Lecture #03

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Multi-Version Concurrency Control (Design Decisions)

@Andy_Pavlo // 15-721 // Spring 2020

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

Writers don't block readers.
Readers don't block writers.

Read-only txns can read a consistent **snapshot** without acquiring locks or txn ids.

→ Use timestamps to determine visibility.

Easily support **time-travel** queries.

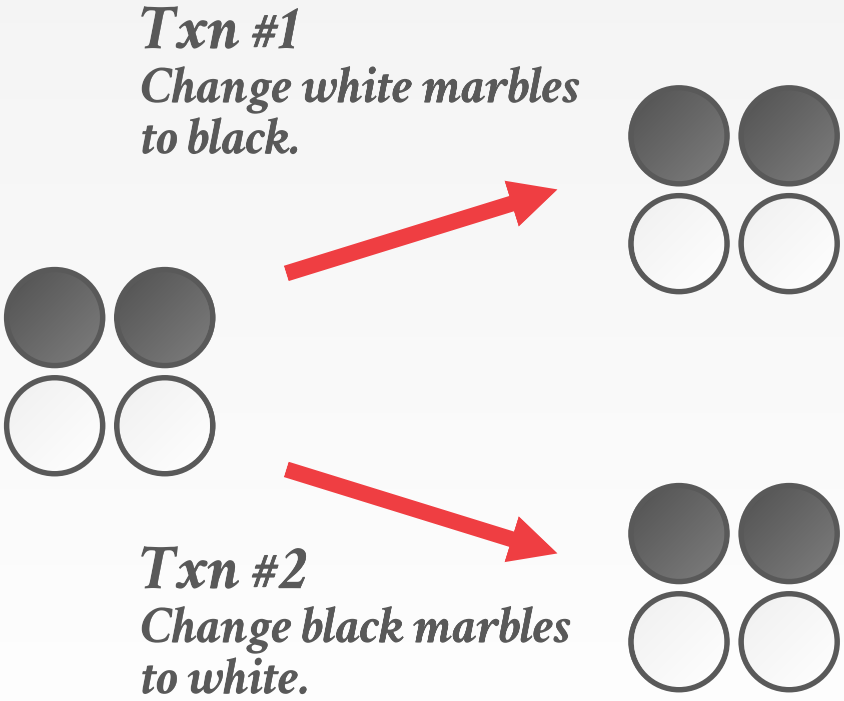
SNAPSHOT ISOLATION (SI)

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 first writer wins.

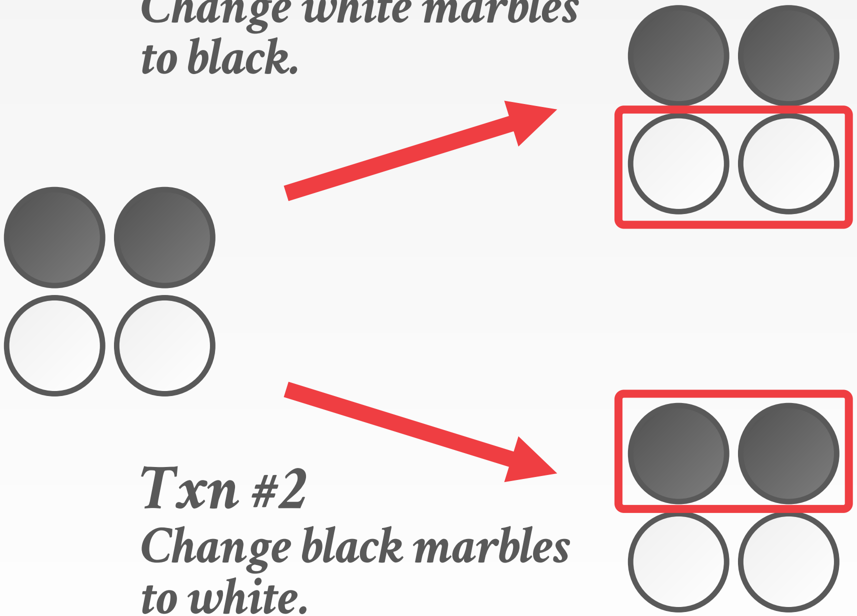
SI is susceptible to the **Write Skew Anomaly**.

WRITE SKEW ANOMALY



WRITE SKEW ANOMALY

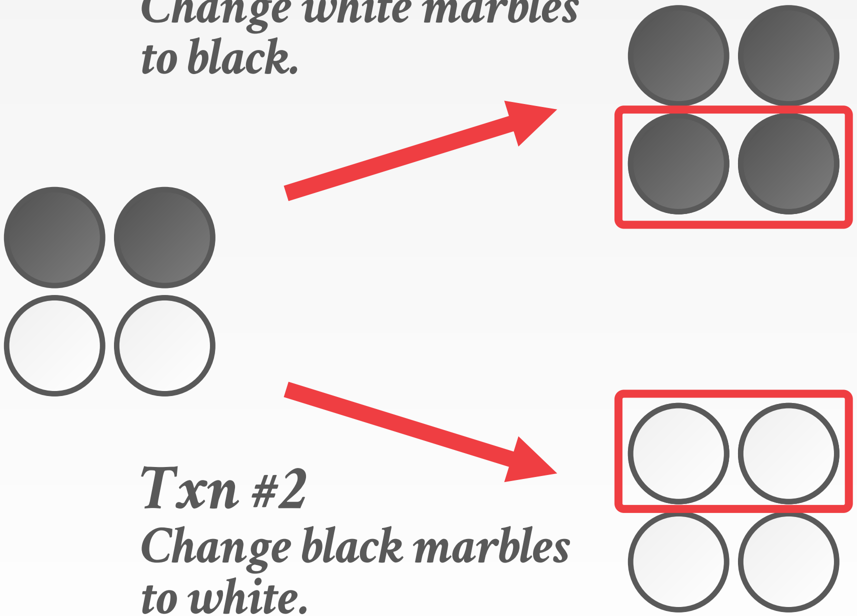
Txn #1
Change white marbles to black.



Txn #2
Change black marbles to white.

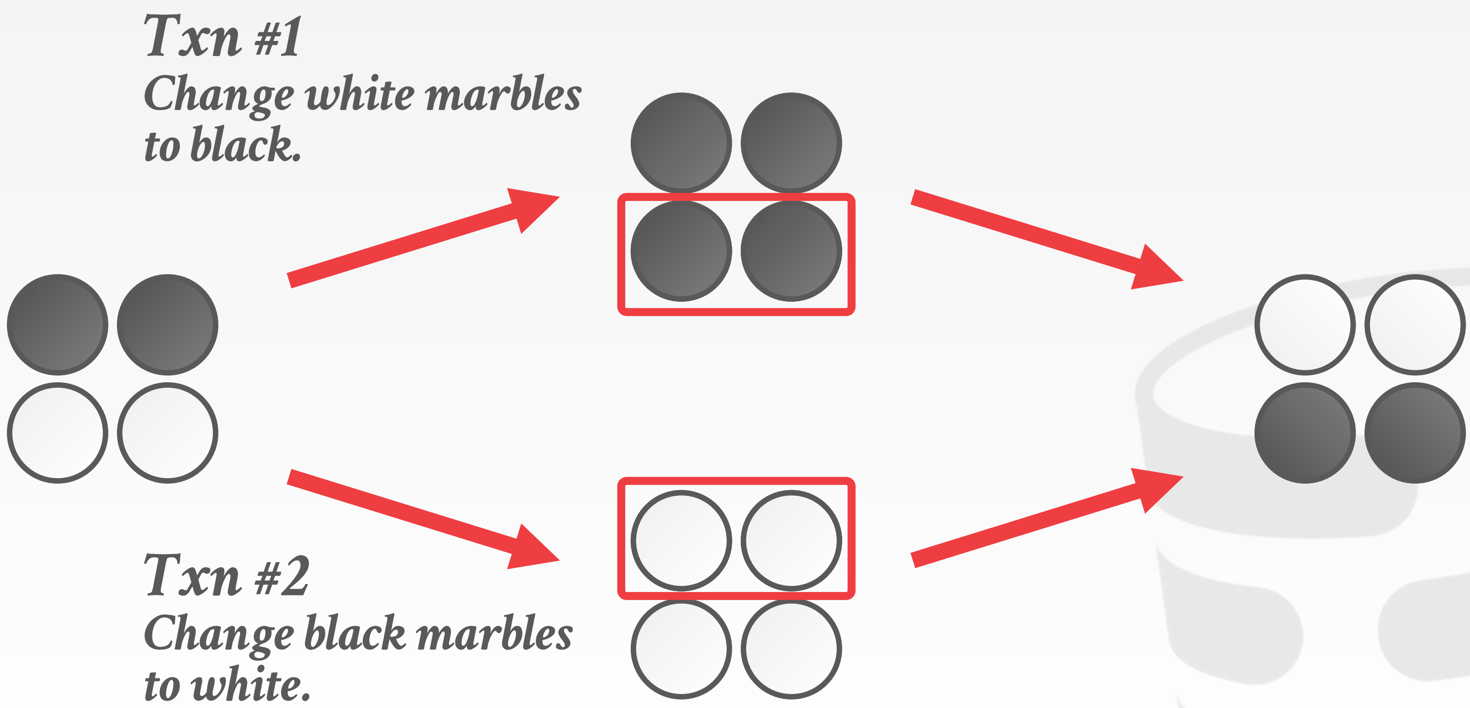
WRITE SKEW ANOMALY

Txn #1
Change white marbles to black.



