Carnegie Mellon University

18 Multi-Version Concurrency Control

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

Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

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

Q&A Session on Wed Nov 11th @ 8:00pm
→ https://cmu.zoom.us/j/96880648178?pwd=Z0loZUVORVV1eURFc2R0aDR6QU5udz09

No class on Wed Nov 11th

UPCOMING DATABASE TALKS

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

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

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

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.

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.

MULTI-VERSION CONCURRENCY CONTROL

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

Read-only txns can read a consistent snapshot without acquiring locks.
→ Use timestamps to determine visibility.

Easily support time-travel queries.

MVCC – EXAMPLE #1

Schedule

$TS(T_1)=1$

$TS(T_2)=2$

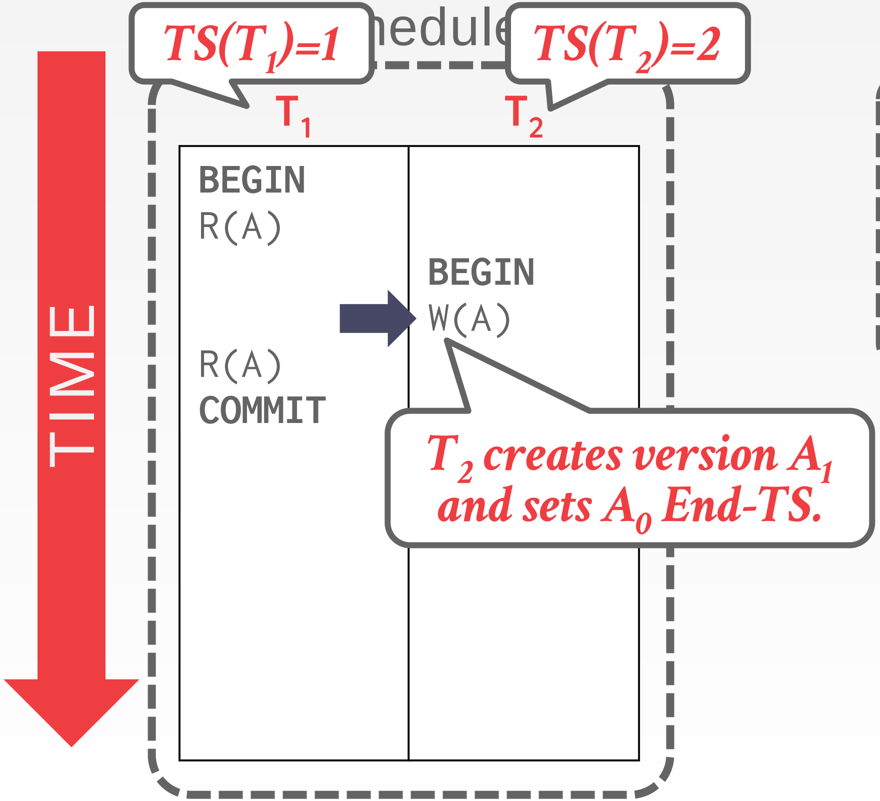
TIME

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

Database

Version	Value	Begin	End
A_0	123	0	-

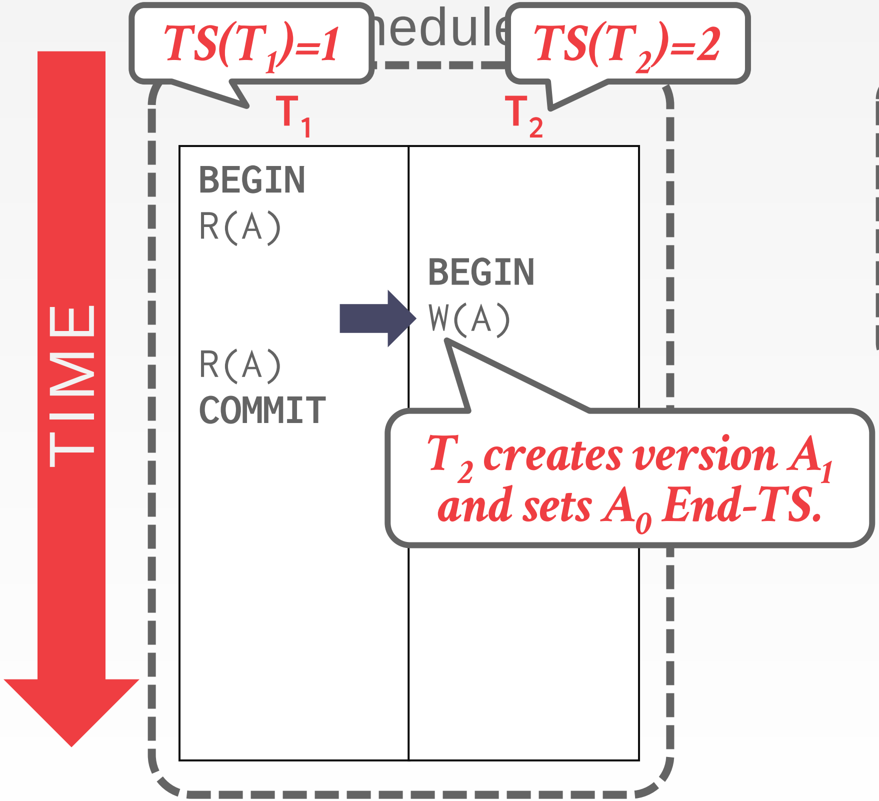
MVCC – EXAMPLE #1



Database

Version	Value	Begin	End
A_0	123	0	-
A_1	456	2	-

MVCC – EXAMPLE #1



Database

Version	Value	Begin	End
A_0	123	0	2
A_1	456	2	-

MVCC – EXAMPLE #1

Schedule

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

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

T_2 creates version A_1 and sets A_0 End-TS.

TIME

Database

Version	Value	Begin	End
A_0	123	0	2
A_1	456	2	-

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active
T_2	2	Active

MVCC – EXAMPLE #1

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

Schedule

TIME

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

Database

Version	Value	Begin	End
A_0	123	0	2
A_1	456	2	-

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active
T_2	2	Active

MVCC – EXAMPLE #1

hedule

TS(T_1)=1

TS(T_2)=2

IME

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

T_1 reads version A_0.

