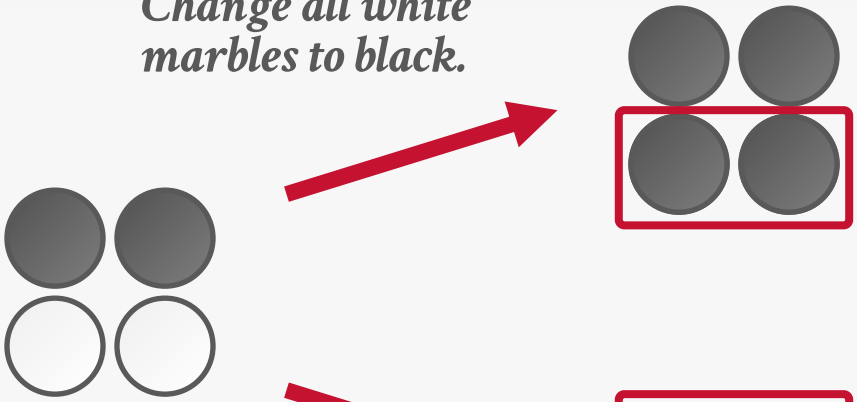


WRITE SKEW ANOMALY



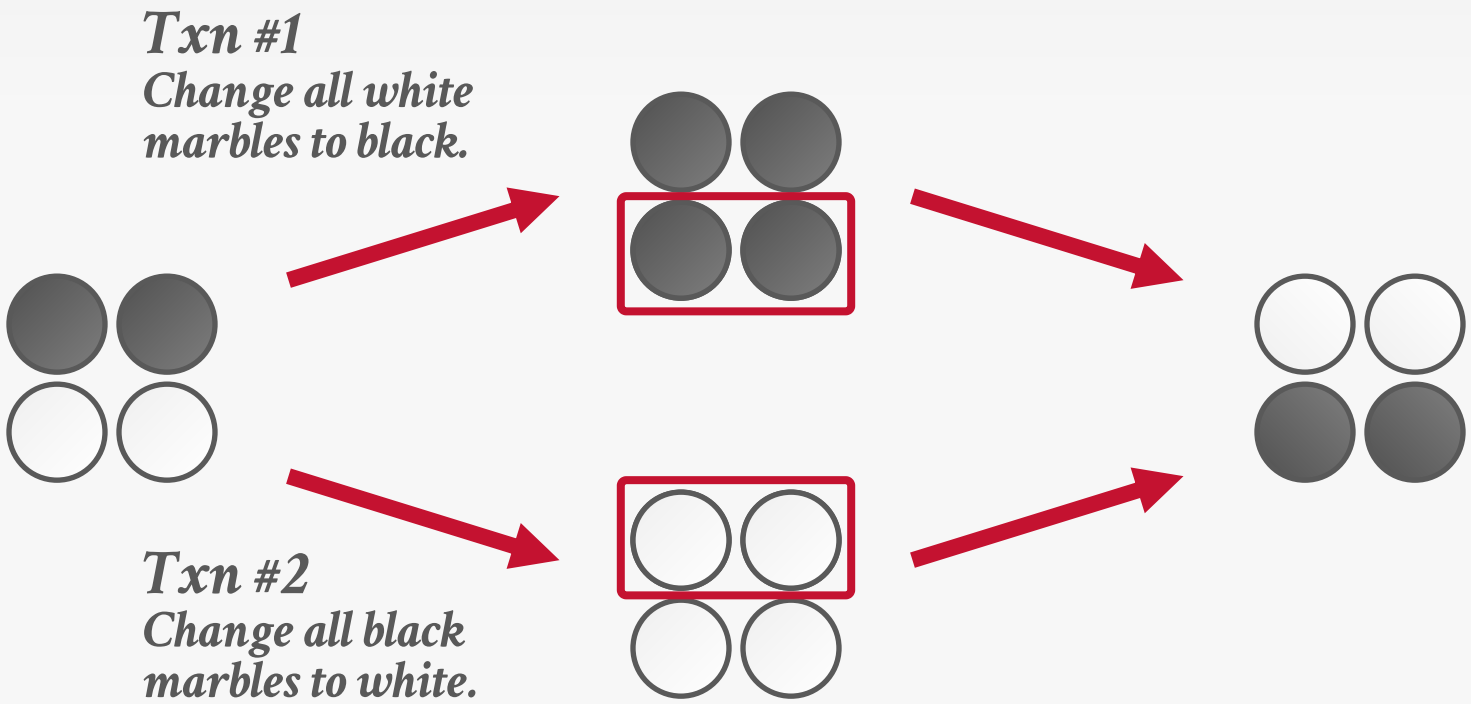
WRITE SKEW ANOMALY

Txn #1
Change all white marbles to black.

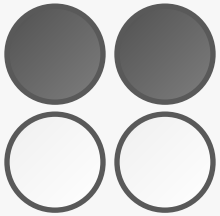


Txn #2
Change all black marbles to white.