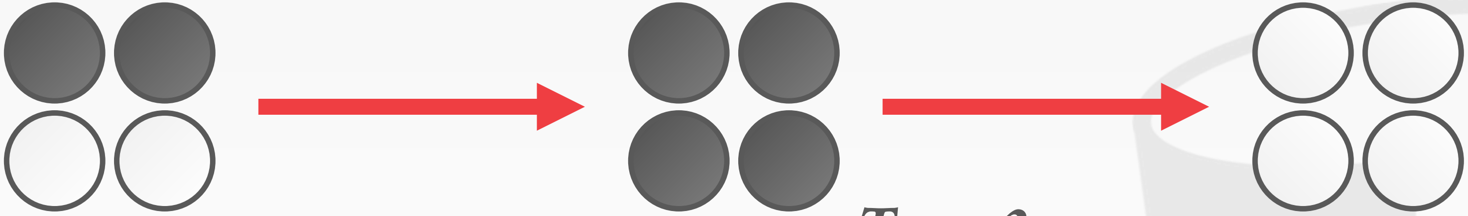
Txn #2
Change black marbles to white.

WRITE SKEW ANOMALY



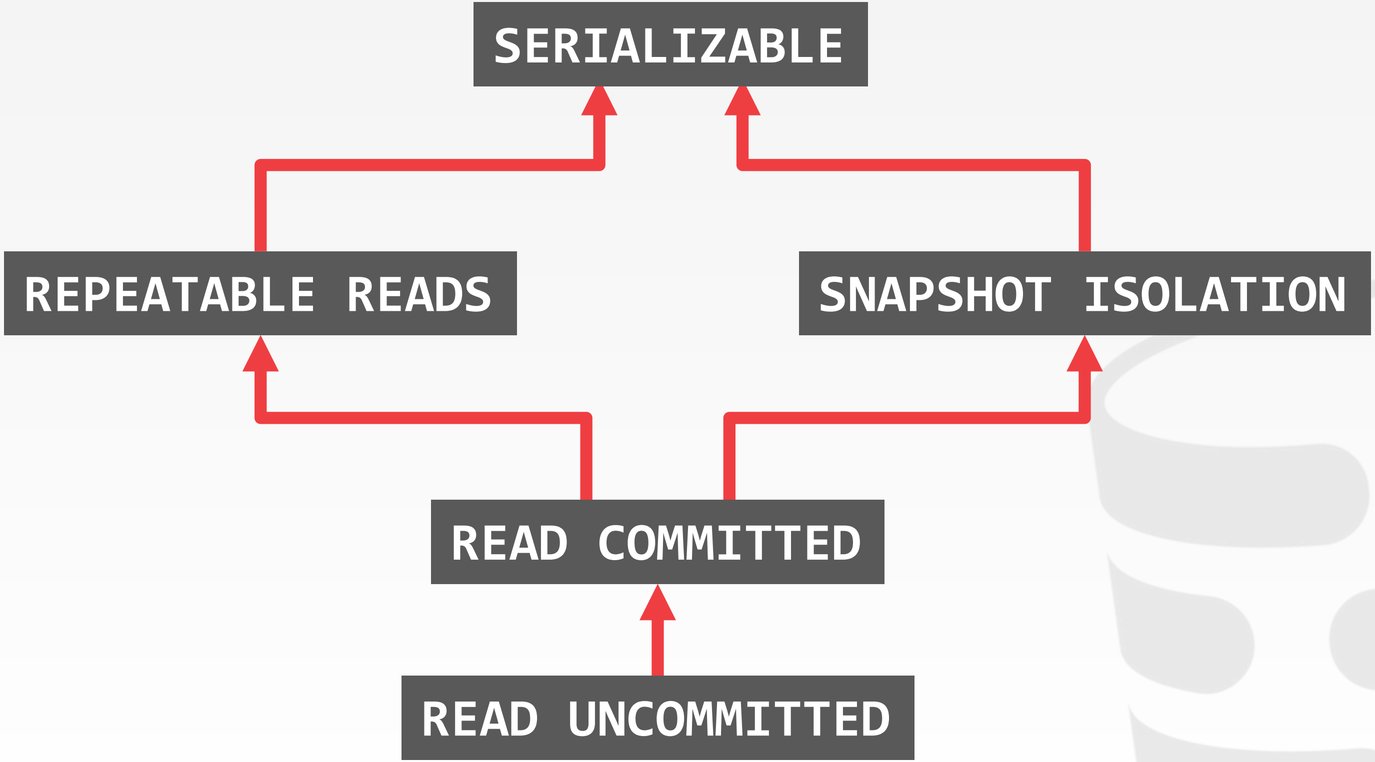
WRITE SKEW ANOMALY

Txn #1
Change white marbles to black.

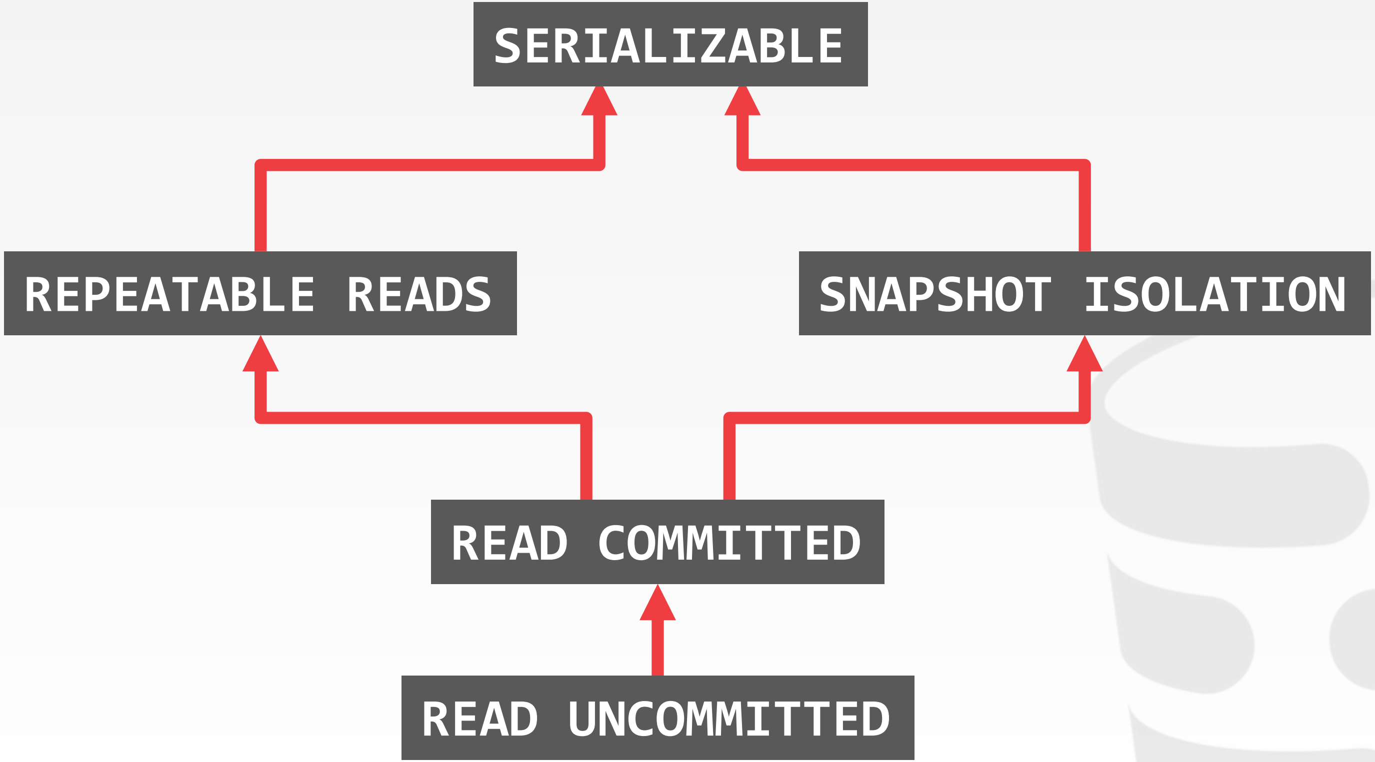


Txn #2
Change black marbles to white.