Database

Version	Value	Begin	End
A_0	123	0	2
A_1	456	2	-

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active
T_2	2	Active

MVCC – EXAMPLE #1

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

Schedule

TIME

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

Database

Version	Value	Begin	End
A_0	123	0	2
A_1	456	2	-

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active
T_2	2	Active

MVCC – EXAMPLE #2

Schedule

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

TIME

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

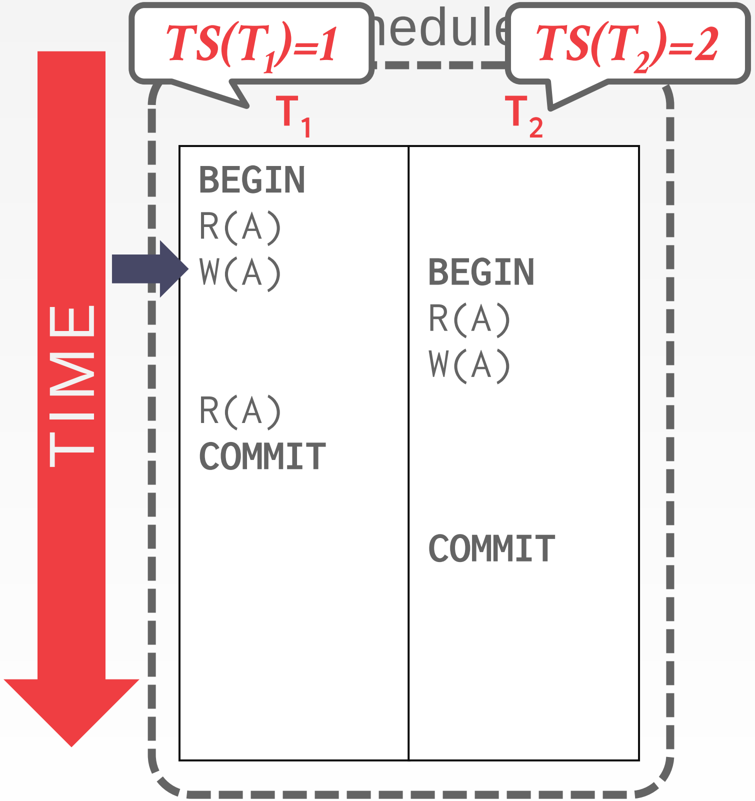
Database

Version	Value	Begin	End
A_0	123	0	

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active

MVCC – EXAMPLE #2



Database

Version	Value	Begin	End
A_0	123	0	
A_1	456	1	-

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active

MVCC – EXAMPLE #2

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

Schedule

TIME

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

Database

Version	Value	Begin	End
A_0	123	0	1
A_1	456	1	-

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active

MVCC – EXAMPLE #2

TS(T1)=1   *TS(T2)=2*

Schedule

TIME

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

T_2 reads version A_0 because T_1 has not committed yet.

Database

Version	Value	Begin	End
A_0	123	0	1
A_1	456	1	-

Txn Status Table

	…tamp	Status
		Active
T_2	2	Active

MVCC – EXAMPLE #2

TIME

Schedule

TS(T_1)=1

TS(T_2)=2

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

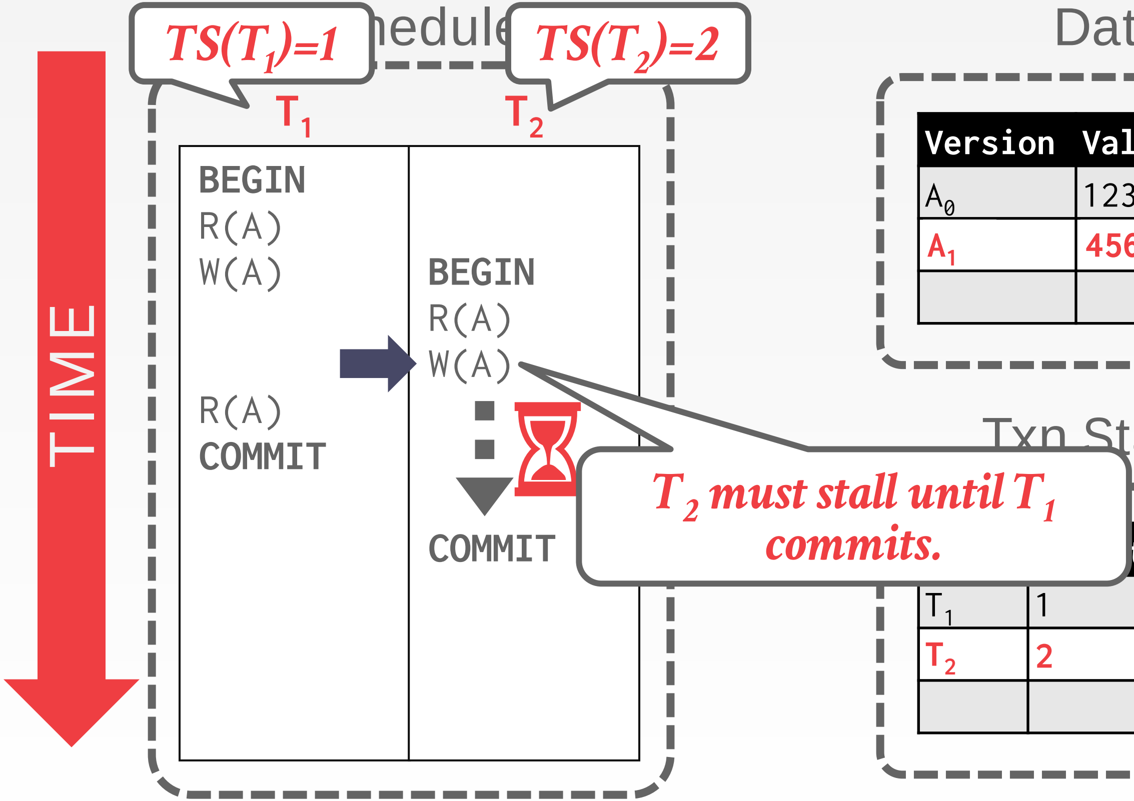
Database

Version	Value	Begin	End
A_0	123	0	1
A_1	456	1	-

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active
T_2	2	Active

MVCC – EXAMPLE #2



Database

Version	Value	Begin	End
A_0	123	0	1
A_1	456	1	-

Txn Status Table

	tamp	Status
T_1	1	Active
T_2	2	Active

MVCC – EXAMPLE #2

Schedule

TIME

$TS(T_1)$=1

$TS(T_2)$=2

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

T_1 reads version A_1 that it wrote earlier.