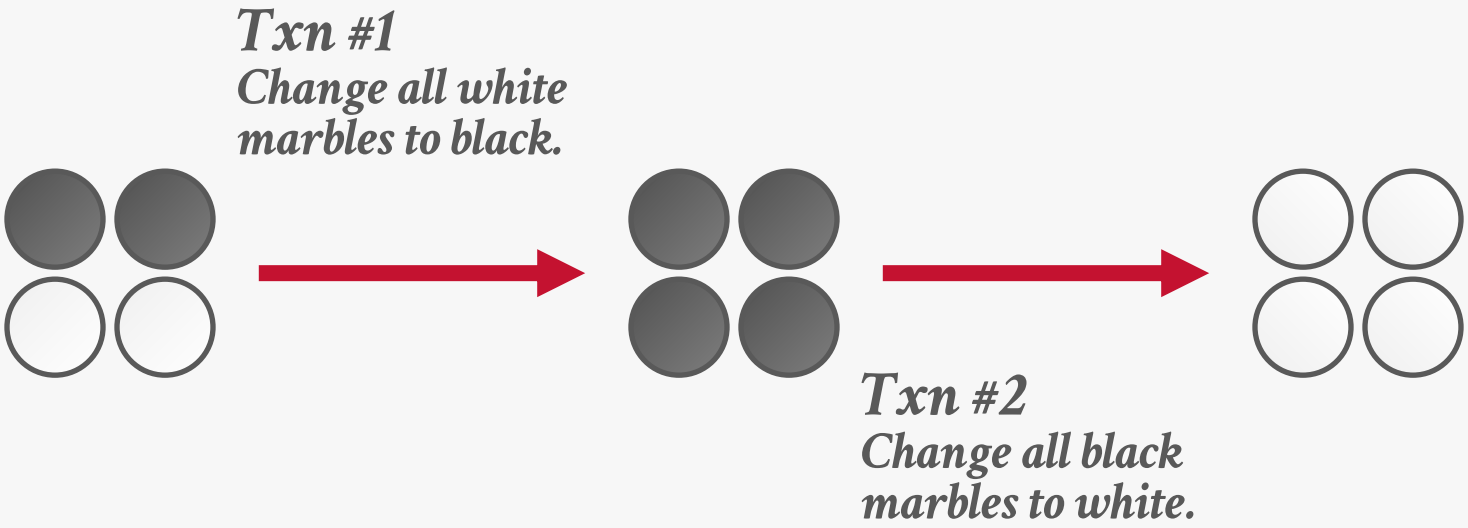
WRITE SKEW ANOMALY



WRITE SKEW ANOMALY

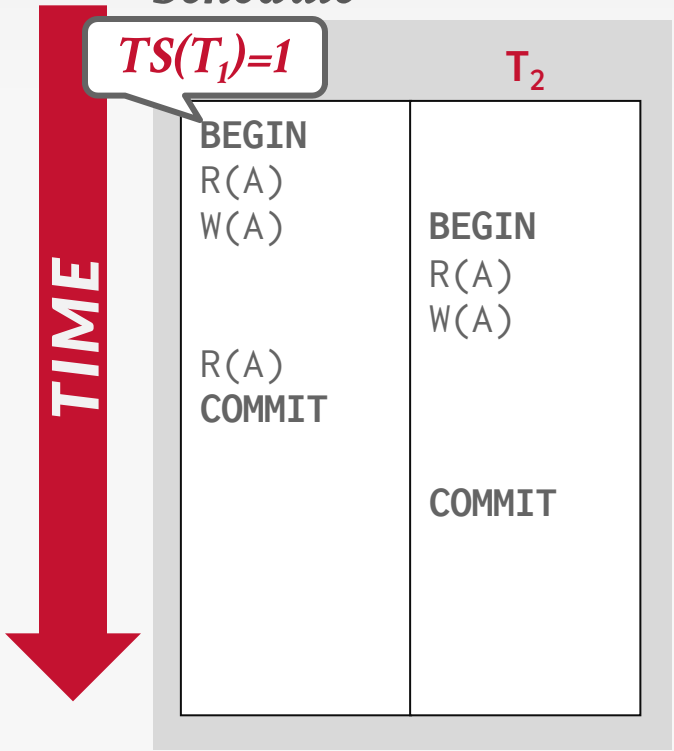


WRITE SKEW ANOMALY



SERIALIZABLE SNAPSHOT ISOLATION

Schedule



Database

	begin-ts	end-ts	value
A_0	0	-	123

Txn Status Table

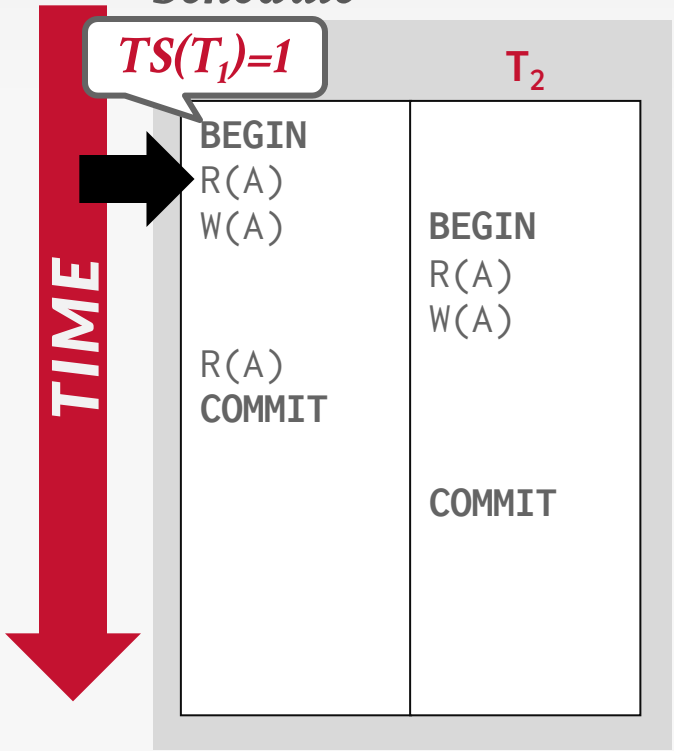
txnid	timestamp	status
T_1	1	Active

Lock Table

obj	owner	waiting

SERIALIZABLE SNAPSHOT ISOLATION

Schedule



Database

	begin-ts	end-ts	value
A_0	0	-	123

Txn Status Table

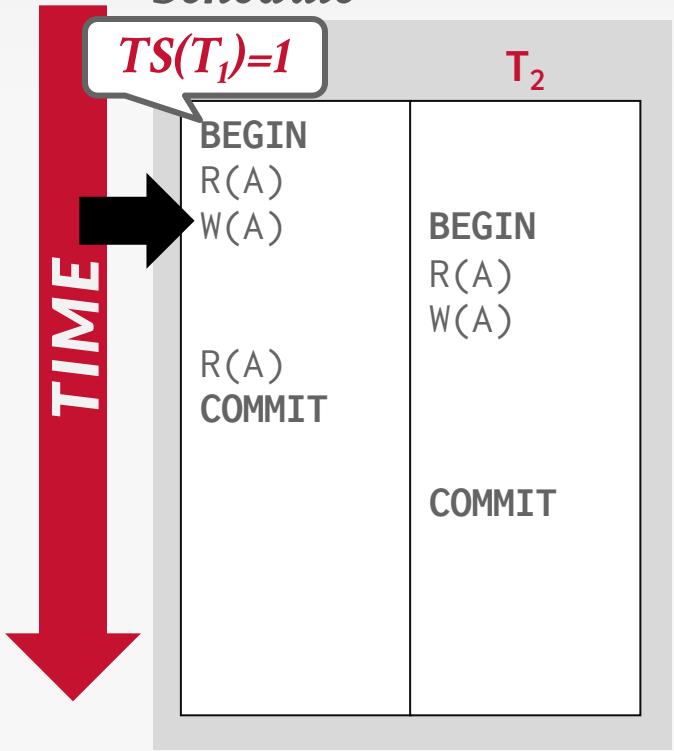
txnid	timestamp	status
T_1	1	Active

Lock Table

obj	owner	waiting

SERIALIZABLE SNAPSHOT ISOLATION

Schedule



Database

	begin-ts	end-ts	value
A_0	0	-	123

Txn Status Table

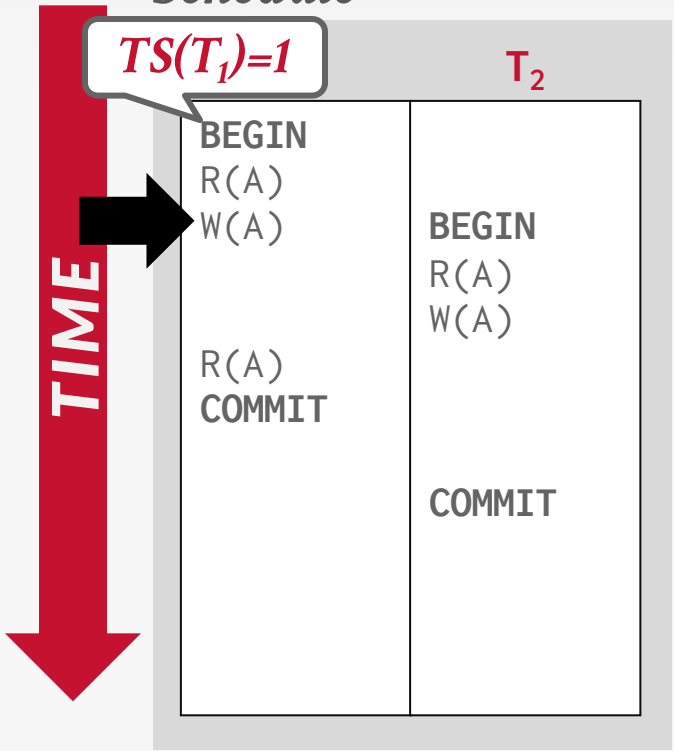
txnid	timestamp	status
T_1	1	Active

Lock Table

obj	owner	waiting

SERIALIZABLE SNAPSHOT ISOLATION

Schedule



Database

	begin-ts	end-ts	value
A_0	0	-	123

Txn Status Table

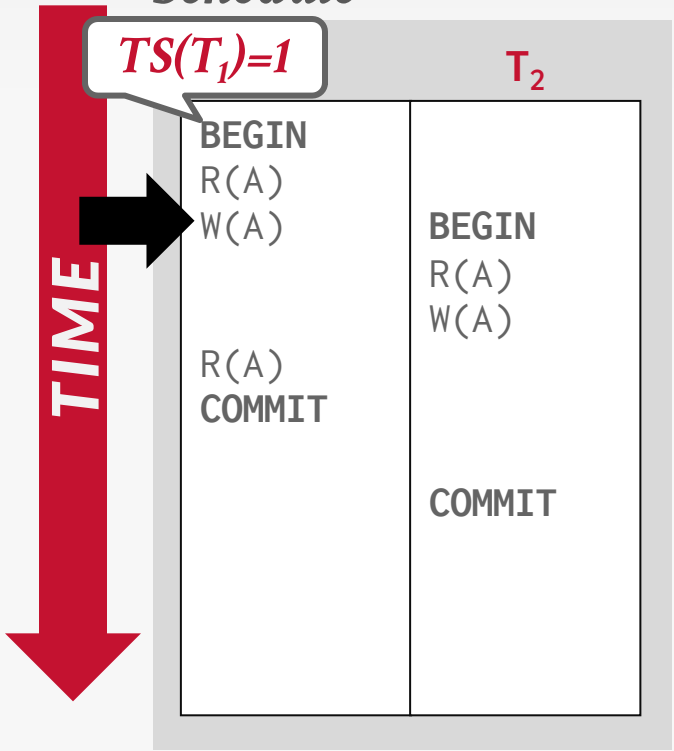
txnid	timestamp	status
T_1	1	Active

Lock Table

obj	owner	waiting
A	T_1 ▸ X	

SERIALIZABLE SNAPSHOT ISOLATION

Schedule



Database

	begin-ts	end-ts	value
A_0	0	-	123
A_1	1	-	456

Txn Status Table

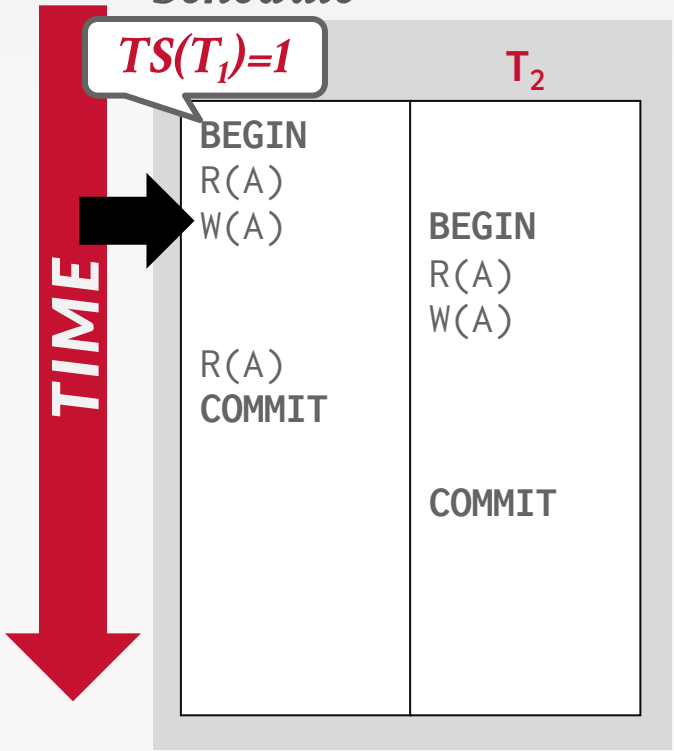
txnid	timestamp	status
T_1	1	Active

Lock Table

obj	owner	waiting
A	T_1 ▸ X	

SERIALIZABLE SNAPSHOT ISOLATION

Schedule



Database

	begin-ts	end-ts	value
A_0	0	-	123
A_1	1	-	456

Txn Status Table

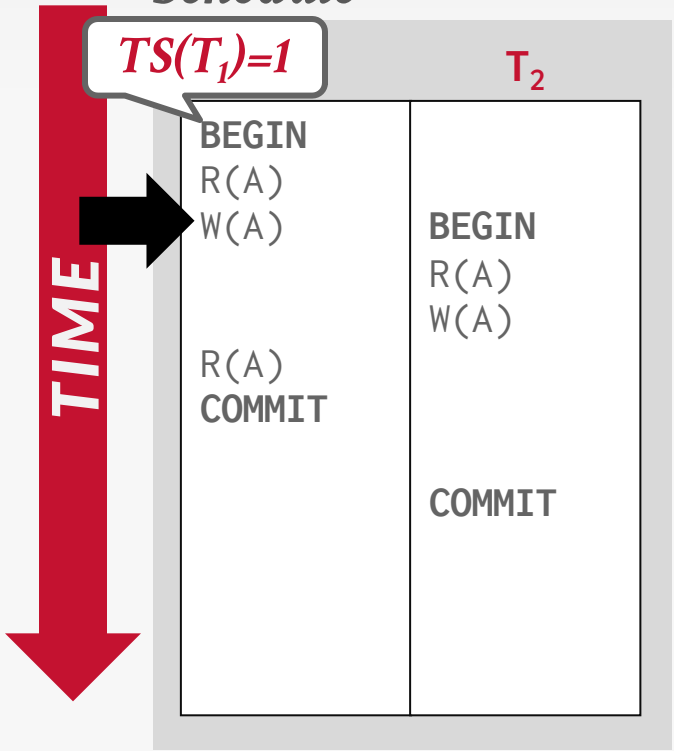
txnid	timestamp	status
T_1	1	Active

Lock Table

obj	owner	waiting
A	T_1 ▸ X	

SERIALIZABLE SNAPSHOT ISOLATION

Schedule



Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

Txn Status Table

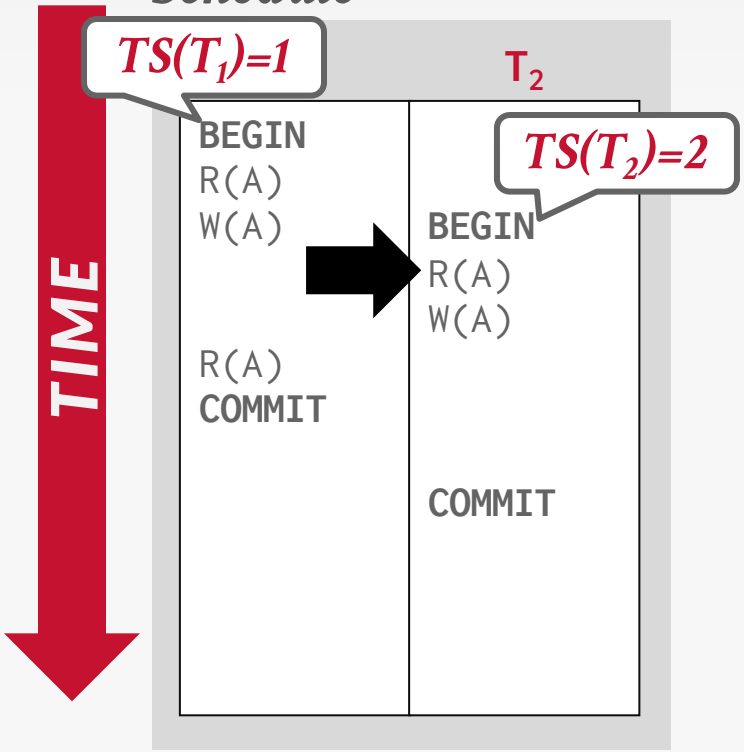
txnid	timestamp	status
T_1	1	Active

Lock Table

obj	owner	waiting
A	T_1 ▸ X	

SERIALIZABLE SNAPSHOT ISOLATION

Schedule



Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

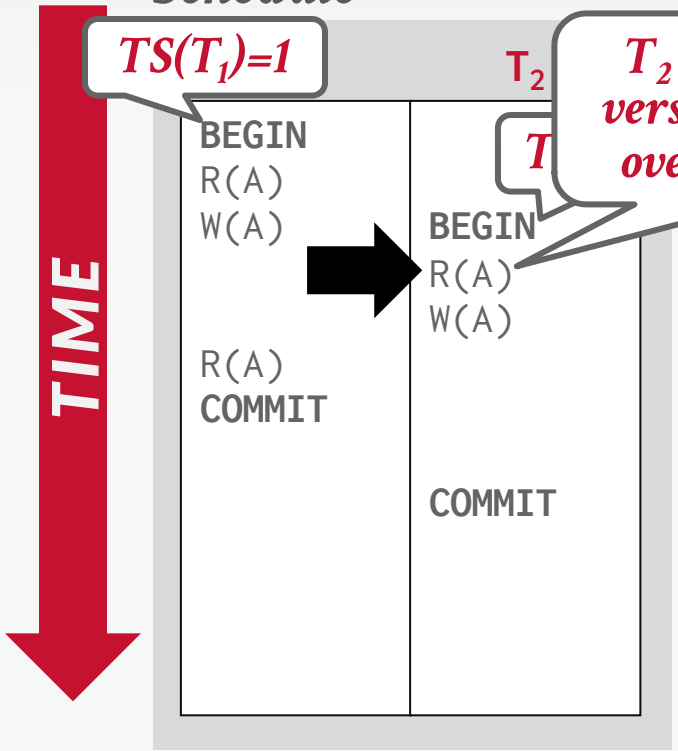
Txn Status Table

txnid	timestamp	status
T_1	1	Active
T_2	2	Active

Lock Table

obj	owner	waiting
A	T_1 ▸ X	

SERIALIZABLE SNAPSHOT ISOLATION



T2 reads an older version of an object overwritten by T1

Database

begin-ts	end-ts	value
0	1	123
1	–	456

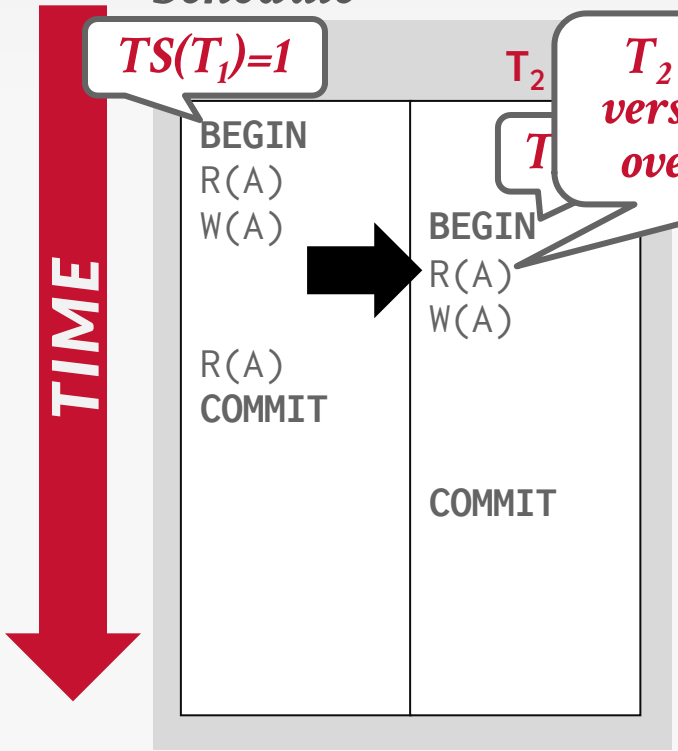
Txn Status Table

txnid	timestamp	status
T_1	1	Active
T_2	2	Active

Lock Table

obj	owner	waiting
A	T_1 ▸ X	

SERIALIZABLE SNAPSHOT ISOLATION



T2 reads an older version of an object overwritten by T1

Database

begin-ts	end-ts	value
0	1	123
1	–	456

Anti-Dependency Graph: T_1 → T_2

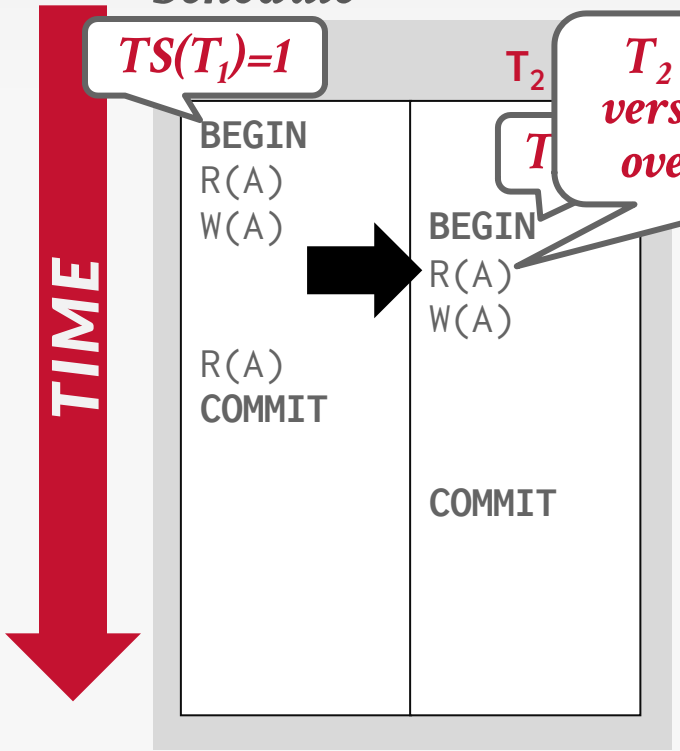
Txn Status Table

txnid	timestamp	status
T_1	1	Active
T_2	2	Active

Lock Table

obj	owner	waiting
A	T_1 ▸ X	

SERIALIZABLE SNAPSHOT ISOLATION



T2 reads an older version of an object overwritten by T1

Database

	begin-ts	end-ts	value
	0	1	123
	1	-	456

Anti-Dependency Graph:



Txn Status Table

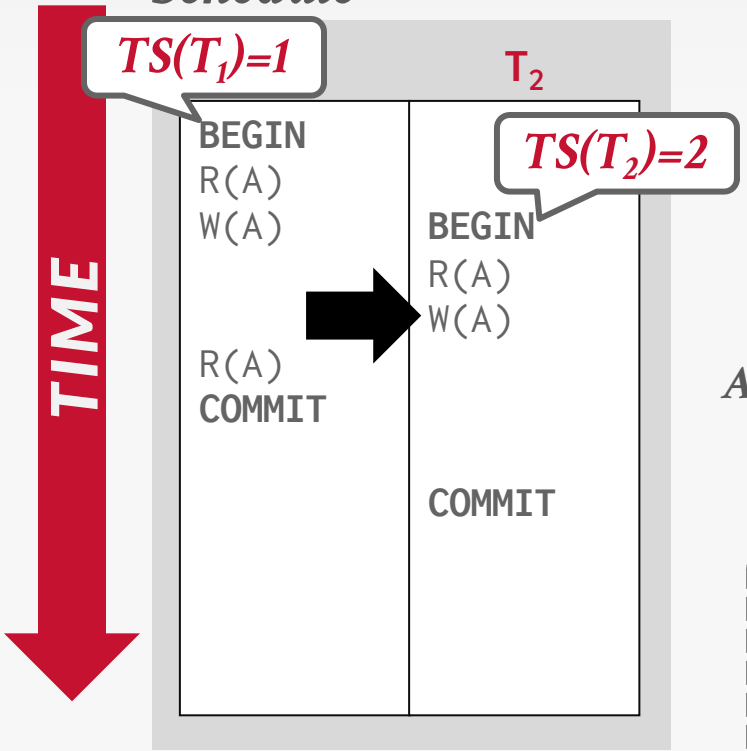
txnid	timestamp	status
T1	1	Active
T2	2	Active

Lock Table

obj	owner	waiting
A	T1 ▸ X	

SERIALIZABLE SNAPSHOT ISOLATION

Schedule



Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

Anti-Dependency Graph:



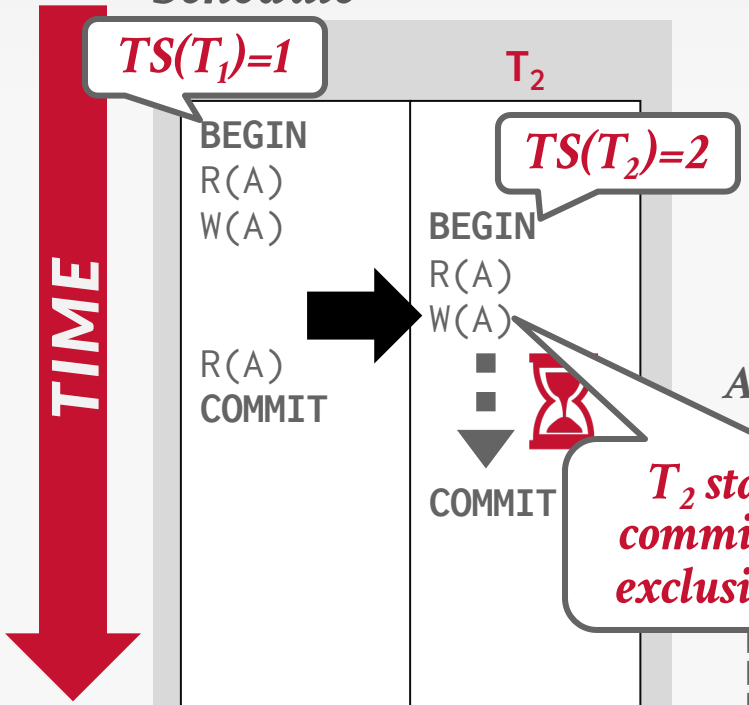
Txn Status Table

txnid	timestamp	status
T_1	1	Active
T_2	2	Active

Lock Table

obj	owner	waiting
A	T_1 ▸ X	T_2 ▸ X

SERIALIZABLE SNAPSHOT ISOLATION



Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

Anti-Dependency Graph:



Transaction Table

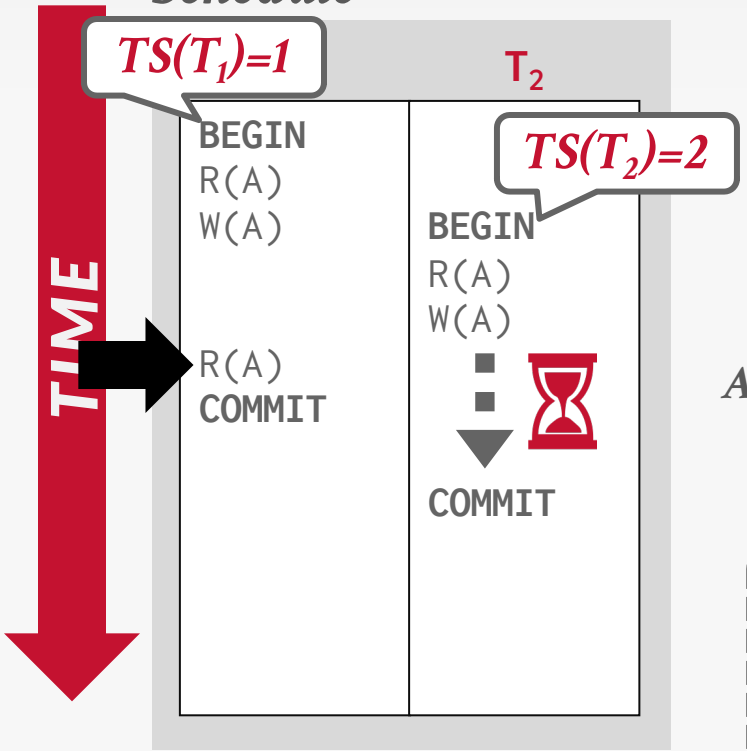
		status
T_1	1	Active
T_2	2	Active

Lock Table

obj	owner	waiting
A	T_1 ▶ X	T_2 ▶ X

SERIALIZABLE SNAPSHOT ISOLATION

Schedule



Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

Anti-Dependency Graph:



Txn Status Table

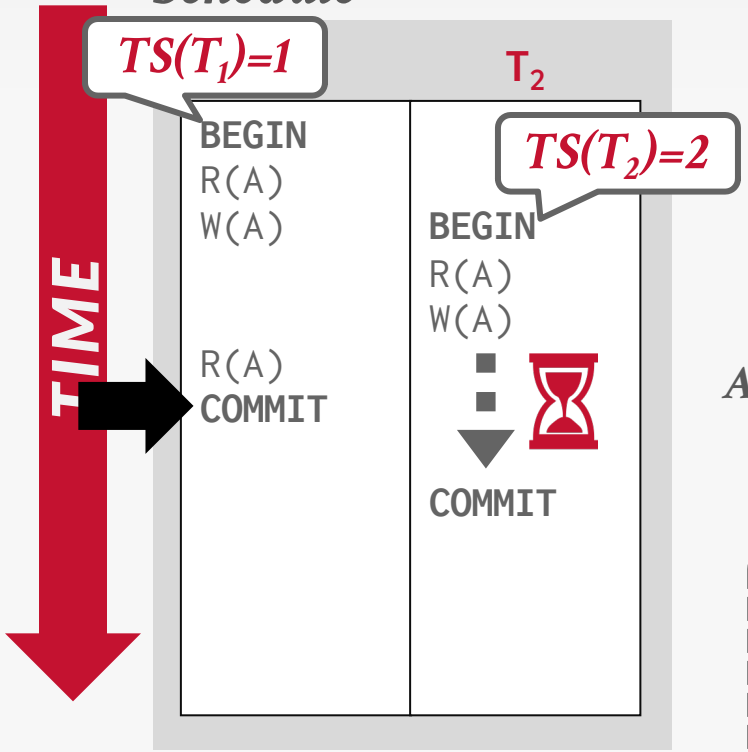
txnid	timestamp	status
T_1	1	Active
T_2	2	Active

Lock Table

obj	owner	waiting
A	T_1 ▸ X	T_2 ▸ X

SERIALIZABLE SNAPSHOT ISOLATION

Schedule



Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

Anti-Dependency Graph:



Txn Status Table

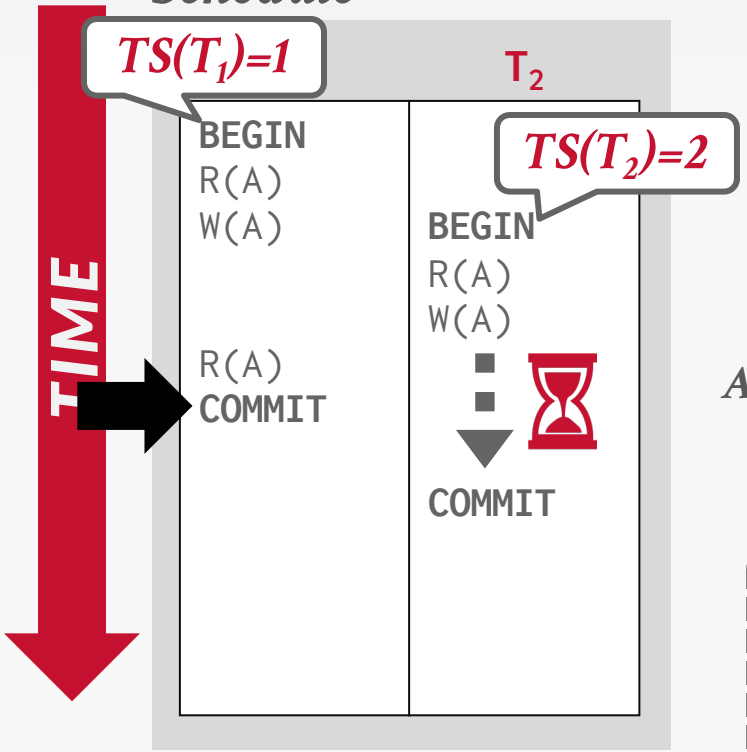
txnid	timestamp	status
T_1	1	Active
T_2	2	Active

Lock Table

obj	owner	waiting
A	T_1 ▸ X	T_2 ▸ X

SERIALIZABLE SNAPSHOT ISOLATION

Schedule



Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

Anti-Dependency Graph:



Txn Status Table

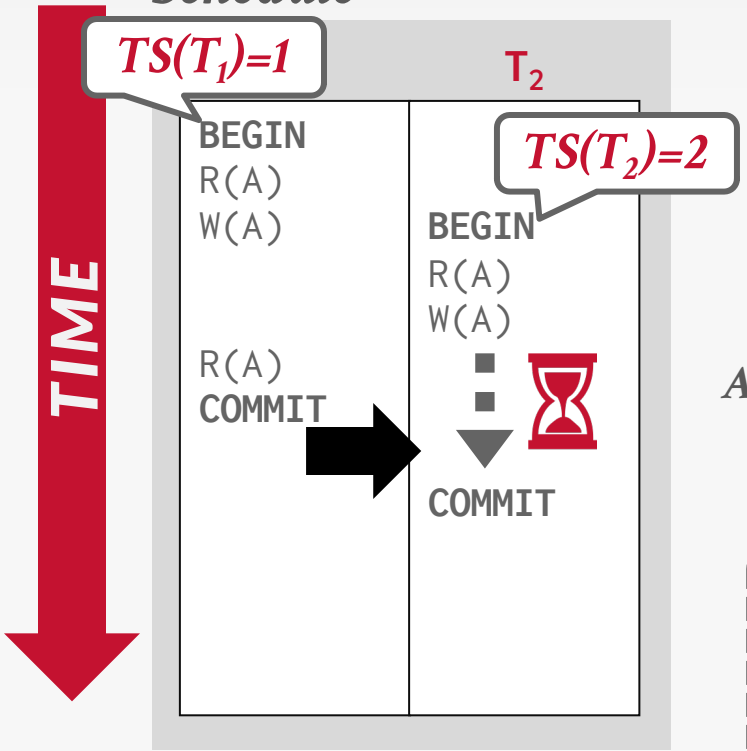
txnid	timestamp	status
T_1	1	Committed
T_2	2	Active

Lock Table

obj	owner	waiting
A	T_1 ▶ X	T_2 ▶ X

SERIALIZABLE SNAPSHOT ISOLATION

Schedule



Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

Anti-Dependency Graph:



Txn Status Table

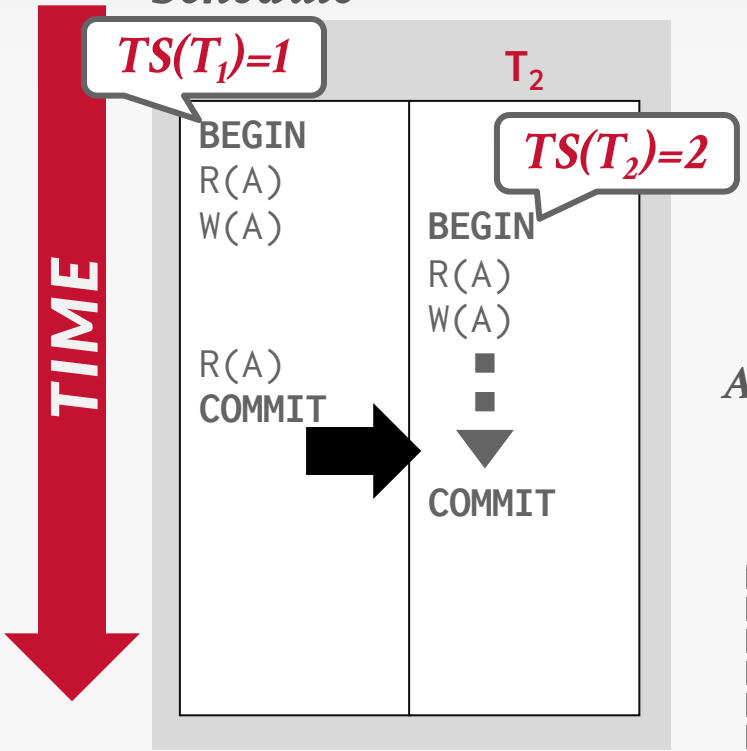
txnid	timestamp	status
T_1	1	Committed
T_2	2	Active

Lock Table

obj	owner	waiting
A		T_2 ▸ X

SERIALIZABLE SNAPSHOT ISOLATION

Schedule



Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	2	456
A_2	2	-	789

Anti-Dependency Graph:



Txn Status Table

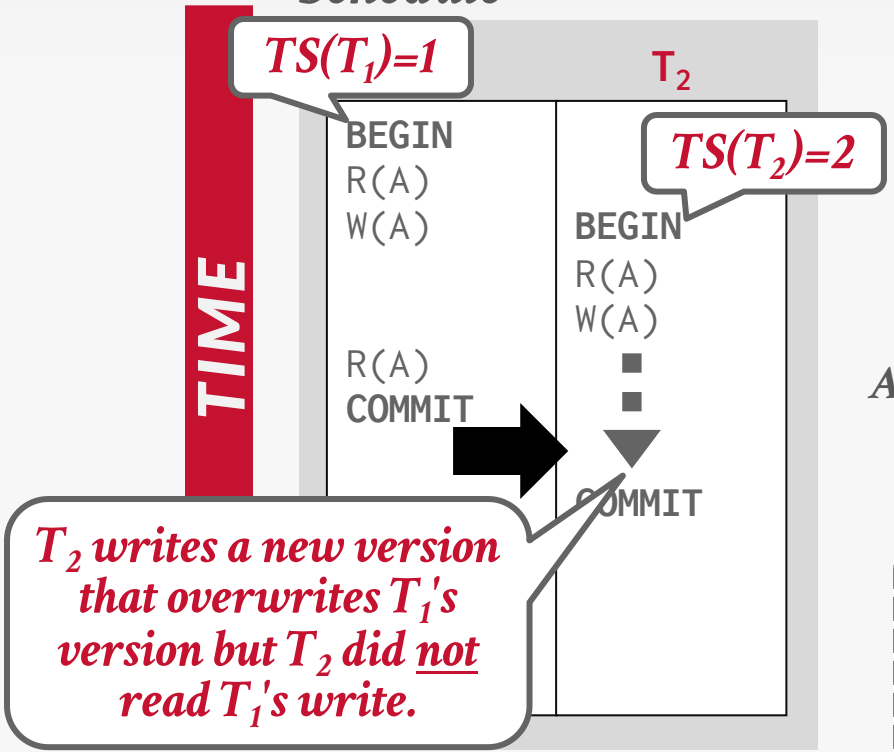
txnid	timestamp	status
T_1	1	Committed
T_2	2	Active