ISOLATION LEVEL HIERARCHY



ISOLATION LEVEL HIERARCHY



MVCC DESIGN DECISIONS

Concurrency Control Protocol

Version Storage

Garbage Collection

Index Management

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

CONCURRENCY CONTROL PROTOCOL

Approach #1: Timestamp Ordering

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

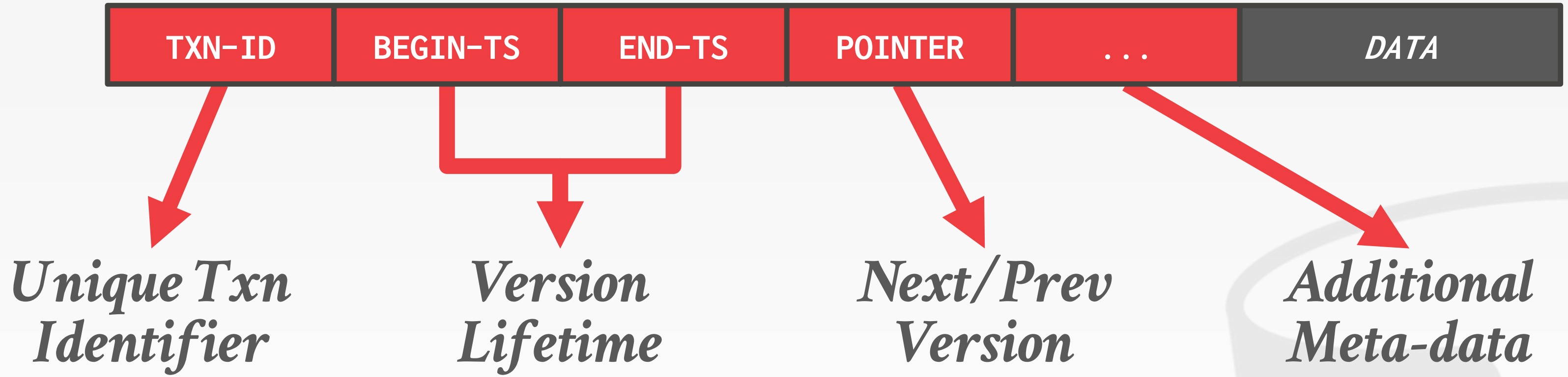
Approach #2: Optimistic Concurrency Control

→ 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.

TUPLE FORMAT



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)

Thread #1

T_{id}=10

READ(A)

WRITE(B)

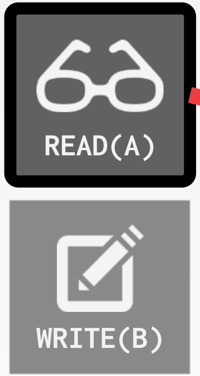
	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	1	1	∞
B_1	0	0	1	∞

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

TIMESTAMP ORDERING (MVTO)

Thread #1

T_{id}=10



	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 can read version if the latch is unset and its T_{id} is between ***begin-ts*** and ***end-ts***.

TIMESTAMP ORDERING (MVTO)

Thread #1

$T_{id}=10$

READ(A)

WRITE(B)

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	$\boxed{1}$	1	∞
B_1	0	0	1	∞

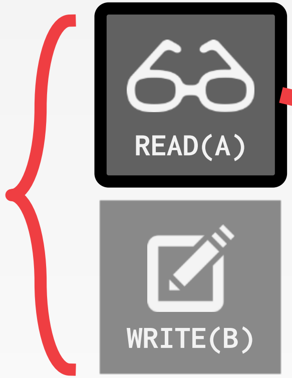
Use ***read-ts*** field in the header to keep track of the timestamp of the last txn that read it.

Txn can read version if the latch is unset and its T_{id} is between ***begin-ts*** and ***end-ts***.

TIMESTAMP ORDERING (MVTO)

Thread #1

T_{id}=10



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

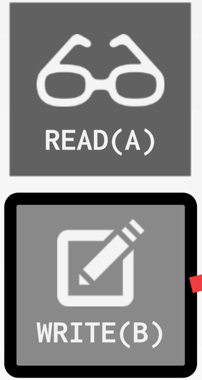
Use ***read-ts*** field in the header to keep track of the timestamp of the last txn that read it.

Txn can read version if the latch is unset and its T_{id} is between ***begin-ts*** and ***end-ts***.

TIMESTAMP ORDERING (MVTO)

Thread #1

T_{id}=10



	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.

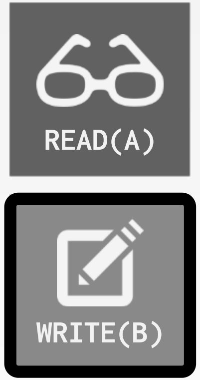
Txn can read version if the latch is unset and its T_{id} is between ***begin-ts*** and ***end-ts***.

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

TIMESTAMP ORDERING (MVTO)

Thread #1

T_{id}=10



	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.

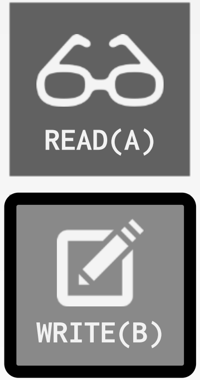
Txn can read version if the latch is unset and its T_{id} is between ***begin-ts*** and ***end-ts***.

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

TIMESTAMP ORDERING (MVTO)

Thread #1

T_{id}=10



	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 last txn that read it.

Txn can read version if the latch is unset and its T_{id} is between ***begin-ts*** and ***end-ts***.

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

TIMESTAMP ORDERING (MVTO)

Thread #1

T_{id}=10

READ(A)

WRITE(B)

	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 can read version if the latch is unset and its T_{id} is between ***begin-ts*** and ***end-ts***.

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

TIMESTAMP ORDERING (MVTO)

Thread #1

T_{id}*=10*

READ(A)

WRITE(B)

	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.

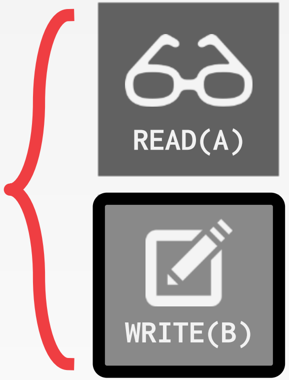
Txn can read version if the latch is unset and its T_{id} is between ***begin-ts*** and ***end-ts***.

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

TIMESTAMP ORDERING (MVTO)

Thread #1

T_{id}=10



	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 can read version if the latch is unset and its T_{id} is between ***begin-ts*** and ***end-ts***.

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

TWO-PHASE LOCKING (MV2PL)

Thread #1

$T_{id}=10$

READ(A)

WRITE(B)

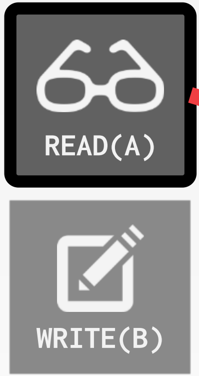
	TXN-ID	READ-CNT	BEGIN-TS	END-TS
A_1	0	0	1	∞
B_1	0	0	1	∞

Txns use the tuple's ***read-cnt*** field as SHARED lock. Use ***txn-id*** and ***read-cnt*** together as EXCLUSIVE lock.

TWO-PHASE LOCKING (MV2PL)

Thread #1

$T_{id}=10$



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

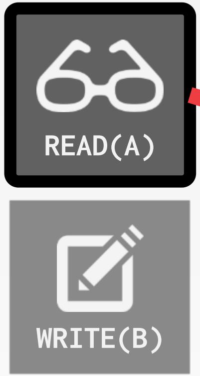
Txns use the tuple's ***read-cnt*** field as SHARED lock. Use ***txn-id*** and ***read-cnt*** together as EXCLUSIVE lock.

If ***txn-id*** is zero, then the txn acquires the SHARED lock by incrementing the ***read-cnt*** field.

TWO-PHASE LOCKING (MV2PL)

Thread #1

$T_{id}=10$



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

Txns use the tuple's ***read-cnt*** field as SHARED lock. Use ***txn-id*** and ***read-cnt*** together as EXCLUSIVE lock.

If ***txn-id*** is zero, then the txn acquires the SHARED lock by incrementing the ***read-cnt*** field.

TWO-PHASE LOCKING (MV2PL)

Thread #1

T_{id}=10

READ(A)

WRITE(B)

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

Txns use the tuple's ***read-cnt*** field as SHARED lock. Use ***txn-id*** and ***read-cnt*** together as EXCLUSIVE lock.

If ***txn-id*** is zero, then the txn acquires the SHARED lock by incrementing the ***read-cnt*** field.

TWO-PHASE LOCKING (MV2PL)

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

Txns use the tuple's ***read-cnt*** field as SHARED lock. Use ***txn-id*** and ***read-cnt*** together as EXCLUSIVE lock.

