Carnegie Mellon University

Lecture #05

ADVANCED DATABASE SYSTEMS

Multi-Version Concurrency Control (Part I)

@Andy_Pavlo // 15-721 // Spring 2018

TODAY'S AGENDA

Compare-and-Swap (CAS)

Isolation Levels

MVCC Design Decisions

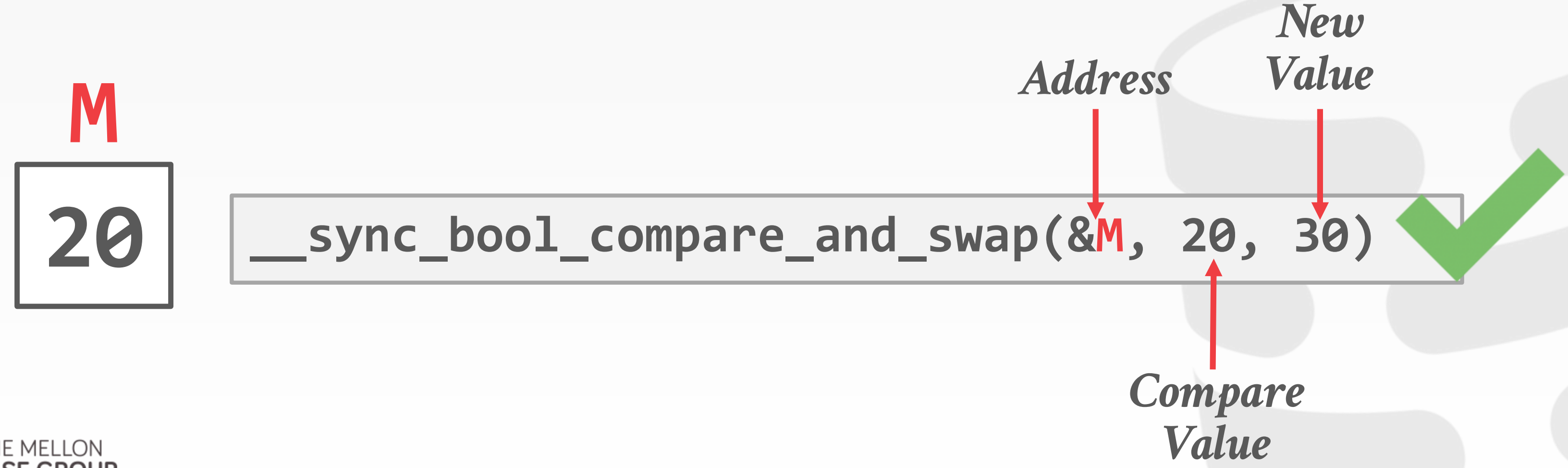
Project #2

COMPARE-AND-SWAP

Atomic instruction that compares contents of a memory location **M** to a given value **V**

→ If values are equal, installs new given value **V'** in **M**

→ Otherwise operation fails



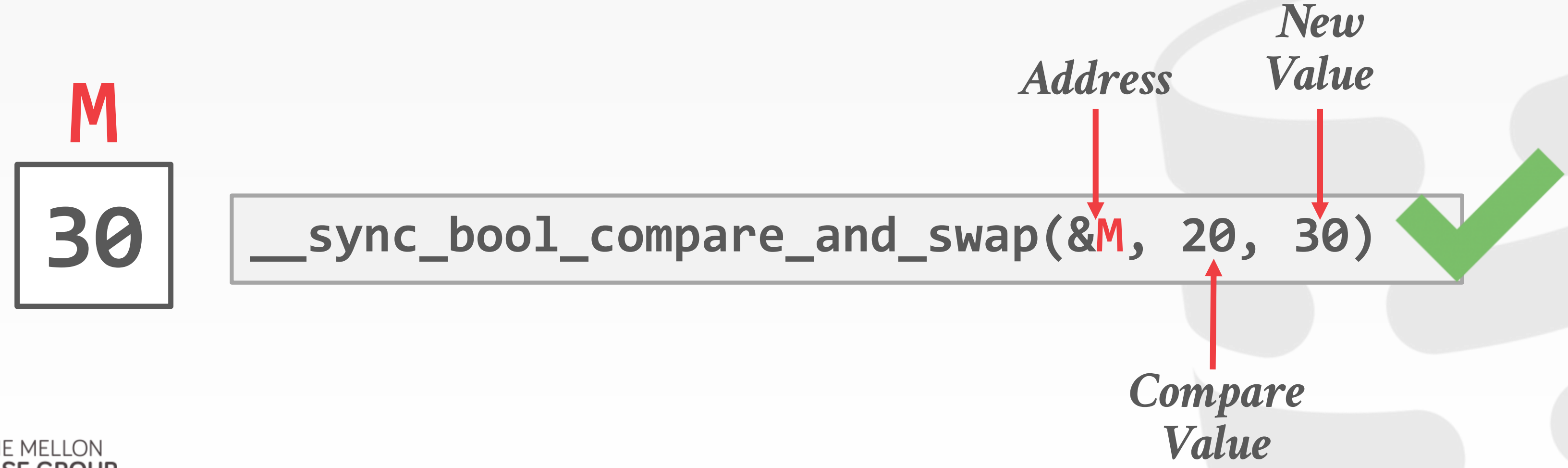
```
__sync_bool_compare_and_swap(&M, 20, 30)
```

COMPARE-AND-SWAP

Atomic instruction that compares contents of a memory location **M** to a given value **V**

→ If values are equal, installs new given value **V’** in **M**

→ Otherwise operation fails



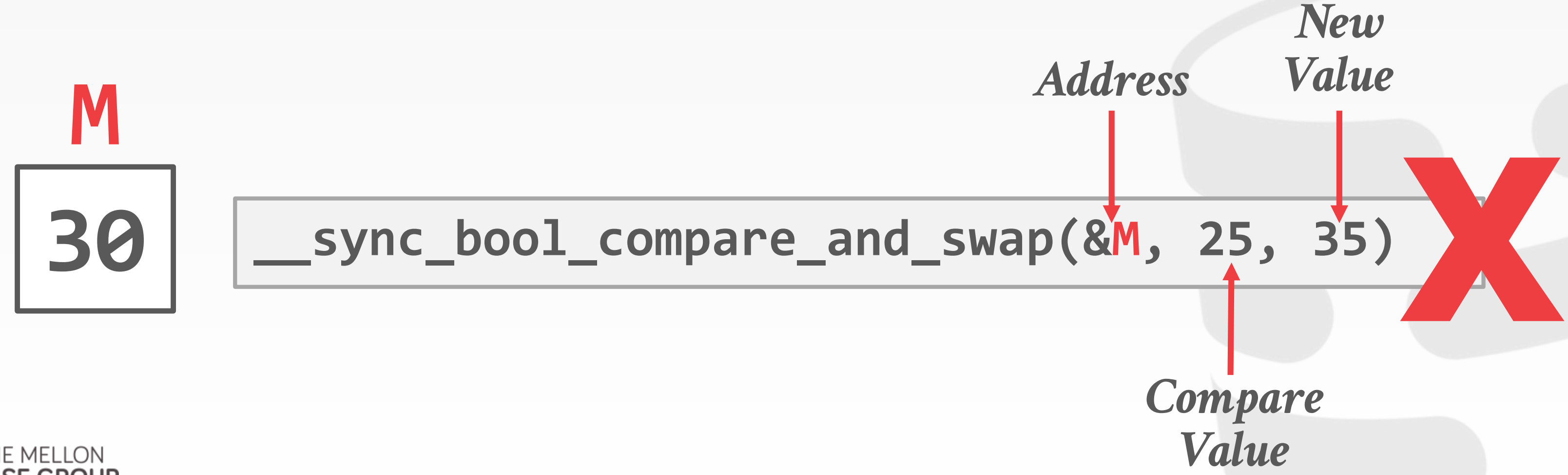
```
__sync_bool_compare_and_swap(&M, 20, 30)
```

COMPARE-AND-SWAP

Atomic instruction that compares contents of a memory location **M** to a given value **V**

→ If values are equal, installs new given value **V’** in **M**

→ Otherwise operation fails



M

30

```
__sync_bool_compare_and_swap(&M, 25, 35)
```

Address

New Value

Compare Value

X

OBSERVATION

Serializability is useful because it allows programmers to ignore concurrency issues but enforcing it may allow too little parallelism and limit performance.

We may want to use a weaker level of consistency to improve scalability.

ISOLATION LEVELS

Controls the extent that a txn is exposed to the actions of other concurrent txns.

Provides for greater concurrency at the cost of exposing txns to uncommitted changes:

→ Dirty Read Anomaly
→ Unrepeatable Reads Anomaly
→ Phantom Reads Anomaly

ANSI ISOLATION LEVELS

SERIALIZABLE

→ No phantoms, all reads repeatable, no dirty reads.

REPEATABLE READS

→ Phantoms may happen.

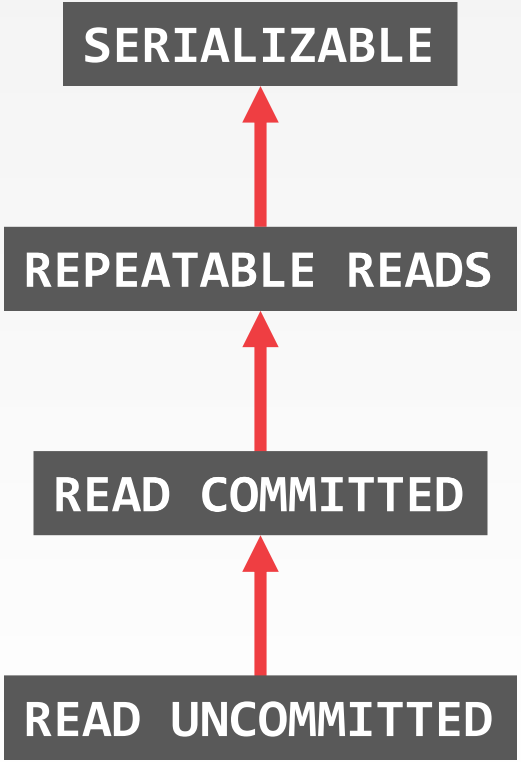
READ COMMITTED

→ Phantoms and unrepeatable reads may happen.

READ UNCOMMITTED

→ All of them may happen.

ISOLATION LEVEL HIERARCHY



SERIALIZABLE

REPEATABLE READS

READ COMMITTED

READ UNCOMMITTED

REAL-WORLD ISOLATION LEVELS

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

Source: Peter Bailis

CRITICISM OF ISOLATION LEVELS

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

Two additional isolation levels:

- → **CURSOR STABILITY**
- → **SNAPSHOT ISOLATION**

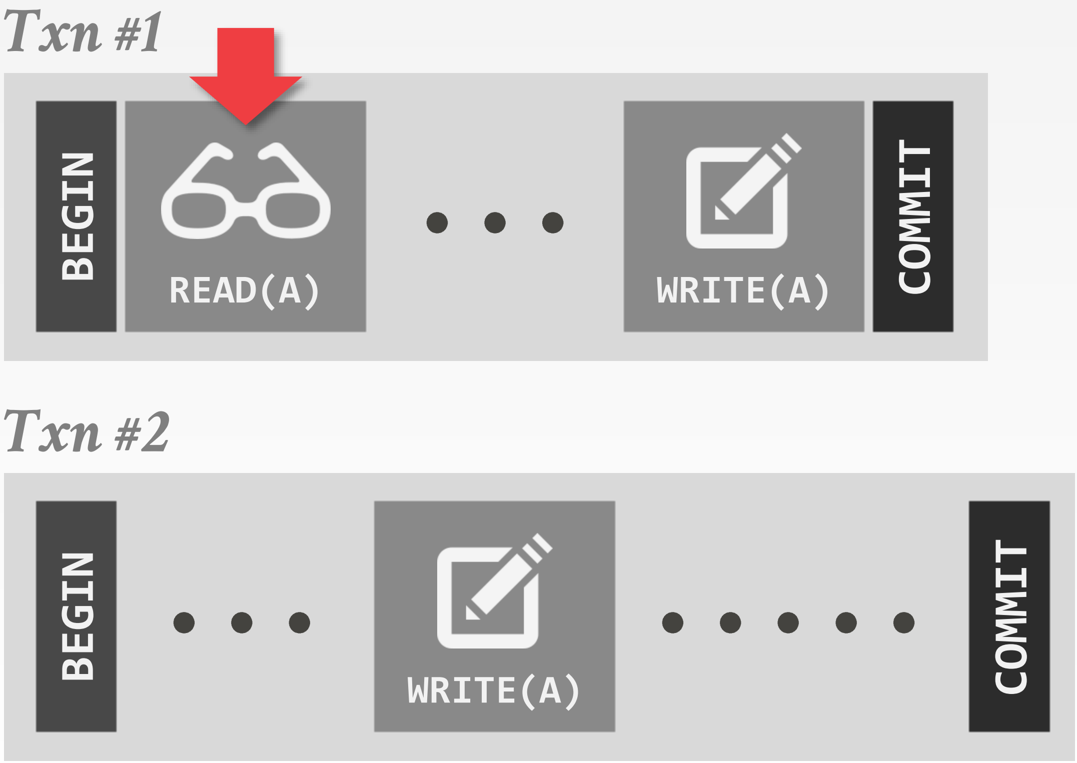
A CRITIQUE OF ANSI SQL ISOLATION LEVELS
SIGMOD 1995

CURSOR STABILITY (CS)

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

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

LOST UPDATE ANOMALY



Txn #1

BEGIN

READ(A)

WRITE(A)

COMMIT

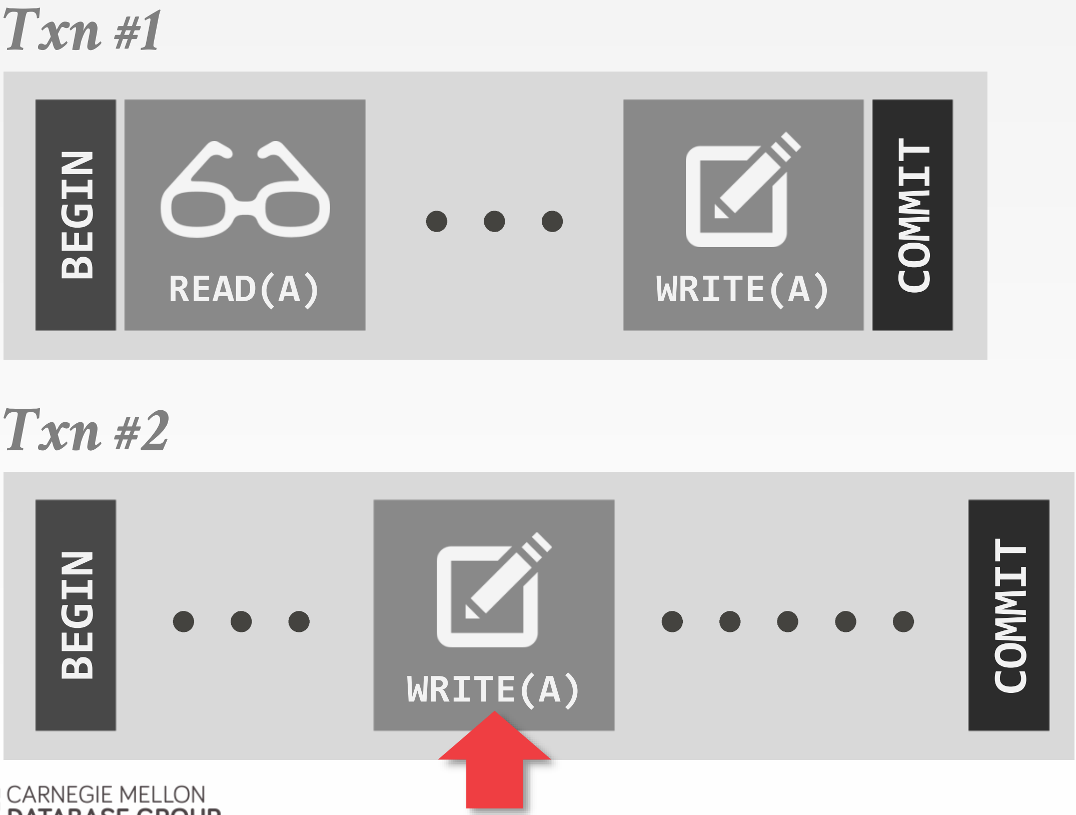
Txn #2

BEGIN

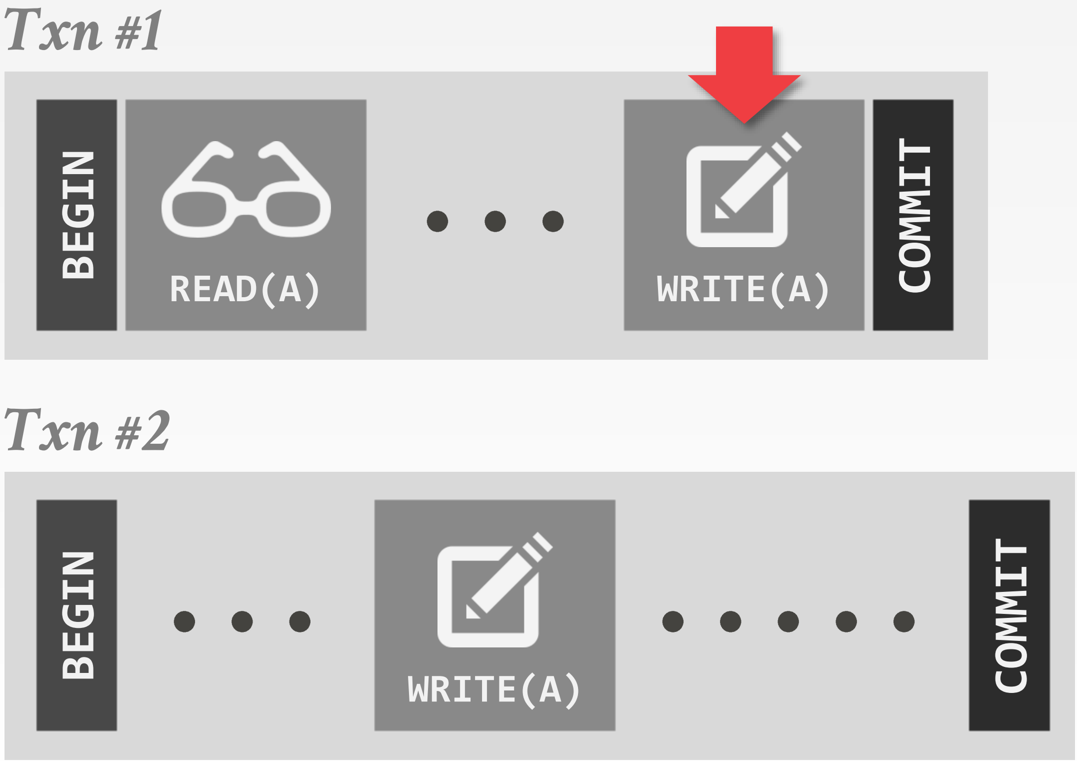
WRITE(A)