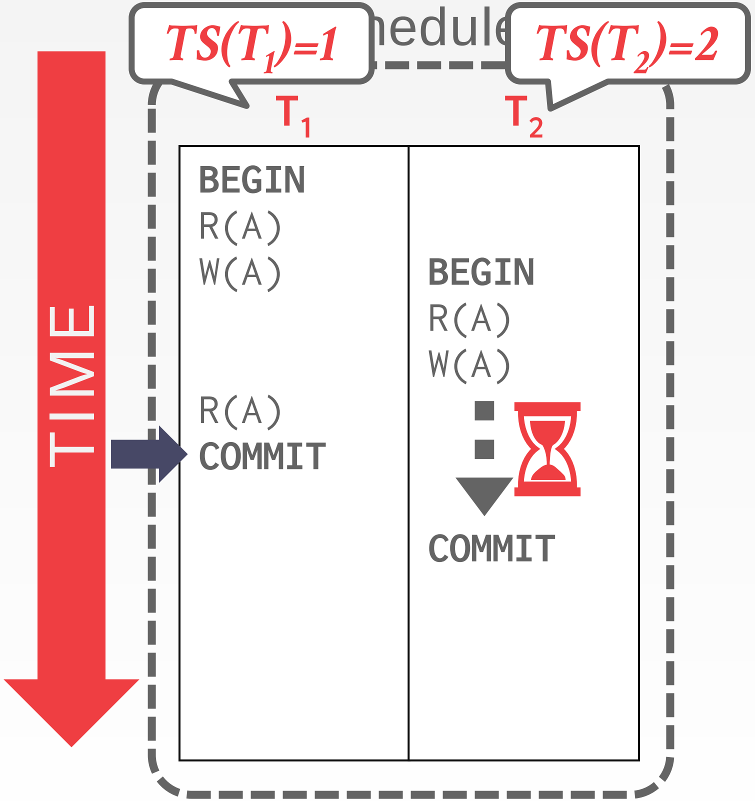
Database

Version	Value	Begin	End
A_0	123	0	1
A_1	456	1	-

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active
T_2	2	Active

MVCC – EXAMPLE #2



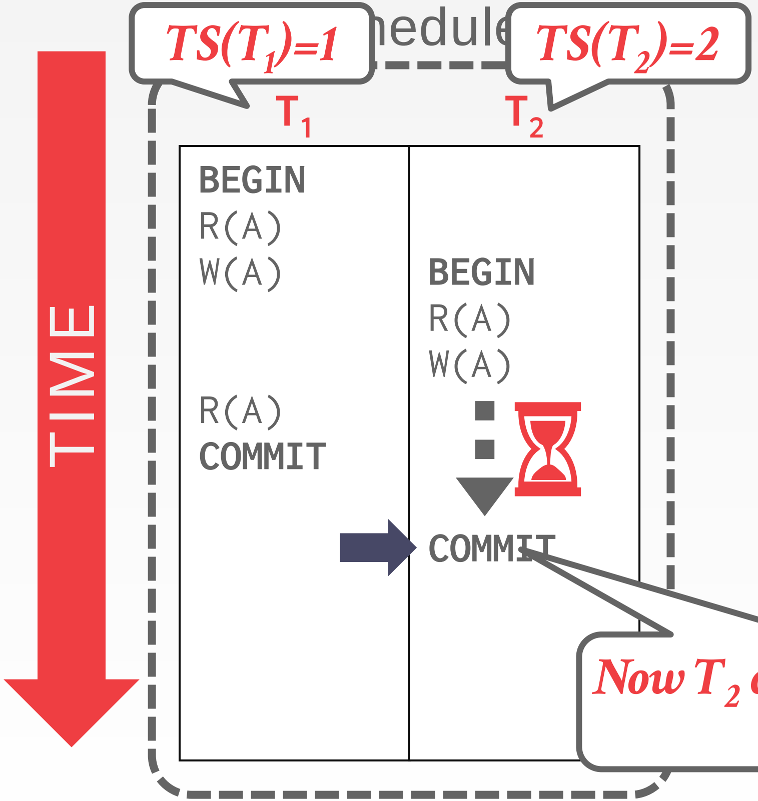
Database

Version	Value	Begin	End
A_0	123	0	1
A_1	456	1	-

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Committed
T_2	2	Active

MVCC – EXAMPLE #2



Now T_2 can create the new version.

Database

Version	Value	Begin	End
A_0	123	0	1
A_1	456	1	2
A_2	789	2	-

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Committed
		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.

MVCC DESIGN DECISIONS

Concurrency Control Protocol

Version Storage

Garbage Collection

Index Management

Deletes

CONCURRENCY CONTROL PROTOCOL

Approach #1: Timestamp Ordering
→ Assign txns timestamps that determine serial order.

Approach #2: Optimistic Concurrency Control
→ Three-phase protocol from last class.
→ Use private workspace for new versions.

Approach #3: Two-Phase Locking
→ Txns acquire appropriate lock on physical version before they can read/write a logical tuple.

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.

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

	VALUE	POINTER
A_0	$111	•
A_1	$222	Ø
B_1	$10	Ø

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

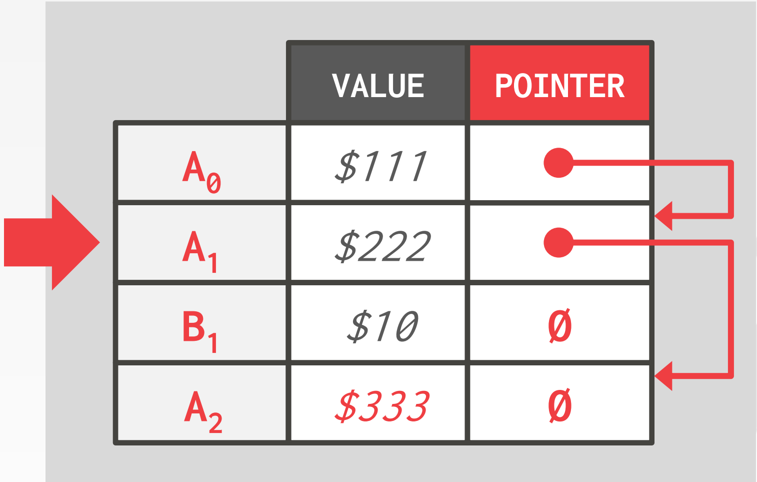
	VALUE	POINTER
A_0	$111	
A_1	$222	Ø
B_1	$10	Ø
A_2	$333	Ø

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



VERSION CHAIN ORDERING

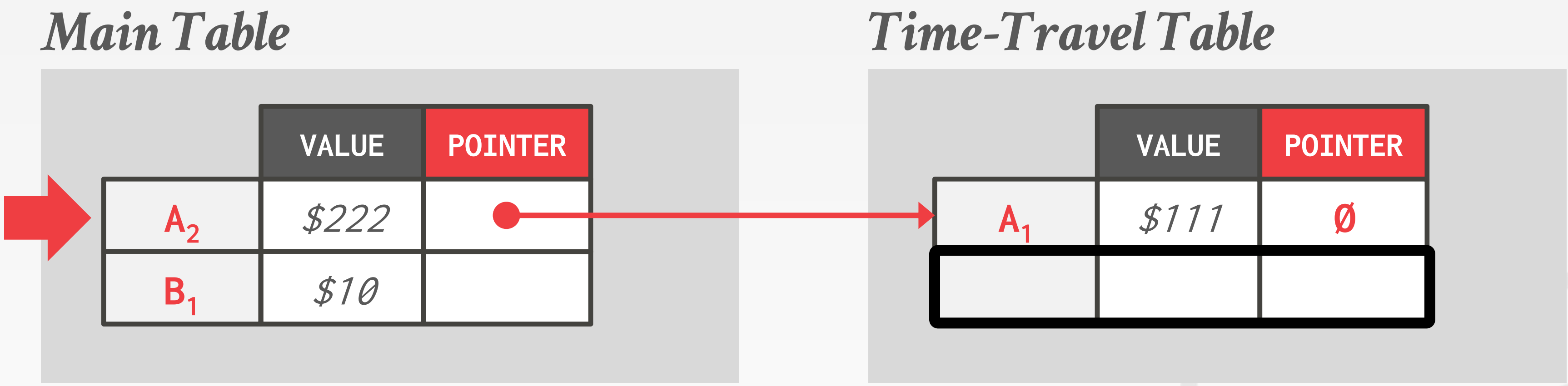
Approach #1: Oldest-to-Newest (O2N)

→ 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.