Lock Table

obj	owner	waiting
A	T_2 ▶ X	

SERIALIZABLE SNAPSHOT ISOLATION

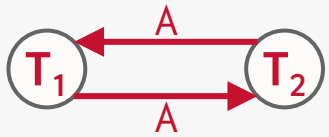
Schedule



Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	2	456
A_2	2	-	789

Anti-Dependency Graph:



Txn Status Table

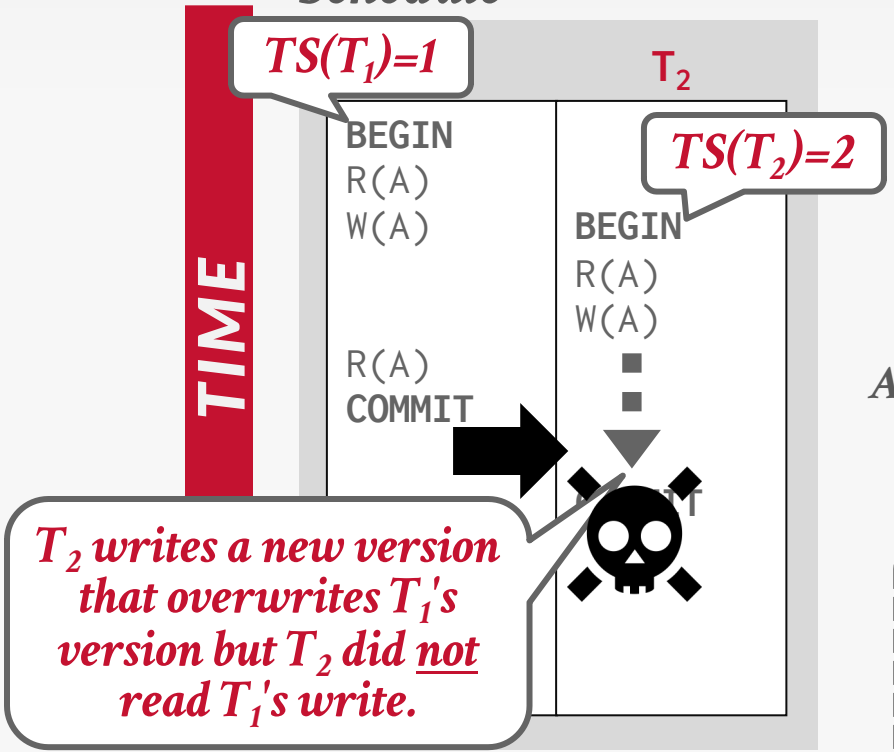
txnid	timestamp	status
T_1	1	Committed
T_2	2	Active

Lock Table

obj	owner	waiting
A	T_2 ▸ X	

SERIALIZABLE SNAPSHOT ISOLATION

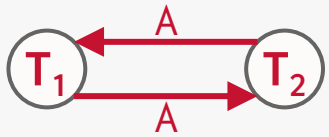
Schedule



Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	2	456
A_2	2	-	789

Anti-Dependency Graph:



Txn Status Table

txnid	timestamp	status
T_1	1	Committed
T_2	2	Active

Lock Table

obj	owner	waiting
A	T_2 ▸ X	

VERSION STORAGE

The DBMS uses the tuples' pointer field to create a **version chain** per logical tuple.

→ This allows the DBMS to find the version that is visible to a particular txn at runtime.

→ Indexes always point to the "head" of the chain.

Different storage schemes determine where/what to store for each version.

VERSION STORAGE

Approach #1: Append-Only Storage

→ New versions are appended to the same table space.

Approach #2: Time-Travel Storage

→ Old versions are copied to separate table space.

Approach #3: Delta Storage

→ The original values of the modified attributes are copied into a separate delta record space.

VERSION STORAGE

Approach #1: Append-Only Storage ← *Less Common*

→ New versions are appended to the same table space.

Approach #2: Time-Travel Storage ← *Rare*

→ Old versions are copied to separate table space.

Approach #3: Delta Storage ← *Common*

→ The original values of the modified attributes are copied into a separate delta record space.

VERSION STORAGE



Don't Do This!

Approach #1: Append-Only Storage ← *Less Common*

→ New versions are appended to the same table space.

Approach #2: Time-Travel Storage ← *Rare*

→ Old versions are copied to separate table space.

Approach #3: Delta Storage ← *Common*

→ The original values of the modified attributes are copied into a separate delta record space.

VERSION STORAGE



Approach #1: Append-Onl

→ New versions are appended to

Approach #2: Time-Trave

→ Old versions are copied to sep

Approach #3: Delta Storag

→ The original values of the mo
separate delta record space.

Andy Pavlo

The Part of PostgreSQL We Hate the Most

Posted on April 26, 2023

This article was written in collaboration with Bohan Zhang and originally appeared on the OtterTune website.

There are a lot of choices in databases (897 as of April 2023). With so many systems, it's hard to know what to pick! But there is an interesting phenomenon where the Internet collectively decides on the default choice for new applications. In the 2000s, the conventional wisdom selected MySQL because rising tech stars like Google and Facebook were using it. Then in the 2010s, it was MongoDB because non-durable writes made it "webscale". In the last five years, PostgreSQL has become the Internet's darling DBMS. And for good reasons! It's dependable, feature-rich, extensible, and well-suited for most operational workloads.

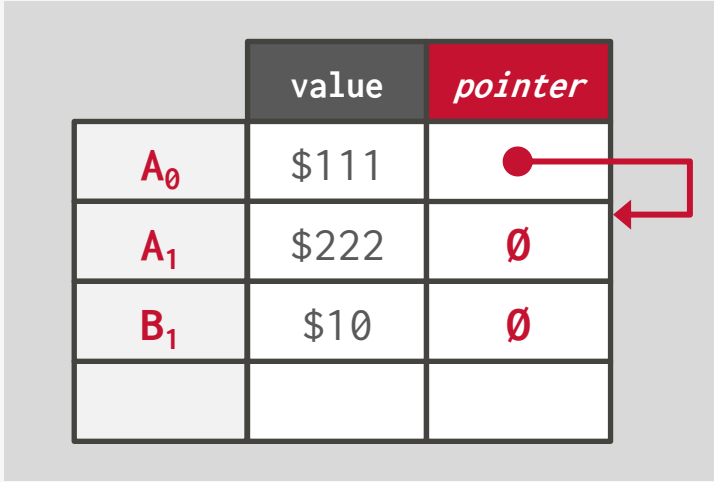
But as much as we love PostgreSQL at OtterTune, certain aspects of it are not great. So instead of writing yet another blog article like everyone else touting the awesomeness of everyone's favorite elephant-themed DBMS, we want to discuss the one major thing that sucks: how PostgreSQL implements multi-version concurrency control (MVCC). Our research at Carnegie Mellon University and experience optimizing PostgreSQL database instances on Amazon RDS have shown that its MVCC implementation is the worst

APPEND-ONLY STORAGE

All the physical versions of a logical tuple are stored in the same table space. The versions are inter-mixed.

On every update, append a new version of the tuple into an empty space in the table.

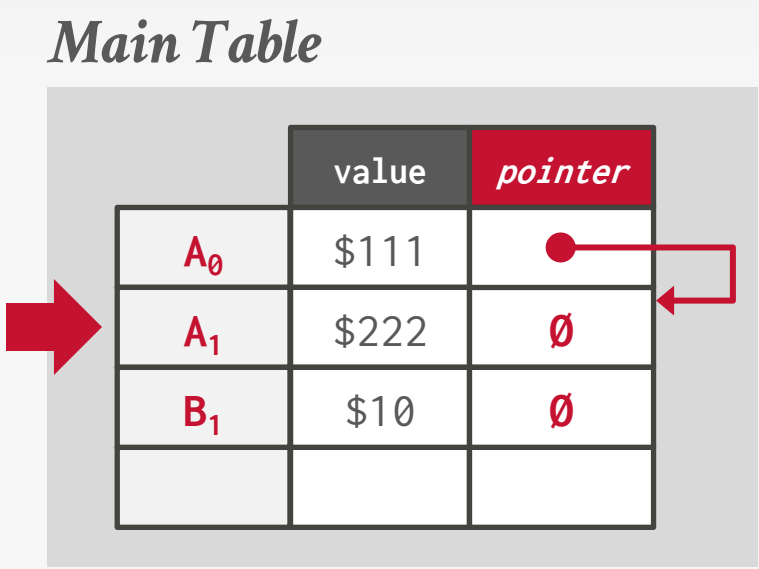
Main Table



APPEND-ONLY STORAGE

All the physical versions of a logical tuple are stored in the same table space. The versions are inter-mixed.

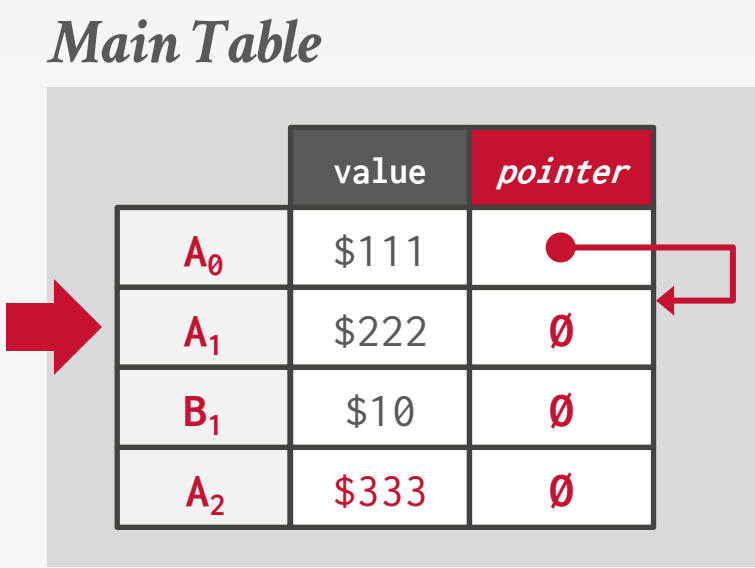
On every update, append a new version of the tuple into an empty space in the table.



APPEND-ONLY STORAGE

All the physical versions of a logical tuple are stored in the same table space. The versions are inter-mixed.

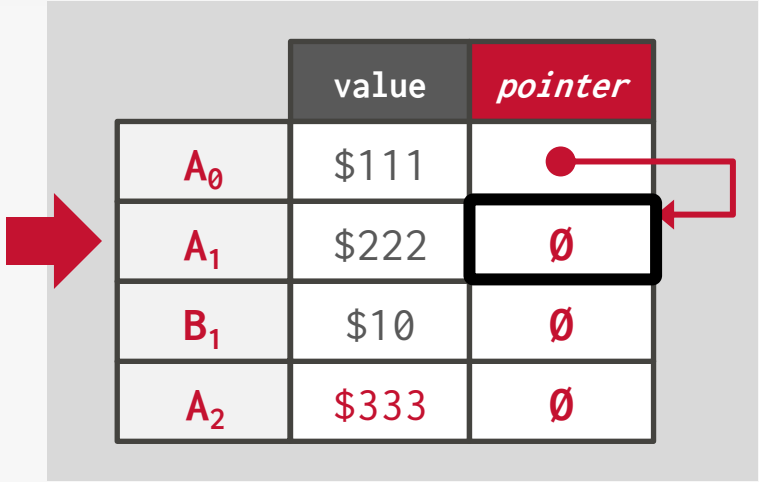
On every update, append a new version of the tuple into an empty space in the table.



APPEND-ONLY STORAGE

All the physical versions of a logical tuple are stored in the same table space. The versions are inter-mixed.

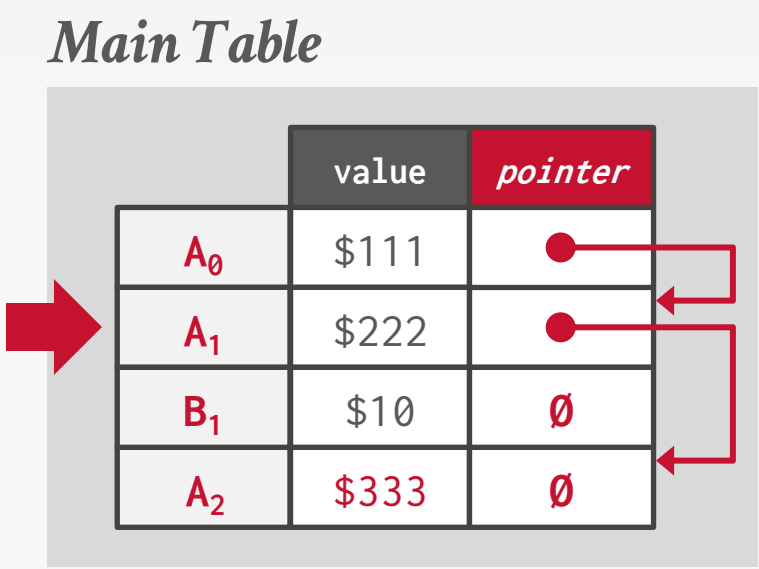
On every update, append a new version of the tuple into an empty space in the table.



APPEND-ONLY STORAGE

All the physical versions of a logical tuple are stored in the same table space. The versions are inter-mixed.

On every update, append a new version of the tuple into an empty space in the table.



VERSION CHAIN ORDERING

Approach #1: Oldest-to-Newest (O2N)

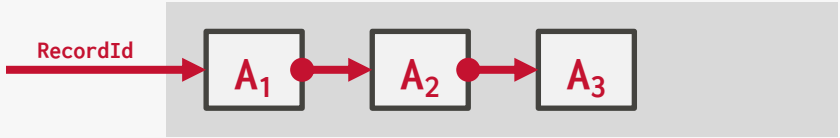
→ Append new version to end of the chain.
→ Must traverse chain on look-ups.

VERSION CHAIN ORDERING

Approach #1: Oldest-to-Newest (O2N)

→ Append new version to end of the chain.

→ Must traverse chain on look-ups.

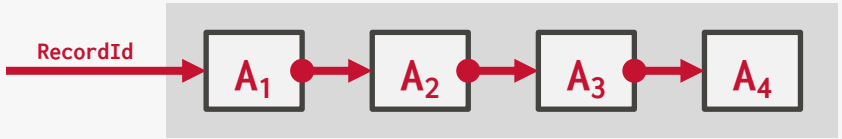


VERSION CHAIN ORDERING

Approach #1: Oldest-to-Newest (O2N)

→ Append new version to end of the chain.

→ Must traverse chain on look-ups.

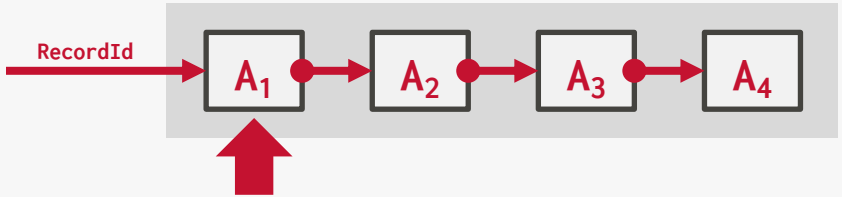


VERSION CHAIN ORDERING

Approach #1: Oldest-to-Newest (O2N)

→ Append new version to end of the chain.

→ Must traverse chain on look-ups.

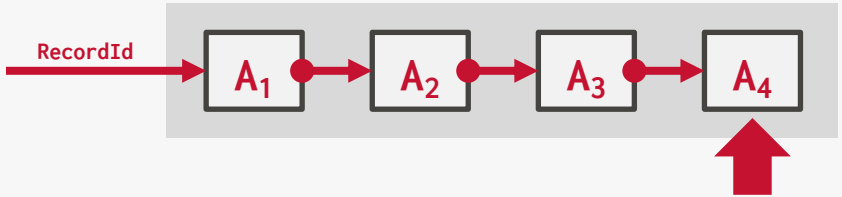


VERSION CHAIN ORDERING

Approach #1: Oldest-to-Newest (O2N)

→ Append new version to end of the chain.

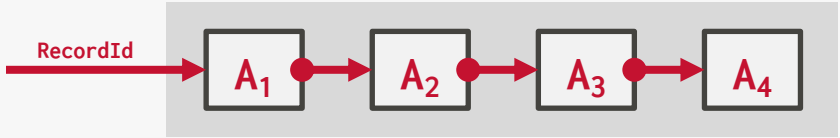
→ Must traverse chain on look-ups.



VERSION CHAIN ORDERING

Approach #1: Oldest-to-Newest (O2N)

→ Append new version to end of the chain.
→ Must traverse chain on look-ups.



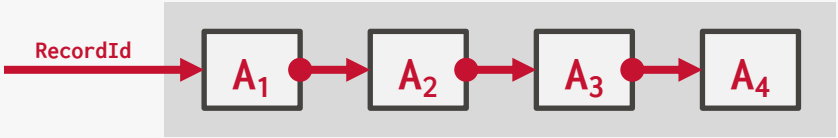
Approach #2: Newest-to-Oldest (N2O)

→ Must update index pointers for every new version.
→ Do not have to traverse chain on look-ups.
→ Better approach if most txns only want the newest version.

VERSION CHAIN ORDERING

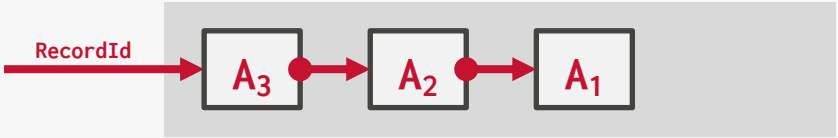
Approach #1: Oldest-to-Newest (O2N)

→ Append new version to end of the chain.
→ Must traverse chain on look-ups.



Approach #2: Newest-to-Oldest (N2O)

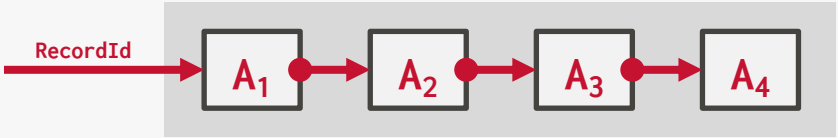
→ Must update index pointers for every new version.
→ Do not have to traverse chain on look-ups.
→ Better approach if most txns only want the newest version.



VERSION CHAIN ORDERING

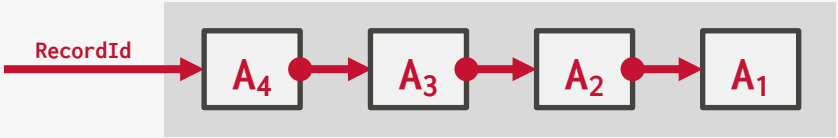
Approach #1: Oldest-to-Newest (O2N)

→ Append new version to end of the chain.
→ Must traverse chain on look-ups.



Approach #2: Newest-to-Oldest (N2O)

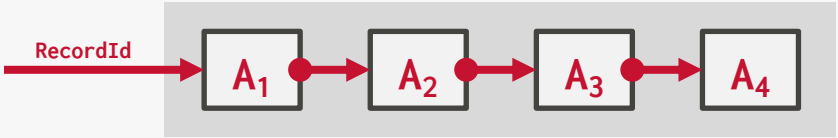
→ Must update index pointers for every new version.
→ Do not have to traverse chain on look-ups.
→ Better approach if most txns only want the newest version.



VERSION CHAIN ORDERING

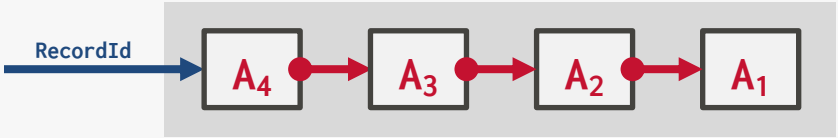
Approach #1: Oldest-to-Newest (O2N)

→ Append new version to end of the chain.
→ Must traverse chain on look-ups.



Approach #2: Newest-to-Oldest (N2O)

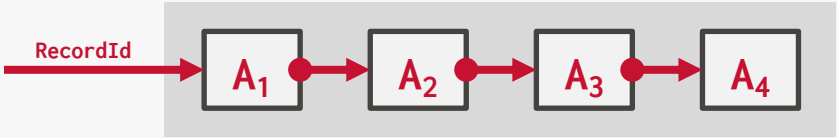
→ Must update index pointers for every new version.
→ Do not have to traverse chain on look-ups.
→ Better approach if most txns only want the newest version.



VERSION CHAIN ORDERING

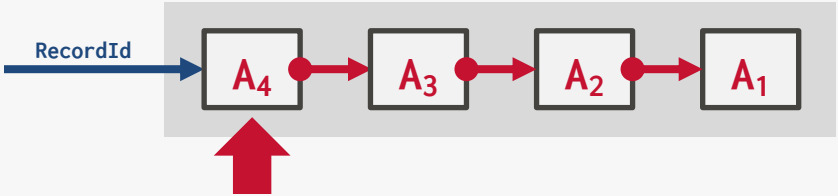
Approach #1: Oldest-to-Newest (O2N)

→ Append new version to end of the chain.
→ Must traverse chain on look-ups.



Approach #2: Newest-to-Oldest (N2O)

→ Must update index pointers for every new version.
→ Do not have to traverse chain on look-ups.
→ Better approach if most txns only want the newest version.