COMMIT

LOST UPDATE ANOMALY



LOST UPDATE ANOMALY



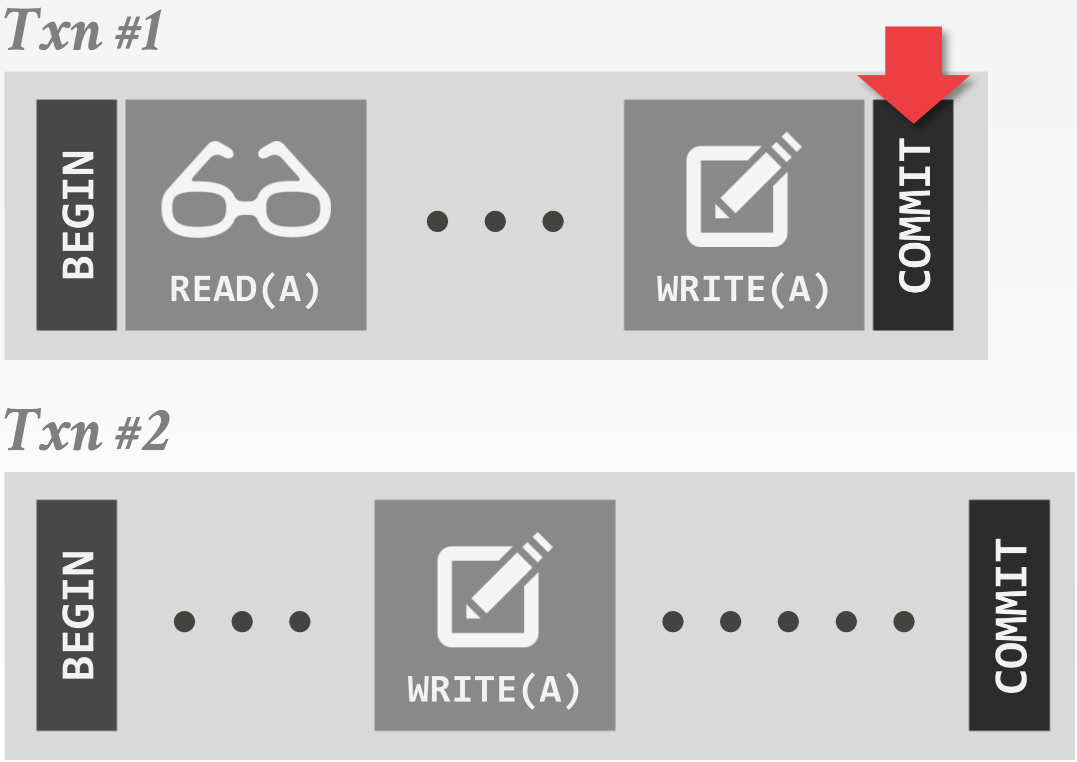
Txn #1

BEGIN READ(A) • • • WRITE(A) COMMIT

Txn #2

BEGIN • • • WRITE(A) • • • • • COMMIT

LOST UPDATE ANOMALY



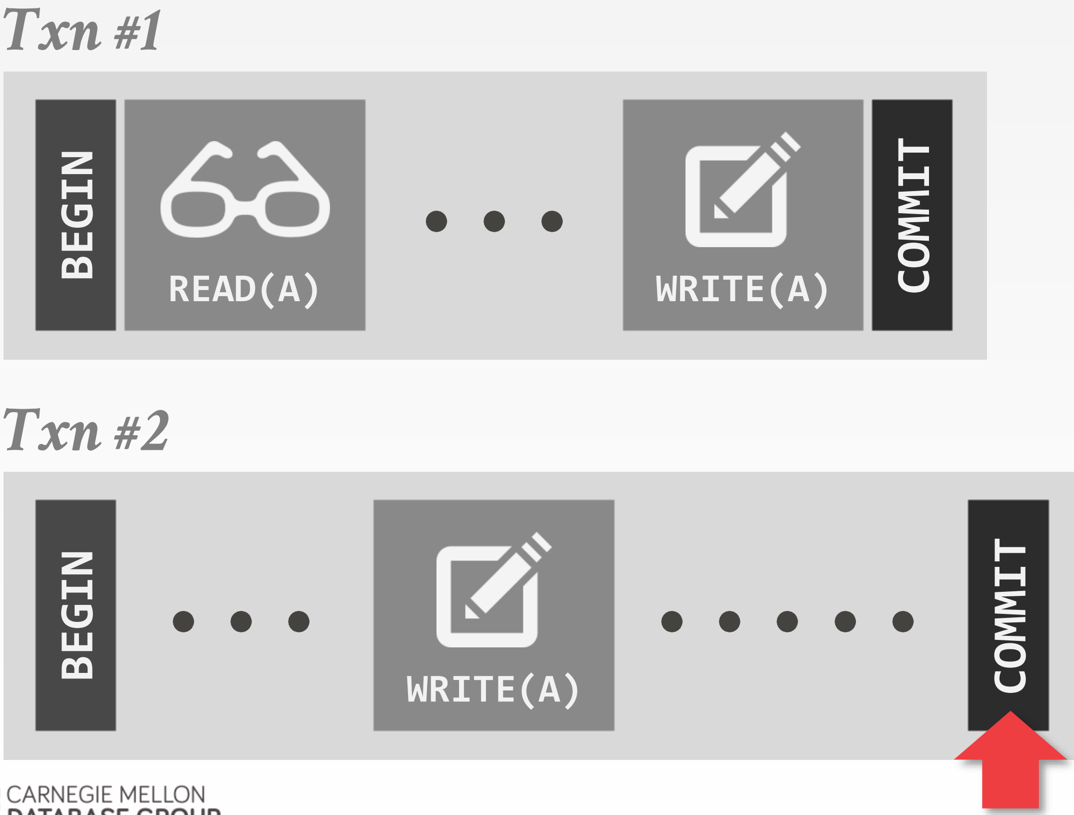
Txn #1

BEGIN READ(A) • • • WRITE(A) COMMIT

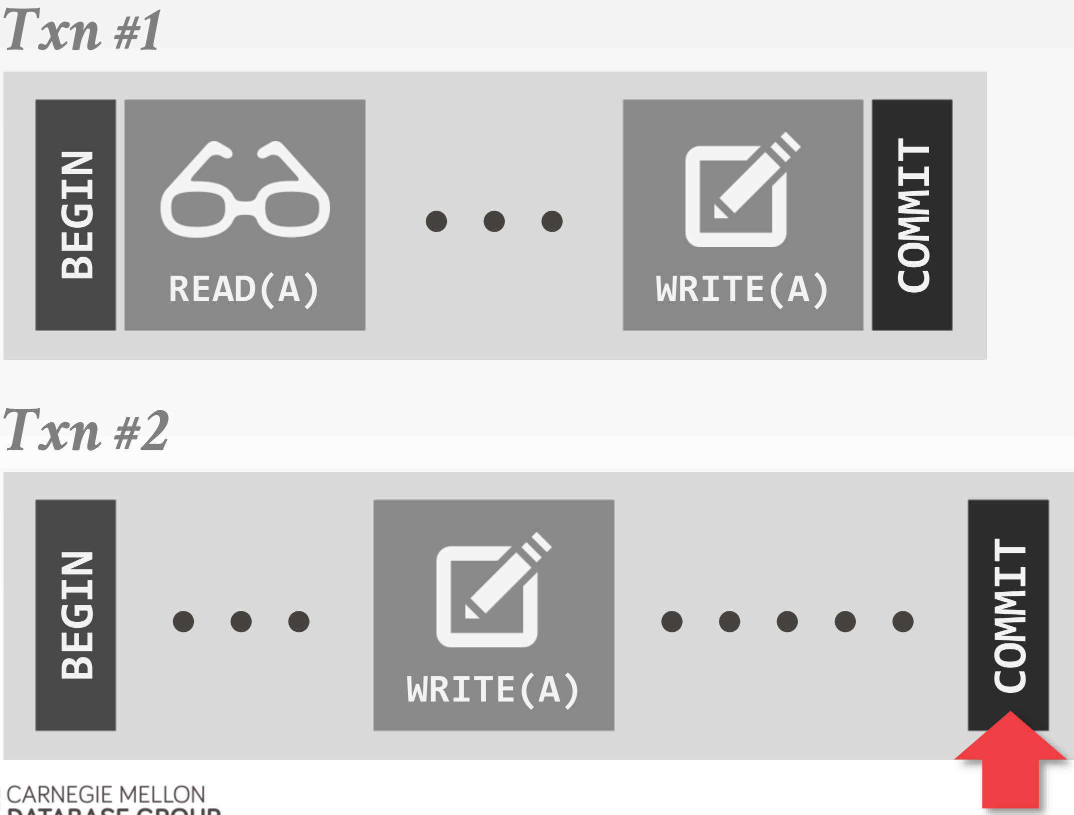
Txn #2

BEGIN • • • WRITE(A) • • • • • COMMIT

LOST UPDATE ANOMALY



LOST UPDATE ANOMALY



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

A **<u>cursor lock</u>** on **A** would prevent this problem (but not always).

SNAPSHOT ISOLATION (SI)

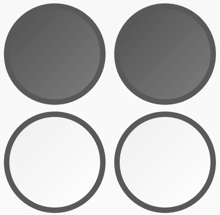
Guarantees that all reads made in a txn see a consistent snapshot of the database that existed at the time the txn started.

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

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

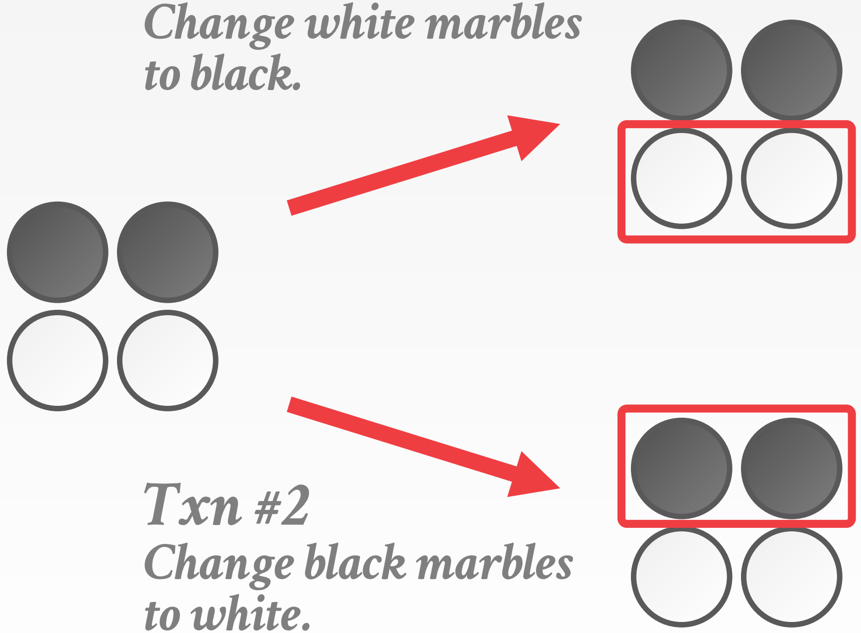
WRITE SKEW ANOMALY

Txn #1
Change white marbles to black.



Txn #2
Change black marbles to white.

WRITE SKEW ANOMALY

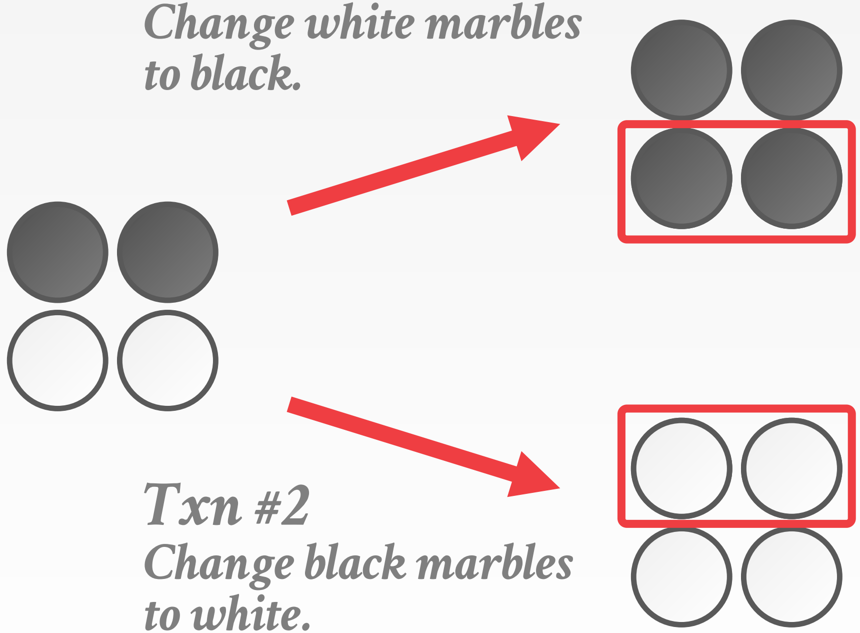


Txn #1
Change white marbles to black.

Txn #2
Change black marbles to white.

WRITE SKEW ANOMALY

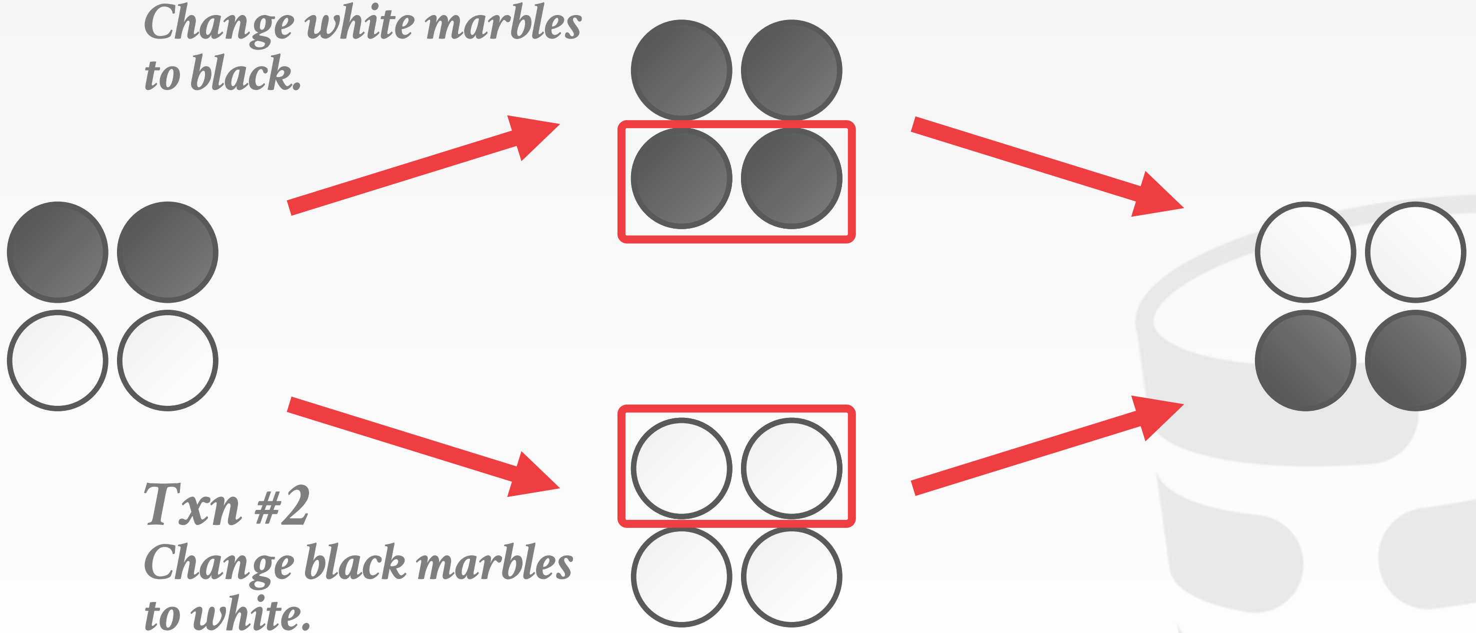
Txn #1
Change white marbles to black.



Txn #2
Change black marbles to white.

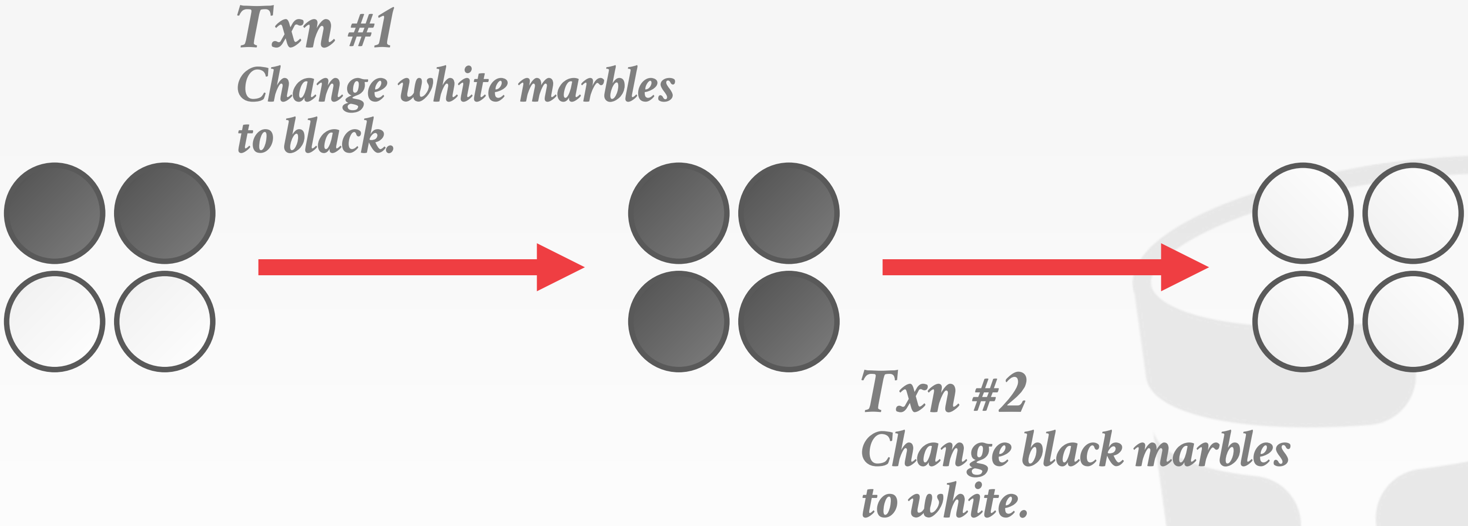
WRITE SKEW ANOMALY

Txn #1
Change white marbles to black.