TIME-TRAVEL STORAGE



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

TIME-TRAVEL STORAGE

Main Table

	VALUE	POINTER
A_2	$222	•
B_1	$10	

Time-Travel Table

	VALUE	POINTER
A_1	$111	Ø
A_2	$222	•

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

TIME-TRAVEL STORAGE

Main Table

	VALUE	POINTER
A_2	$222	
B_1	$10	

Time-Travel Table

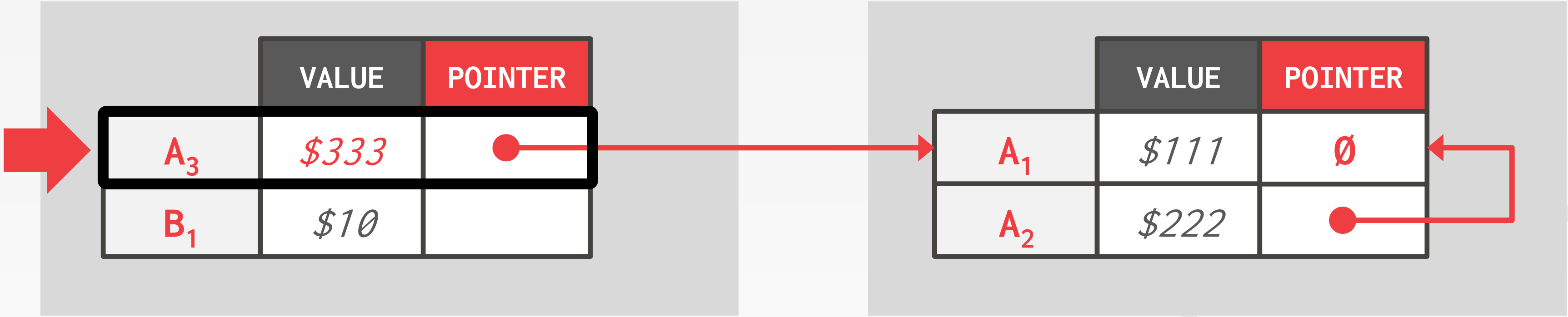
	VALUE	POINTER
A_1	$111	Ø
A_2	$222	

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

	VALUE	POINTER
A_3	$333	•
B_1	$10	

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

Time-Travel Table

	VALUE	POINTER
A_1	$111	Ø
A_2	$222	•

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

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

DELTA STORAGE

Main Table

	VALUE	POINTER
A_2	$222	●
B_1	$10	

Delta Storage Segment

	DELTA	POINTER
A_1	(VALUE→$111)	Ø

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

DELTA STORAGE

Main Table

	VALUE	POINTER
A_2	*$222*	● → A_1
B_1	*$10*	

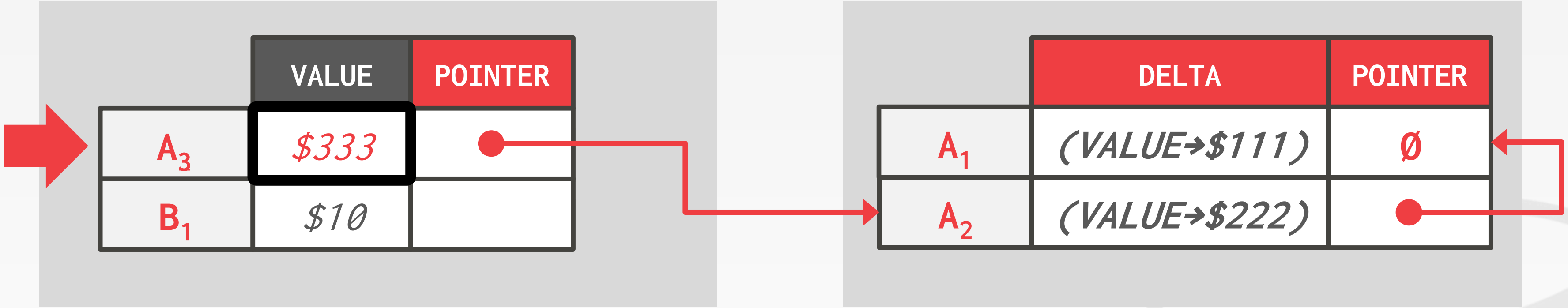
Delta Storage Segment

	DELTA	POINTER
A_1	*(VALUE→$111)*	Ø
A_2	*(VALUE→$222)*	● → A_1

On every update, copy only the 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 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

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

Thread #1

TS(T_1)=12

Thread #2

TS(T_2)=25

VERSION	BEGIN	END
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

Thread #1

$TS(T_1)=12$

Thread #2

$TS(T_2)=25$

Vacuum

VERSION	BEGIN	END
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

Thread #1

$TS(T_1)=12$

Thread #2

$TS(T_2)=25$

Vacuum

VERSION	BEGIN	END
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

Thread #1

TS(T_1)=12

Thread #2

TS(T_2)=25

Vacuum

VERSION	BEGIN	END
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

Thread #1

$TS(T_1)=12$

Thread #2

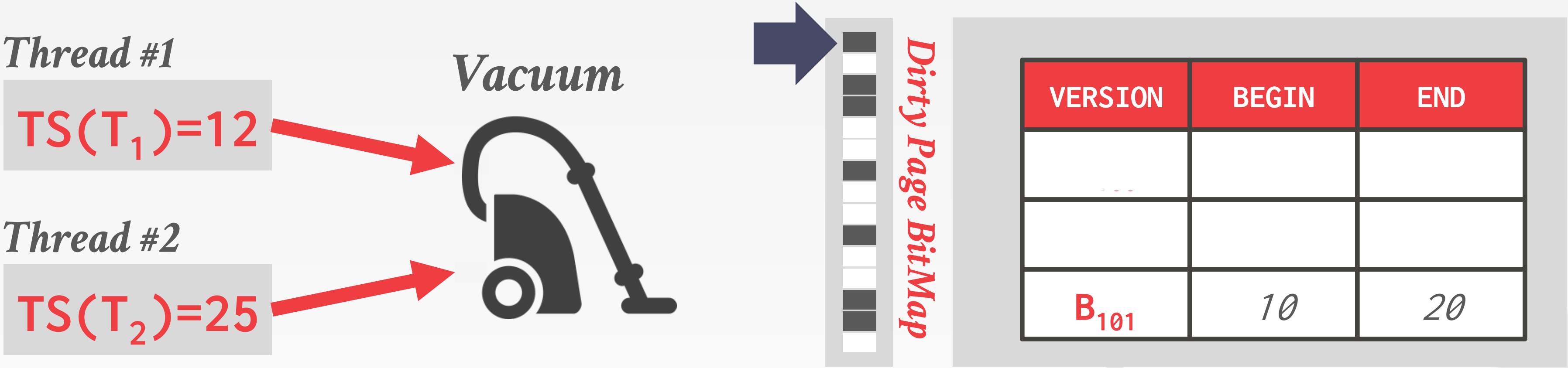
$TS(T_2)=25$

Vacuum

VERSION	BEGIN	END
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

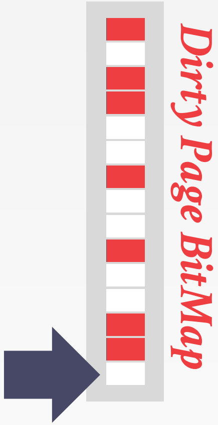
Thread #1

$TS(T_1)=12$

Thread #2

$TS(T_2)=25$

Vacuum



VERSION	BEGIN	END
B_{101}	10	20

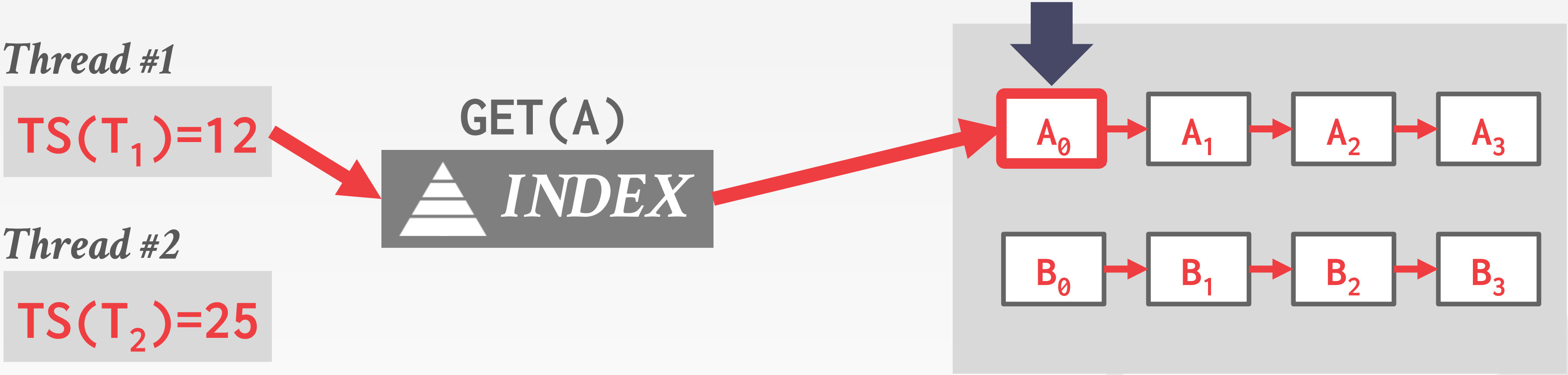
Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