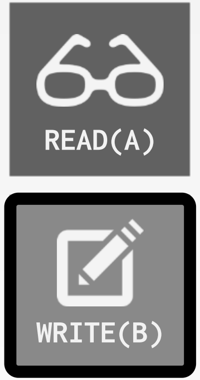
If ***txn-id*** is zero, then the txn acquires the SHARED lock by incrementing the ***read-cnt*** field.

If both ***txn-id*** and ***read-cnt*** are zero, then txn acquires the EXCLUSIVE lock by setting both of them.

TWO-PHASE LOCKING (MV2PL)

Thread #1

T_{id}=10



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

Txns use the tuple's ***read-cnt*** field as SHARED lock. Use ***txn-id*** and ***read-cnt*** together as EXCLUSIVE lock.

If ***txn-id*** is zero, then the txn acquires the SHARED lock by incrementing the ***read-cnt*** field.

If both ***txn-id*** and ***read-cnt*** are zero, then txn acquires the EXCLUSIVE lock by setting both of them.

TWO-PHASE LOCKING (MV2PL)

Thread #1

T_{id}=10

READ(A)

WRITE(B)

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

Txns use the tuple's ***read-cnt*** field as SHARED lock. Use ***txn-id*** and ***read-cnt*** together as EXCLUSIVE lock.

If ***txn-id*** is zero, then the txn acquires the SHARED lock by incrementing the ***read-cnt*** field.

If both ***txn-id*** and ***read-cnt*** are zero, then txn acquires the EXCLUSIVE lock by setting both of them.

TWO-PHASE LOCKING (MV2PL)

Thread #1

T_{id}=10

READ(A)

WRITE(B)

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

Txns use the tuple's ***read-cnt*** field as SHARED lock. Use ***txn-id*** and ***read-cnt*** together as EXCLUSIVE lock.

If ***txn-id*** is zero, then the txn acquires the SHARED lock by incrementing the ***read-cnt*** field.

If both ***txn-id*** and ***read-cnt*** are zero, then txn acquires the EXCLUSIVE lock by setting both of them.

TWO-PHASE LOCKING (MV2PL)

Thread #1

T_{id}=10

READ(A)

WRITE(B)

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

Txns use the tuple's ***read-cnt*** field as SHARED lock. Use ***txn-id*** and ***read-cnt*** together as EXCLUSIVE lock.

If ***txn-id*** is zero, then the txn acquires the SHARED lock by incrementing the ***read-cnt*** field.

If both ***txn-id*** and ***read-cnt*** are zero, then txn acquires the EXCLUSIVE lock by setting both of them.

OBSERVATION

Thread #1

$T_{id}=2^{31}\text{-}1$

WRITE(A)

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	-	99999	∞

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

OBSERVATION

Thread #1

$T_{id}=2^{31}-1$

WRITE(A)

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	$2^{31}-1$	-	99999	$2^{31}-1$
A_2	$2^{31}-1$	-	$2^{31}-1$	∞

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

OBSERVATION

Thread #1

$T_{id}=2^{31}-1$

WRITE(A)

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	-	99999	$2^{31}-1$
A_2	0	-	$2^{31}-1$	∞

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

OBSERVATION

Thread #1

$T_{id}=2^{31}-1$

Thread #2

$T_{id}=1$

WRITE(A)

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	-	99999	$2^{31}-1$
A_2	0	-	$2^{31}-1$	∞

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

OBSERVATION

Thread #1

$T_{id}=2^{31}-1$

Thread #2

$T_{id}=1$

WRITE(A)

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	-	99999	$2^{31}-1$
A_2	1	-	$2^{31}-1$	1
A_3	1	-	1	∞

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

OBSERVATION

Thread #1

$T_{id}=2^{31}-1$

Thread #2

$T_{id}=1$

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	-	99999	$2^{31}-1$
A_2	0	-	$\boxed{2^{31}-1}$	$\boxed{1}$
A_3	0	-	1	∞

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

POSTGRES TXN ID WRAPAROUND

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.

Otherwise it must stop accepting new commands when the system gets close to the max txn id.

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.

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 mixed together.

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 mixed together.

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 mixed together.

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	●
B_1	\$10	Ø
A_2	\$333	Ø

VERSION CHAIN ORDERING

Approach #1: Oldest-to-Newest (O2N)
→ Append every 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.
→ Don't have to traverse chain on look ups.

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

TIME-TRAVEL STORAGE

Main Table

	VALUE	POINTER
A_2	*$222*	•
B_1	*$10*	

Time-Travel Table

	VALUE	POINTER
A_1	*$111*	Ø

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	

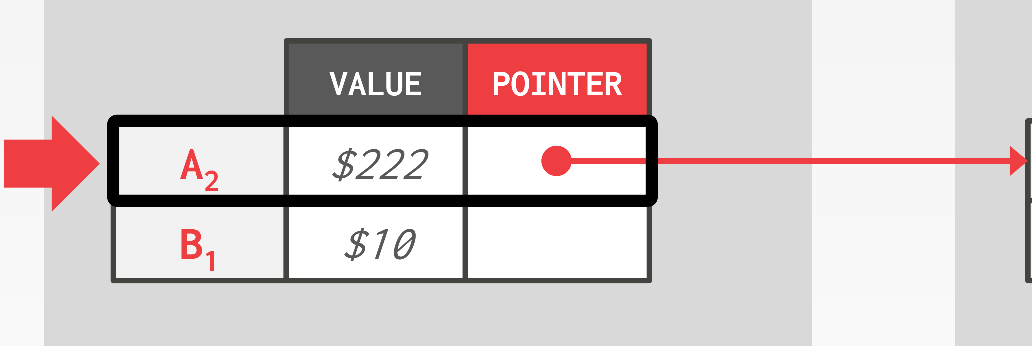
Time-Travel Table

	VALUE	POINTER
A_1	$111	Ø
A_2	$222	●

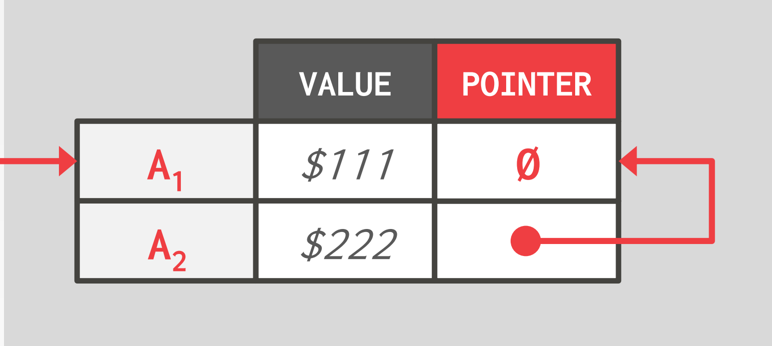
On every update, copy the current version to the time-travel table. Update pointers.

TIME-TRAVEL STORAGE

Main Table



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	

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

	VALUE	POINTER
A_3	*$333*	
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

	VALUE	POINTER
A_3	*$333*	•
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.

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_1	$111	
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*	•
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	
B_1	$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

	VALUE	POINTER
A_3	$333	●
B_1	$10	

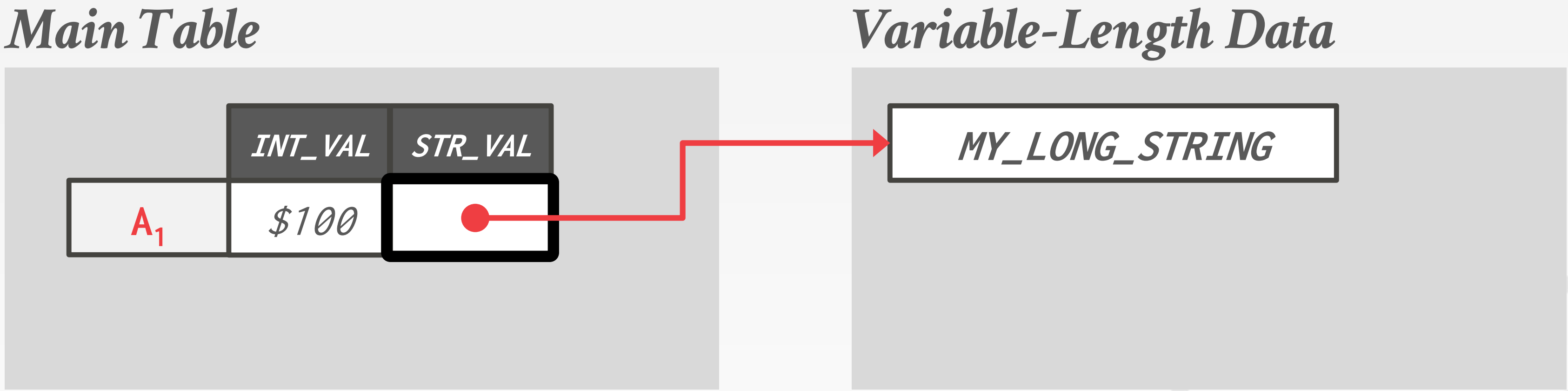
On every update, copy only the values that were modified to the delta storage and overwrite the master version.