Txn #2
Change black marbles to white.

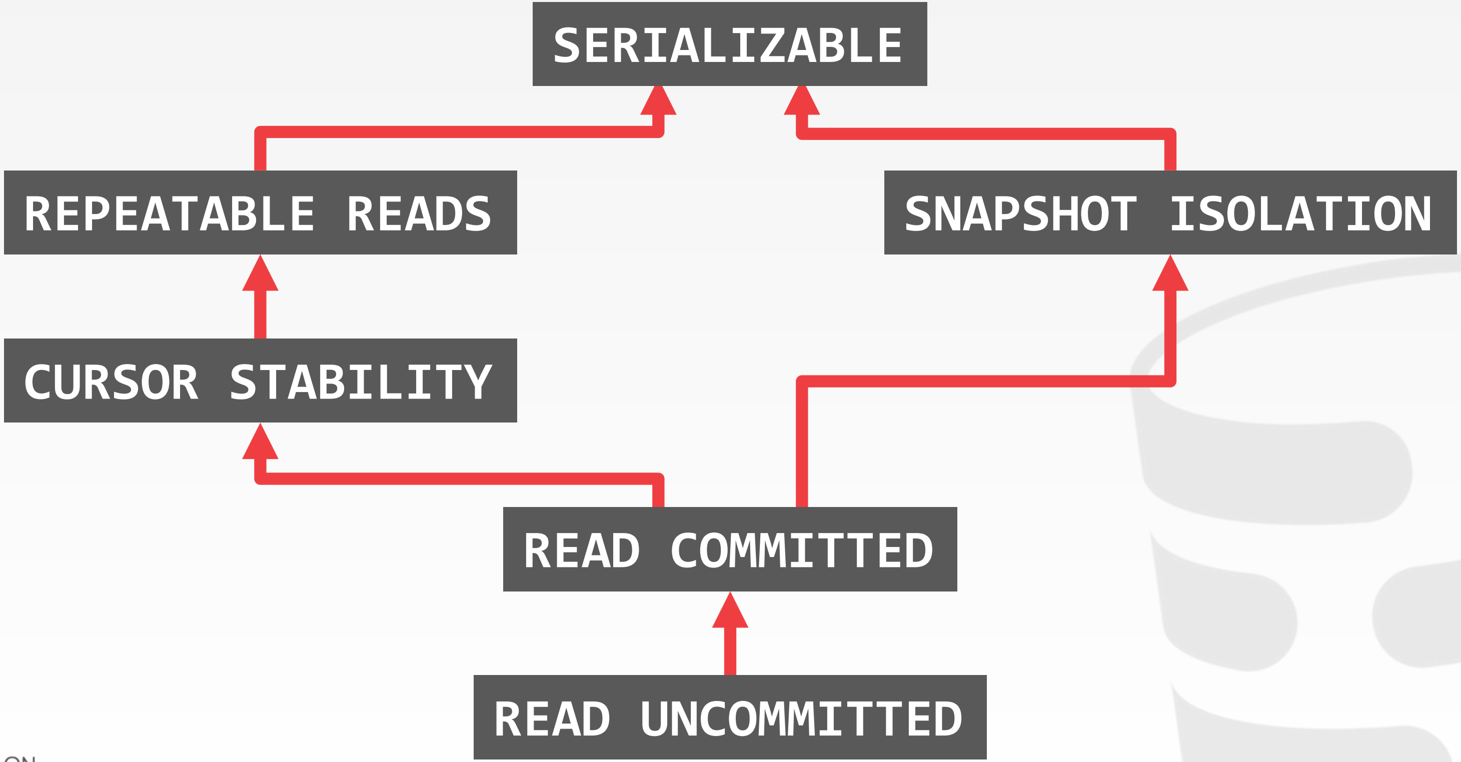
WRITE SKEW ANOMALY



Txn #1
Change white marbles to black.

Txn #2
Change black marbles to white.

ISOLATION LEVEL HIERARCHY



ISO HY

REPEATAE OLATION

CURSOR S

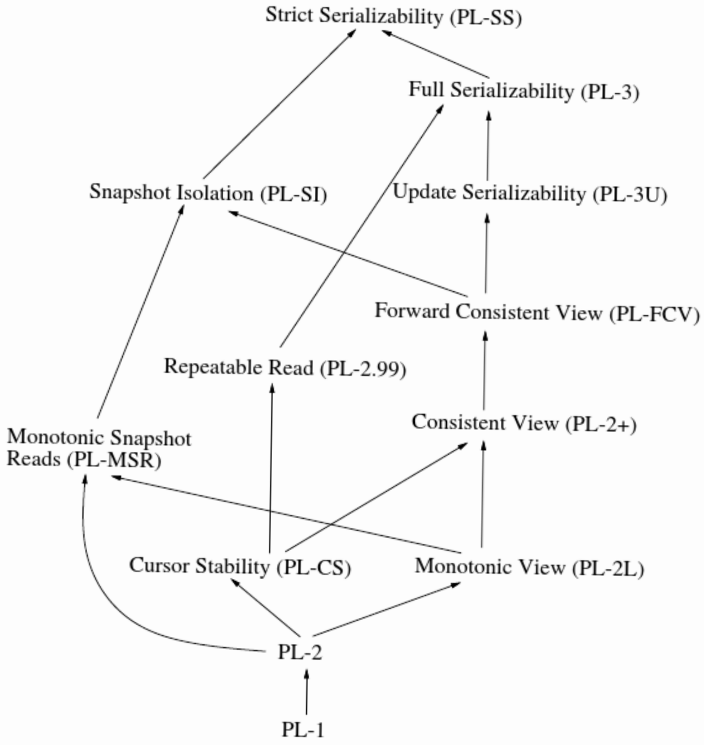

Strict Serializability (PL-SS)
Full Serializability (PL-3)
Snapshot Isolation (PL-SI)
Update Serializability (PL-3U)
Forward Consistent View (PL-FCV)
Repeatable Read (PL-2.99)
Consistent View (PL-2+)
Monotonic Snapshot Reads (PL-MSR)
Cursor Stability (PL-CS)
Monotonic View (PL-2L)
PL-2
PL-1

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

Source: Atul Adya

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.

First proposed in 1978 MIT PhD dissertation.

First implementation was InterBase (Firebird).
Used in almost every new DBMS in last 10 years.

MULTI-VERSION CONCURRENCY CONTROL

Main benefits:

→ Writers don't block readers.
→ Read-only txns can read a consistent snapshot without acquiring locks.
→ Easily support time-travel queries.

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

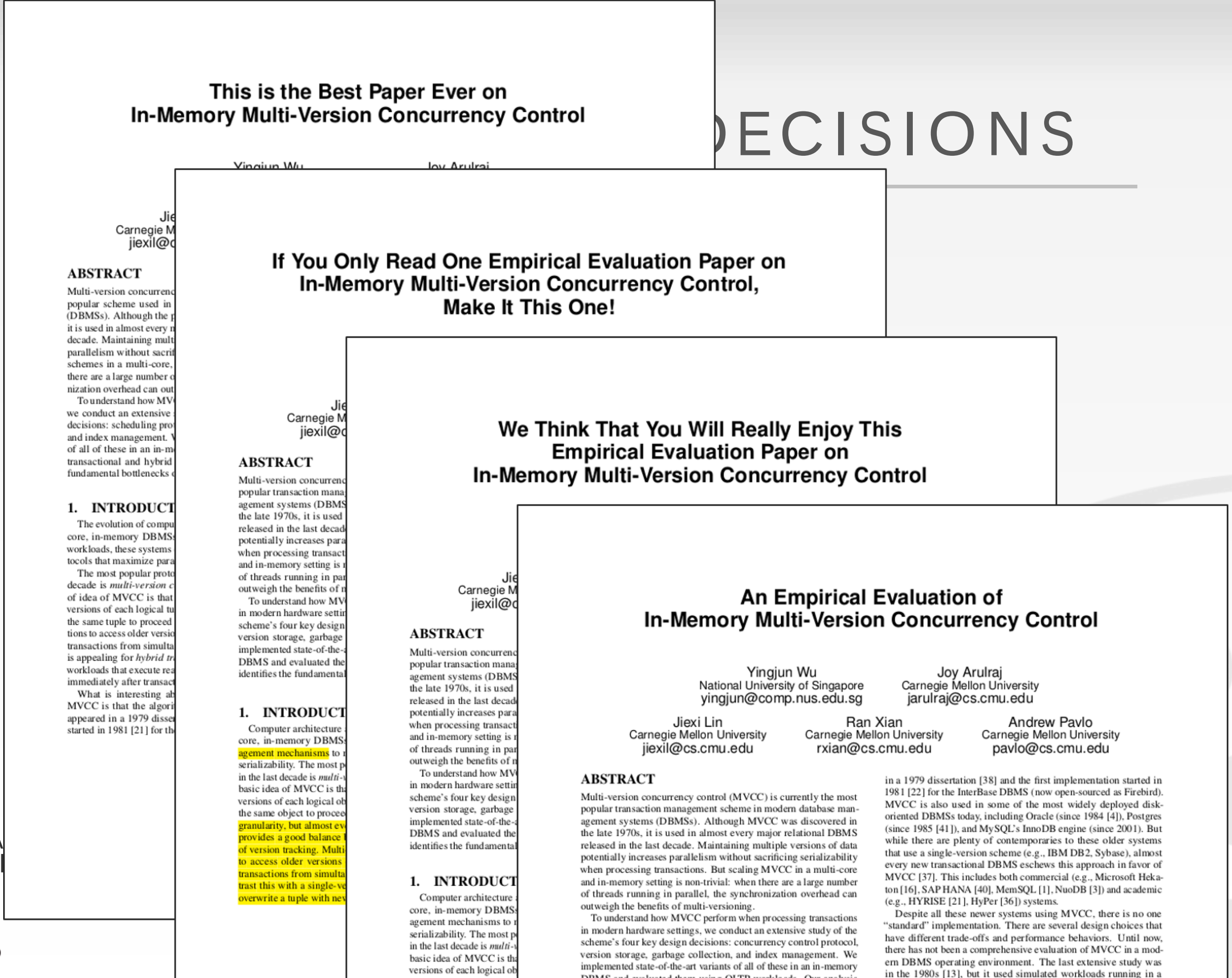
Garbage Collection

Index Management

Txn Id Wraparound (New)

AN EMPIRICAL EVALUATION OF IN-MEMORY MULTI-VERSION CONCURRENCY CONTROL
VLDB 2017

DECISIONS



This is the Best Paper Ever on In-Memory Multi-Version Concurrency Control

If You Only Read One Empirical Evaluation Paper on In-Memory Multi-Version Concurrency Control, Make It This One!

We Think That You Will Really Enjoy This Empirical Evaluation Paper on In-Memory Multi-Version Concurrency Control

An Empirical Evaluation of In-Memory Multi-Version Concurrency Control

Yingjun Wu
National University of Singapore
yingjun@comp.nus.edu.sg

Joy Arulraj
Carnegie Mellon University
jarulraj@cs.cmu.edu

Jiexi Lin
Carnegie Mellon University
jiexil@cs.cmu.edu

Ran Xian
Carnegie Mellon University
rxian@cs.cmu.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

ABSTRACT

Multi-version concurrency control (MVCC) is currently the most popular transaction management scheme in modern database management systems (DBMSs). Although MVCC was discovered in the late 1970s, it is used in almost every major relational DBMS released in the last decade. Maintaining multiple versions of data potentially increases parallelism without sacrificing serializability when processing transactions. But scaling MVCC in a multi-core and in-memory setting is non-trivial: when there are a large number of threads running in parallel, the synchronization overhead can outweigh the benefits of multi-versioning.

To understand how MVCC perform when processing transactions in modern hardware settings, we conduct an extensive study of the scheme's four key design decisions: concurrency control protocol, version storage, garbage collection, and index management. We implemented state-of-the-art variants of all of these in an in-memory DBMS and evaluated them using OLTP workloads. Our analysis

in a 1979 dissertation [38] and the first implementation started in 1981 [22] for the InterBase DBMS (now open-sourced as Firebird). MVCC is also used in some of the most widely deployed disk-oriented DBMSs today, including Oracle (since 1984 [4]), Postgres (since 1985 [41]), and MySQL's InnoDB engine (since 2001). But while there are plenty of contemporaries to these older systems that use a single-version scheme (e.g., IBM DB2, Sybase), almost every new transactional DBMS eschews this approach in favor of MVCC [37]. This includes both commercial (e.g., Microsoft Hekaton [16], SAP HANA [40], MemSQL [1], NuoDB [3]) and academic (e.g., HYRISE [21], HyPer [36]) systems.

Despite all these newer systems using MVCC, there is no one "standard" implementation. There are several design choices that have different trade-offs and performance behaviors. Until now, there has not been a comprehensive evaluation of MVCC in a modern DBMS operating environment. The last extensive study was in the 1980s [13], but it used simulated workloads running in a

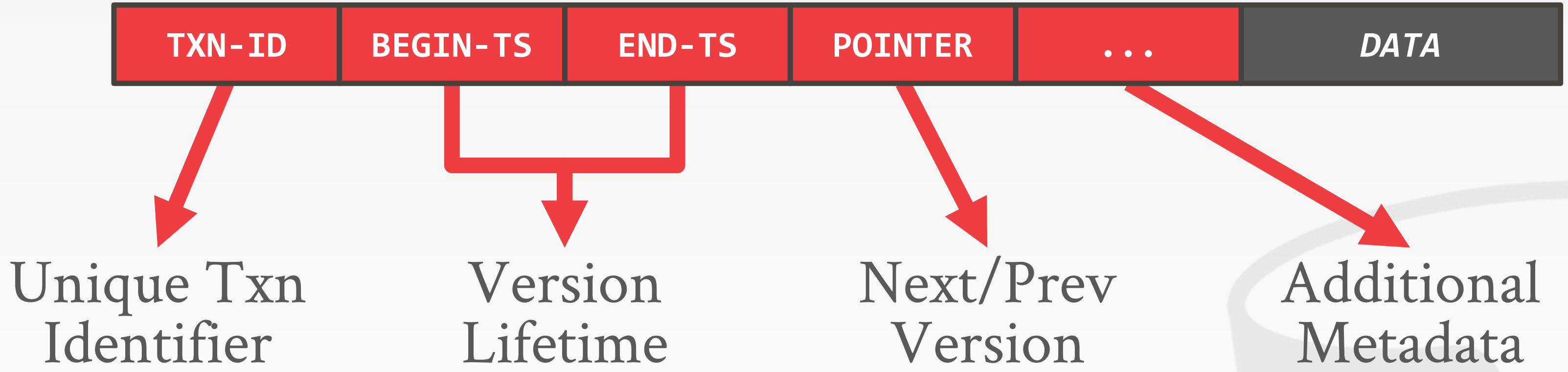
AN EMPIRICA
VERSION CO
VLDB 2017

CARNEGIE MELLON
DATABASE GROUP

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

TUPLE FORMAT



CONCURRENCY CONTROL PROTOCOL

Approach #1: Timestamp Ordering

→ Assign txns timestamps that determine serial order.
→ Considered to be original MVCC protocol.

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.

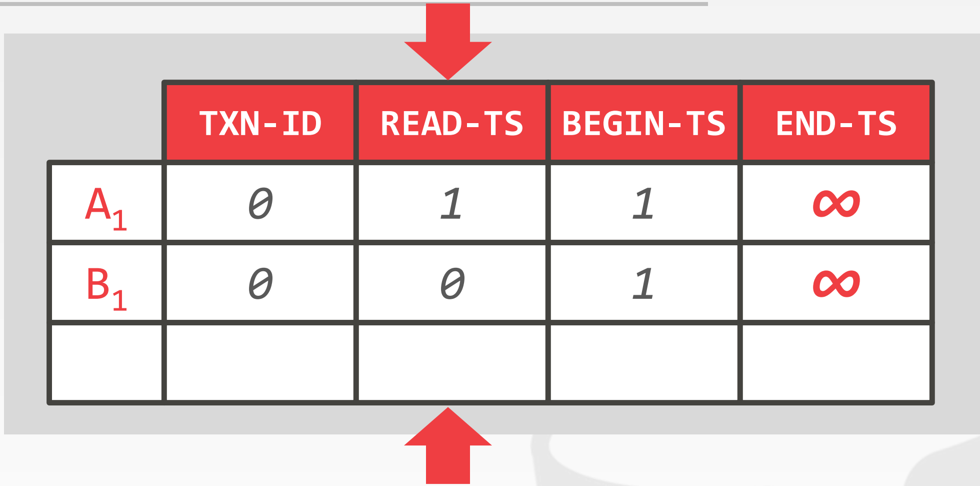
TIMESTAMP ORDERING (MVTO)

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	1	1	∞
B_1	0	0	1	∞

TIMESTAMP ORDERING (MVTO)

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	1	1	∞
B_1	0	0	1	∞

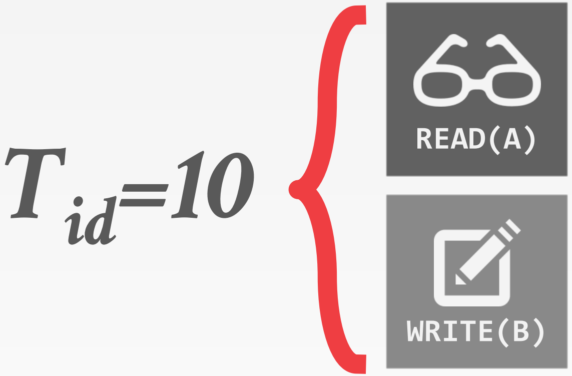
TIMESTAMP ORDERING (MVTO)



	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	1	1	∞
B_1	0	0	1	∞

Use “read-ts” field in the header to keep track of the timestamp of the last txn that read it.