TIME-TRAVEL STORAGE

Main Table

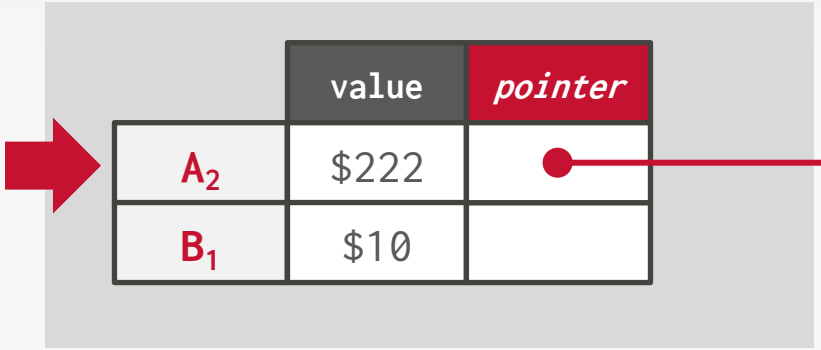
	value	*pointer*
A_2	$222	●
B_1	$10	

Time-Travel Table

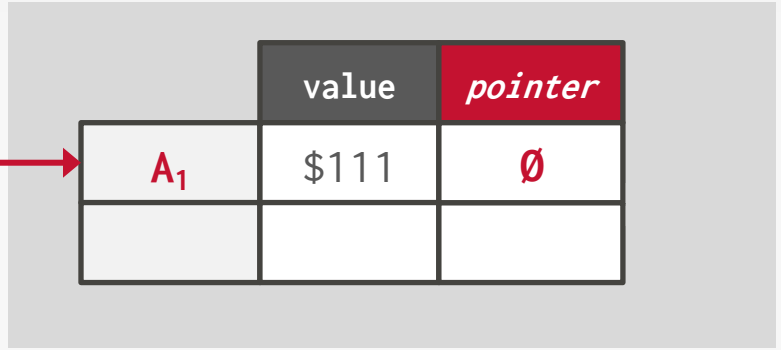
	value	*pointer*
A_1	$111	Ø

TIME-TRAVEL STORAGE

Main Table



Time-Travel Table

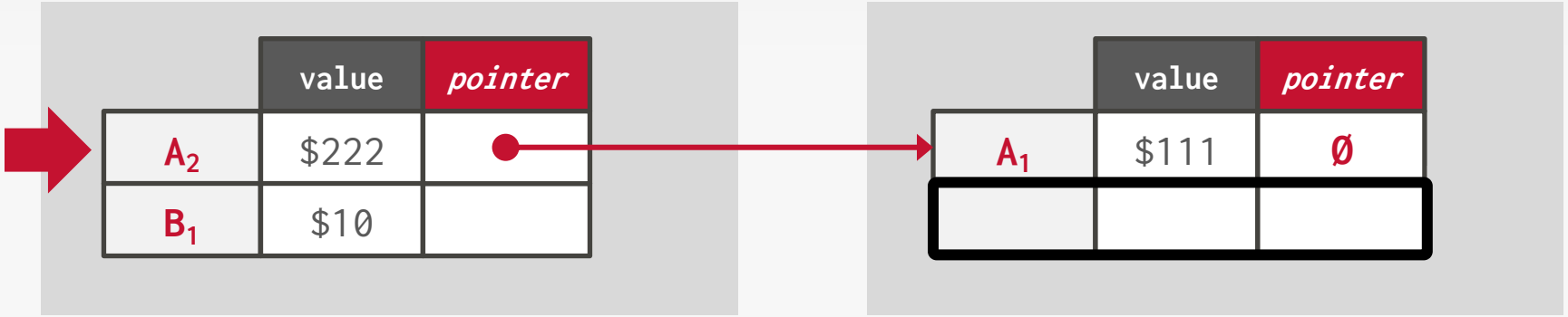


On every update, copy the current version to the time-travel table. Update pointers.

TIME-TRAVEL STORAGE

Main Table

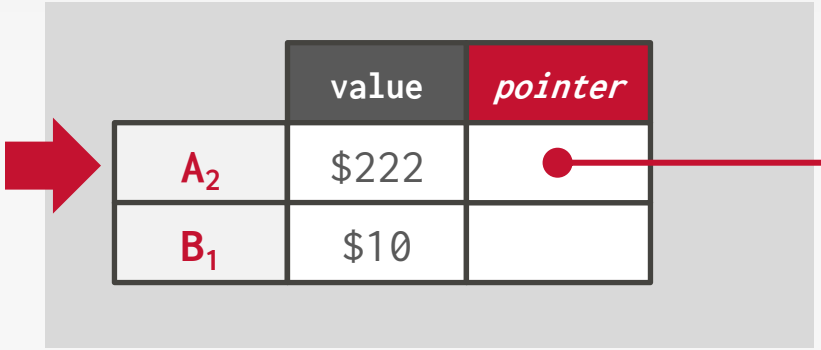
Time-Travel Table



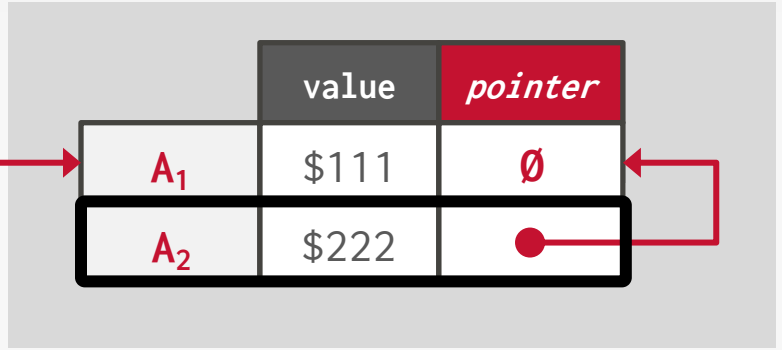
On every update, copy the current version to the time-travel table. Update pointers.

TIME-TRAVEL STORAGE

Main Table



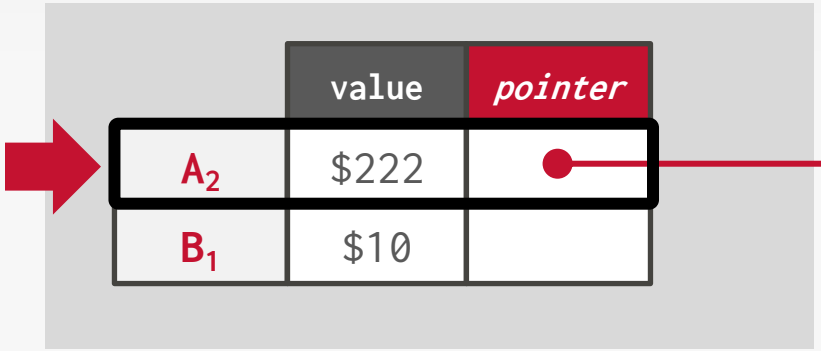
Time-Travel Table



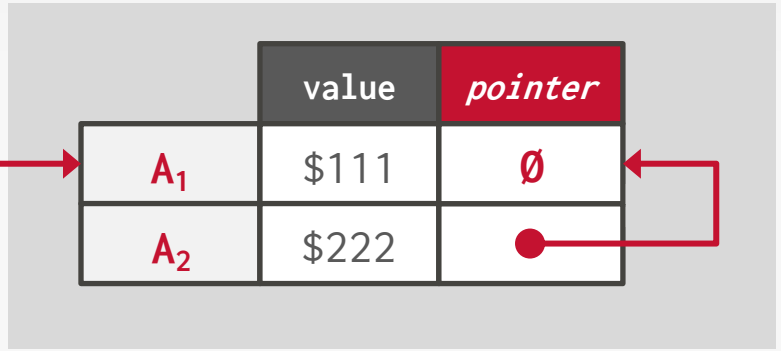
On every update, copy the current version to the time-travel table. Update pointers.

TIME-TRAVEL STORAGE

Main Table



Time-Travel Table

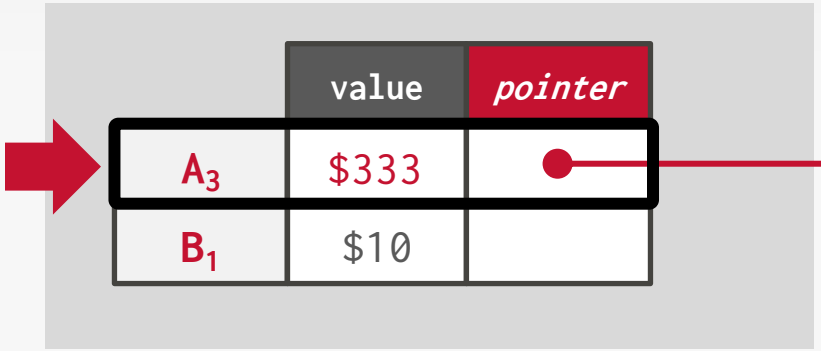


On every update, copy the current version to the time-travel table. Update pointers.

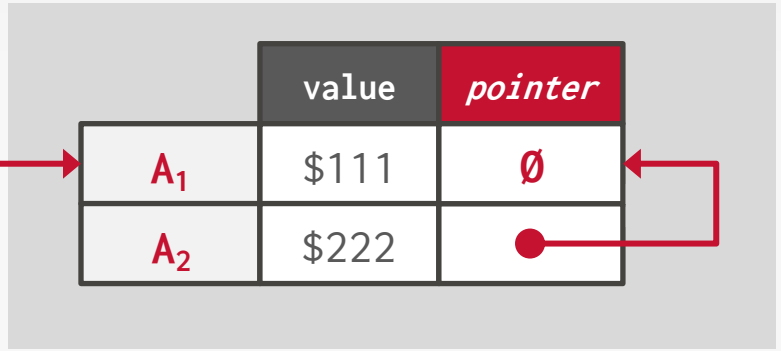
Overwrite master version in the main table and update pointers.

TIME-TRAVEL STORAGE

Main Table



Time-Travel Table

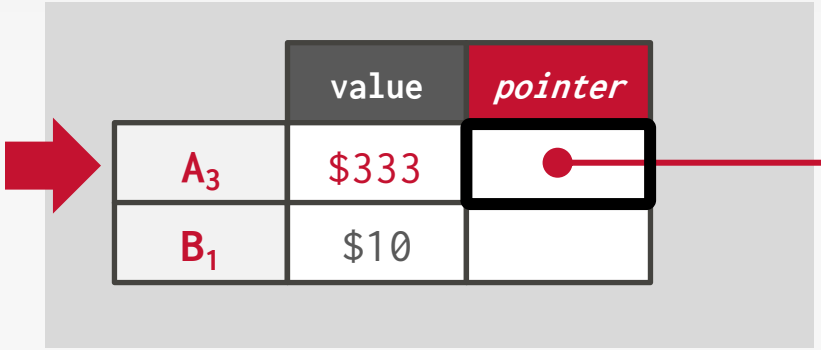


On every update, copy the current version to the time-travel table. Update pointers.

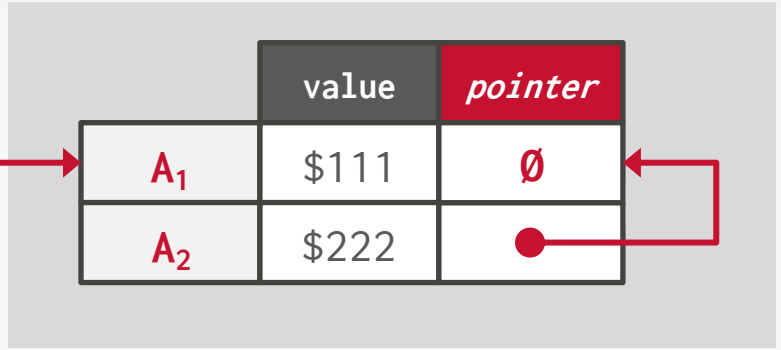
Overwrite master version in the main table and update pointers.

TIME-TRAVEL STORAGE

Main Table



Time-Travel Table

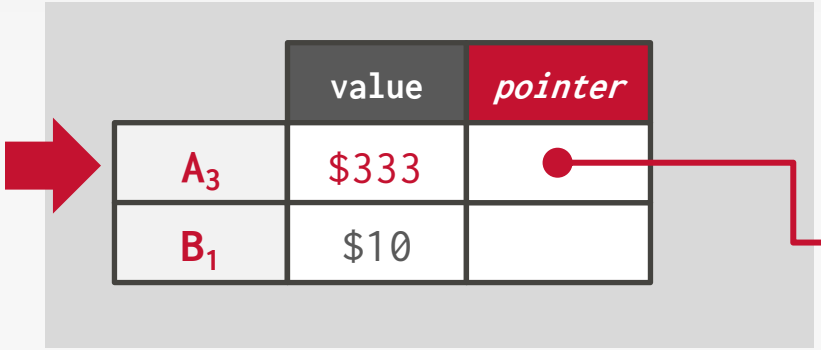


On every update, copy the current version to the time-travel table. Update pointers.

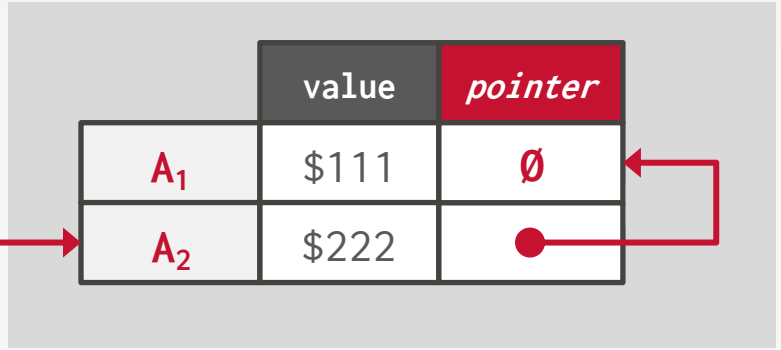
Overwrite master version in the main table and update pointers.

TIME-TRAVEL STORAGE

Main Table



Time-Travel Table



On every update, copy the current version to the time-travel table. Update pointers.

Overwrite master version in the main table and update pointers.

DELTA STORAGE

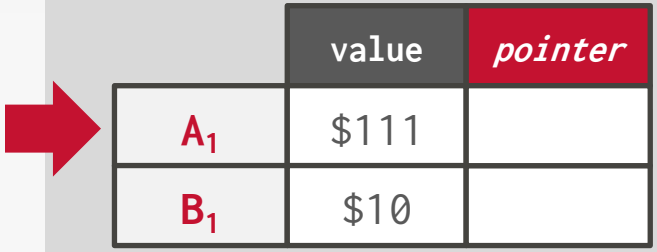
Main Table

	value	*pointer*
A_1	$111	
B_1	$10	

Delta Storage Segment

DELTA STORAGE

Main Table

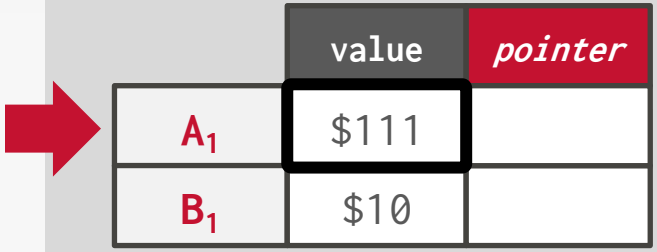


Delta Storage Segment

On every update, copy only the column values that were modified to the delta storage and overwrite the master version.

DELTA STORAGE

Main Table

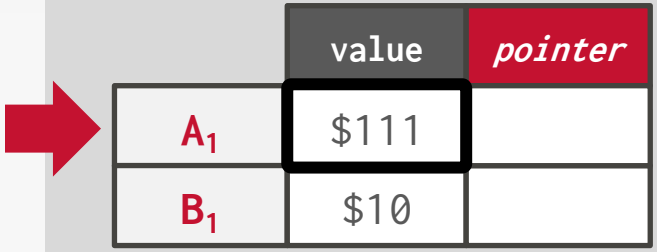


Delta Storage Segment

On every update, copy only the column values that were modified to the delta storage and overwrite the master version.

DELTA STORAGE

Main Table



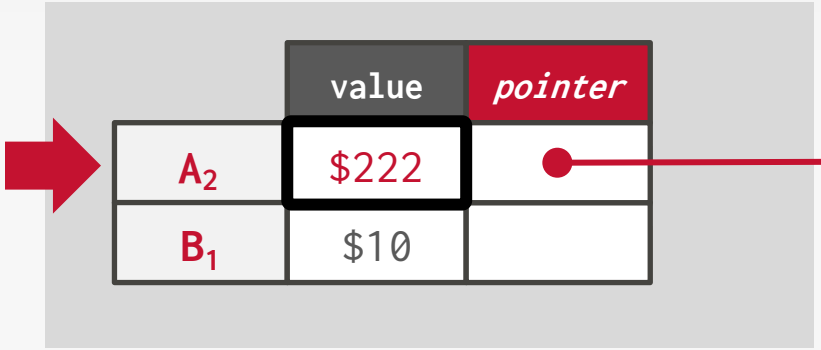
Delta Storage Segment



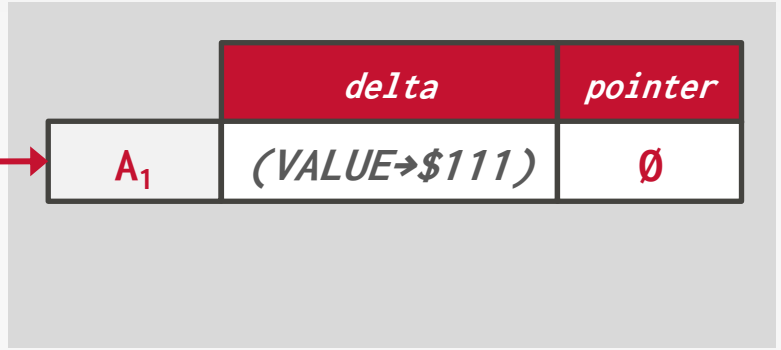
On every update, copy only the column values that were modified to the delta storage and overwrite the master version.

DELTA STORAGE

Main Table

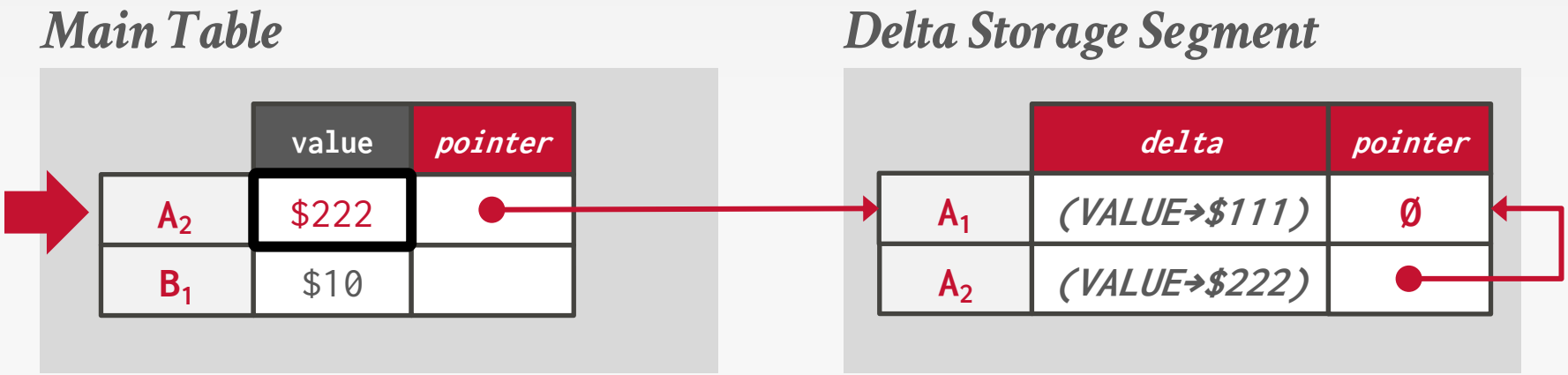


Delta Storage Segment



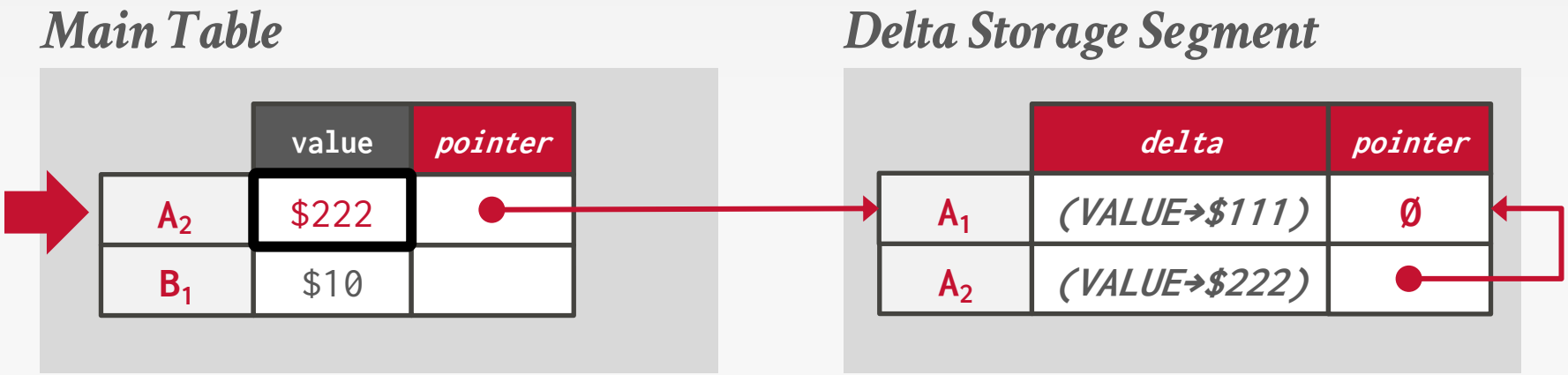
On every update, copy only the column values that were modified to the delta storage and overwrite the master version.