Delta Storage Segment

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

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

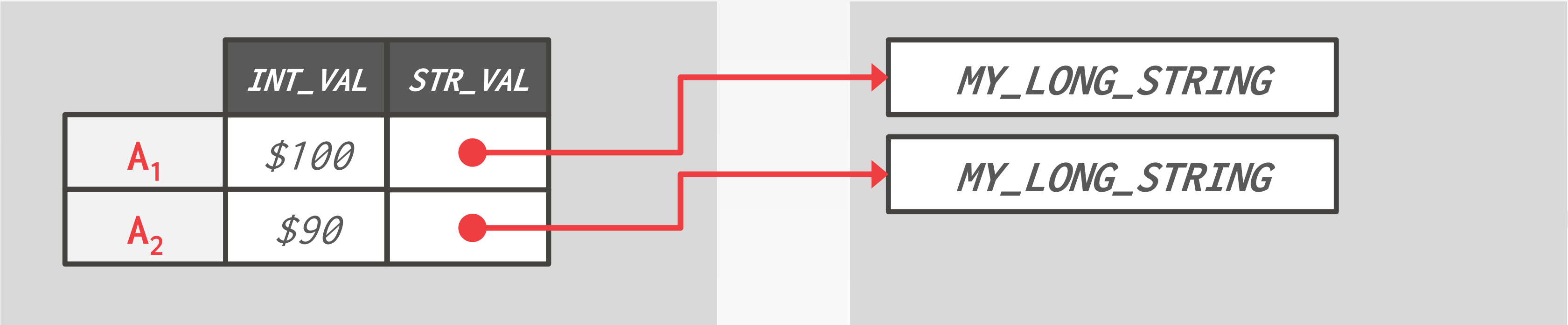
NON-INLINE ATTRIBUTES



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

NON-INLINE ATTRIBUTES

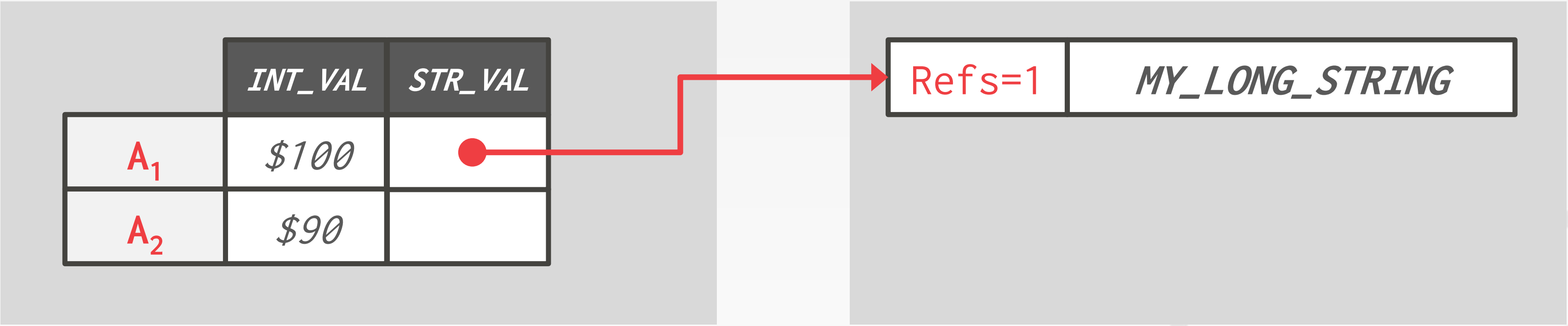
Main Table | *Variable-Length Data*



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

NON-INLINE ATTRIBUTES

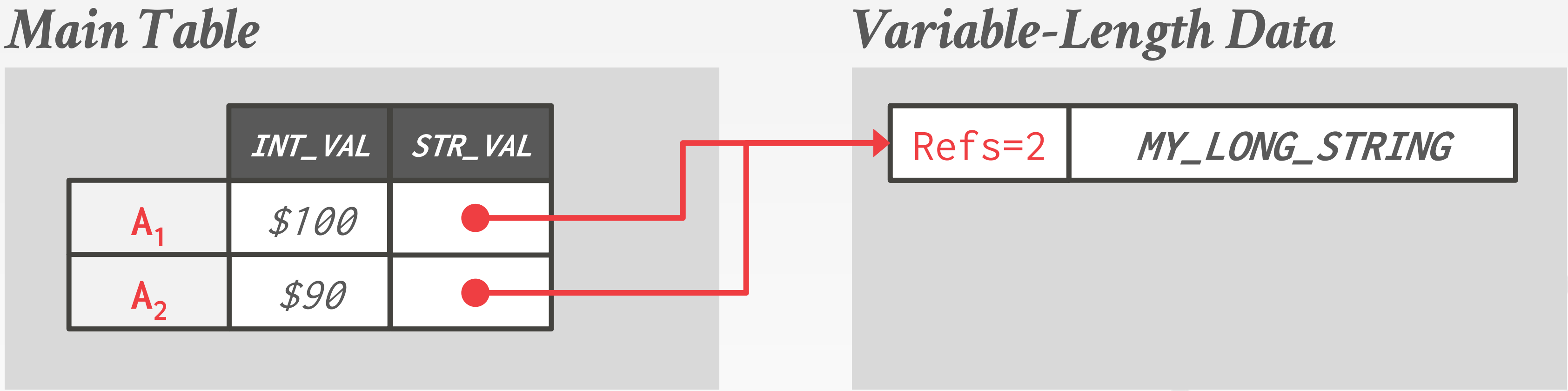
Main Table ***Variable-Length Data***



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

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

NON-INLINE ATTRIBUTES



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

Requires reference counters to know when it is 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.

Three additional design decisions:
→ How to look for expired versions?
→ How to decide when it is safe to reclaim memory?
→ Where to look for expired versions?

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

	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

Thread #1

T_{id}=12

Thread #2

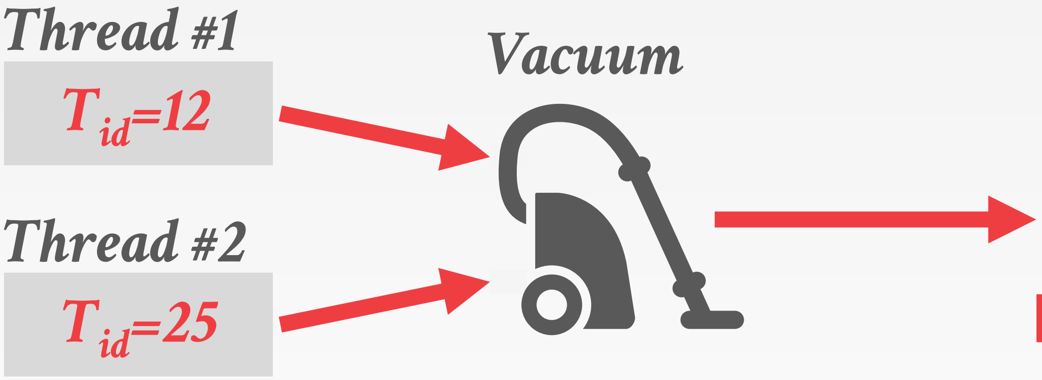
T_{id}=25

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.

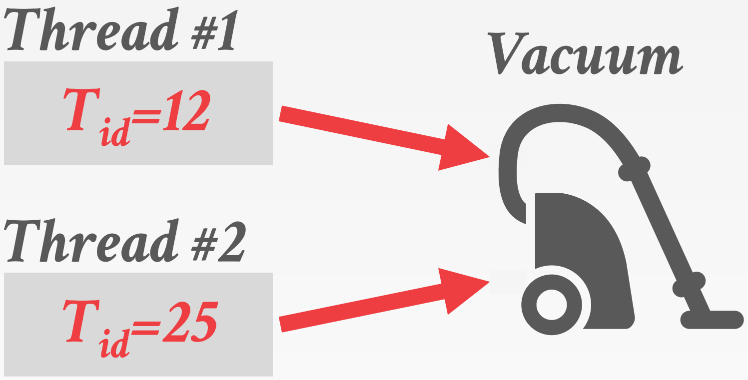
TUPLE-LEVEL GC



	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.