TIMESTAMP ORDERING (MVTO)



T_{id}=10

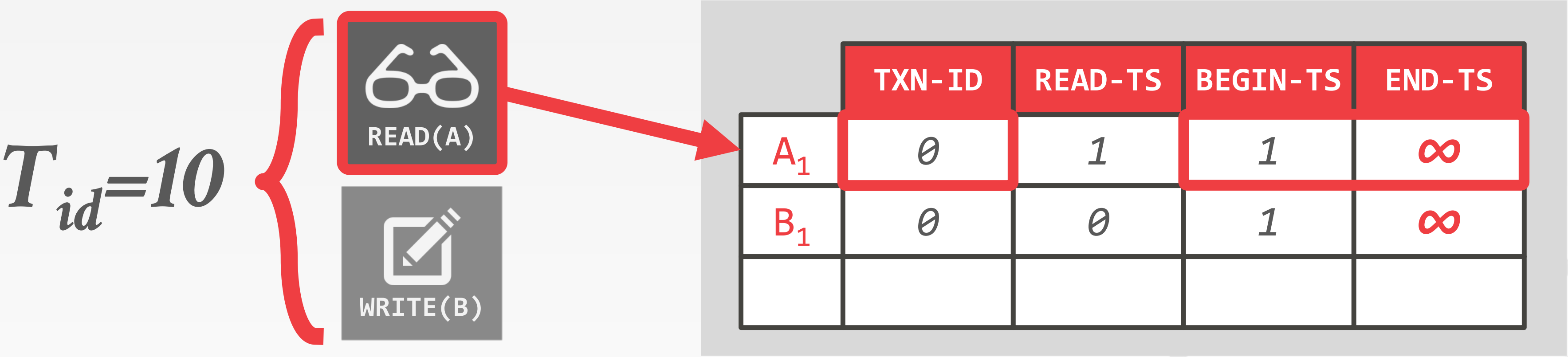
READ(A)

WRITE(B)

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	1	1	∞
B_1	0	0	1	∞

Use “read-ts” field in the header to keep track of the timestamp of the last txn that read it.

TIMESTAMP ORDERING (MVTO)

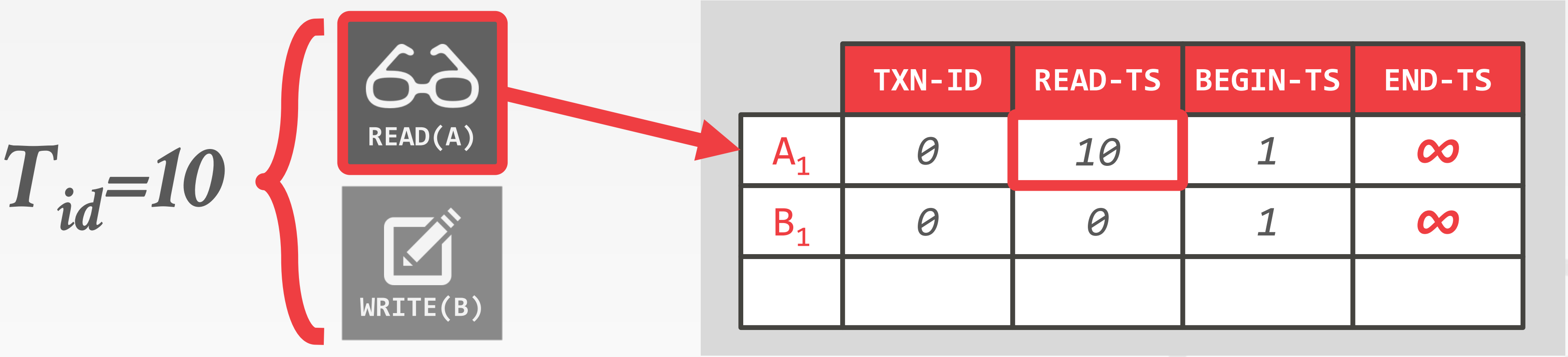


	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	1	1	∞
B_1	0	0	1	∞

Use “read-ts” field in the header to keep track of the timestamp of the last txn that read it.

Txn is allowed to read version if the lock is unset and its T_{id} is between “begin-ts” and “end-ts”.

TIMESTAMP ORDERING (MVTO)

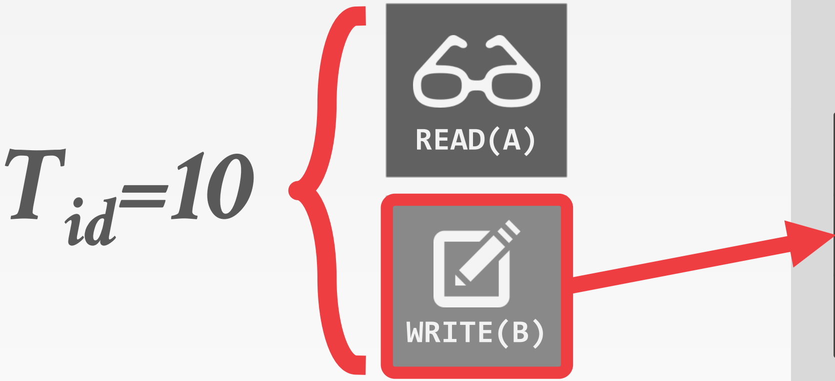


	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	10	1	∞
B_1	0	0	1	∞

Use "read-ts" field in the header to keep track of the timestamp of the last txn that read it.

Txn is allowed to read version if the lock is unset and its T_{id} is between "begin-ts" and "end-ts".

TIMESTAMP ORDERING (MVTO)



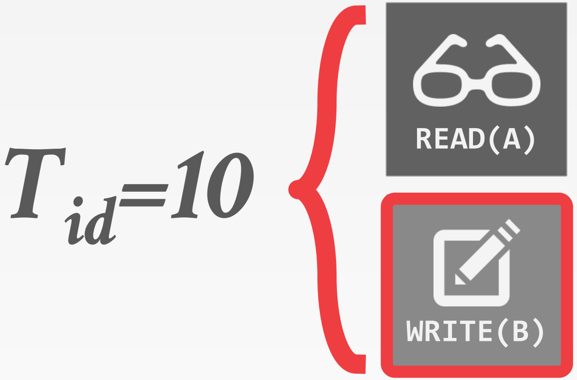
	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	10	1	∞
B_1	0	0	1	∞

Use “read-ts” field in the header to keep track of the timestamp of the last txn that read it.

Txn is allowed to read version if the lock is unset and its T_{id} is between “begin-ts” and “end-ts”.

Txn creates a new version if no other txn holds lock and T_{id} is greater than “read-ts”.

TIMESTAMP ORDERING (MVTO)



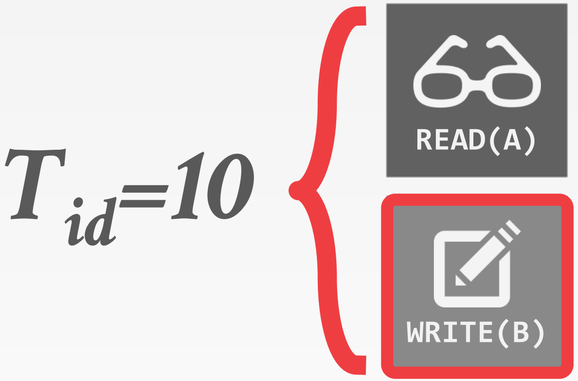
	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	10	1	∞
🔒 B_1	10	0	1	∞

Use "read-ts" field in the header to keep track of the timestamp of the last txn that read it.

Txn is allowed to read version if the lock is unset and its T_{id} is between "begin-ts" and "end-ts".

Txn creates a new version if no other txn holds lock and T_{id} is greater than "read-ts".

TIMESTAMP ORDERING (MVTO)



	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	10	1	∞
🔒 B_1	10	0	1	∞
🔒 B_2	10	0	10	∞

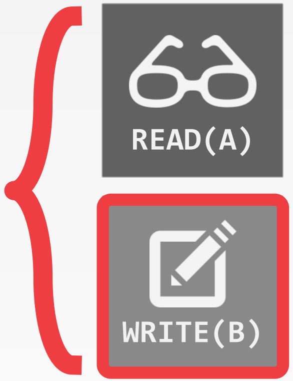
Use “read-ts” field in the header to keep track of the timestamp of the last txn that read it.

Txn is allowed to read version if the lock is unset and its T_{id} is between “begin-ts” and “end-ts”.

Txn creates a new version if no other txn holds lock and T_{id} is greater than “read-ts”.

TIMESTAMP ORDERING (MVTO)

T_{id}=10



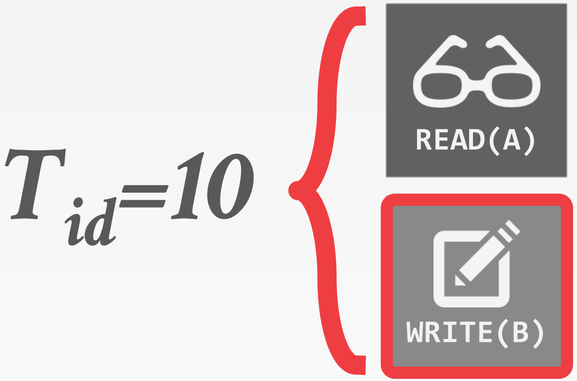
	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	10	1	∞
🔒 B_1	10	0	1	10
🔒 B_2	10	0	10	∞

Use “read-ts” field in the header to keep track of the timestamp of the last txn that read it.

Txn is allowed to read version if the lock is unset and its T_{id} is between “begin-ts” and “end-ts”.

Txn creates a new version if no other txn holds lock and T_{id} is greater than “read-ts”.

TIMESTAMP ORDERING (MVTO)



$T_{id}=10$

READ(A)

WRITE(B)

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	10	1	∞
B_1	0	0	1	10
B_2	0	0	10	∞

Use “read-ts” field in the header to keep track of the timestamp of the last txn that read it.

Txn is allowed to read version if the lock is unset and its T_{id} is between “begin-ts” and “end-ts”.

Txn creates a new version if no other txn holds lock and T_{id} is greater than “read-ts”.

VERSION STORAGE

The DBMS uses the tuples' pointer field to create a latch-free **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.

Threads store versions in "local" memory regions to avoid contention on centralized data structures.

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

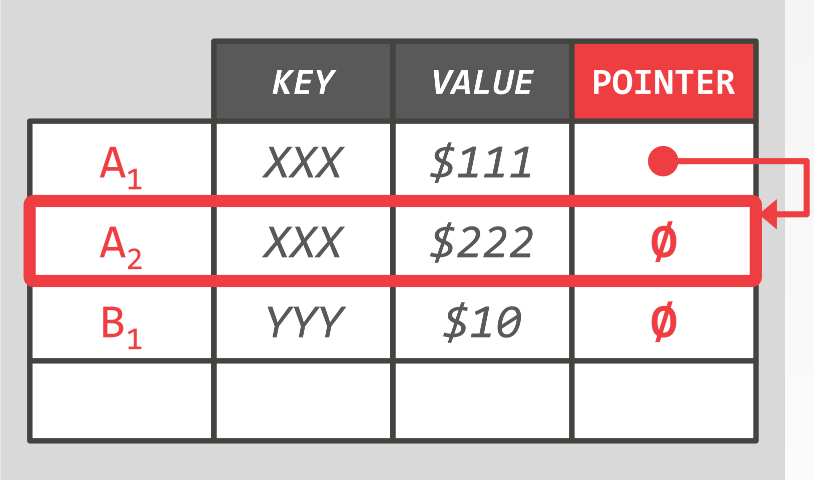
→ Old versions are copied to separate table space.

Approach #3: Delta Storage

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

APPEND-ONLY STORAGE

Main Table



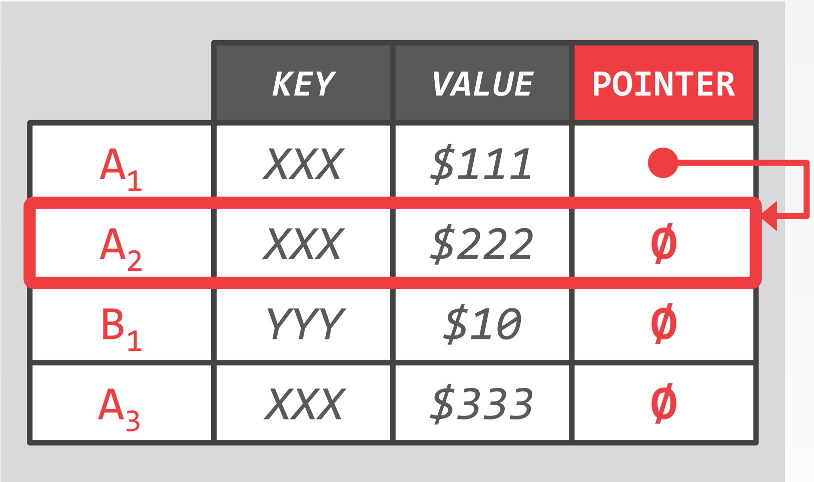
	KEY	VALUE	POINTER
A_1	XXX	$111	●
A_2	XXX	$222	Ø
B_1	YYY	$10	Ø

All of the physical versions of a logical tuple are stored in the same table space

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

APPEND-ONLY STORAGE

Main Table



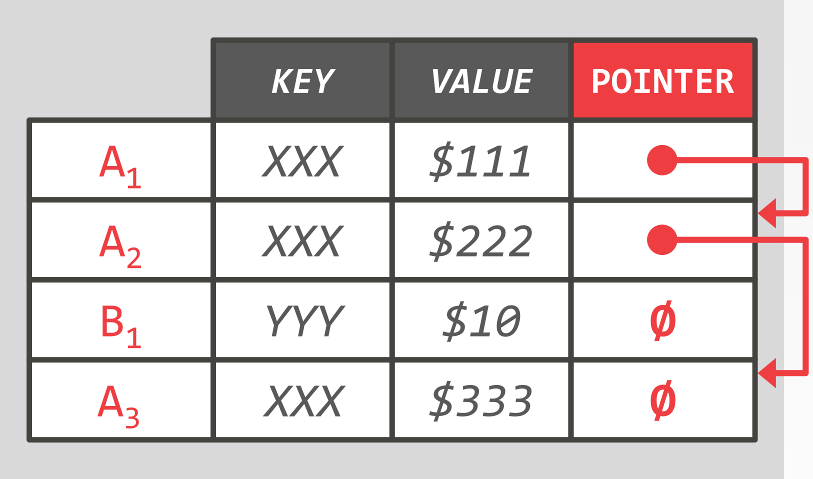
	KEY	VALUE	POINTER
A_1	XXX	$111	●
A_2	XXX	$222	Ø
B_1	YYY	$10	Ø
A_3	XXX	$333	Ø

All of the physical versions of a logical tuple are stored in the same table space

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

APPEND-ONLY STORAGE

Main Table



	KEY	VALUE	POINTER
A_1	XXX	\$111	●
A_2	XXX	\$222	●
B_1	YYY	\$10	Ø
A_3	XXX	\$333	Ø

All of the physical versions of a logical tuple are stored in the same table space

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

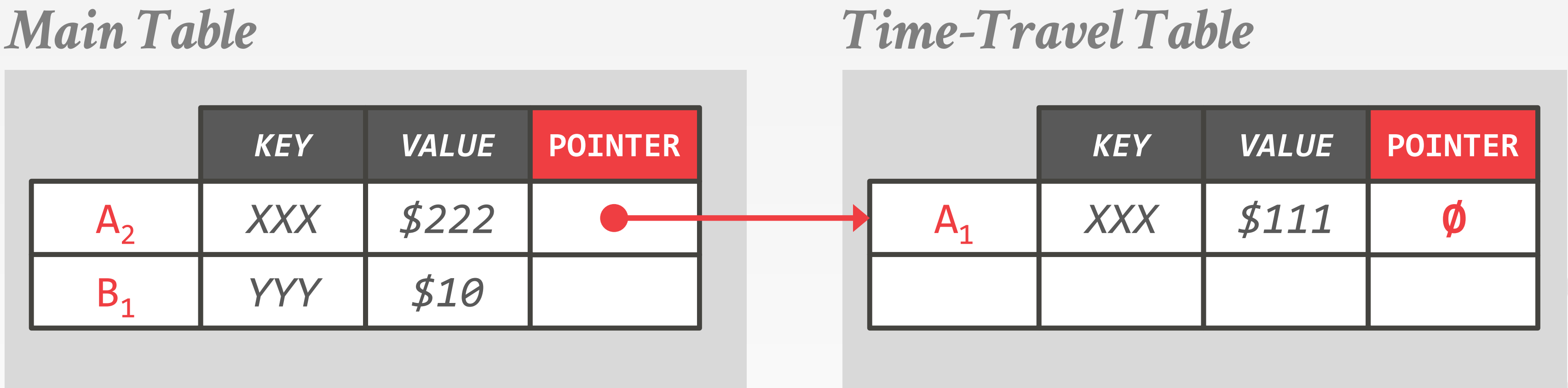
VERSION CHAIN ORDERING

Approach #1: Oldest-to-Newest (O2N)
→ Just append new version to end of the chain.
→ Have to traverse chain on look-ups.

Approach #2: Newest-to-Oldest (N2O)
→ Have to update index pointers for every new version.
→ Don't have to traverse chain on look ups.

The ordering of the chain has different performance trade-offs.