DELTA STORAGE



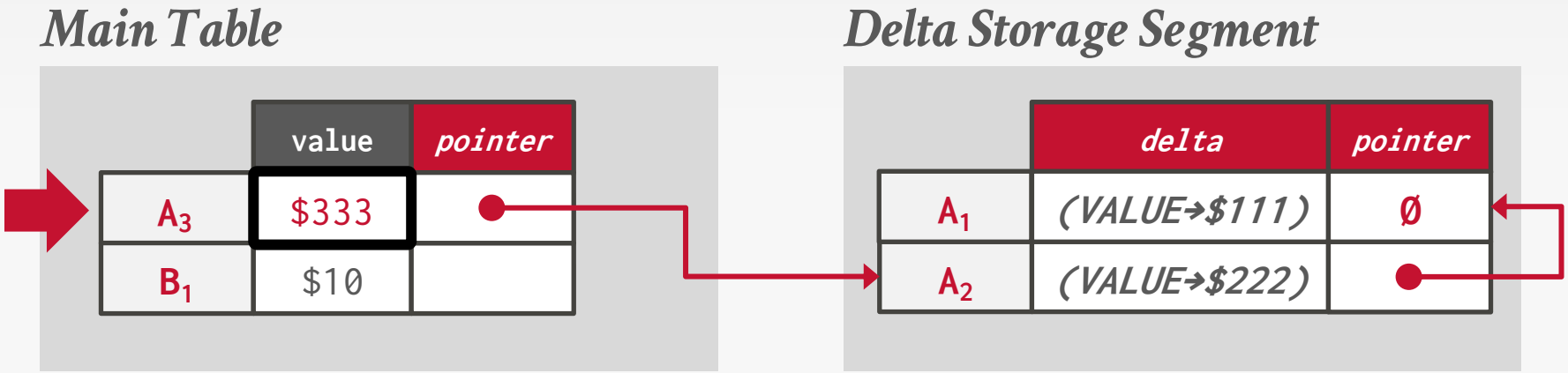
On every update, copy only the column values that were modified to the delta storage and overwrite the master version.

DELTA STORAGE



On every update, copy only the column values that were modified to the delta storage and overwrite the master version.

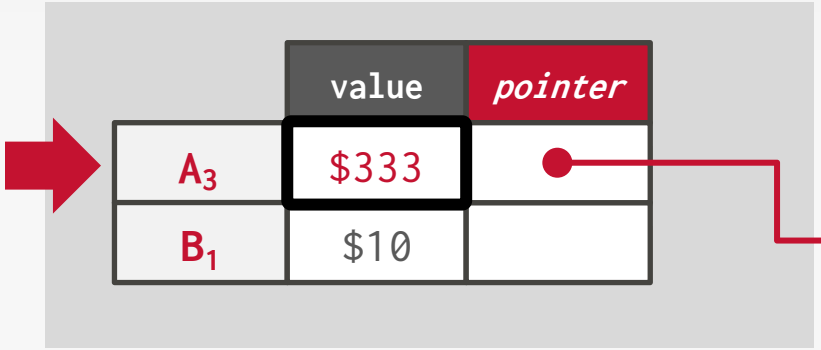
DELTA STORAGE



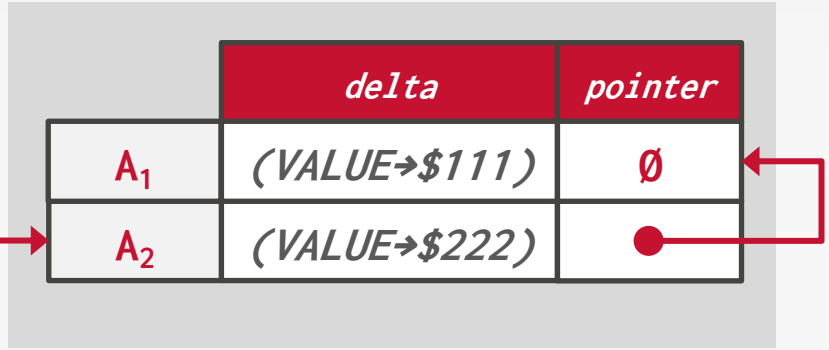
On every update, copy only the column values that were modified to the delta storage and overwrite the master version.

DELTA STORAGE

Main Table



Delta Storage Segment



On every update, copy only the column values that were modified to the delta storage and overwrite the master version.

Txns can recreate old versions by applying the delta in reverse order.

GARBAGE COLLECTION

The DBMS needs to remove **reclaimable** physical versions from the database over time.

→ No active txn in the DBMS can "see" that version (SI).
→ The version was created by an aborted txn.

Two additional design decisions:

→ How to look for expired versions?
→ How to decide when it is safe to reclaim memory?

GARBAGE COLLECTION

The DBMS needs to remove **reclaimable** physical versions from the database over time.

→ No active txn in the DBMS can "see" that version (SI).
→ The version was created by an aborted txn.

Two additional design decisions:

→ How to look for expired versions?
→ How to decide when it is safe to reclaim memory?

GARBAGE COLLECTION

Approach #1: Tuple-level

→ Find old versions by examining tuples directly.
→ Background Vacuuming vs. Cooperative Cleaning

Approach #2: Transaction-level

→ Txns keep track of their old versions so the DBMS does not have to scan tuples to determine visibility.

TUPLE-LEVEL GC

Txn #1

TS=12

Txn #2

TS=25

	begin-ts	end-ts
A_{100}	*1*	*9*
B_{100}	*1*	*9*
B_{101}	*10*	*20*

TUPLE-LEVEL GC

Txn #1

TS=12

Txn #2

TS=25

	begin-ts	end-ts
A_{100}	*1*	*9*
B_{100}	*1*	*9*
B_{101}	*10*	*20*

Background Vacuuming:

Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

TUPLE-LEVEL GC

Txn #1

TS=12

Txn #2

TS=25

Vacuum

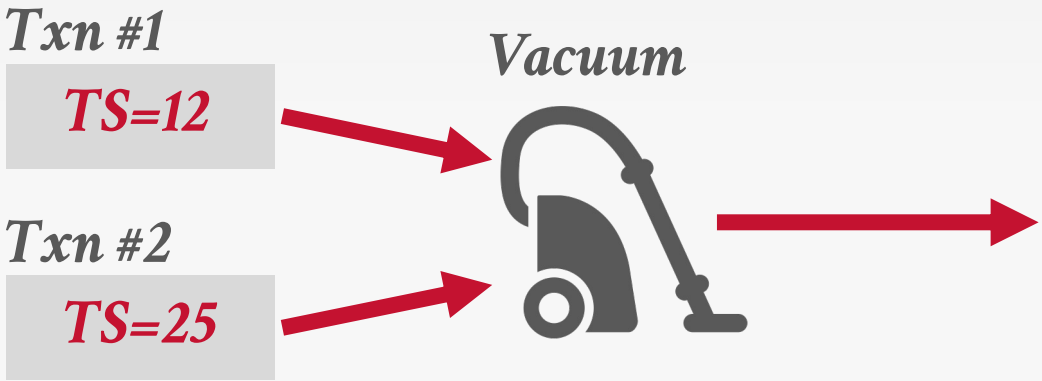


	begin-ts	end-ts
A_{100}	1	9
B_{100}	1	9
B_{101}	10	20

Background Vacuuming:

Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

TUPLE-LEVEL GC

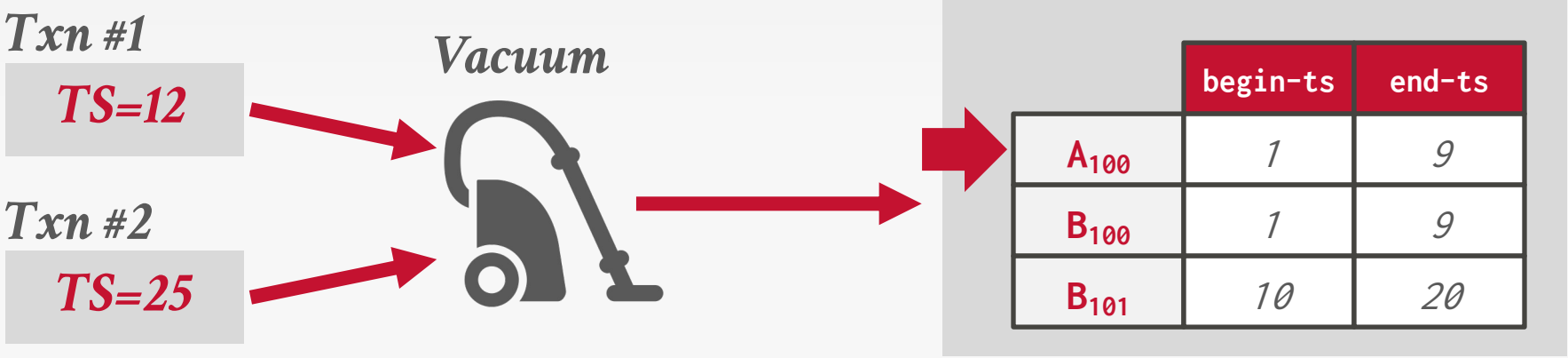


	begin-ts	end-ts
A_{100}	1	9
B_{100}	1	9
B_{101}	10	20

Background Vacuuming:

Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

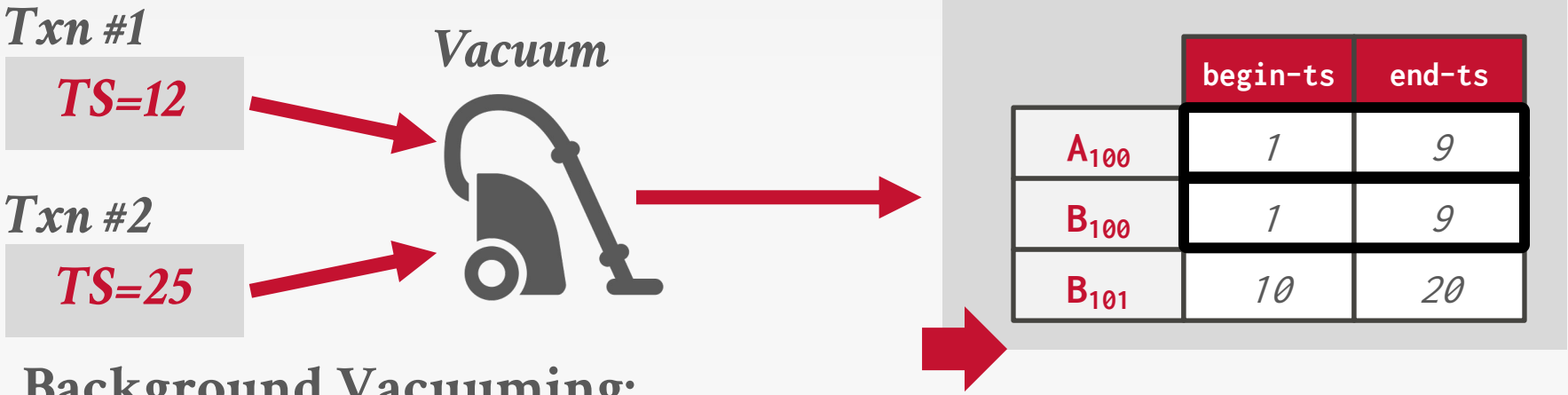
TUPLE-LEVEL GC



Background Vacuuming:

Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

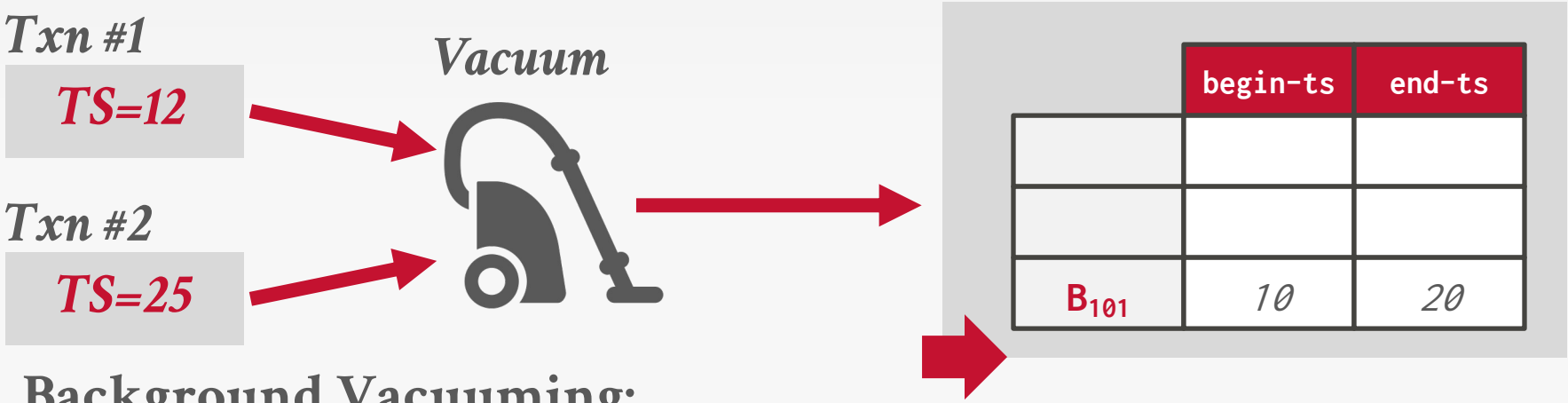
TUPLE-LEVEL GC



Background Vacuuming:

Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

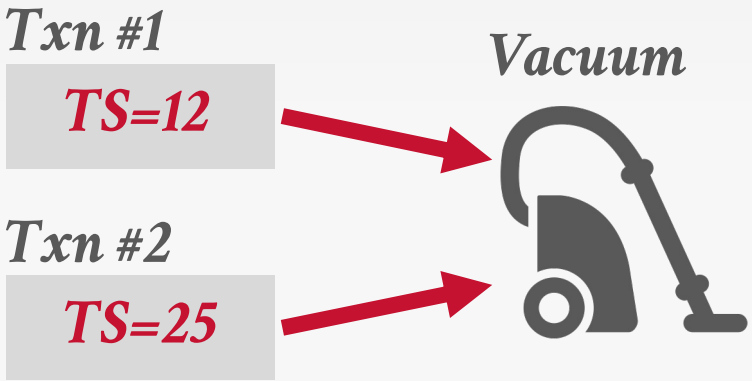
TUPLE-LEVEL GC

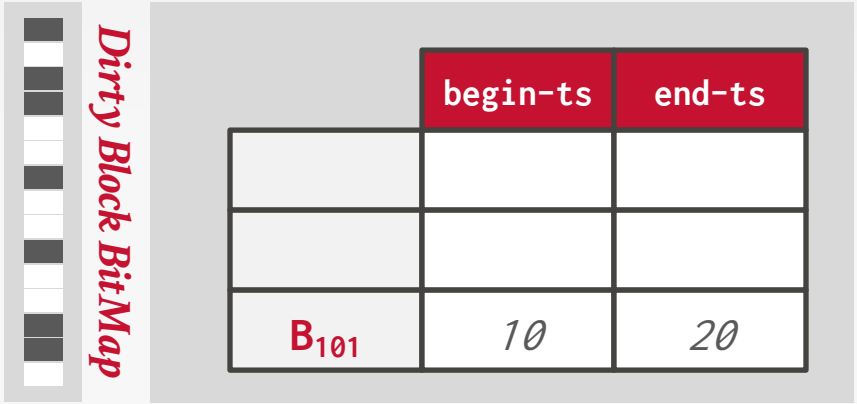


Background Vacuuming:

Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

TUPLE-LEVEL GC

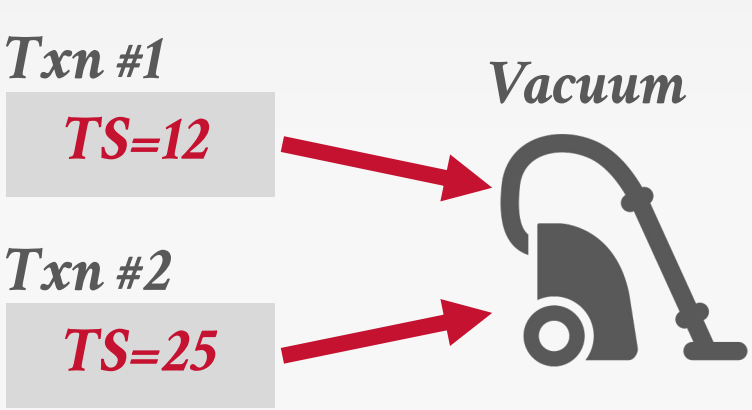


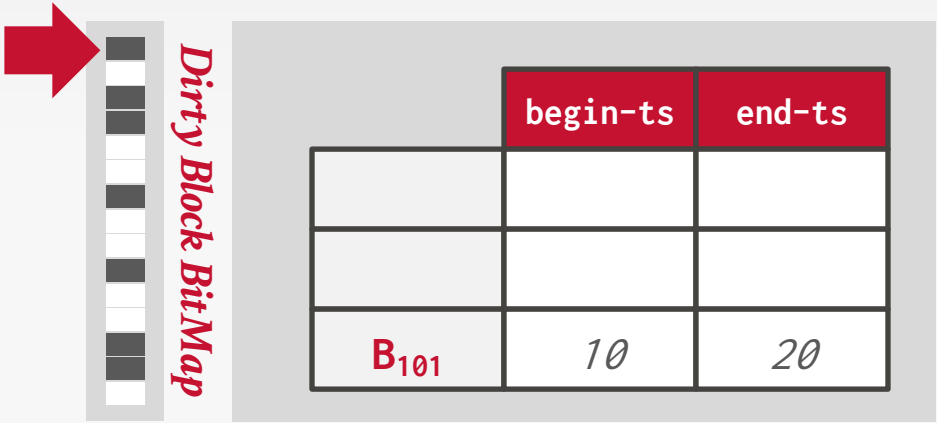


Background Vacuuming:

Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

TUPLE-LEVEL GC

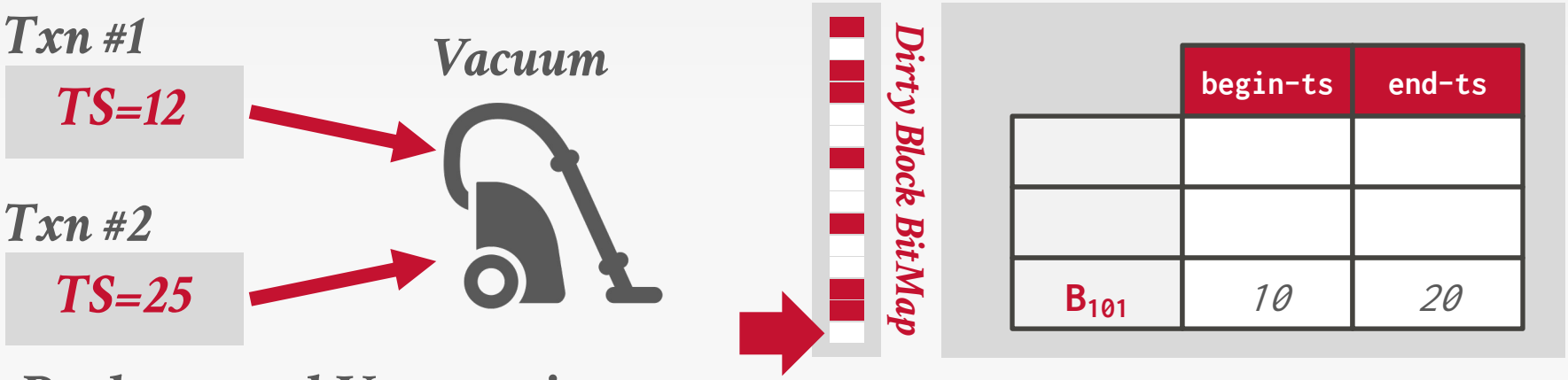




Background Vacuuming:

Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

TUPLE-LEVEL GC



Background Vacuuming:

Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

TUPLE-LEVEL GC

Txn #1

TS=12

Txn #2

TS=25

Background Vacuuming:

Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

TUPLE-LEVEL GC

Txn #1

TS=12

Txn #2

TS=25

Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

Cooperative Cleaning:
Worker threads identify reclaimable versions as they traverse version chain. Only works with O2N.

TUPLE-LEVEL GC

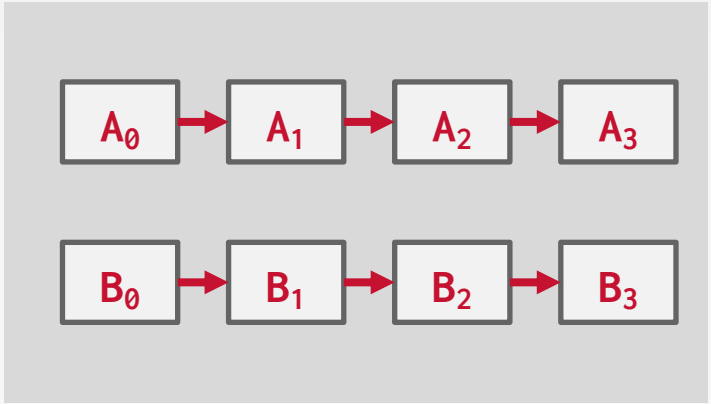
Txn #1

TS=12

Txn #2

TS=25





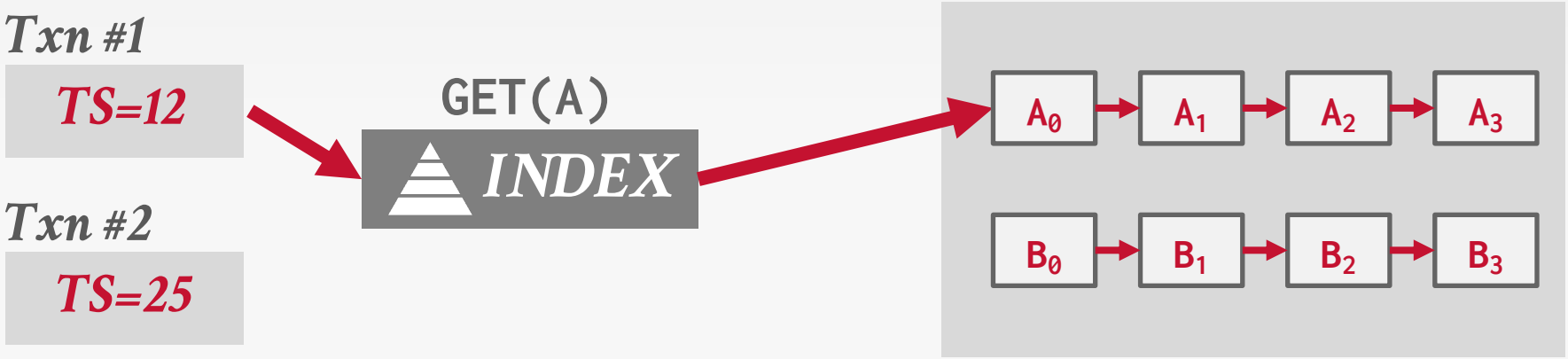
Background Vacuuming:

Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

Cooperative Cleaning:

Worker threads identify reclaimable versions as they traverse version chain. Only works with O2N.