TUPLE-LEVEL GC

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

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

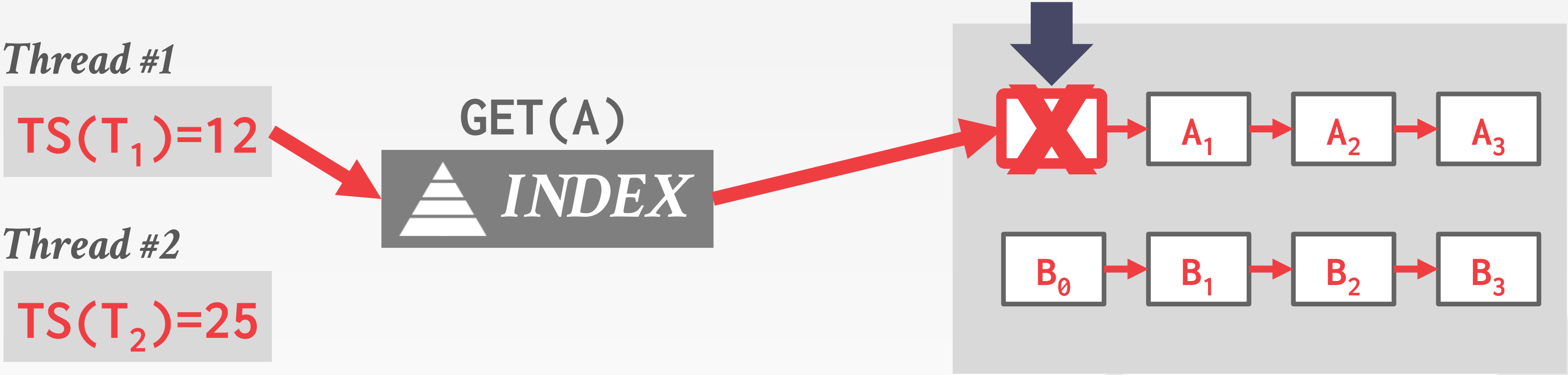
TUPLE-LEVEL GC



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.

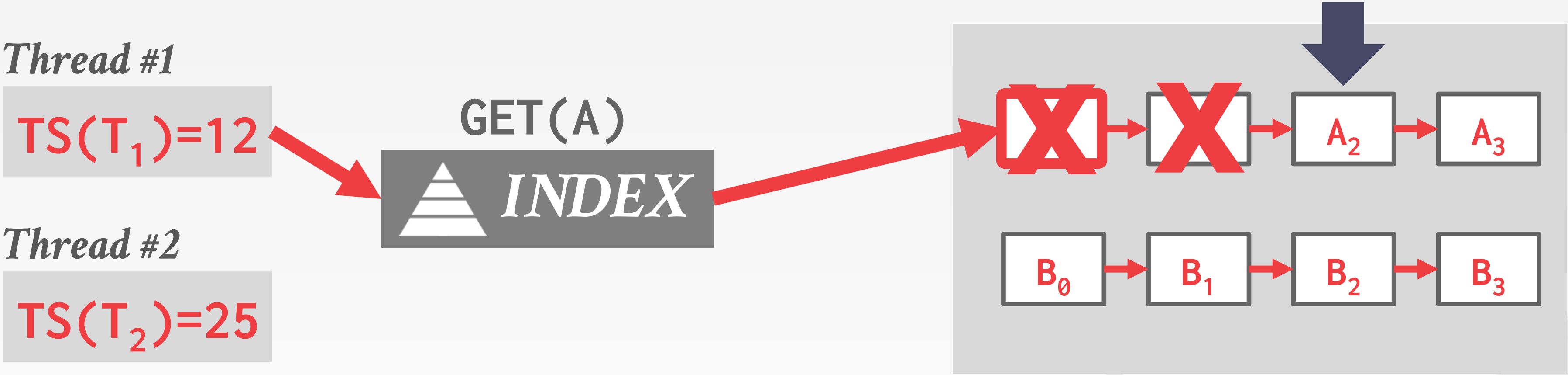
TUPLE-LEVEL GC



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.

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

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…

UBER Engineering

JOIN THE TEAM

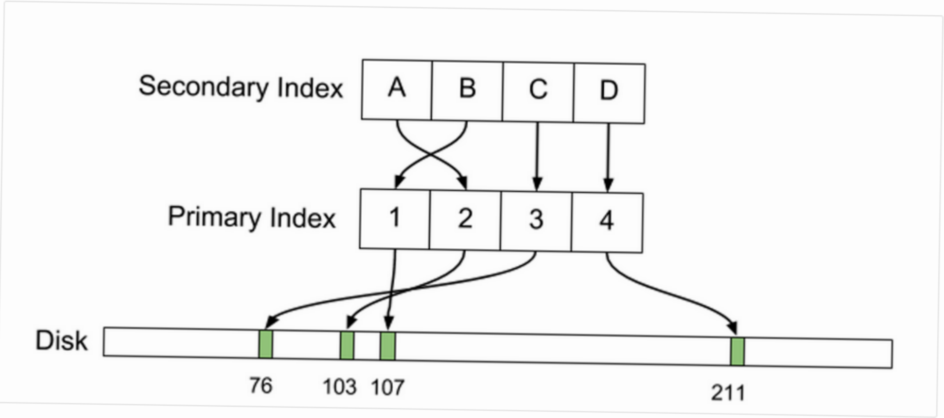
MEET THE PEOPLE

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

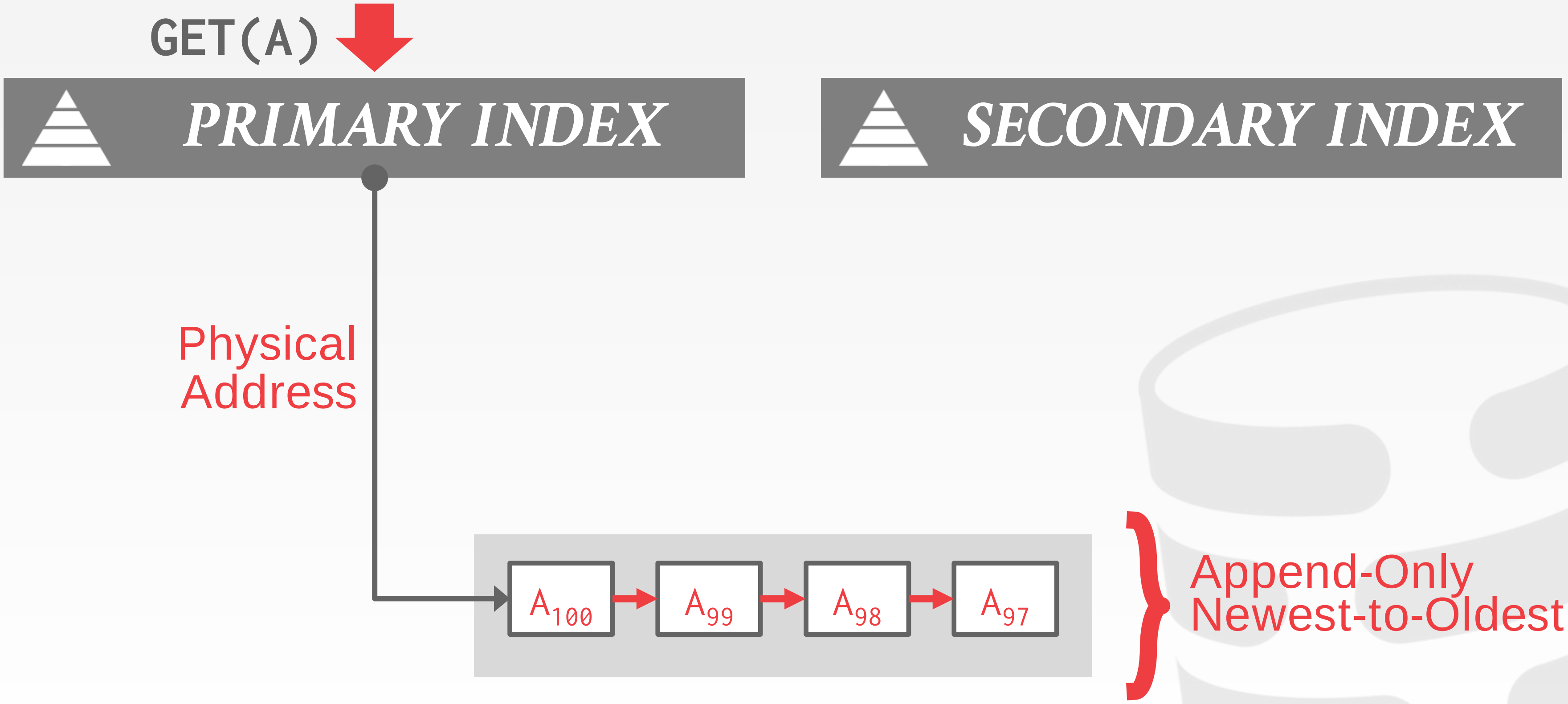
→ Use the physical address to the version chain head.