TIME-TRAVEL STORAGE



Main Table

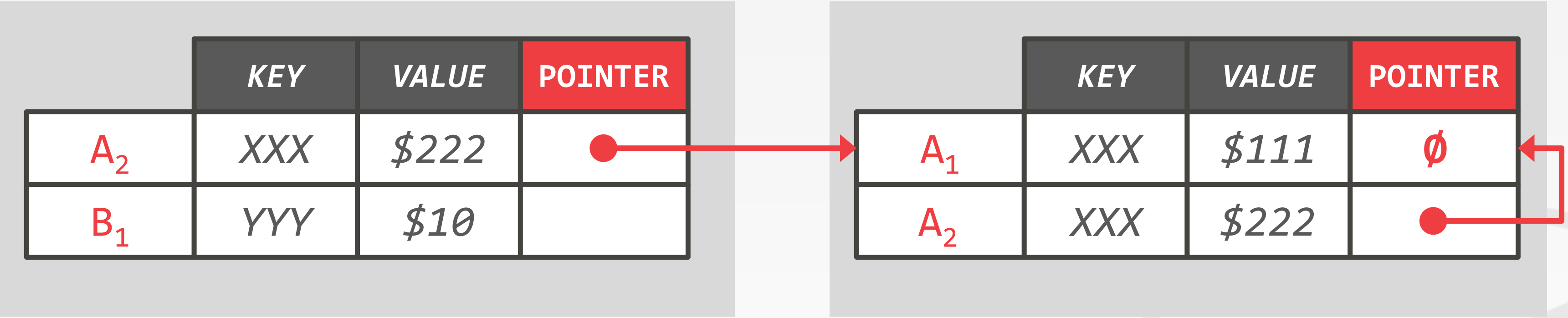
	KEY	*VALUE*	POINTER
A_2	*XXX*	*$222*	●
B_1	*YYY*	*$10*	

Time-Travel Table

	KEY	*VALUE*	POINTER
A_1	*XXX*	*$111*	Ø

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

TIME-TRAVEL STORAGE



Main Table

	KEY	VALUE	POINTER
A_2	XXX	$222	●
B_1	YYY	$10	

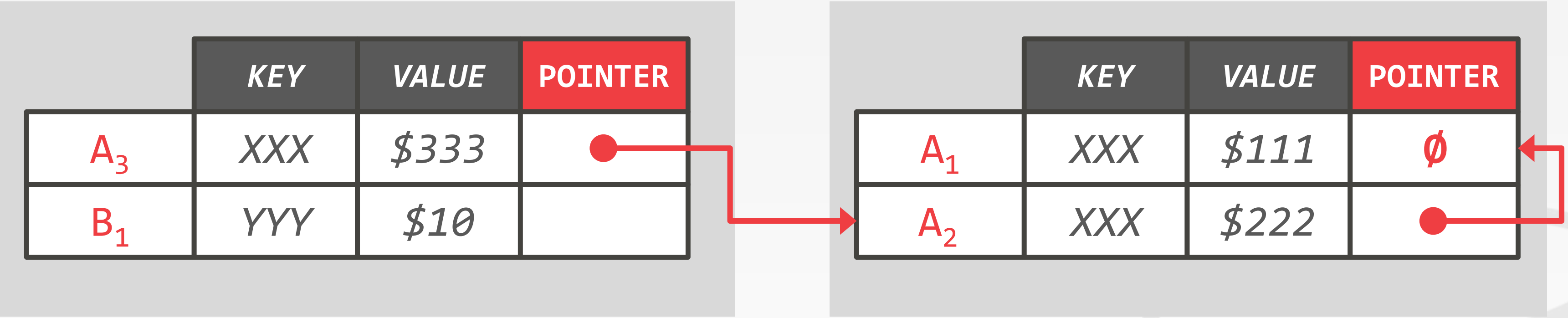
Time-Travel Table

	KEY	VALUE	POINTER
A_1	XXX	$111	∅
A_2	XXX	$222	●

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

Overwrite master version in the main table. Update pointers.

TIME-TRAVEL STORAGE



Main Table

	KEY	VALUE	POINTER
A_3	XXX	\$333	●
B_1	YYY	\$10	

Time-Travel Table

	KEY	VALUE	POINTER
A_1	XXX	\$111	Ø
A_2	XXX	\$222	●

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

Overwrite master version in the main table. Update pointers.

DELTA STORAGE

Main Table

	KEY	VALUE	POINTER
A_1	XXX	$111	
B_1	YYY	$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

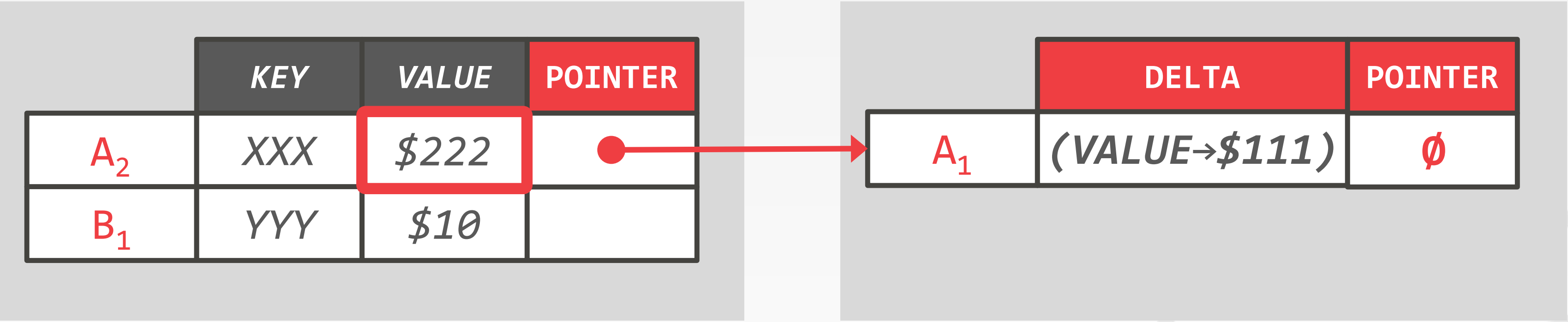
	KEY	VALUE	POINTER
A_1	XXX	$111	
B_1	YYY	$10	

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

	KEY	VALUE	POINTER
A_2	*XXX*	*$222*	●
B_1	*YYY*	*$10*	

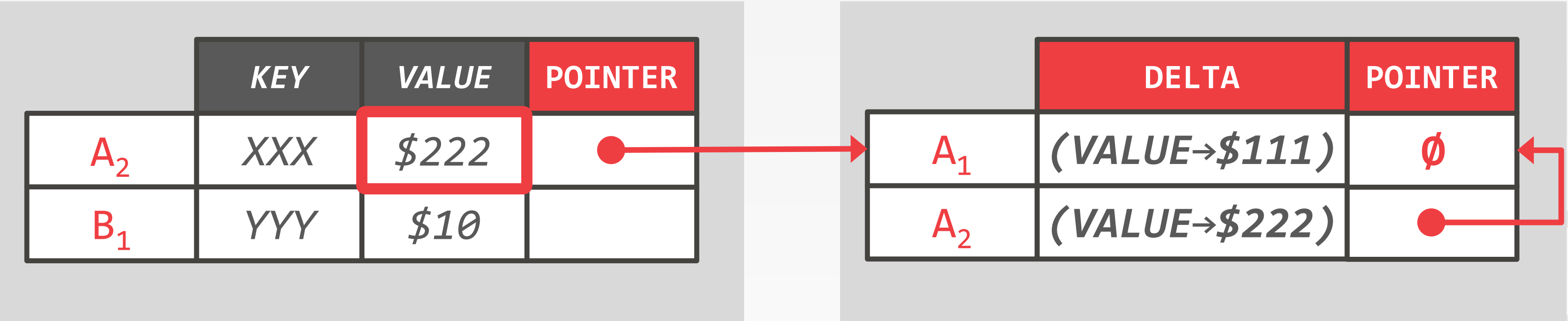
Delta Storage Segment

	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 *Delta Storage Segment*

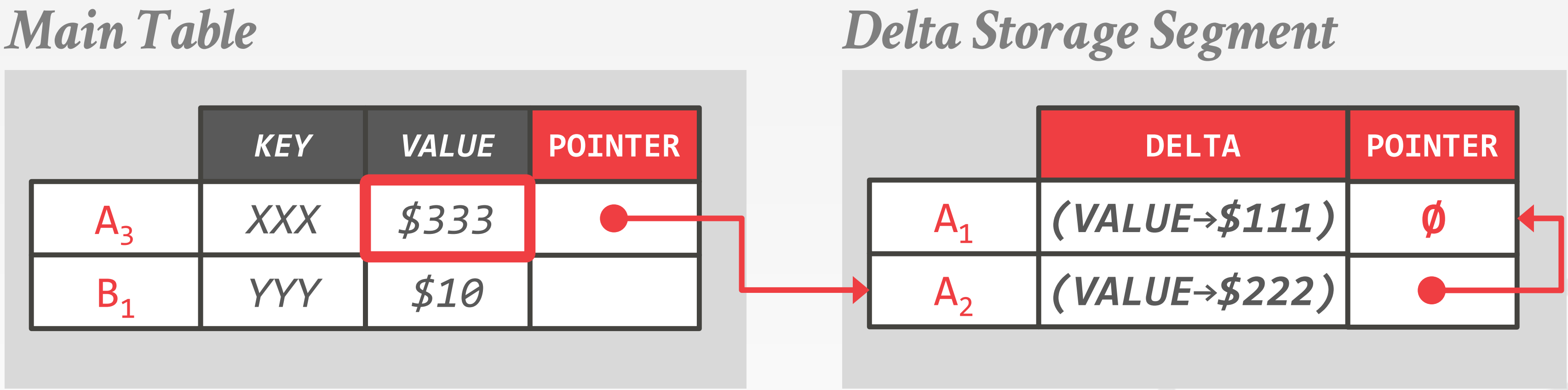


	KEY	VALUE	POINTER
A_2	XXX	$222	●
B_1	YYY	$10	

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

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

DELTA STORAGE



Main Table

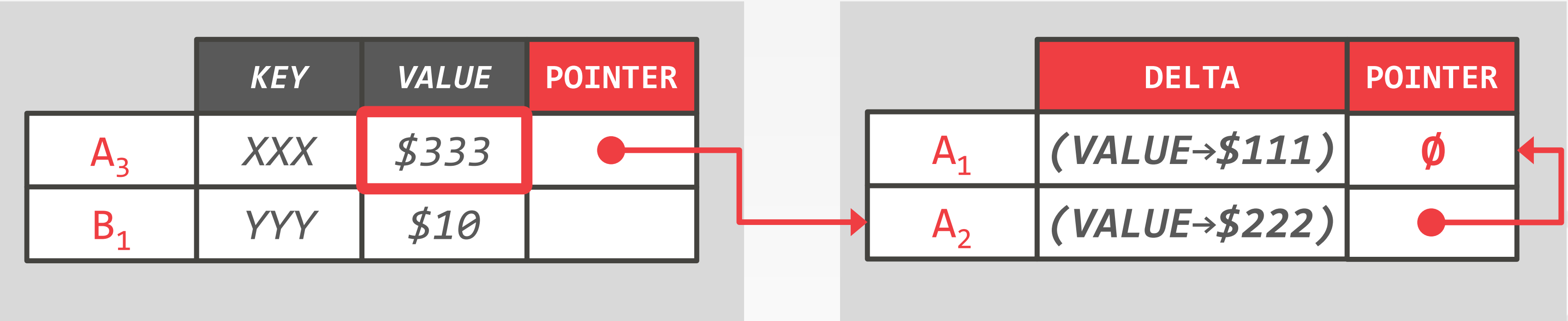
	KEY	VALUE	POINTER
A_3	XXX	$333	●
B_1	YYY	$10	

Delta Storage Segment

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

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

DELTA STORAGE



Main Table

	KEY	VALUE	POINTER
A_3	XXX	$333	●
B_1	YYY	$10	

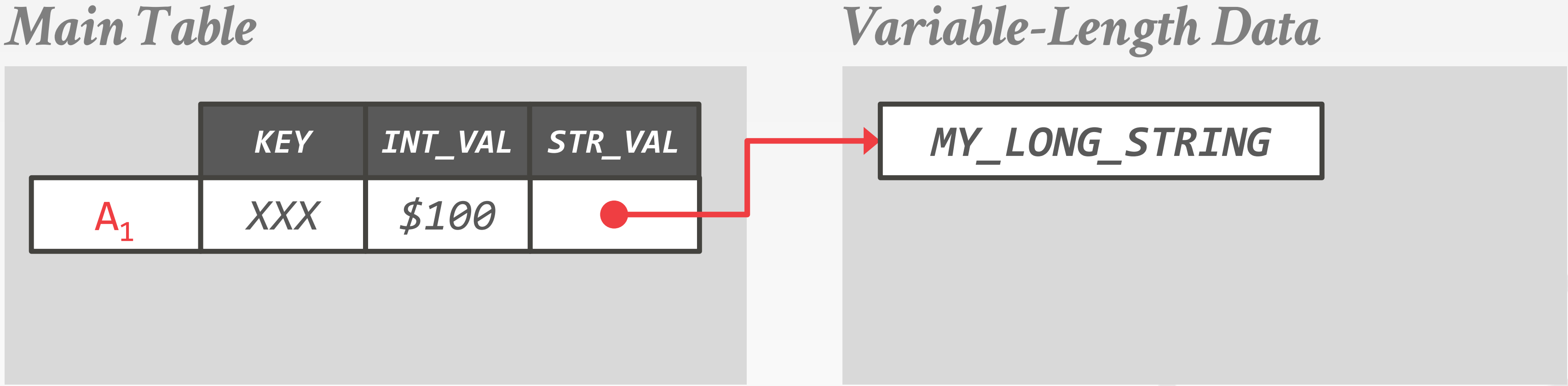
Delta Storage Segment

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

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.

NON-INLINE ATTRIBUTES



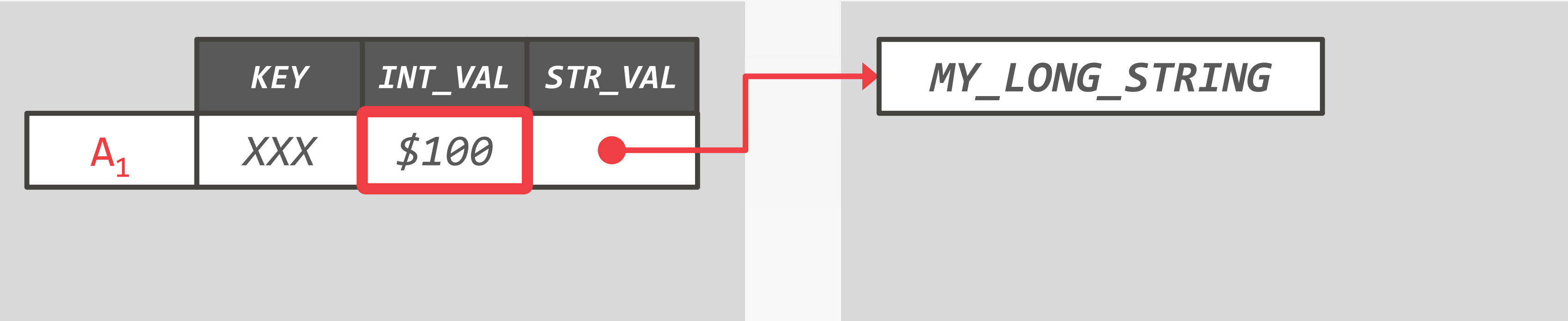
Main Table

	KEY	INT_VAL	STR_VAL
A_1	XXX	$100	●

Variable-Length Data

NON-INLINE ATTRIBUTES

Main Table *Variable-Length Data*



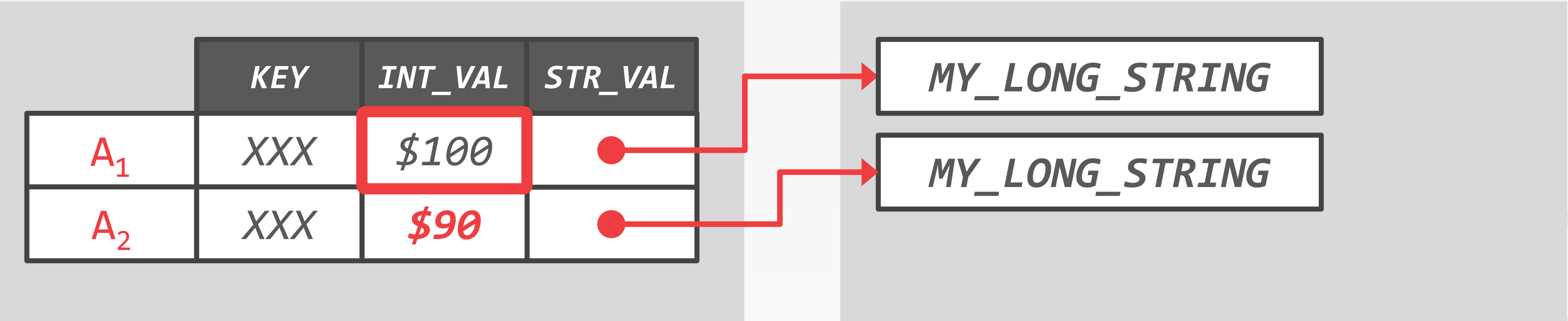
	KEY	INT_VAL	STR_VAL
A_1	XXX	$100	●

MY_LONG_STRING

NON-INLINE ATTRIBUTES

Main Table

Variable-Length Data



	KEY	INT_VAL	STR_VAL
A_1	XXX	$100	●
A_2	XXX	$90	●

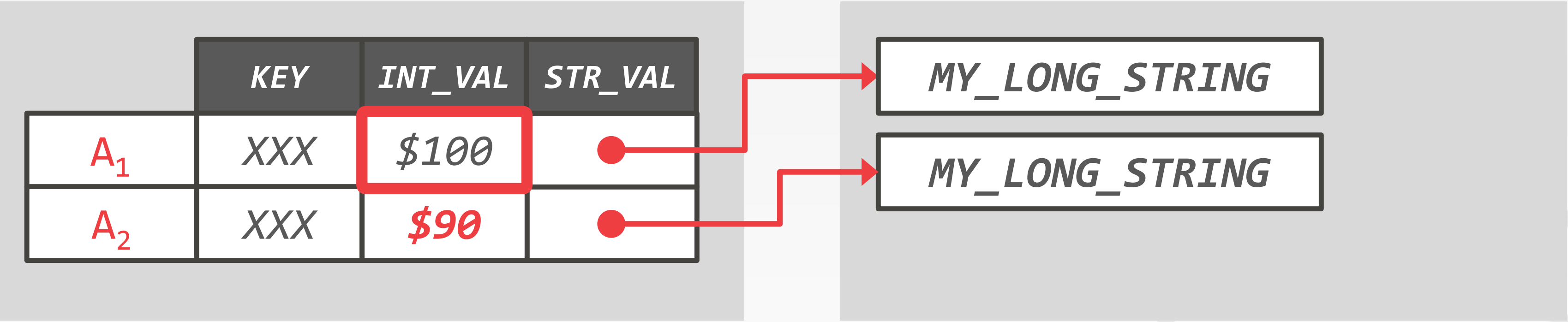
MY_LONG_STRING

MY_LONG_STRING

NON-INLINE ATTRIBUTES

Main Table

Variable-Length Data



	KEY	INT_VAL	STR_VAL
A_1	XXX	$100	●
A_2	XXX	$90	●

MY_LONG_STRING

MY_LONG_STRING

Reuse pointers to variable-length pool for values that do not change between versions.