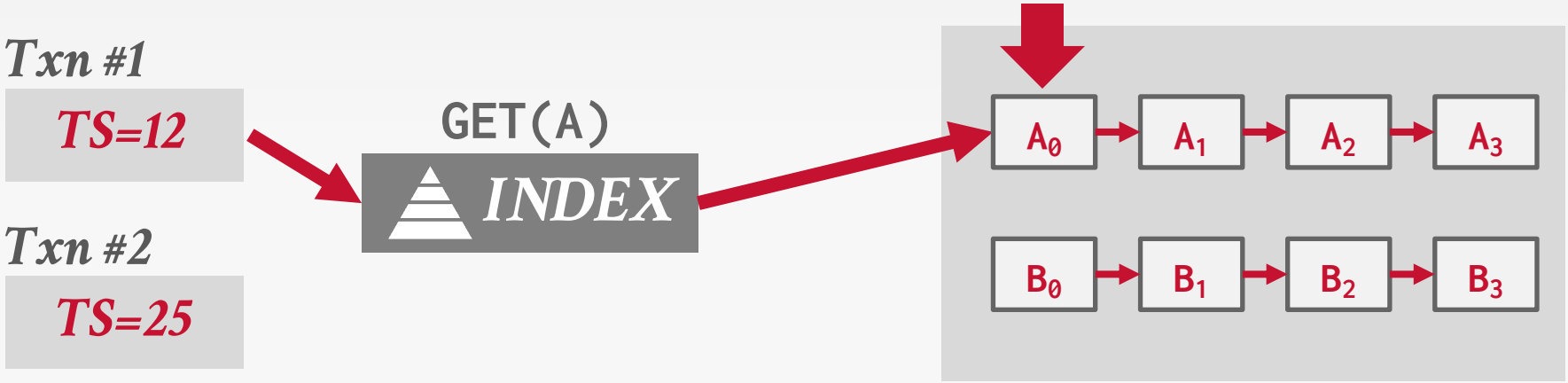
TUPLE-LEVEL GC



Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

Cooperative Cleaning:
Worker threads identify reclaimable versions as they traverse version chain. Only works with O2N.

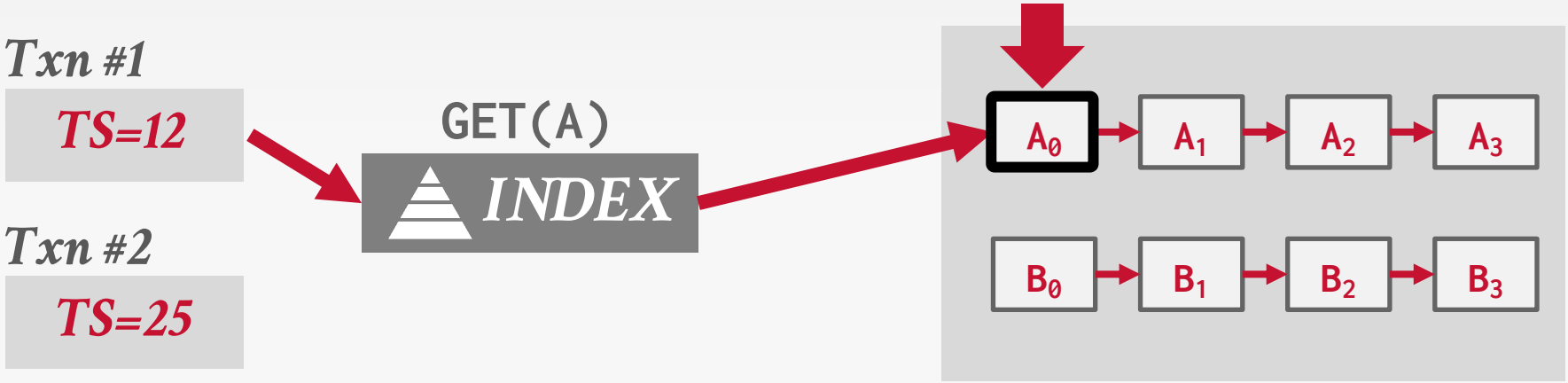
TUPLE-LEVEL GC



Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

Cooperative Cleaning:
Worker threads identify reclaimable versions as they traverse version chain. Only works with O2N.

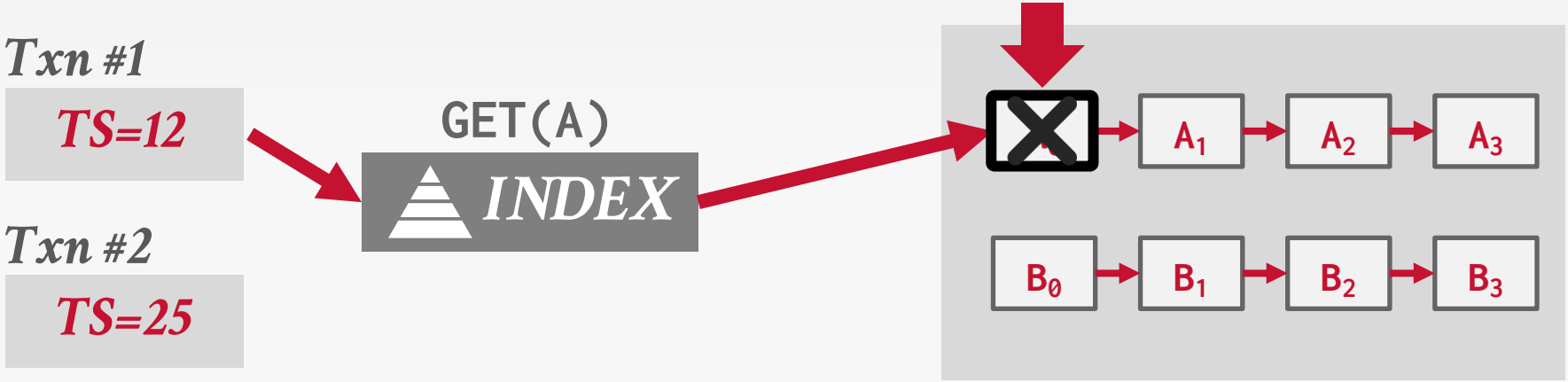
TUPLE-LEVEL GC



Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

Cooperative Cleaning:
Worker threads identify reclaimable versions as they traverse version chain. Only works with O2N.

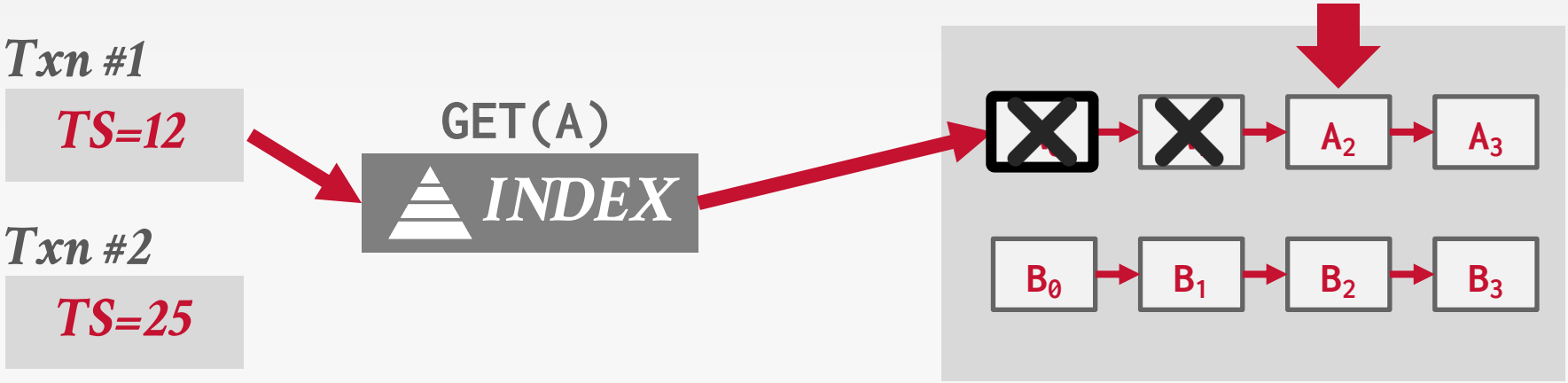
TUPLE-LEVEL GC



Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

Cooperative Cleaning:
Worker threads identify reclaimable versions as they traverse version chain. Only works with O2N.

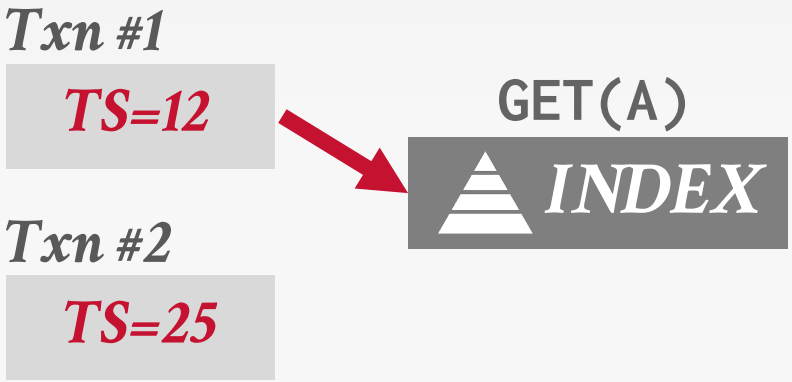
TUPLE-LEVEL GC

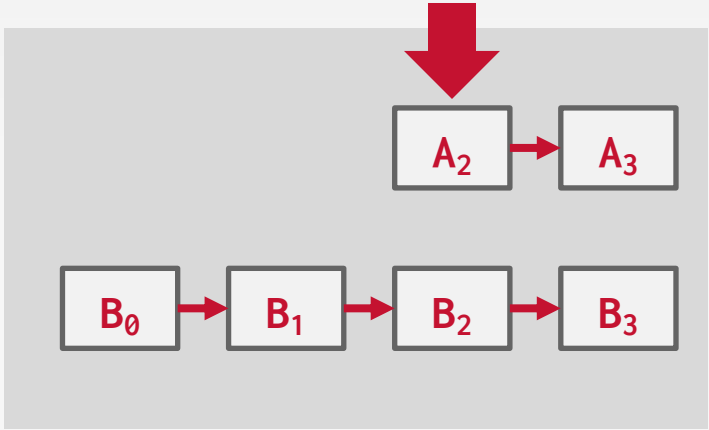


Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

Cooperative Cleaning:
Worker threads identify reclaimable versions as they traverse version chain. Only works with O2N.

TUPLE-LEVEL GC





Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

Cooperative Cleaning:
Worker threads identify reclaimable versions as they traverse version chain. Only works with O2N.

TUPLE-LEVEL GC



Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

Cooperative Cleaning:
Worker threads identify reclaimable versions as they traverse version chain. Only works with O2N.

TRANSACTION-LEVEL GC

Each txn keeps track of its read/write set.

On commit/abort, the txn provides this information to a centralized vacuum worker.

The DBMS periodically determines when all versions created by a finished txn are no longer visible.

TRANSACTION-LEVEL GC

Txn #1

```
BEGIN TS=10
```

	begin-ts	end-ts	value
A_2	1	∞	-
B_6	8	∞	-

TRANSACTION-LEVEL GC

Txn #1

```
BEGIN TS=10
```



	begin-ts	end-ts	value
A_2	1	∞	-
B_6	8	∞	-

TRANSACTION-LEVEL GC

Txn #1

```
BEGIN TS=10
```



	begin-ts	end-ts	value
A_2	1	∞	-
B_6	8	∞	-

TRANSACTION-LEVEL GC

Txn #1

```
BEGIN TS=10
```



	begin-ts	end-ts	value
A_2	1	10	-
B_6	8	∞	-
A_3	10	∞	-

TRANSACTION-LEVEL GC

Txn #1

BEGIN TS=10

Old Versions

A_2



	begin-ts	end-ts	value
A_2	1	10	-
B_6	8	∞	-
A_3	10	∞	-

TRANSACTION-LEVEL GC

Txn #1

BEGIN TS=10

Old Versions

A_2

UPDATE(A)

	begin-ts	end-ts	value
A_2	1	10	-
B_6	8	∞	-
A_3	10	∞	-

TRANSACTION-LEVEL GC

Txn #1

BEGIN TS=10

Old Versions

A_2







	begin-ts	end-ts	value
A_2	1	10	-
B_6	8	∞	-
A_3	10	∞	-

TRANSACTION-LEVEL GC

Txn #1

BEGIN TS=10

Old Versions

A_2

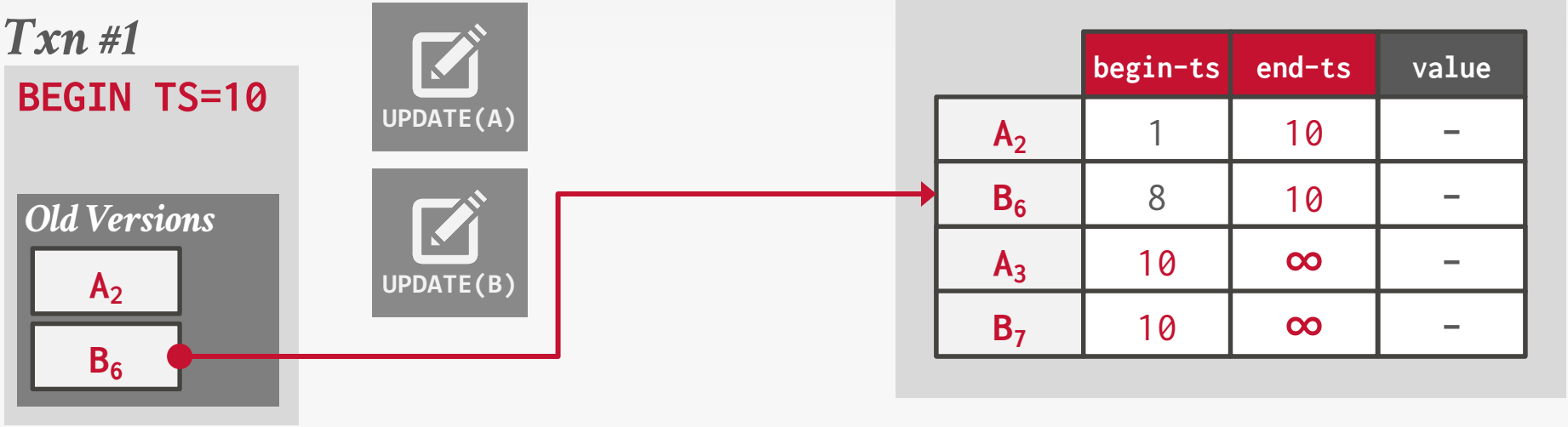






	begin-ts	end-ts	value
A_2	1	10	-
B_6	8	10	-
A_3	10	∞	-
B_7	10	∞	-

TRANSACTION-LEVEL GC



TRANSACTION-LEVEL GC

Txn #1

BEGIN TS=10
COMMIT TS=15

Old Versions

- A_2
- B_6





	begin-ts	end-ts	value
A_2	1	10	-
B_6	8	10	-
A_3	10	∞	-
B_7	10	∞	-

TRANSACTION-LEVEL GC

Txn #1

```
BEGIN TS=10
COMMIT TS=15
```

Old Versions

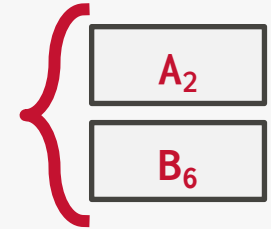




	begin-ts	end-ts	value
A_2	1	10	-
B_6	8	10	-
A_3	10	∞	-
B_7	10	∞	-

Vacuum

TS<10





INDEX MANAGEMENT

Primary key indexes point to version chain head.

→ How often the DBMS must update the pkey index depends on whether the system creates new versions when a tuple is updated.

→ If a txn updates a tuple's pkey attribute(s), then this is treated as a **DELETE** followed by an **INSERT**.

Secondary indexes are more complicated…

IND

Primary key inde
→ How often the D
whether the sys
updated.
→ If a txn updates
a **DELETE** follo

Secondary inde

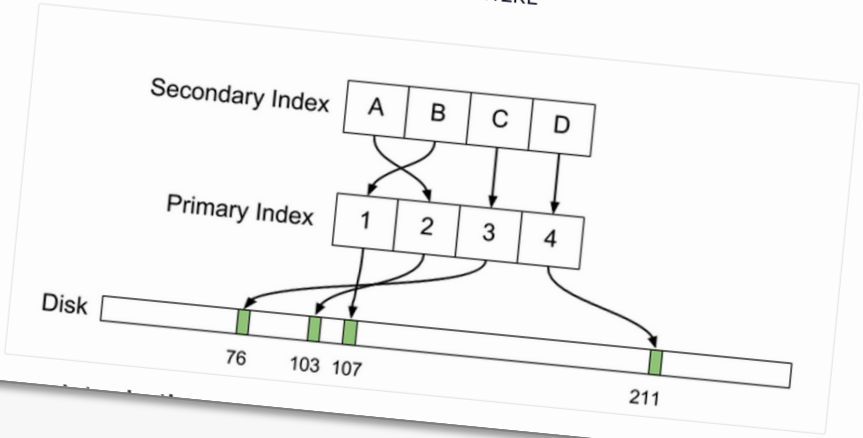
UBER Engineering

JOIN THE TEAM | MEET THE PEOPLE

ARCHITECTURE

WHY UBER ENGINEERING SWITCHED FROM POSTGRES TO MYSQL

JULY 26, 2016
BY EVAN KLITZKE



SECONDARY INDEXES

Approach #1: Logical Pointers

→ Use a fixed identifier per tuple that does not change.
→ Requires an extra indirection layer.
→ Primary Key vs. Tuple Id

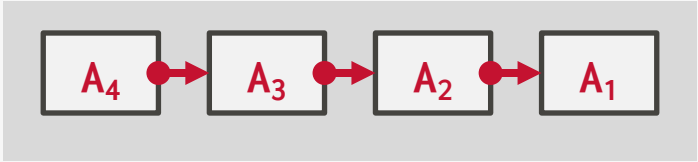
Approach #2: Physical Pointers

→ Use the physical address to the version chain head.

INDEX POINTERS: APPEND-ONLY

PRIMARY INDEX

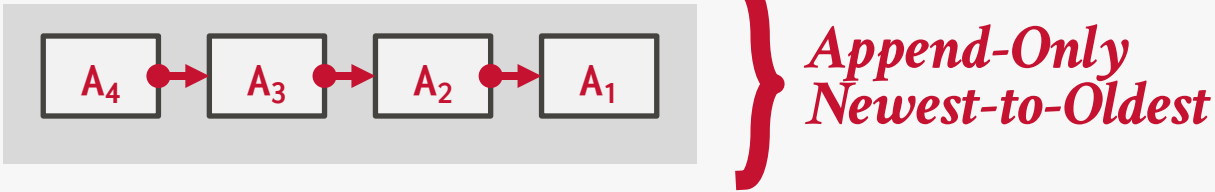
SECONDARY INDEX



INDEX POINTERS: APPEND-ONLY

PRIMARY INDEX

SECONDARY INDEX



INDEX POINTERS: APPEND-ONLY

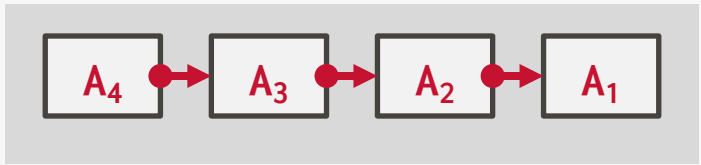
GET(A)