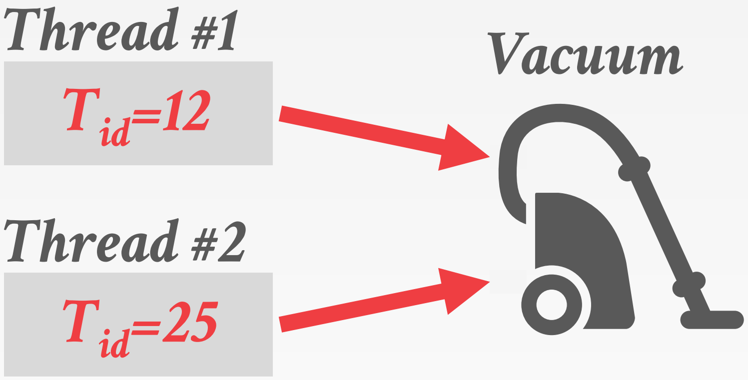
TUPLE-LEVEL GC

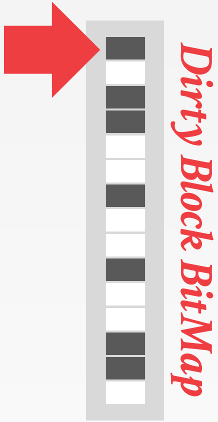


	BEGIN-TS	END-TS
B_{101}	10	20

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
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

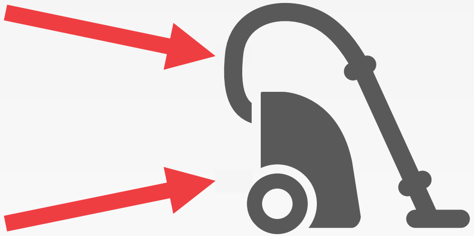
Thread #1

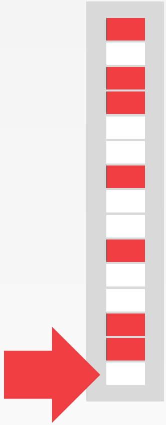
T_{id}=12

Thread #2

T_{id}=25

Vacuum





	BEGIN-TS	END-TS
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

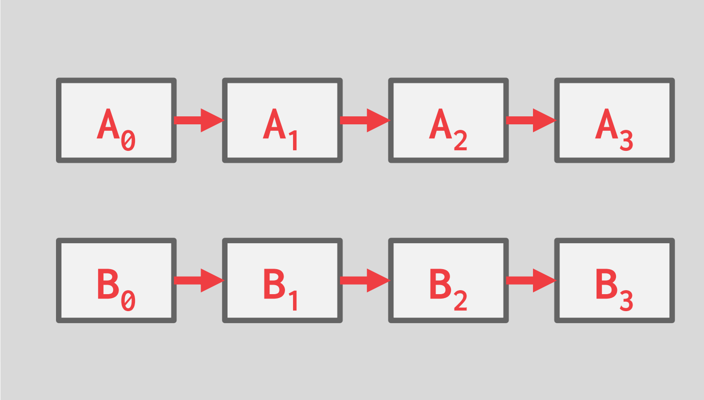
Thread #1

T_{id}=12

Thread #2

T_{id}=25

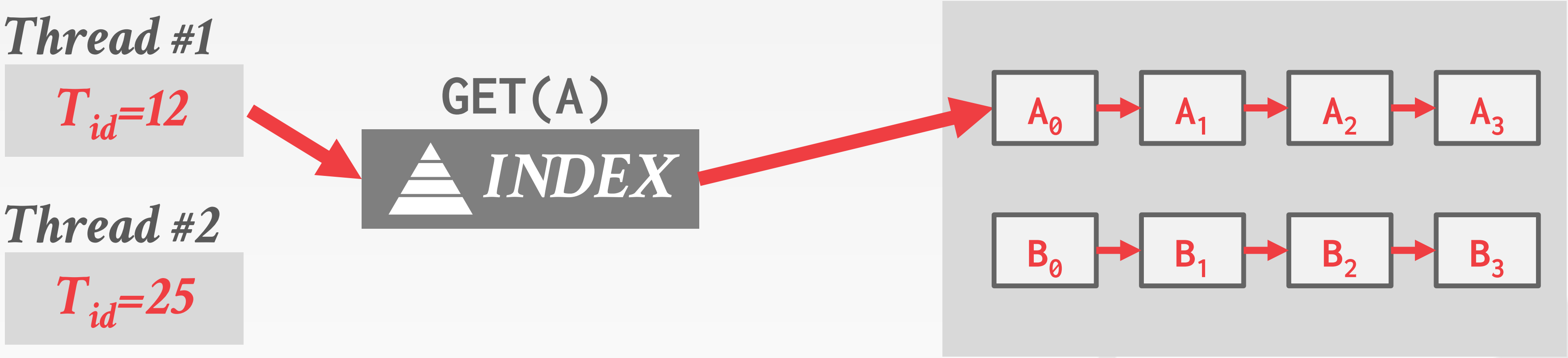
INDEX



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.

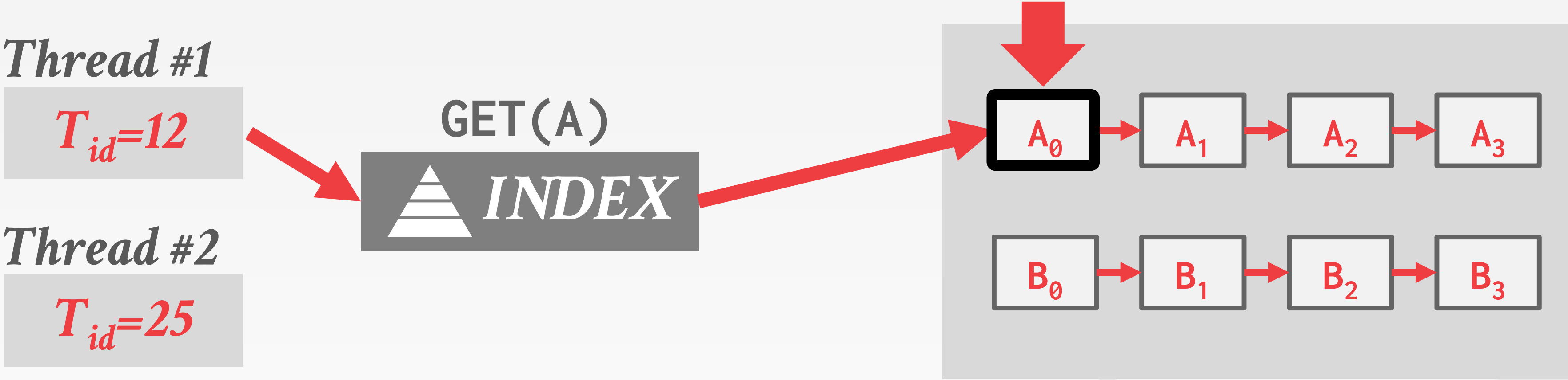
TUPLE-LEVEL GC



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.

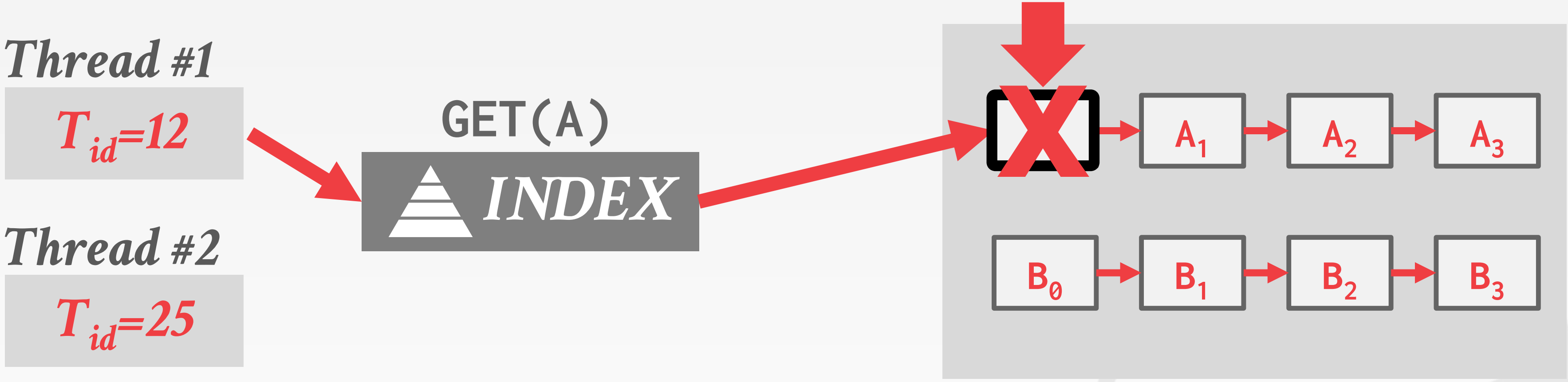
TUPLE-LEVEL GC



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.