NON-INLINE ATTRIBUTES

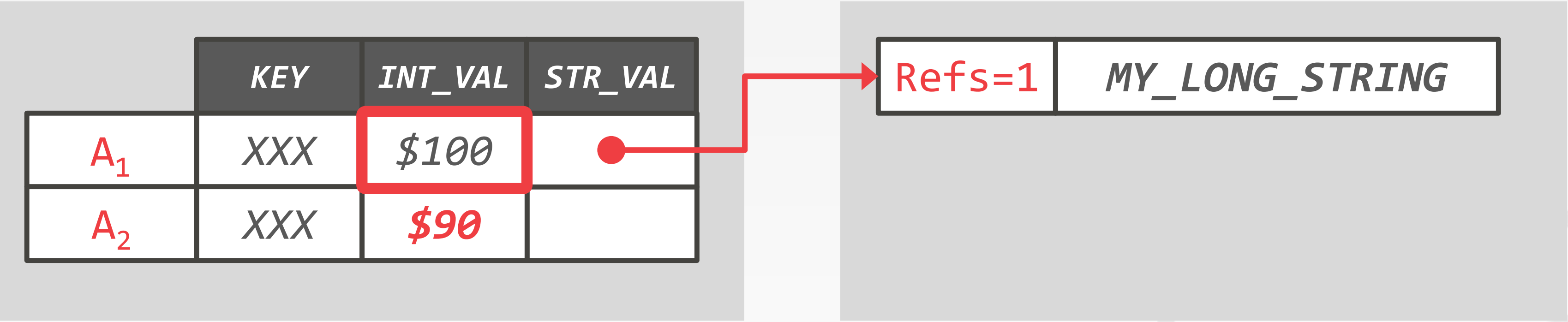
Main Table

	KEY	INT_VAL	STR_VAL
A_1	XXX	$100	
A_2	XXX	**$90**	

Variable-Length Data

Reuse pointers to variable-length pool for values that do not change between versions.

NON-INLINE ATTRIBUTES



Main Table

	KEY	INT_VAL	STR_VAL
A_1	XXX	$100	●
A_2	XXX	$90	

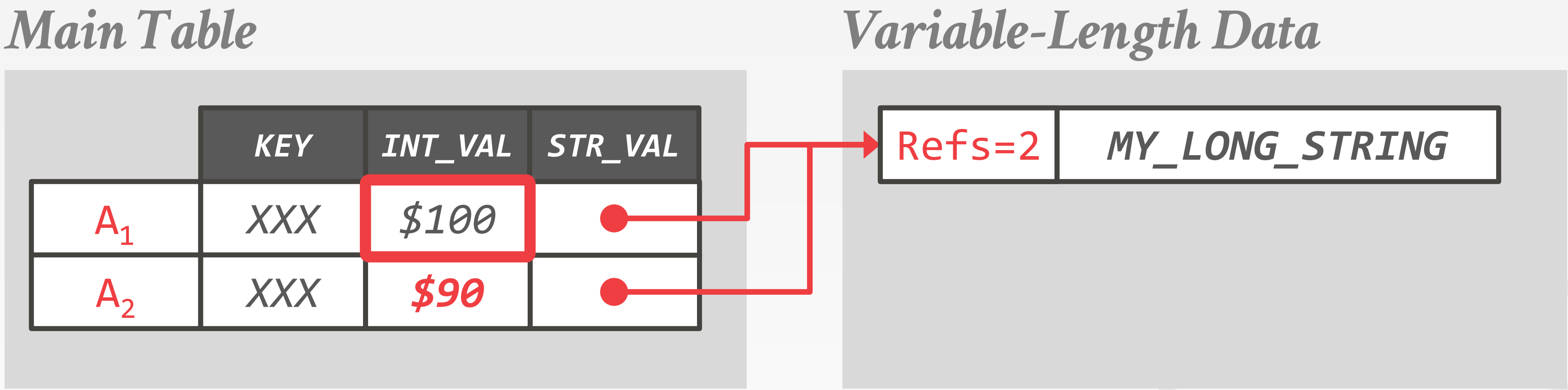
Variable-Length Data

Refs=1	MY_LONG_STRING

Reuse pointers to variable-length pool for values that do not change between versions.

Requires reference counters to know when it safe to free memory. Unable to relocate memory easily.

NON-INLINE ATTRIBUTES



Main Table

	KEY	INT_VAL	STR_VAL
A_1	XXX	$100	●
A_2	XXX	$90	●

Variable-Length Data

Refs=2	MY_LONG_STRING

Reuse pointers to variable-length pool for values that do not change between versions.

Requires reference counters to know when it safe to free memory. Unable to relocate memory easily.

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

T_{id}=12

Thread #2

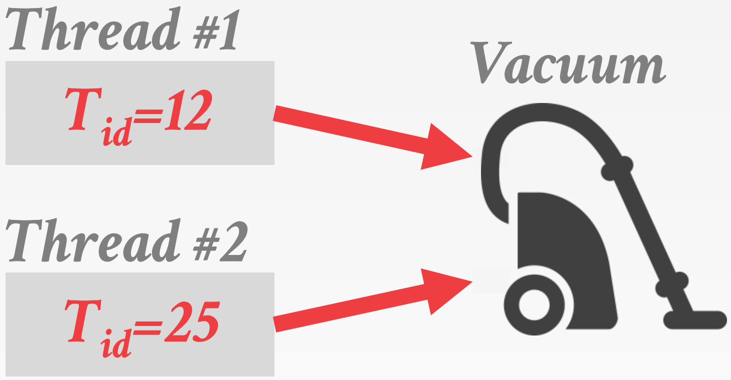
T_{id}=25

Vacuum

	TXN-ID	BEGIN-TS	END-TS
A_1	0	1	9
B_1	0	1	9
B_2	0	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

T_{id}=12

Thread #2

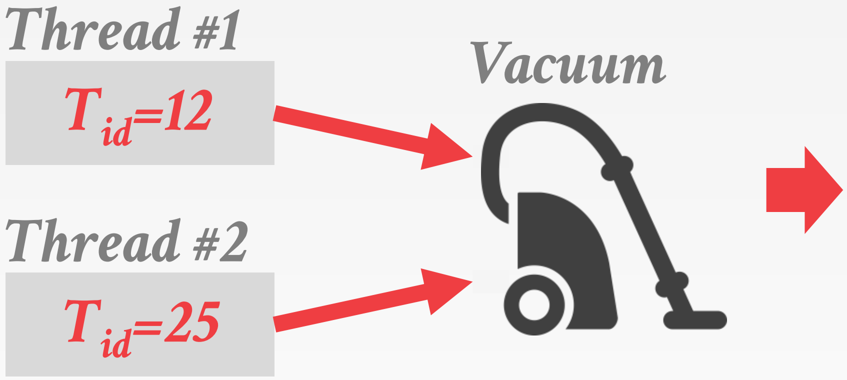
T_{id}=25

Vacuum

	TXN-ID	BEGIN-TS	END-TS
A_1	0	1	9
B_1	0	1	9
B_2	0	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

T_{id}=12

Vacuum

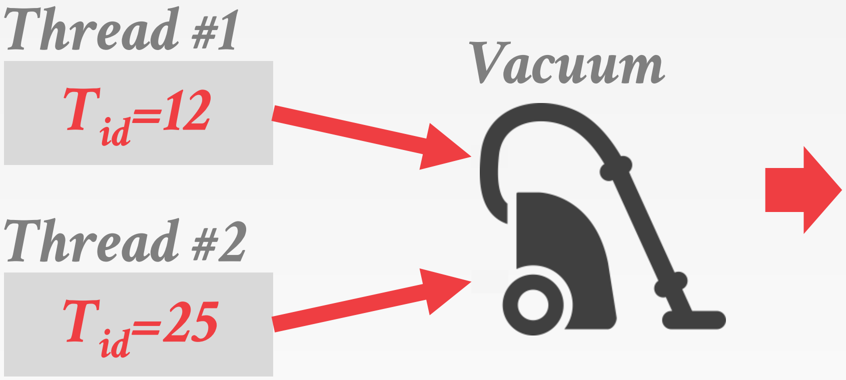
Thread #2

T_{id}=25

	TXN-ID	BEGIN-TS	END-TS
A_1	0	1	9
B_1	0	1	9
B_2	0	10	20

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

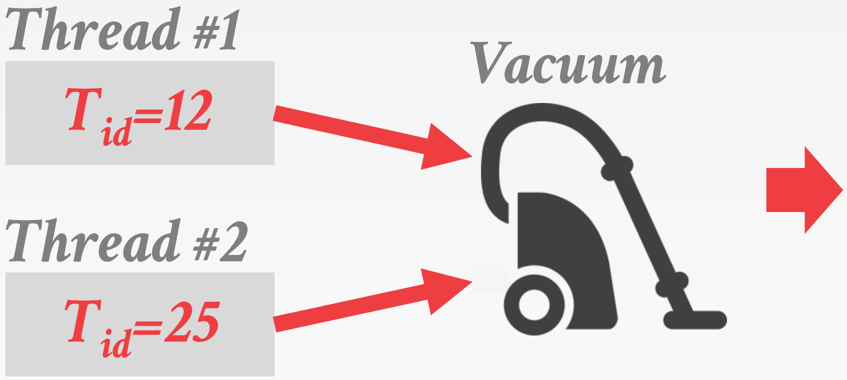
TUPLE-LEVEL GC



	TXN-ID	BEGIN-TS	END-TS
B_2	0	10	20

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

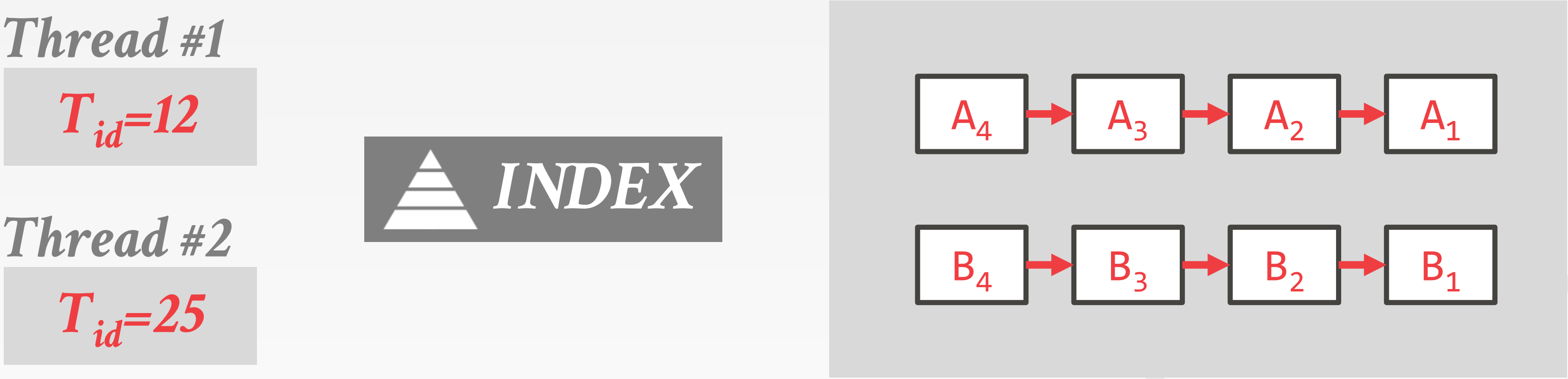
TUPLE-LEVEL GC



Dirty?	TXN-ID	BEGIN-TS	END-TS
B_2	0	10	20

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

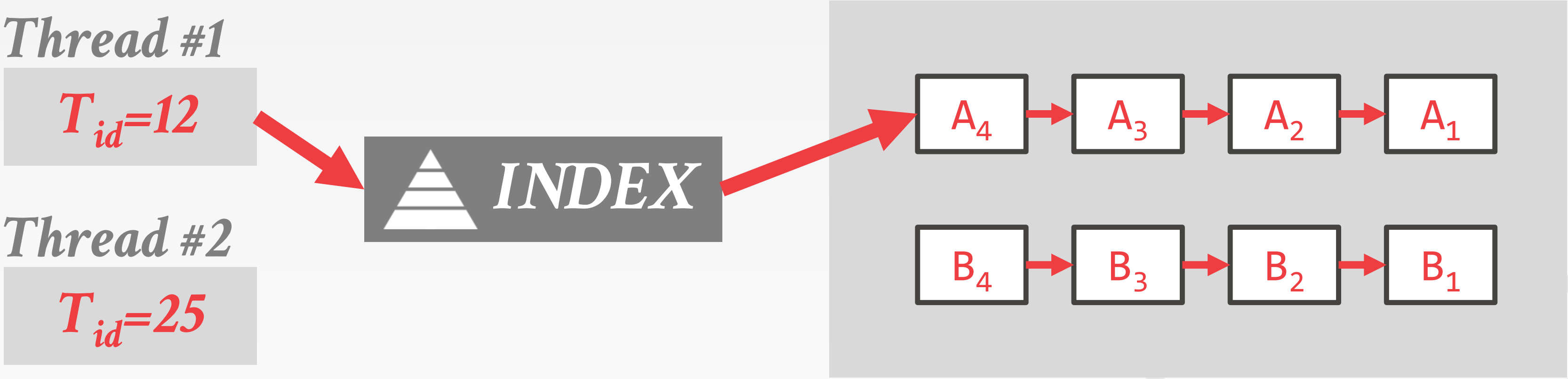
TUPLE-LEVEL GC



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**.

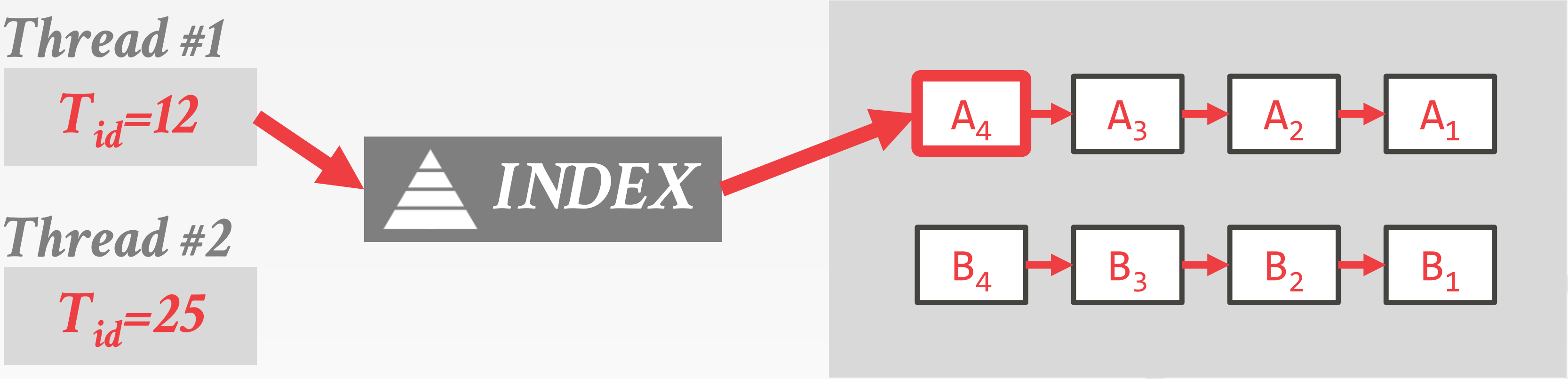
TUPLE-LEVEL GC



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**.

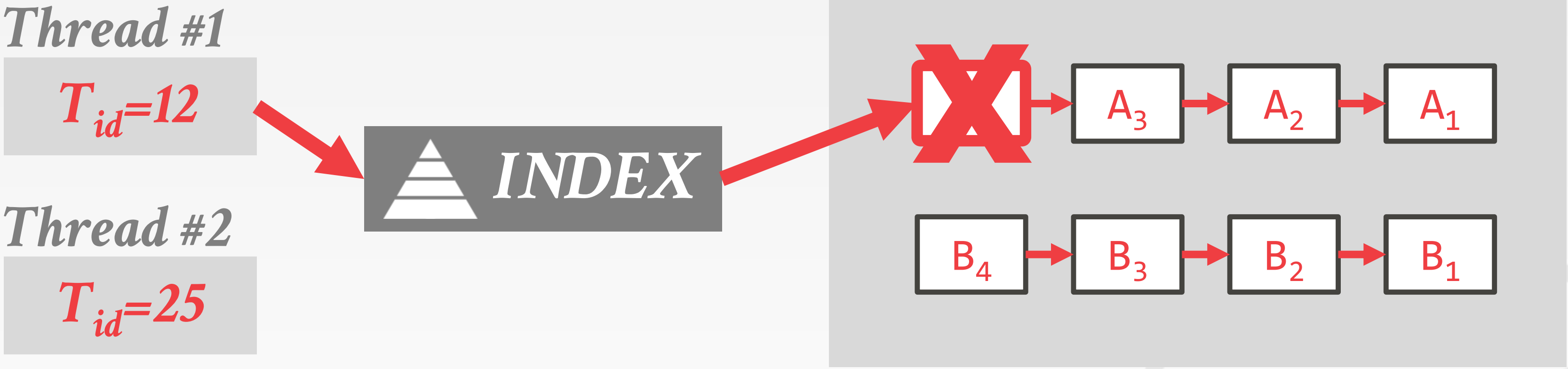
TUPLE-LEVEL GC



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**.

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.

May still require multiple threads to reclaim the memory fast enough for the workload.

OBSERVATION

Thread #1

T_{id}=1

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	$2^{31}-1$	$2^{31}-2$	∞
B_1	0	$2^{31}-1$	$2^{31}-2$	∞

If the DBMS reaches the max value for its timestamps, it will have to wrap around and start at zero. This will make all previous versions be in the "future" from new transactions.

OBSERVATION

Thread #1

T_{id}=1

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	$2^{31}-1$	$2^{31}-2$	∞
B_1	0	$2^{31}-1$	$2^{31}-2$	∞

If the DBMS reaches the max value for its timestamps, it will have to wrap around and start at zero. This will make all previous versions be in the "future" from new transactions.

OBSERVATION

Thread #1

$T_{id}=1$

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	$2^{31}-1$	$2^{31}-2$	∞
B_1	0	$2^{31}-1$	$2^{31}-2$	∞
B_2	0	0	10	∞

If the DBMS reaches the max value for its timestamps, it will have to wrap around and start at zero. This will make all previous versions be in the "future" from new transactions.

POSTGRES TXN ID WRAPAROUND

Stop accepting new commands when the system gets close to the max txn id.

Set a flag in each tuple header that says that it is "frozen" in the past. Any new txn id will always be newer than a frozen version.