INDEX POINTERS

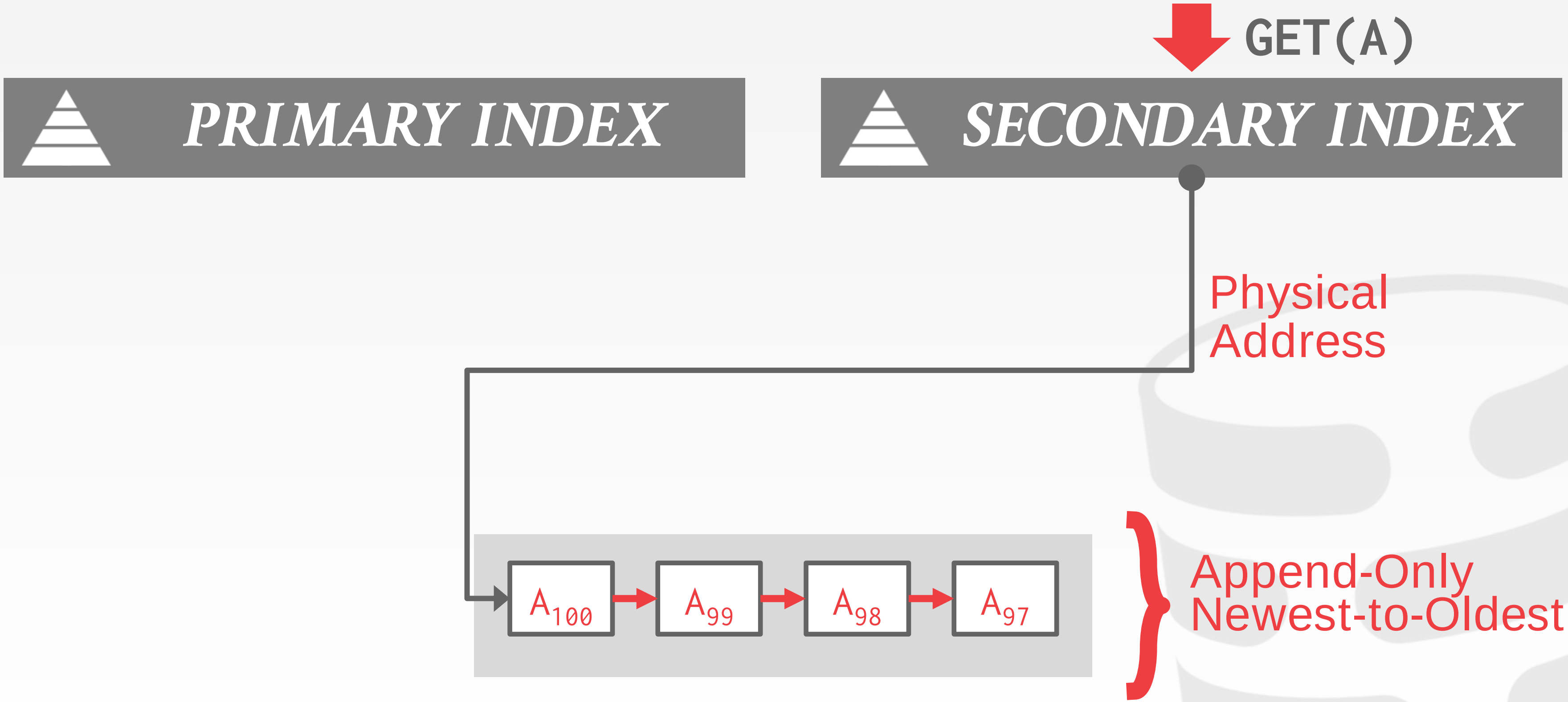
PRIMARY INDEX

SECONDARY INDEX

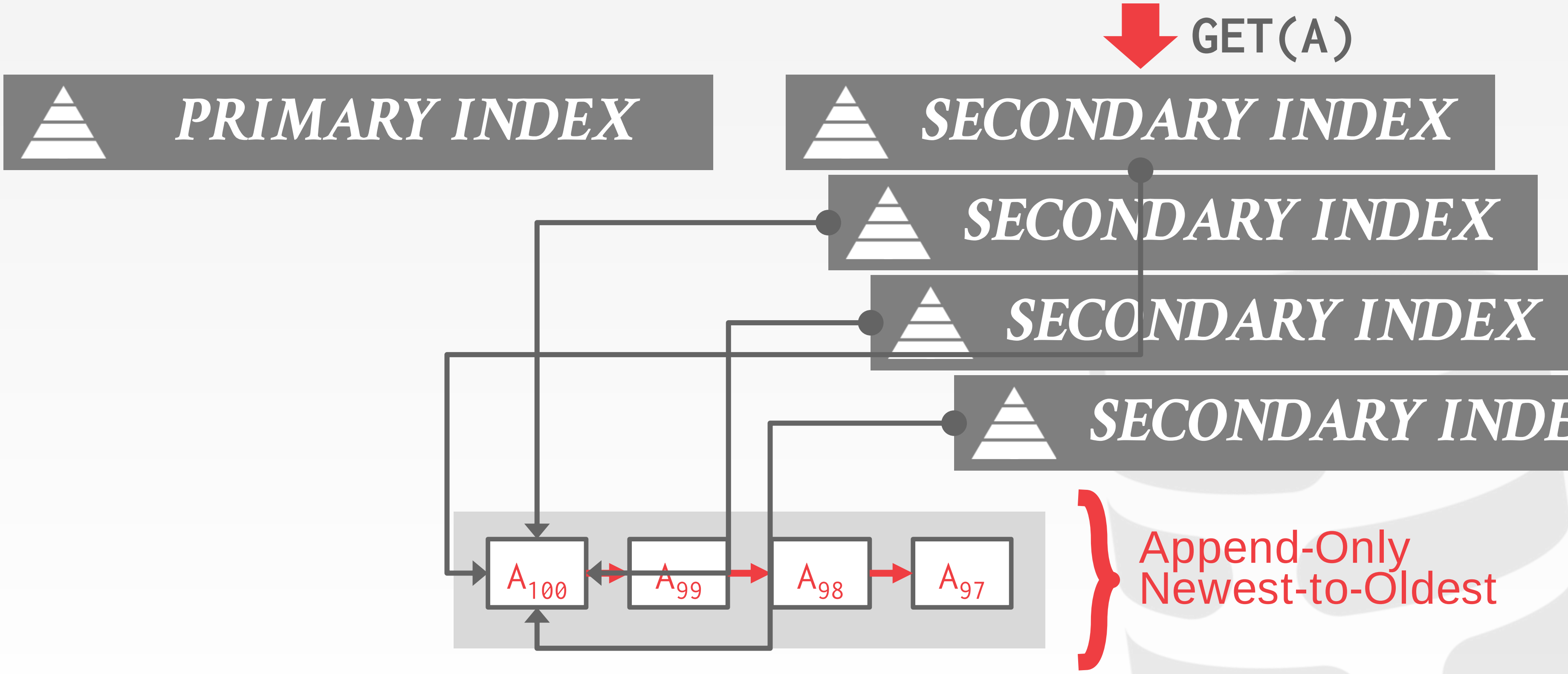
INDEX POINTERS



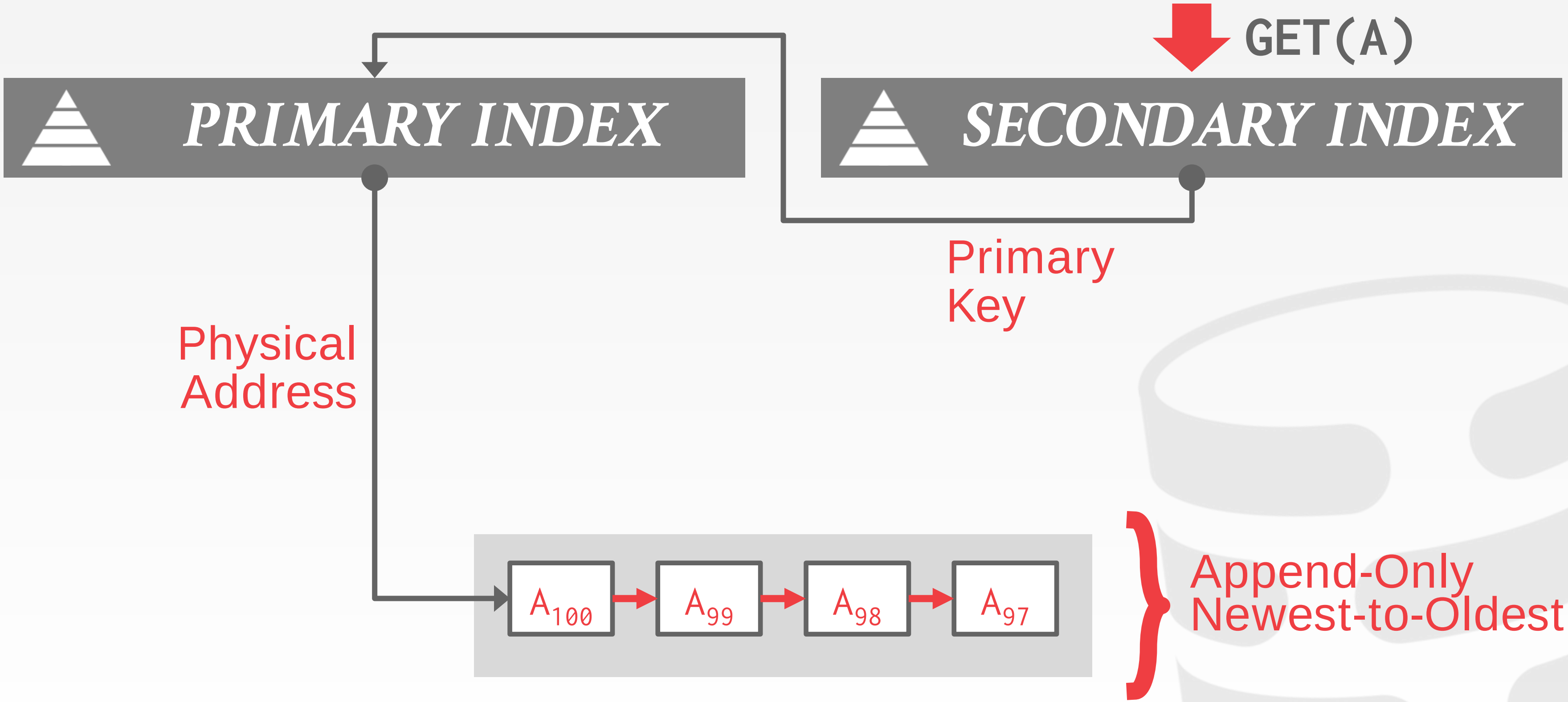
INDEX POINTERS