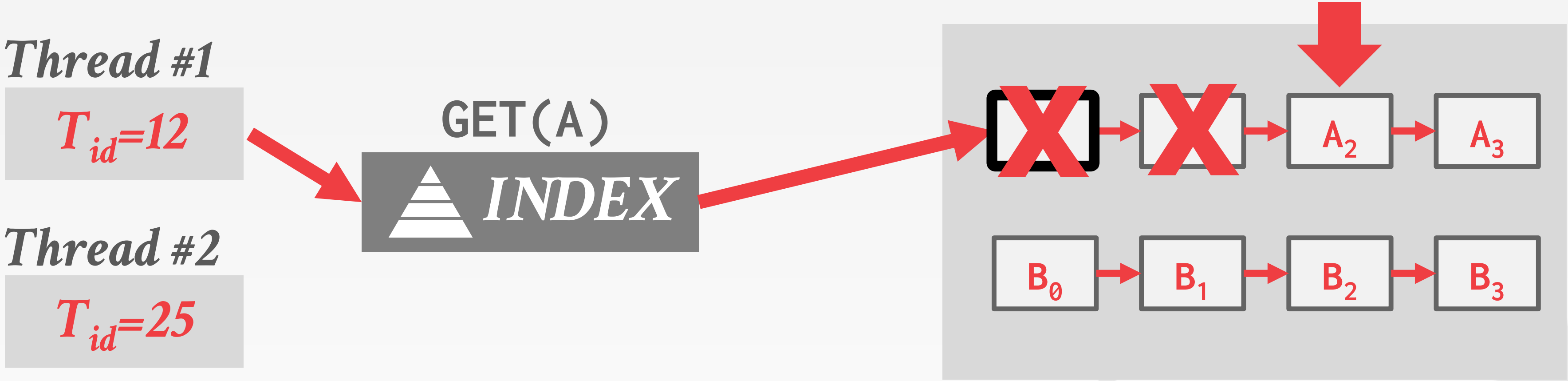
TUPLE-LEVEL GC



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.