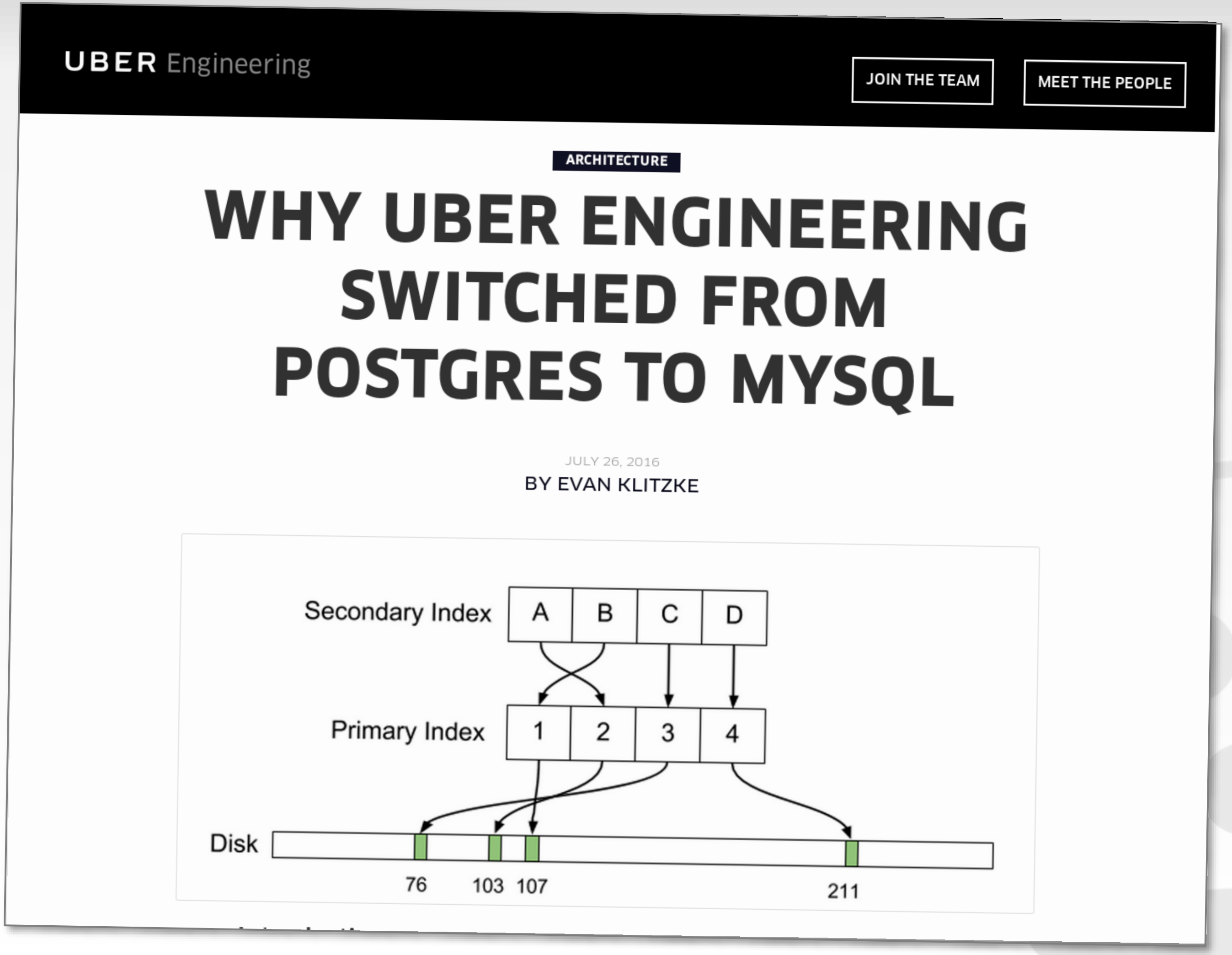
Runs the vacuum before the system gets close to this upper limit.

INDEX MANAGEMENT

PKey indexes always point to version chain head.

- → How often the DBMS has to 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 an **DELETE** followed by an **INSERT**.

Secondary indexes are more complicated…



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

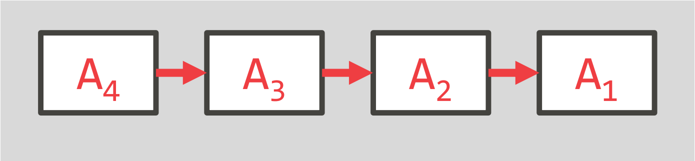
Approach #2: Physical Pointers

→ Use the physical address to the version chain head.

INDEX POINTERS

PRIMARY INDEX

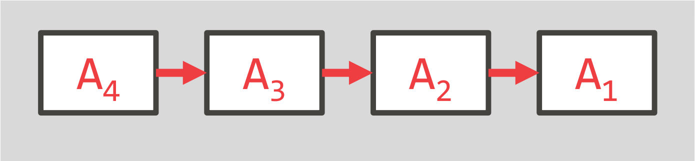
SECONDARY INDEX



INDEX POINTERS

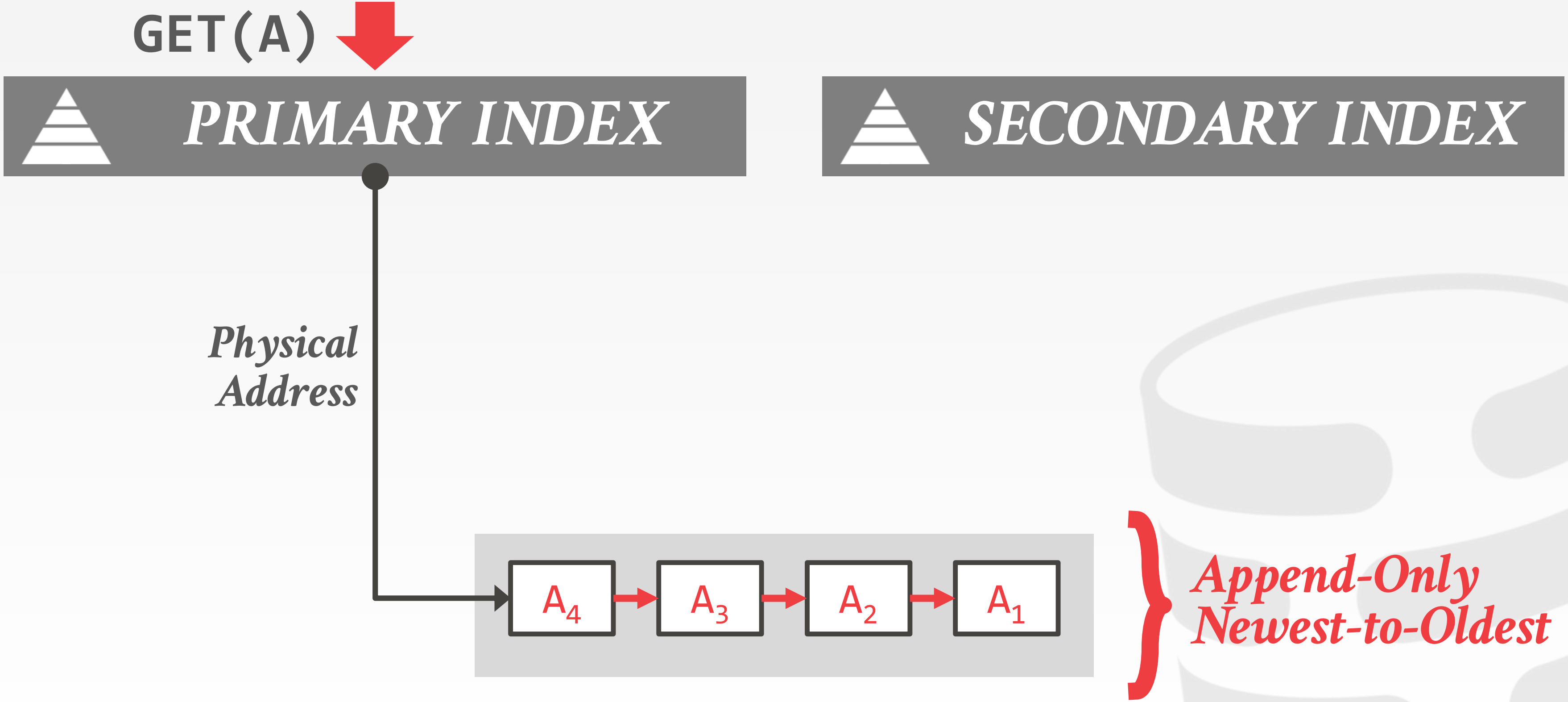
PRIMARY INDEX

SECONDARY INDEX

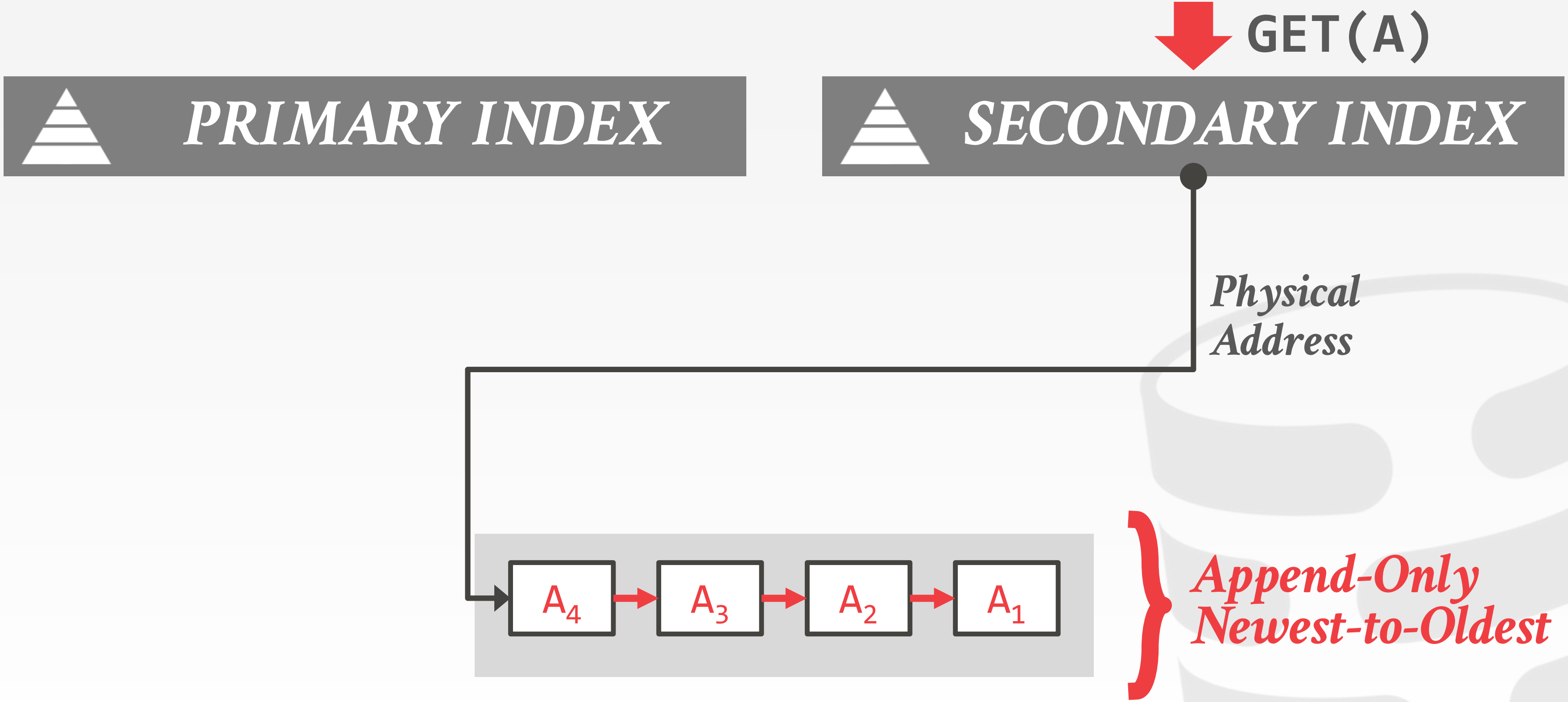


Append-Only
Newest-to-Oldest

INDEX POINTERS

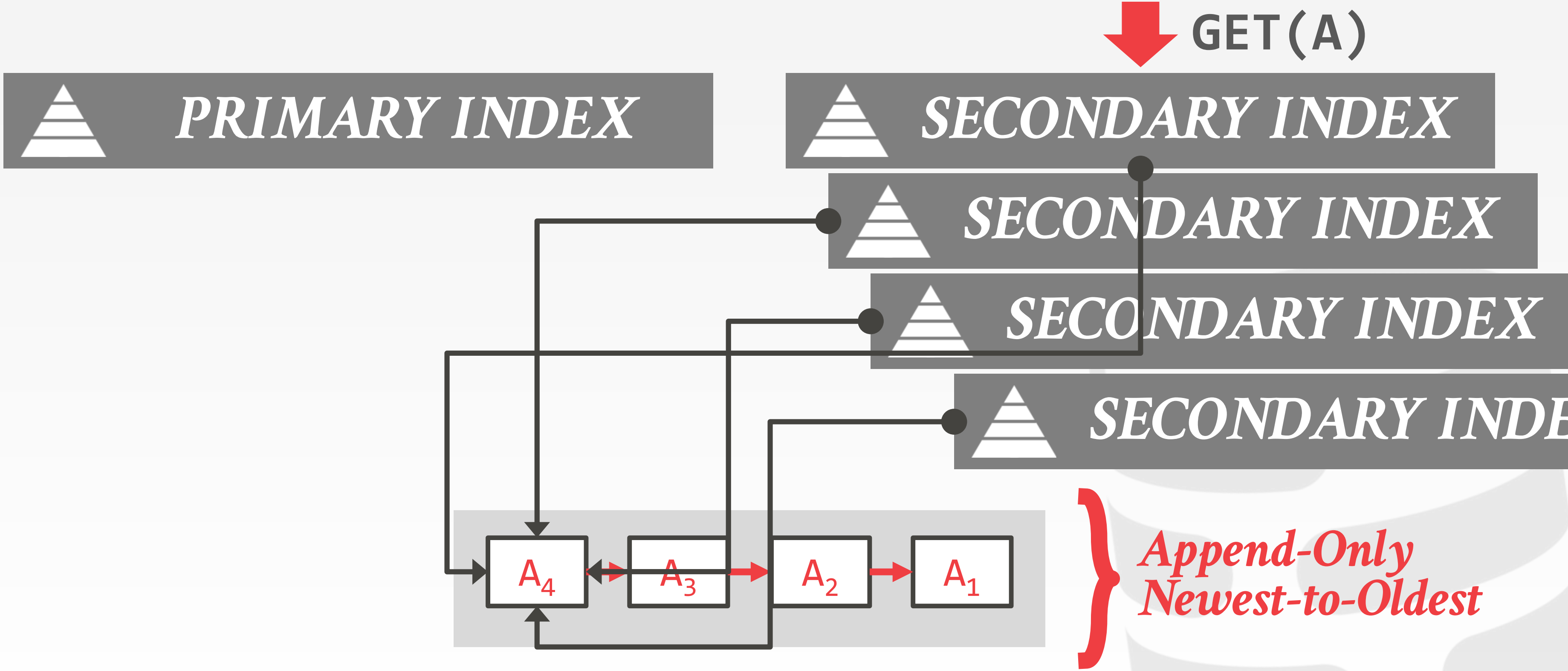


INDEX POINTERS

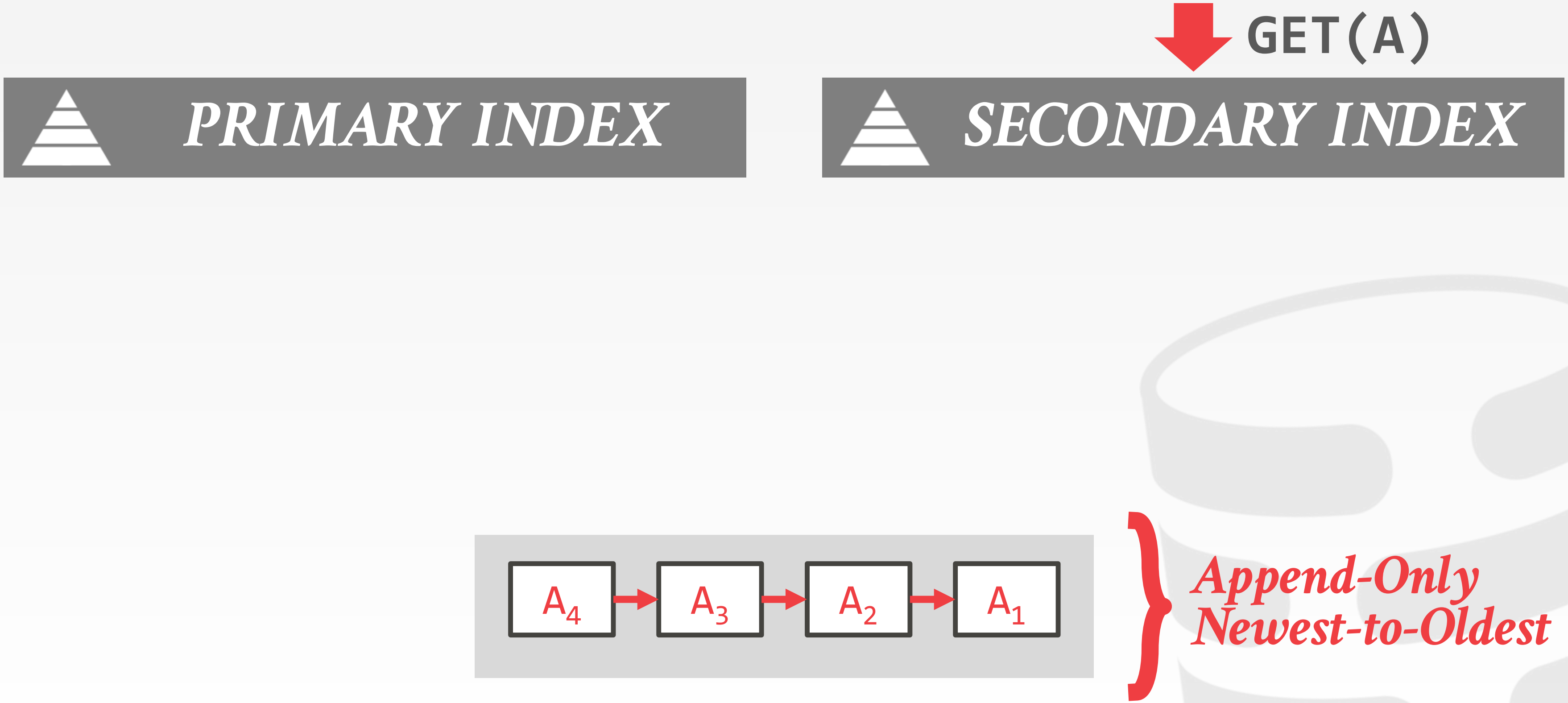


Append-Only
Newest-to-Oldest

INDEX POINTERS



INDEX POINTERS



GET(A)