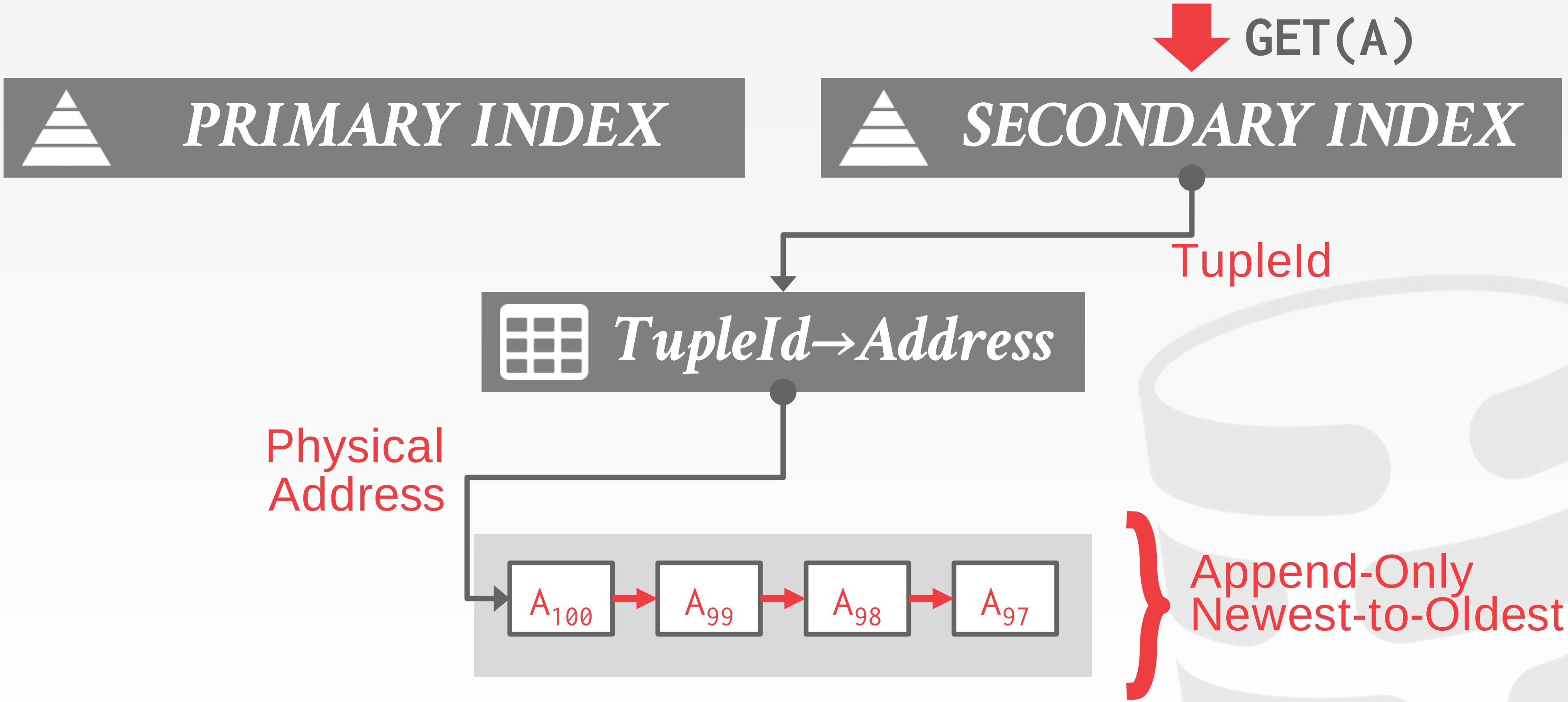
INDEX POINTERS



INDEX POINTERS



INDEX POINTERS



MVCC INDEXES

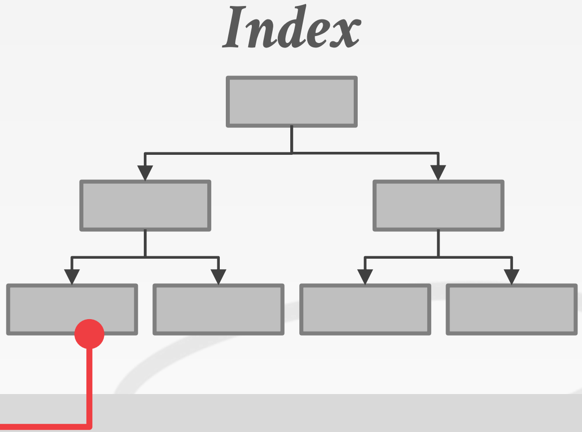
MVCC DBMS indexes (usually) do not store version information about tuples with their keys.