PRIMARY INDEX

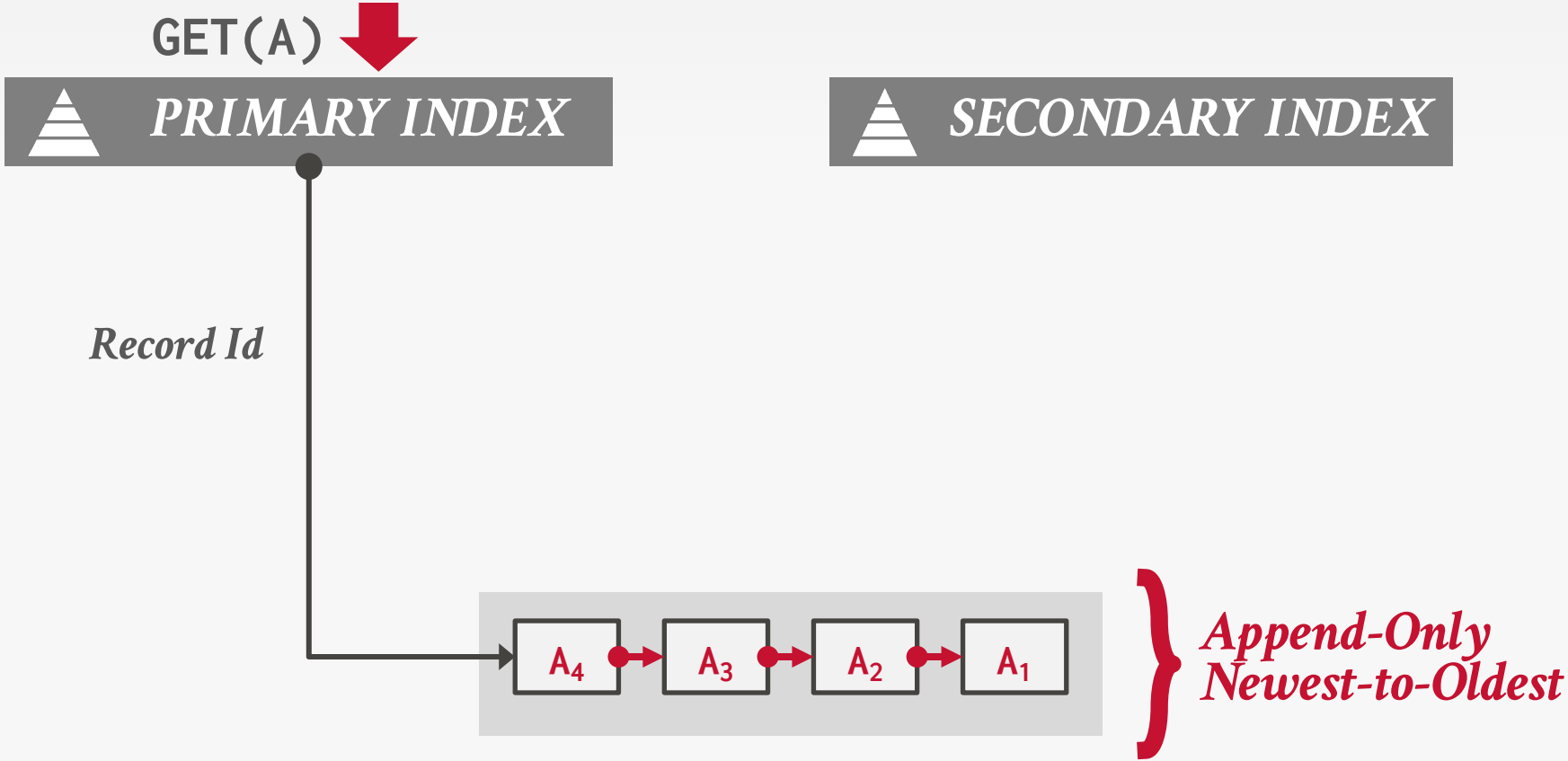


SECONDARY INDEX



Append-Only
Newest-to-Oldest

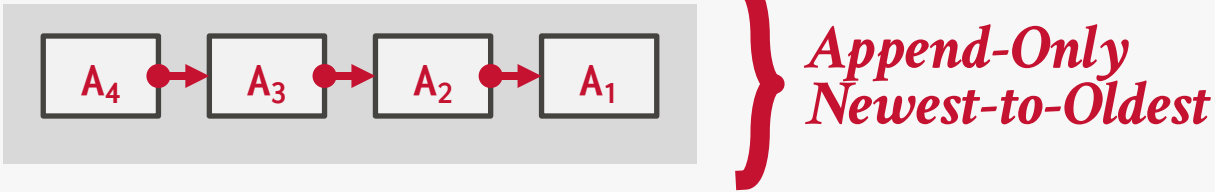
INDEX POINTERS: APPEND-ONLY



INDEX POINTERS: APPEND-ONLY

PRIMARY INDEX

SECONDARY INDEX



INDEX POINTERS: APPEND-ONLY

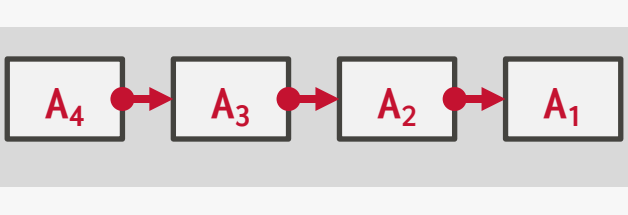
GET(A)



PRIMARY INDEX

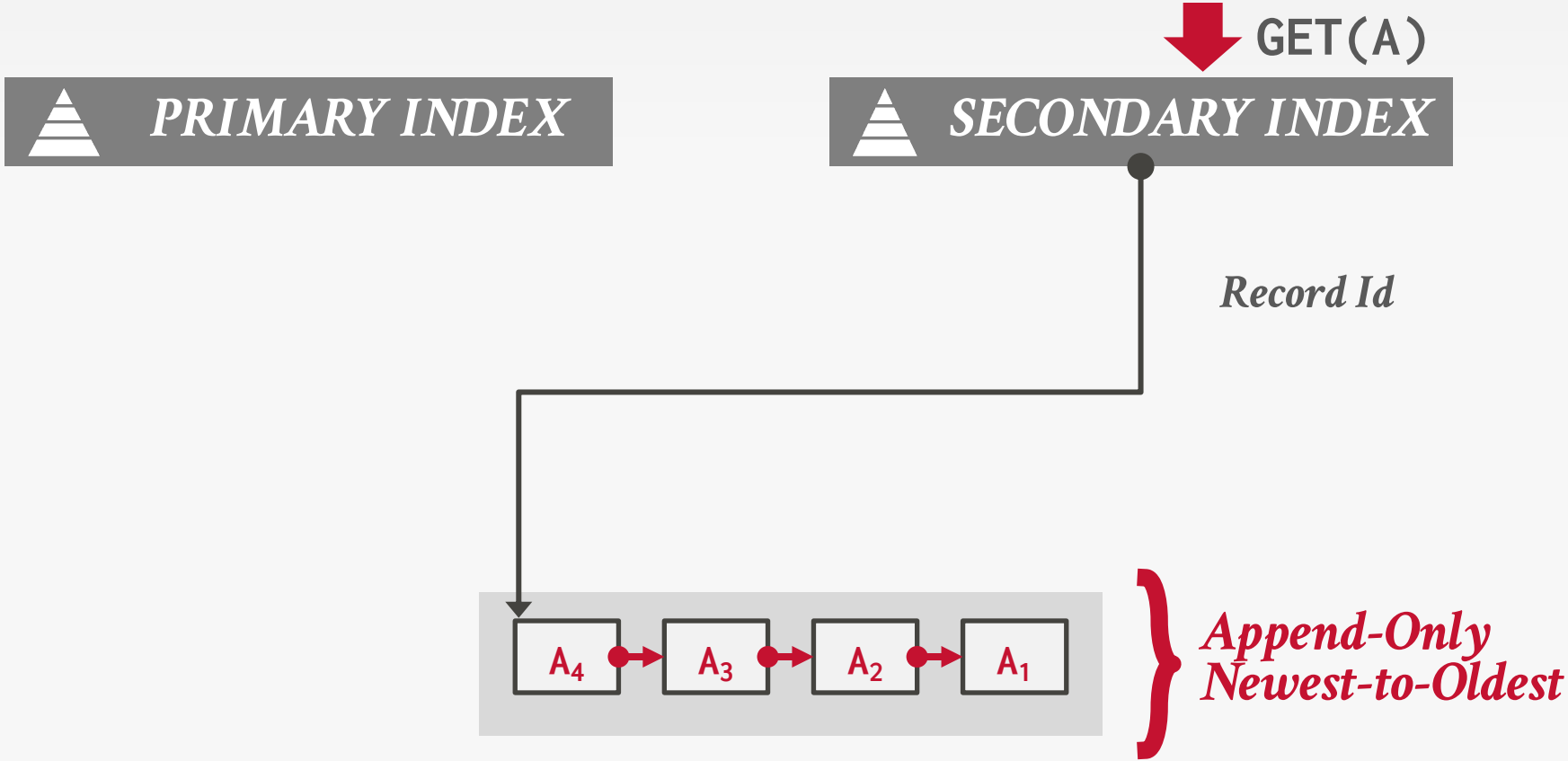


SECONDARY INDEX

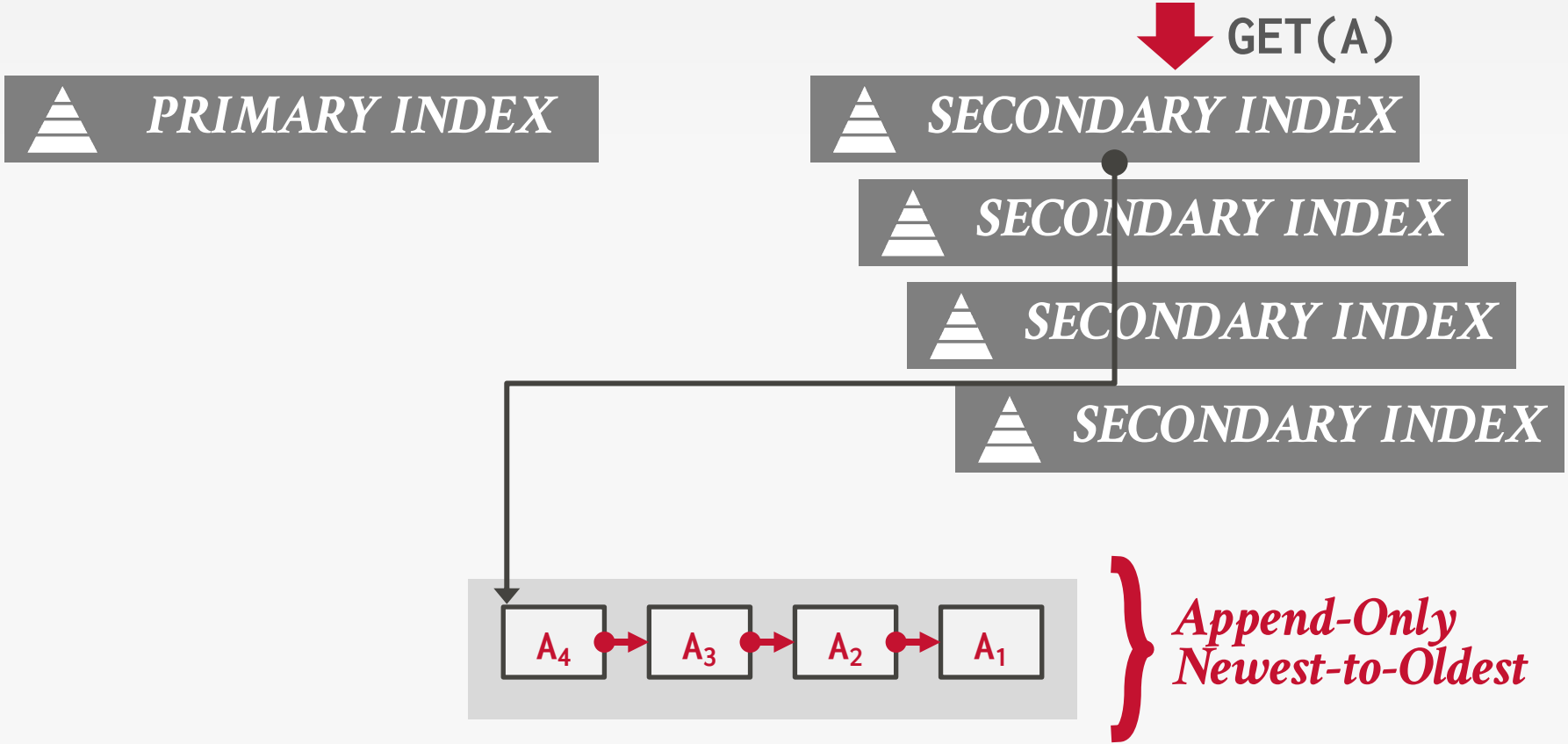


Append-Only
Newest-to-Oldest

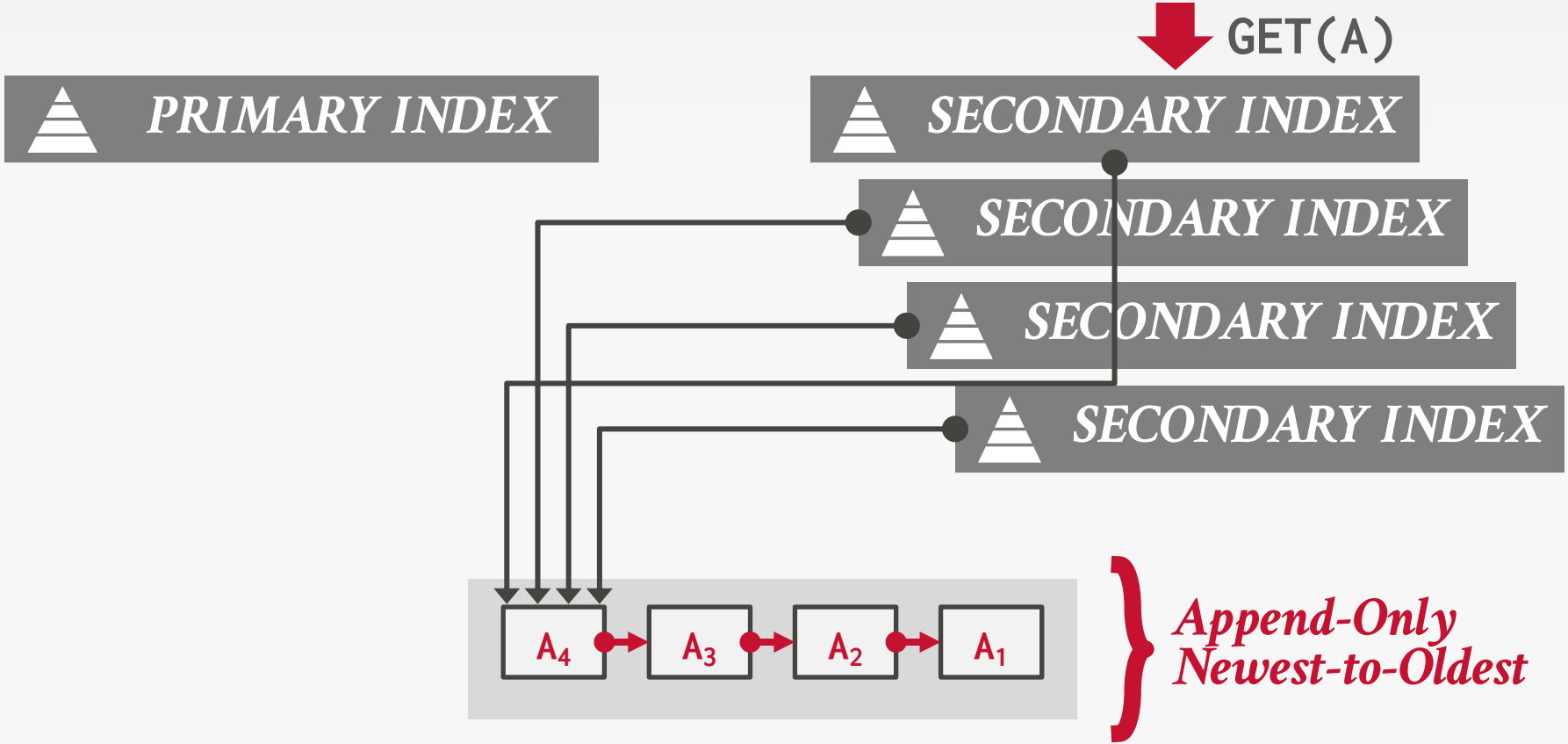
INDEX POINTERS: APPEND-ONLY



INDEX POINTERS: APPEND-ONLY



INDEX POINTERS: APPEND-ONLY



INDEX POINTERS: APPEND-ONLY

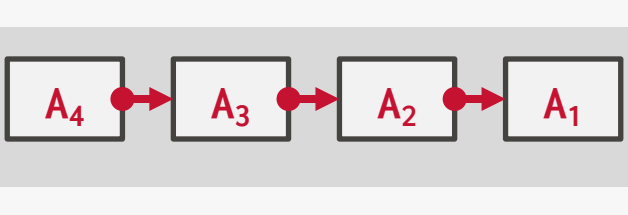
GET(A)



PRIMARY INDEX

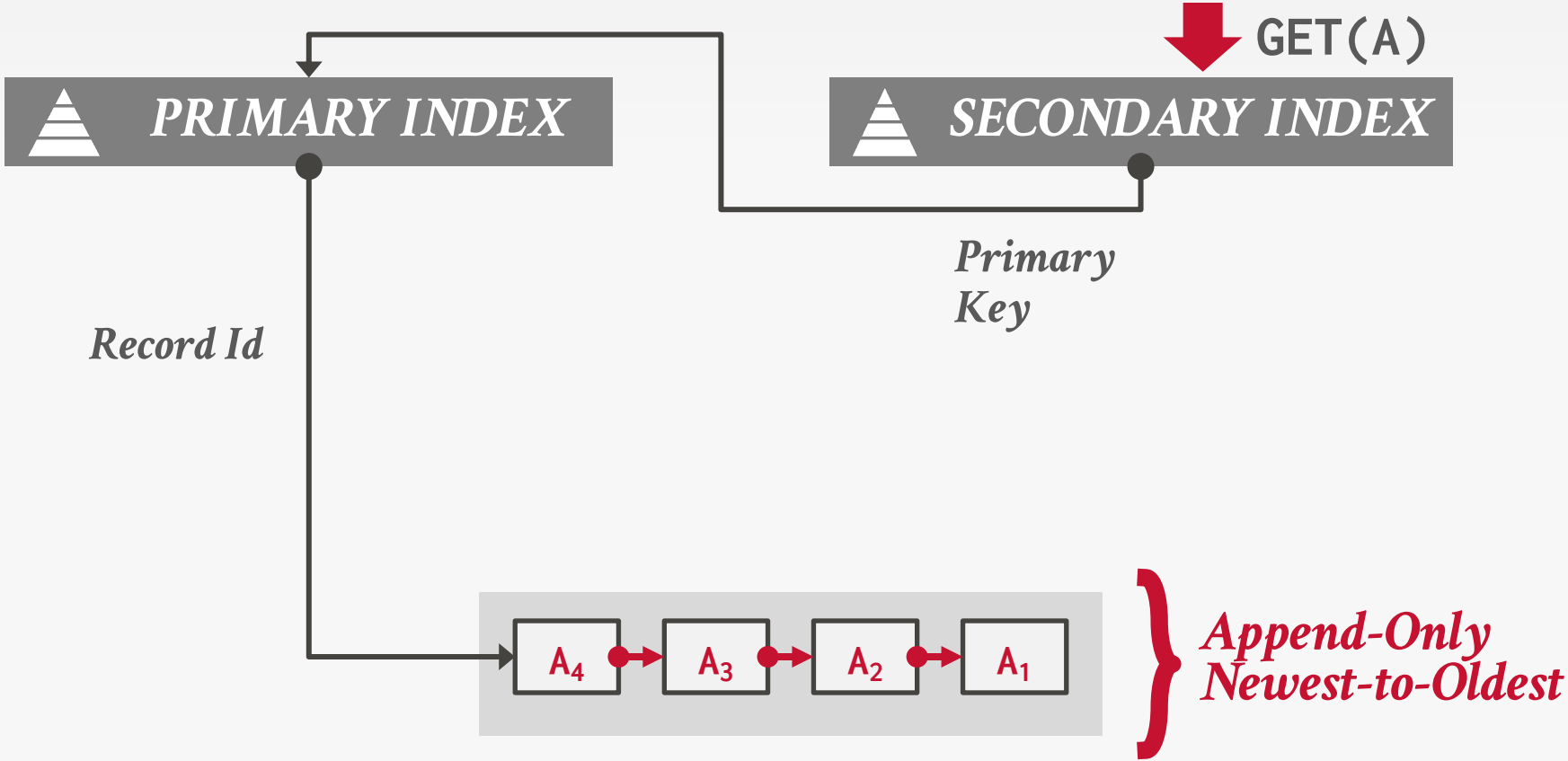


SECONDARY INDEX



Append-Only
Newest-to-Oldest

INDEX POINTERS: APPEND-ONLY



INDEX POINTERS: APPEND-ONLY

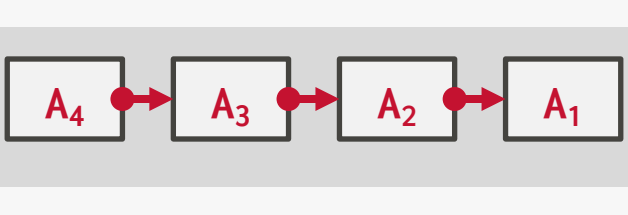
GET(A)



PRIMARY INDEX



SECONDARY INDEX



Append-Only
Newest-to-Oldest

MVCC INDEXES

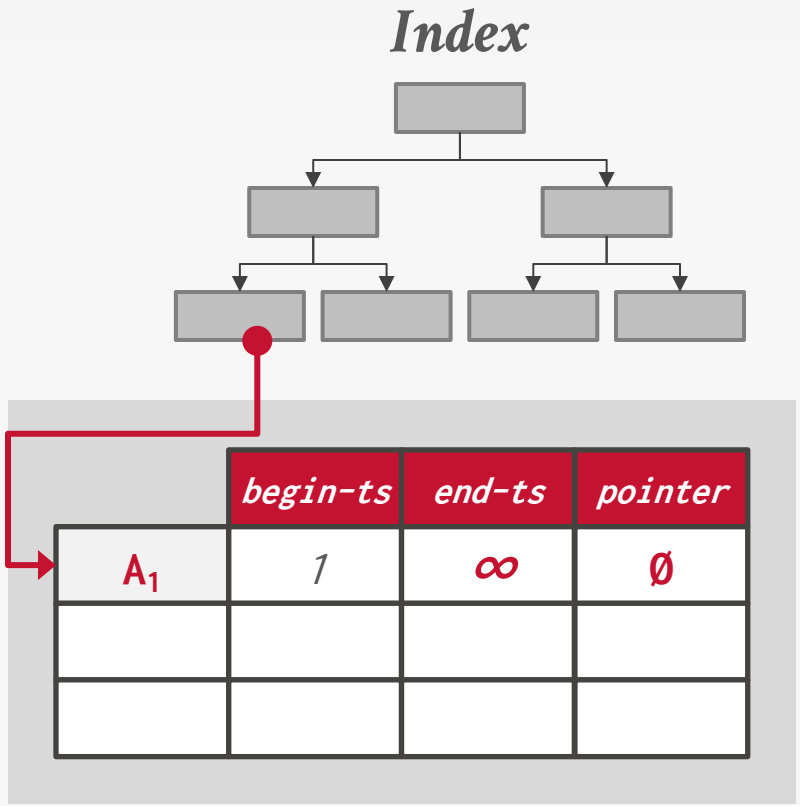
MVCC DBMS indexes (usually) do not store version information about tuples with their keys.

→ Exception: Index-organized tables (e.g., MySQL)

Every index must support duplicate keys from different snapshots:

→ The same key may point to different logical tuples in different snapshots.