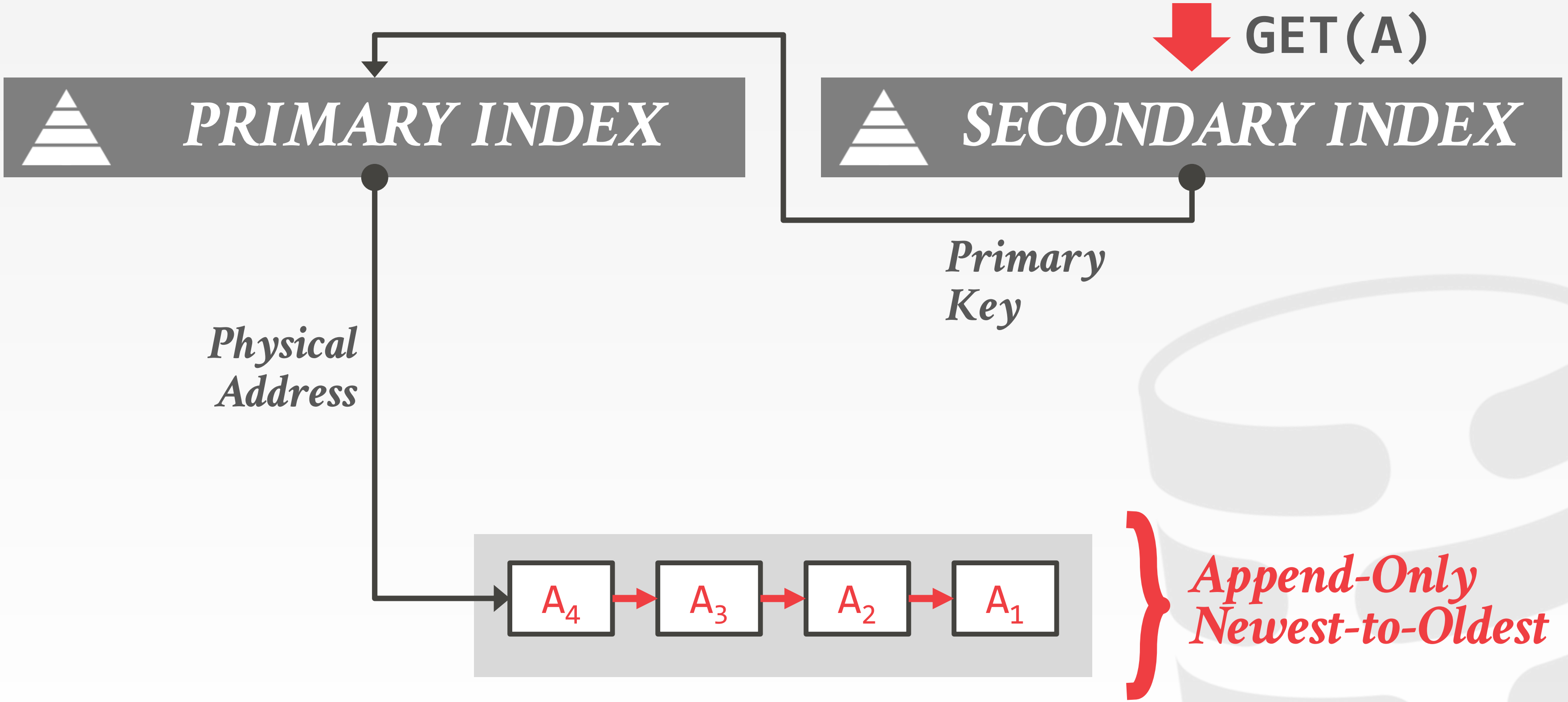
PRIMARY INDEX

SECONDARY INDEX

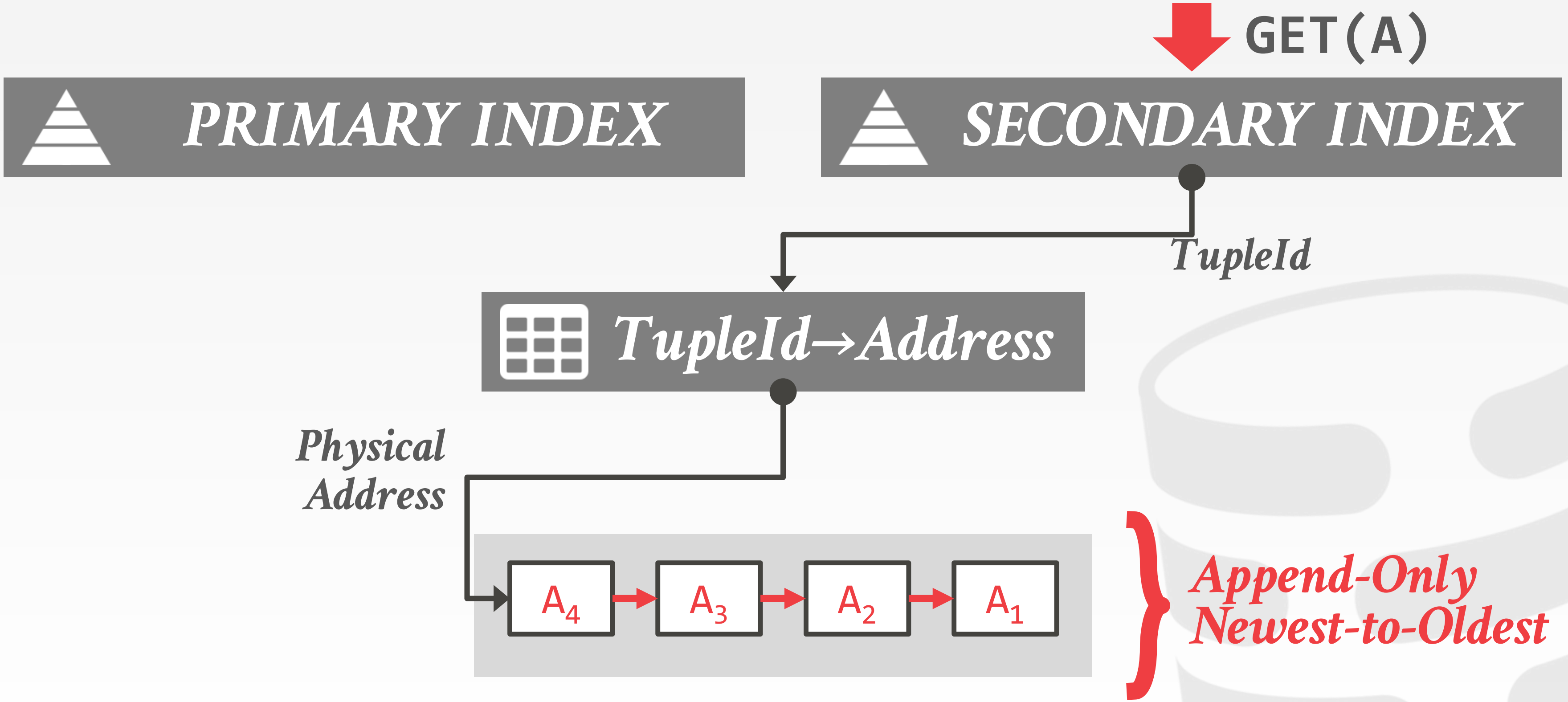
A_4 → A_3 → A_2 → A_1

Append-Only
Newest-to-Oldest

INDEX POINTERS

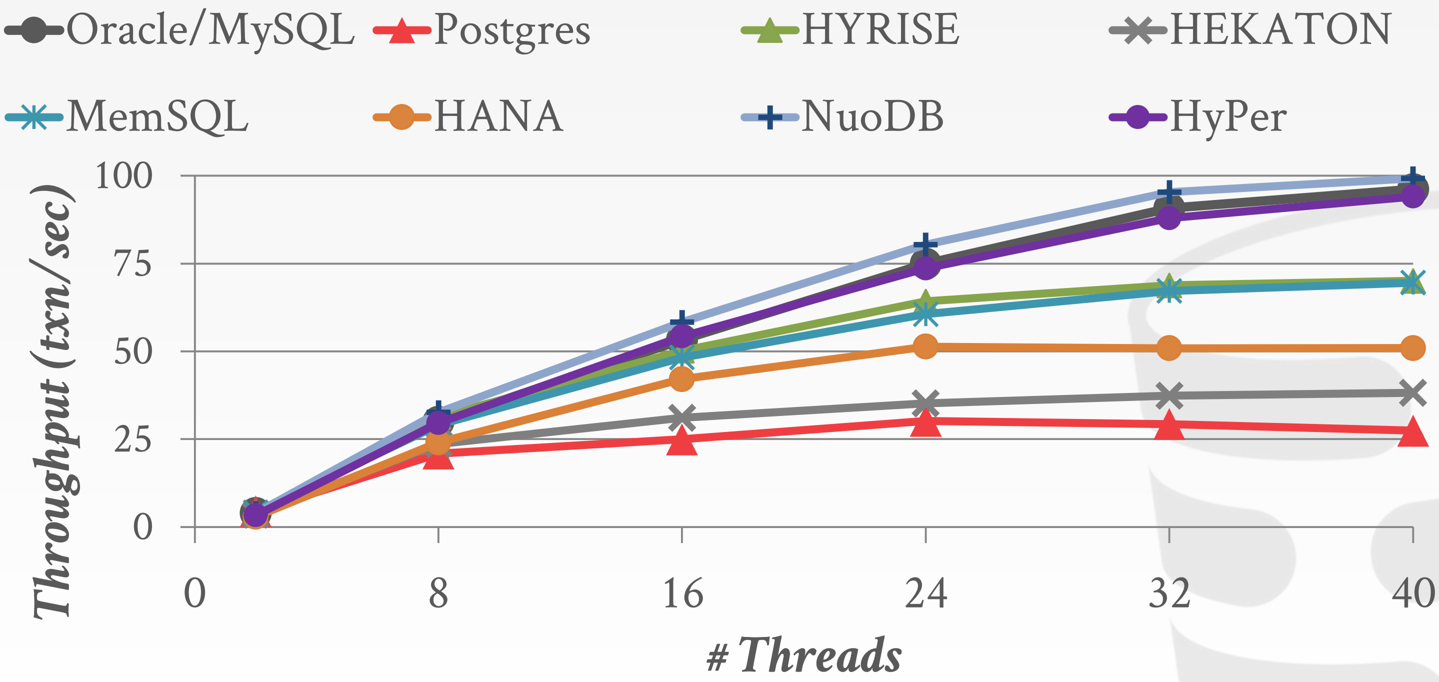


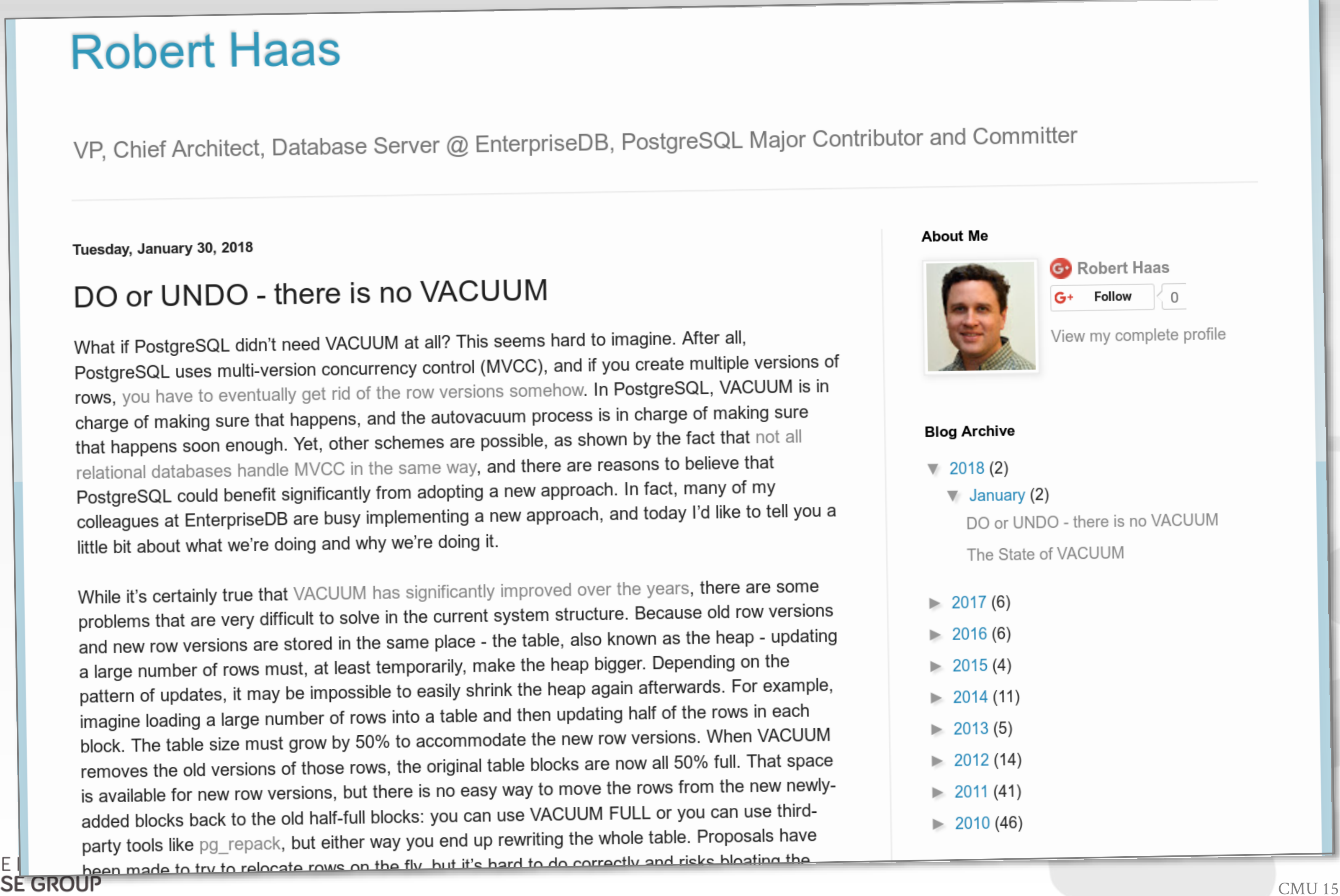
INDEX POINTERS



MVCC CONFIGURATION EVALUATION

Database: TPC-C Benchmark (40 Warehouses)
Processor: 4 sockets, 10 cores per socket





Robert Haas

VP, Chief Architect, Database Server @ EnterpriseDB, PostgreSQL Major Contributor and Committer

Tuesday, January 30, 2018

DO or UNDO - there is no VACUUM

What if PostgreSQL didn't need VACUUM at all? This seems hard to imagine. After all, PostgreSQL uses multi-version concurrency control (MVCC), and if you create multiple versions of rows, you have to eventually get rid of the row versions somehow. In PostgreSQL, VACUUM is in charge of making sure that happens, and the autovacuum process is in charge of making sure that happens soon enough. Yet, other schemes are possible, as shown by the fact that not all relational databases handle MVCC in the same way, and there are reasons to believe that PostgreSQL could benefit significantly from adopting a new approach. In fact, many of my colleagues at EnterpriseDB are busy implementing a new approach, and today I'd like to tell you a little bit about what we're doing and why we're doing it.

While it's certainly true that VACUUM has significantly improved over the years, there are some problems that are very difficult to solve in the current system structure. Because old row versions and new row versions are stored in the same place - the table, also known as the heap - updating a large number of rows must, at least temporarily, make the heap bigger. Depending on the pattern of updates, it may be impossible to easily shrink the heap again afterwards. For example, imagine loading a large number of rows into a table and then updating half of the rows in each block. The table size must grow by 50% to accommodate the new row versions. When VACUUM removes the old versions of those rows, the original table blocks are now all 50% full. That space is available for new row versions, but there is no easy way to move the rows from the new newly-added blocks back to the old half-full blocks: you can use VACUUM FULL or you can use third-party tools like pg_repack, but either way you end up rewriting the whole table. Proposals have been made to try to relocate rows on the fly, but it's hard to do correctly and risks bloating the

About Me

Robert Haas

Follow 0

View my complete profile

Blog Archive

- ▼ 2018 (2)
  - ▼ January (2)
    - DO or UNDO - there is no VACUUM
    - The State of VACUUM
- ► 2017 (6)
- ► 2016 (6)
- ► 2015 (4)
- ► 2014 (11)
- ► 2013 (5)
- ► 2012 (14)
- ► 2011 (41)
- ► 2010 (46)

PARTING THOUGHTS

MVCC is currently the best approach for supporting txns in mixed workoads

We only discussed MVCC for OLTP.
→ Design decisions may be different for HTAP

Interesting MVCC research/project Topics:
→ Block compaction
→ Version compression
→ On-line schema changes

PROJECT #2

Implement a latch-free Skip List in Peloton.

→ Forward / Reverse Iteration

→ Garbage Collection

Must be able to support both unique and non-unique keys.

PROJECT #2 – DESIGN

We will provide you with a header file with the index API that you have to implement.

→ Data serialization and predicate evaluation will be taken care of for you.

There are several design decisions that you are going to have to make.

→ There is no right answer.

→ Do not expect us to guide you at every step of the development process.

PROJECT #2 – TESTING

We are providing you with C++ unit tests for you to check your implementation.

We also have a BwTree implementation to compare against.

We **strongly** encourage you to do your own additional testing.

PROJECT #2 – DOCUMENTATION

You must write sufficient documentation and comments in your code to explain what you are doing in all different parts.

We will inspect the submissions manually.

PROJECT #2 – GRADING

We will run additional tests beyond what we provided you for grading.

→ Bonus points will be given to the groups with the fastest implementation.

→ We will use Valgrind when testing your code.

All source code must pass ClangFormat syntax formatting checker.

→ See Peloton documentation for formatting guidelines.

PROJECT #2 – GROUPS

This is a group project.

→ Everyone should contribute equally.

→ I will review commit history.

Email me if you do not have a group.

PROJECT #2

Due Date: March 12th @ 11:59pm

Projects will be turned in using Autolab.

Full description and instructions:

http://15721.courses.cs.cmu.edu/spring2018/project2.html

NEXT CLASS

Modern MVCC Implementations

- → CMU Cicada
- → Microsoft Hekaton
- → TUM HyPer
- → Serializable Snapshot Isolation