→ 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

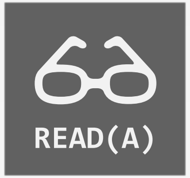


VERSION	BEGIN-TS	END-TS	POINTER
A_1	1	∞	Ø

MVCC DUPLICATE KEY PROBLEM

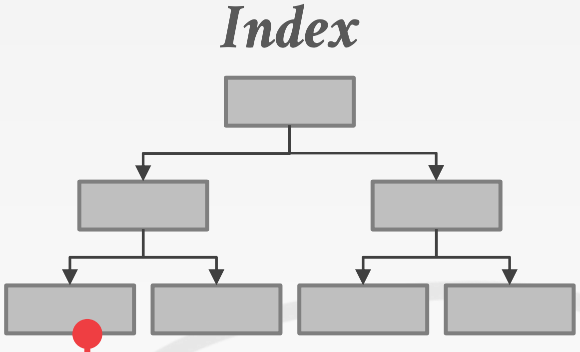
Thread #1

Begin @ 10



Thread #2

Begin @ 20



VERSION	BEGIN-TS	END-TS	POINTER
A_1	1	20	•
A_2	20	∞	Ø

MVCC DUPLICATE KEY PROBLEM

Thread #1

Begin @ 10

READ(A)

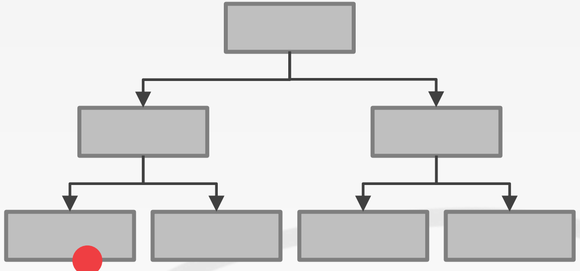
Thread #2

Begin @ 20

UPDATE(A)

DELETE(A)

Index



VERSION	BEGIN-TS	END-TS	POINTER
A_1	1	20	•
~~A_2~~	20	∞	Ø

MVCC DUPLICATE KEY PROBLEM

Thread #1

Begin @ 10

READ(A)

Thread #2

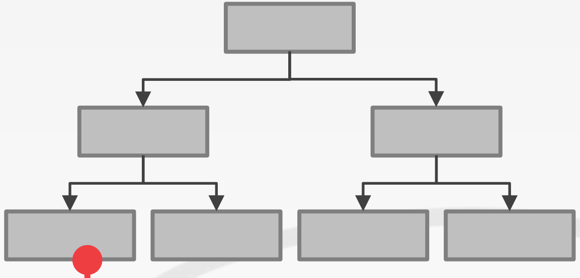
Begin @ 20

Commit @ 25

UPDATE(A)

DELETE(A)

Index



VERSION	BEGIN-TS	END-TS	POINTER
A_1	1	20	●
	20	∞	Ø

MVCC DUPLICATE KEY PROBLEM

Thread #1

Begin @ 10

READ(A)

Thread #2

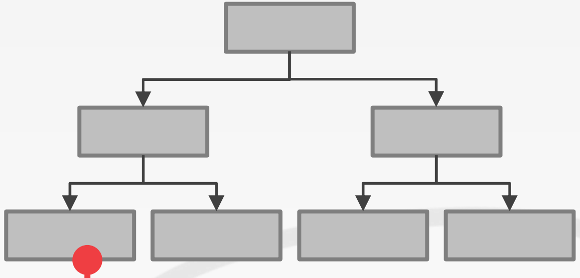
Begin @ 20

Commit @ 25

UPDATE(A)

DELETE(A)

Index



VERSION	BEGIN-TS	END-TS	POINTER
A_1	1	25	•
~~A_2~~	25	25	Ø

MVCC DUPLICATE KEY PROBLEM

Thread #1

Begin @ 10

READ(A)

Thread #2

Begin @ 20
Commit @ 25

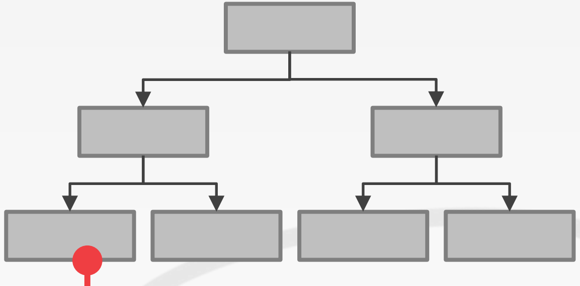
UPDATE(A) DELETE(A)

Thread #3

Begin @ 30

INSERT(A)

Index



VERSION	BEGIN-TS	END-TS	POINTER
A_1	1	25	•
X	25	25	Ø

MVCC DUPLICATE KEY PROBLEM

Thread #1

Begin @ 10

READ(A)

Thread #2

Begin @ 20
Commit @ 25

UPDATE(A)

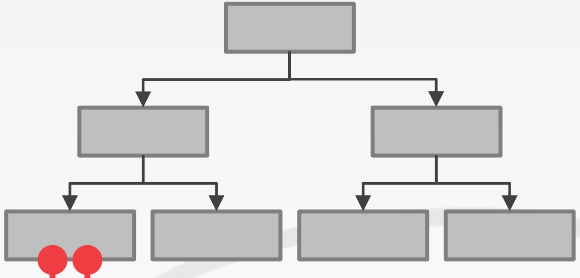
DELETE(A)

Thread #3

Begin @ 30

INSERT(A)

Index



VERSION	BEGIN-TS	END-TS	POINTER
A_1	1	25	•
X	25	25	Ø
A_1	30	∞	Ø

MVCC DUPLICATE KEY PROBLEM

Thread #1

Begin @ 10

READ(A) READ(A)

Thread #2

Begin @ 20
Commit @ 25

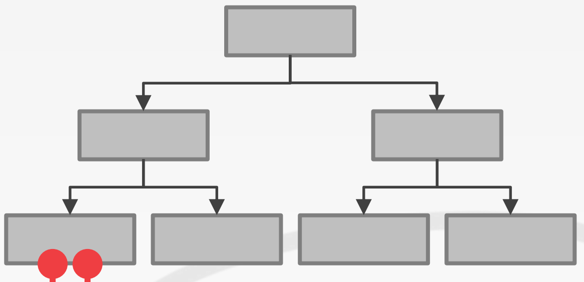
UPDATE(A) DELETE(A)

Thread #3

Begin @ 30

INSERT(A)

Index



VERSION	BEGIN-TS	END-TS	POINTER
A_1	1	25	•
X	25	25	Ø
A_1	30	∞	Ø

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
HYRISE	MV-OCC	Append-Only	–	Physical
Hekaton	MV-OCC	Append-Only	Cooperative	Physical
MemSQL	MV-OCC	Append-Only	Vacuum	Physical
SAP HANA	MV-2PL	Time-travel	Hybrid	Logical
NuoDB	MV-2PL	Append-Only	Vacuum	Logical
HyPer	MV-OCC	Delta	Txn-level	Logical
CMU's TBD	MV-OCC	Delta	Txn-level	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

No class on Wed November 11^{th}