MVCC DUPLICATE KEY PROBLEM

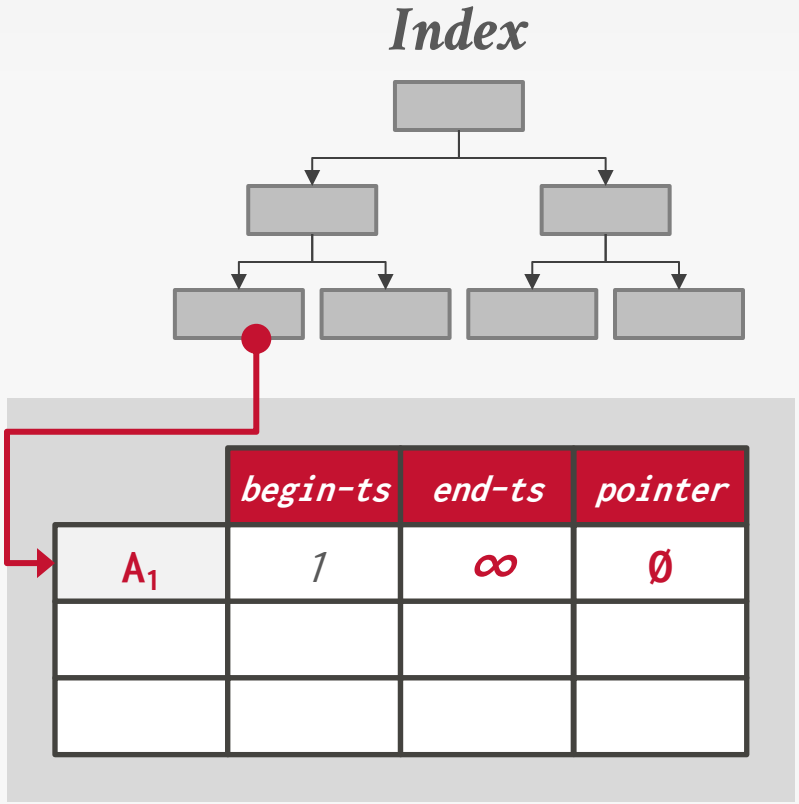


MVCC DUPLICATE KEY PROBLEM

Txn #1

```
BEGIN TS=10
```





MVCC DUPLICATE KEY PROBLEM

Txn #1

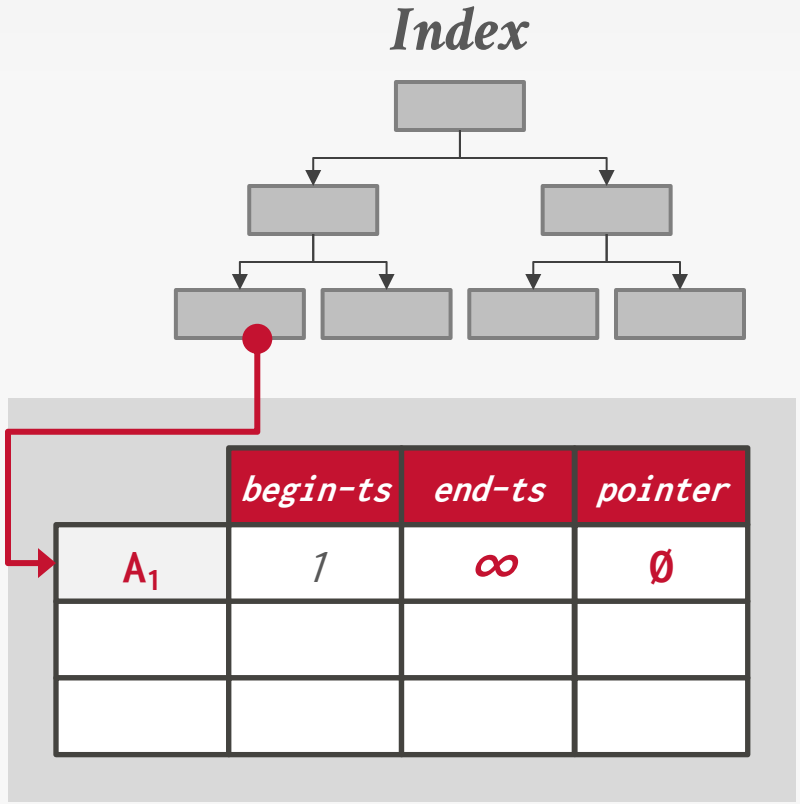
```
BEGIN TS=10
```

READ(A)

Txn #2

```
BEGIN TS=20
```

UPDATE(A)



MVCC DUPLICATE KEY PROBLEM

Txn #1

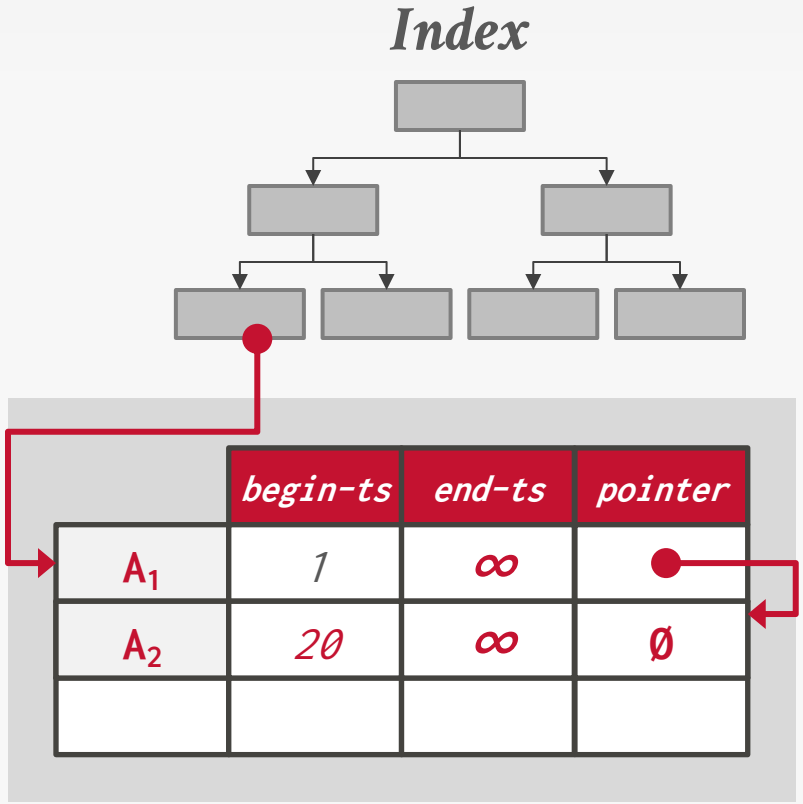
```
BEGIN TS=10
```

READ(A)

Txn #2

```
BEGIN TS=20
```

UPDATE(A)



MVCC DUPLICATE KEY PROBLEM

Txn #1

```
BEGIN TS=10
```

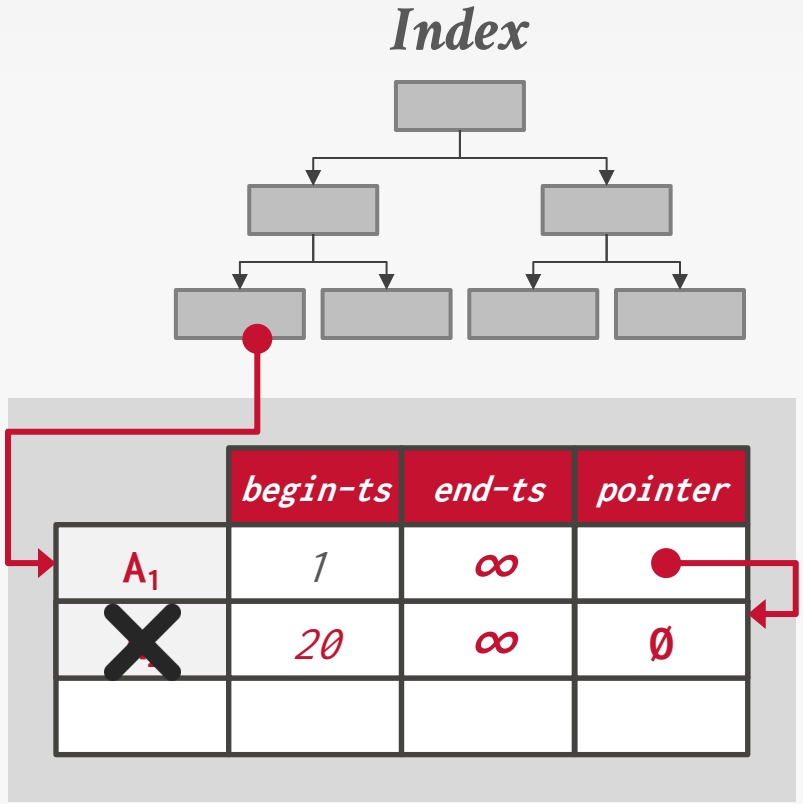
READ(A)

Txn #2

```
BEGIN TS=20
```

UPDATE(A)

DELETE(A)



MVCC DUPLICATE KEY PROBLEM

Txn #1

```
BEGIN TS=10
```

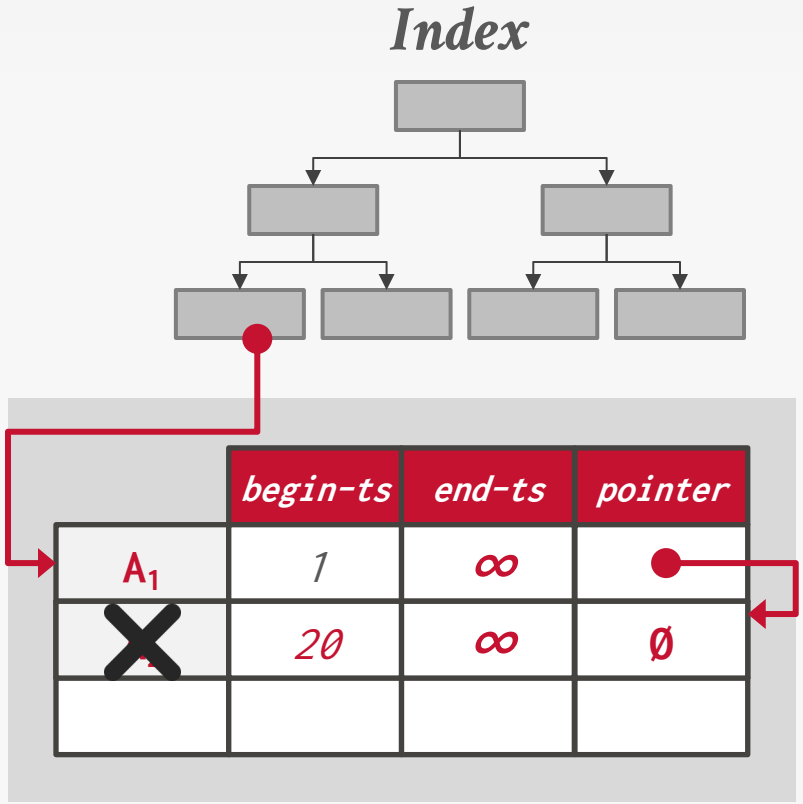
READ(A)

Txn #2

```
BEGIN TS=20
COMMIT TS=25
```

UPDATE(A)

DELETE(A)



MVCC DUPLICATE KEY PROBLEM

Txn #1

```
BEGIN TS=10
```

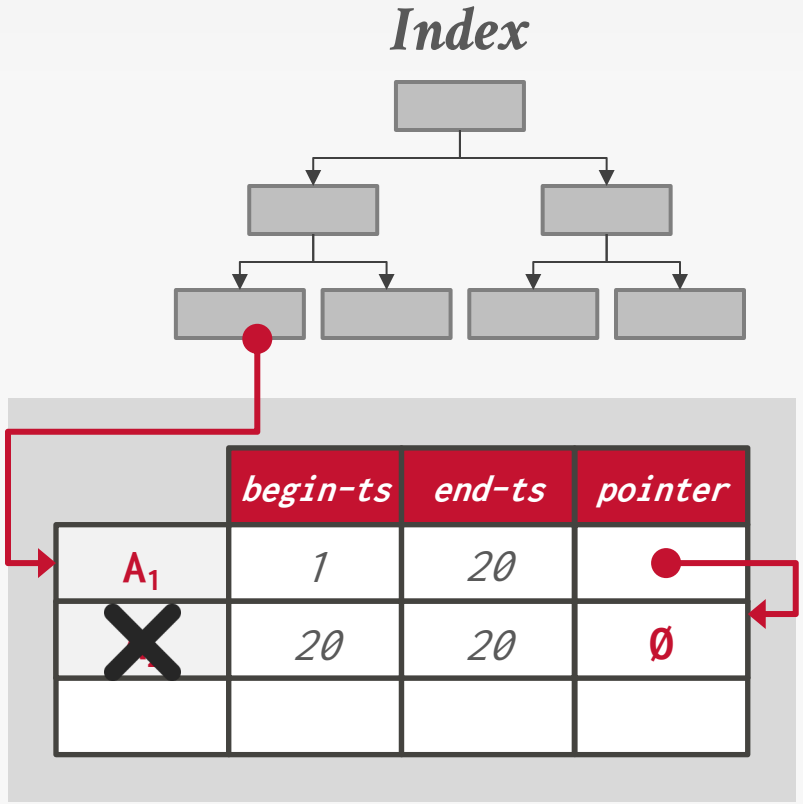
READ(A)

Txn #2

```
BEGIN TS=20
COMMIT TS=25
```

UPDATE(A)

DELETE(A)



MVCC DUPLICATE KEY PROBLEM

Txn #1

```
BEGIN TS=10
```

READ(A)

Txn #2

```
BEGIN TS=20
COMMIT TS=25
```

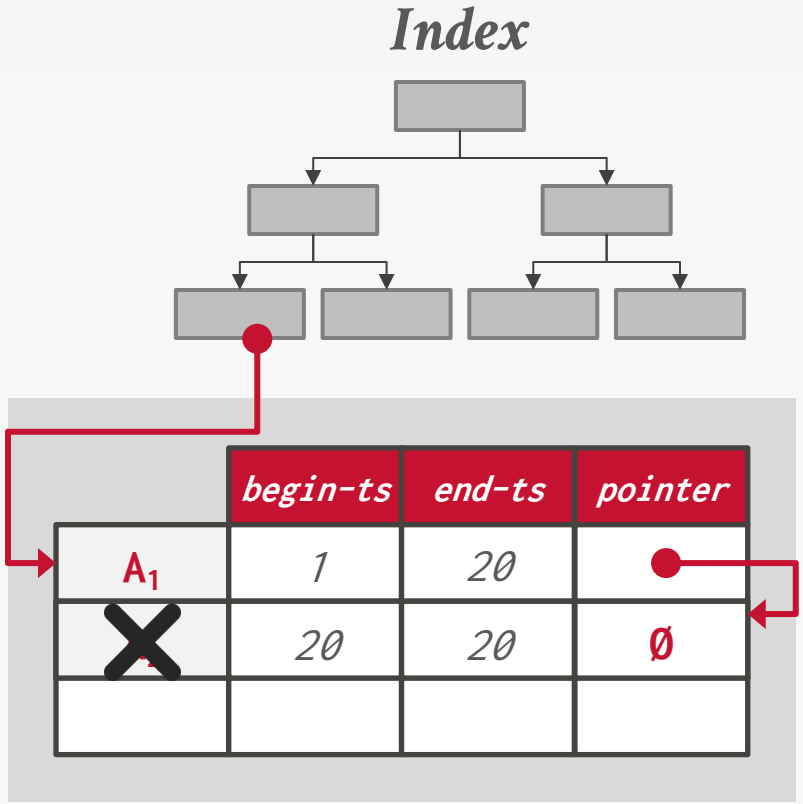
UPDATE(A)

DELETE(A)

Txn #3

```
BEGIN TS=30
```

INSERT(A)



MVCC DUPLICATE KEY PROBLEM

Txn #1

```
BEGIN TS=10
```

READ(A)

Txn #2

```
BEGIN TS=20
COMMIT TS=25
```

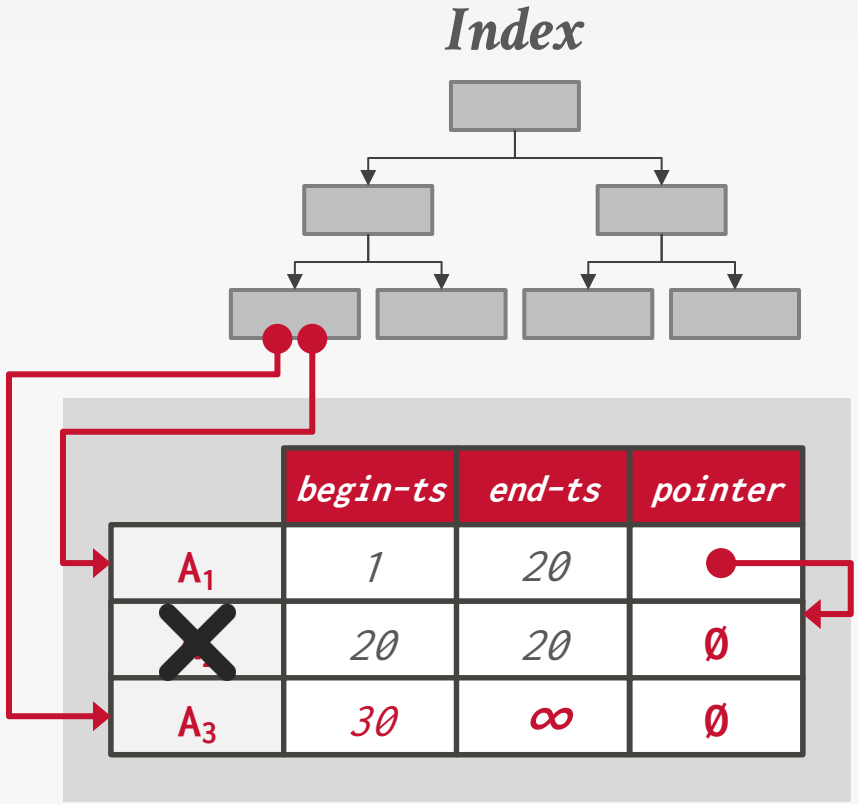
UPDATE(A)

DELETE(A)

Txn #3

```
BEGIN TS=30
```

INSERT(A)



MVCC DUPLICATE KEY PROBLEM

Txn #1

```
BEGIN TS=10
```

READ(A)

READ(A)

Txn #2

```
BEGIN TS=20
COMMIT TS=25
```

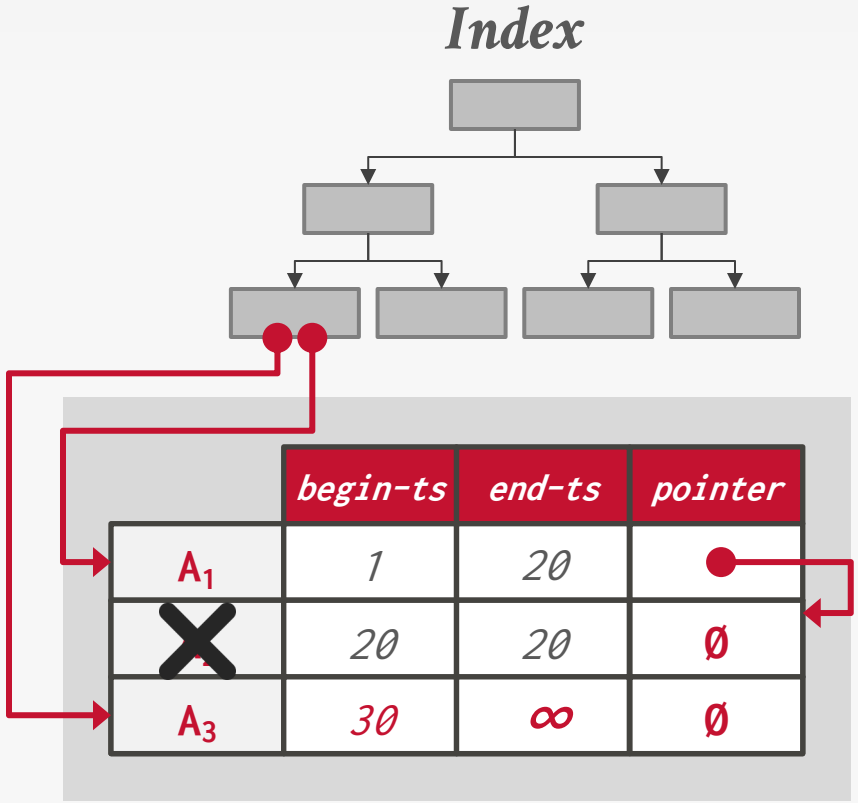
UPDATE(A)

DELETE(A)

Txn #3

```
BEGIN TS=30
```

INSERT(A)



MVCC INDEXES

Each index's underlying data structure must support the storage of non-unique keys.

Use additional execution logic to perform conditional inserts for pkey / unique indexes.
→ Atomically check whether the key exists and then insert.

Workers may get back multiple entries for a single fetch. They then must follow the pointers to find the proper physical version.

MVCC DELETES

The DBMS physically deletes a tuple from the database only when all versions of a logically deleted tuple are not visible.

→ If a tuple is deleted, then there cannot be a new version of that tuple after the newest version.

→ No write-write conflicts / first-writer wins

We need a way to denote that tuple has been logically delete at some point in time.

MVCC DELETES

Approach #1: Deleted Flag

→ Maintain a flag to indicate that the logical tuple has been deleted after the newest physical version.
→ Can either be in tuple header or a separate column.

Approach #2: Tombstone Tuple

→ Create an empty physical version to indicate that a logical tuple is deleted.
→ Use a separate pool for tombstone tuples with only a special bit pattern in version chain pointer to reduce the storage overhead.

MVCC IMPLEMENTATIONS

	Protocol	*Version Storage*	*Garbage Collection*	*Indexes*
Oracle	MV2PL	Delta	Vacuum	Logical
Postgres	MV-2PL/MV-TO	Append-Only	Vacuum	Physical
MySQL-InnoDB	MV-2PL	Delta	Vacuum	Logical
MSSQL Hekaton	MV-OCC	Append-Only	Cooperative	Physical
SingleStore	MV-OCC	Delta	Vacuum	Physical
SAP HANA	MV-2PL	Time-travel	Hybrid	Logical
DuckDB	MV-OCC	Delta	Txn-Level	Logical
HyPer	MV-OCC	Delta	Txn-level	Logical
CockroachDB	MV-2PL	Delta (LSM)	Compaction	Logical

CONCLUSION

MVCC is the widely used scheme in DBMSs.
Even systems that do not support multi-statement txns (e.g., NoSQL) use it.

NEXT CLASS

Logging and recovery!