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

INDEX MANAGEMENT

PKey indexes always 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

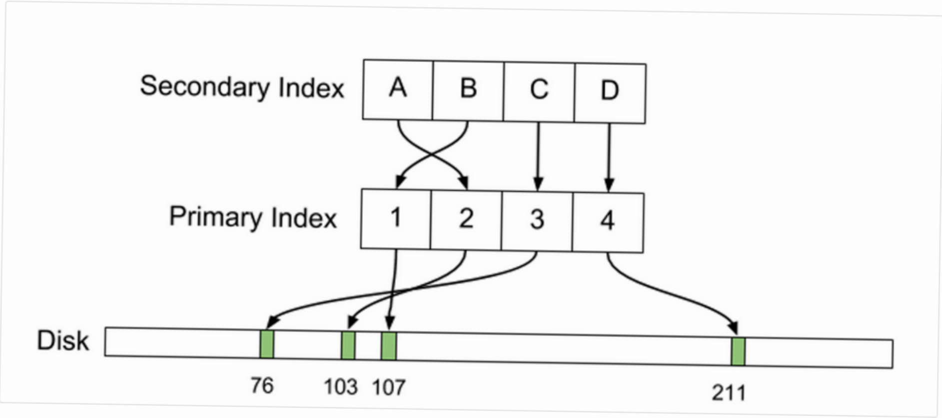
JOIN THE TEAM MEET THE PEOPLE

ARCHITECTURE

WHY UBER ENGINEERING SWITCHED FROM POSTGRES TO MYSQL

JULY 26, 2016

BY EVAN KLITZKE



SECONDARY INDEXES

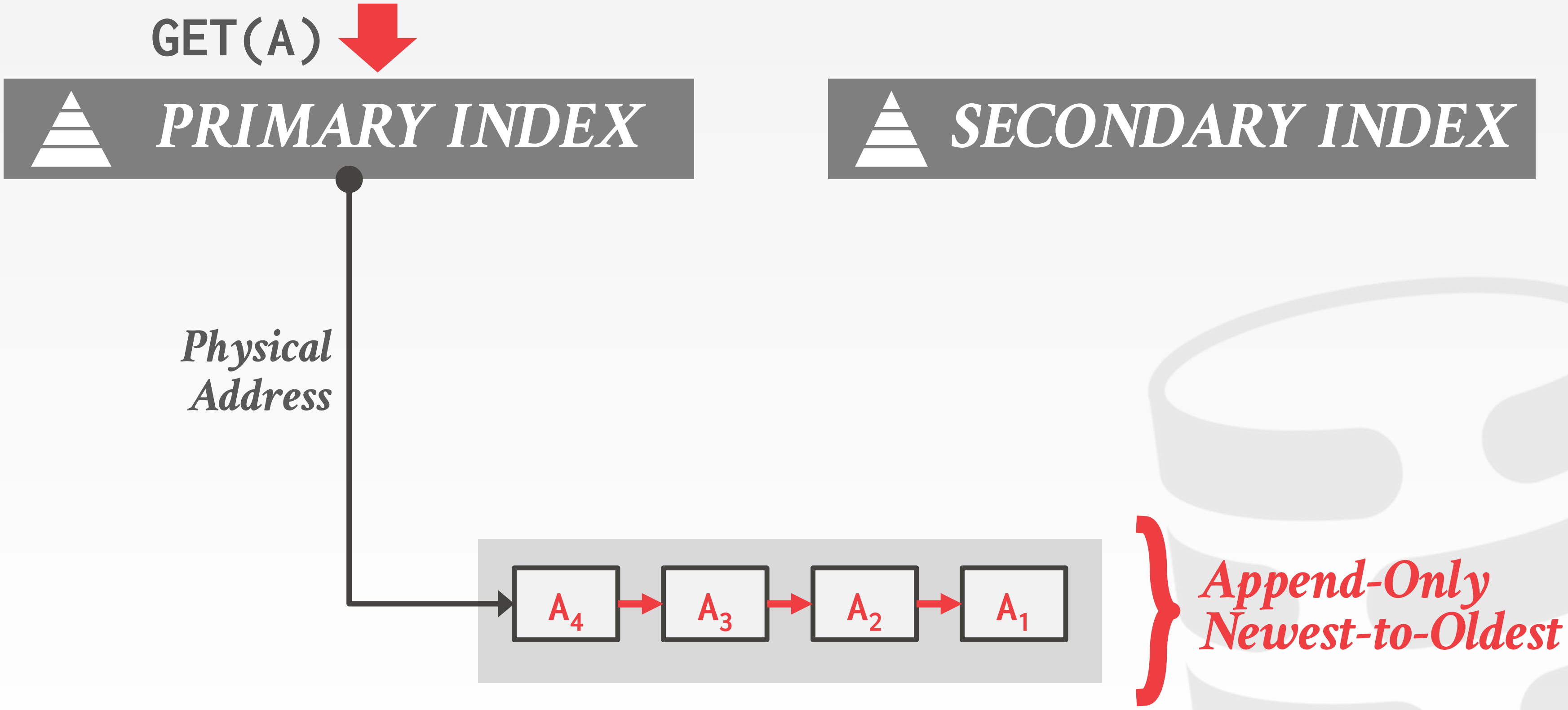
Approach #1: Logical Pointers

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

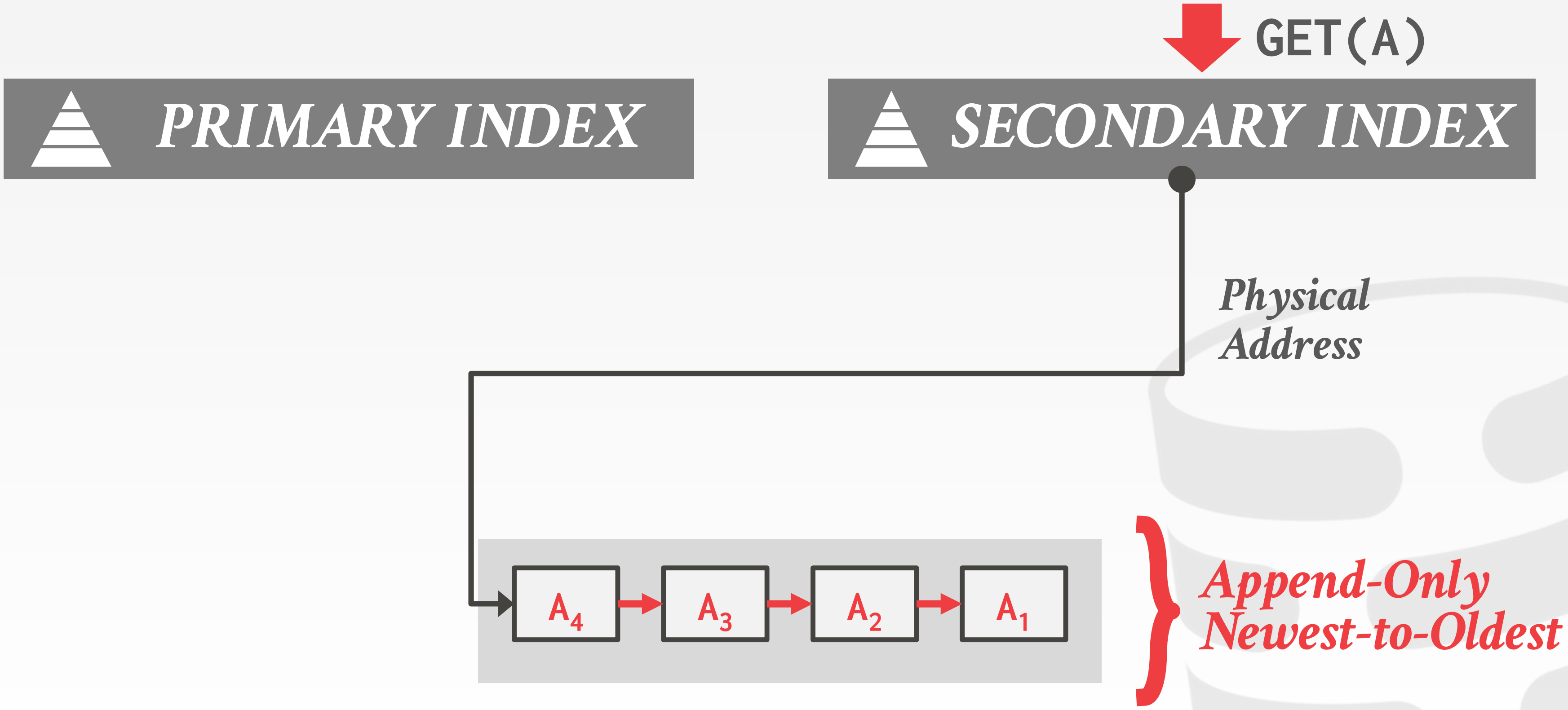
Approach #2: Physical Pointers

→ Use the physical address to the version chain head.

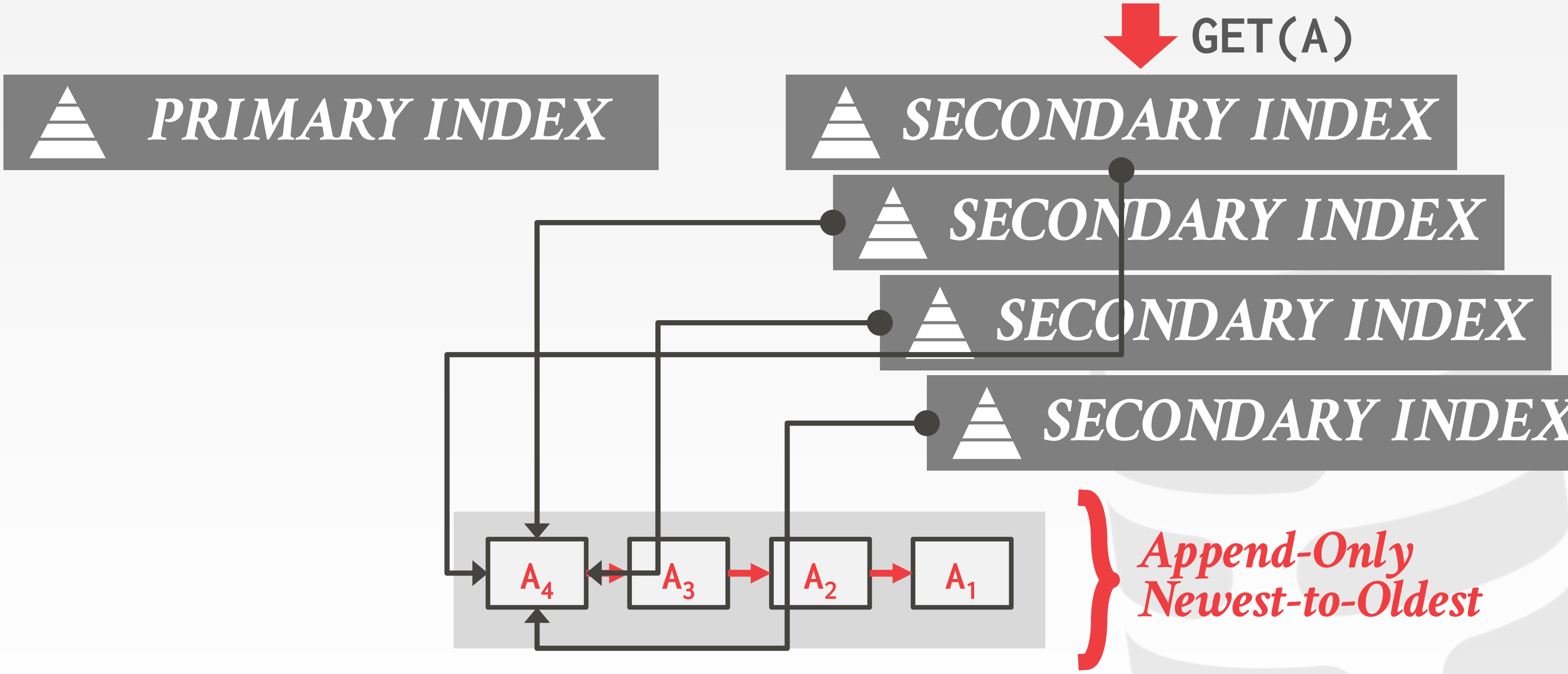
INDEX POINTERS



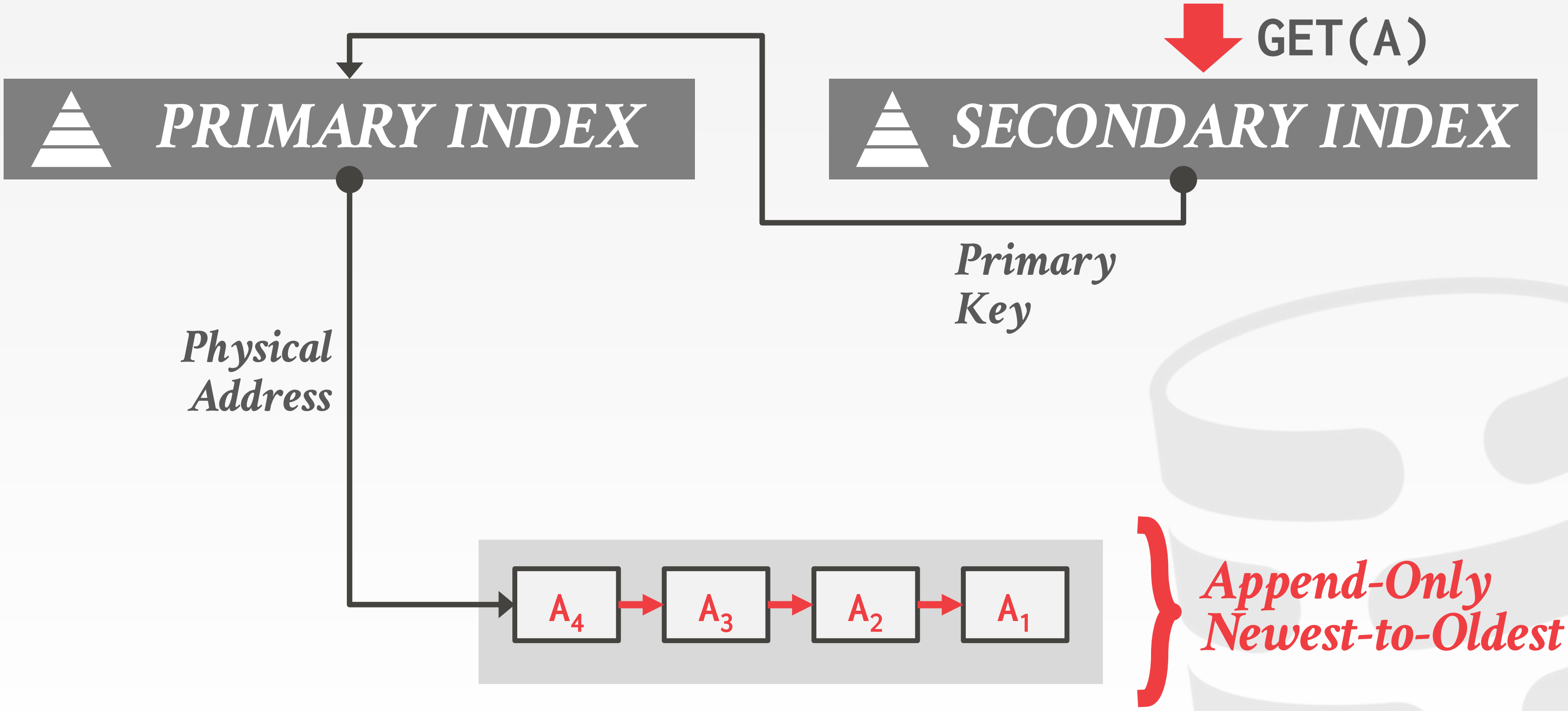
INDEX POINTERS



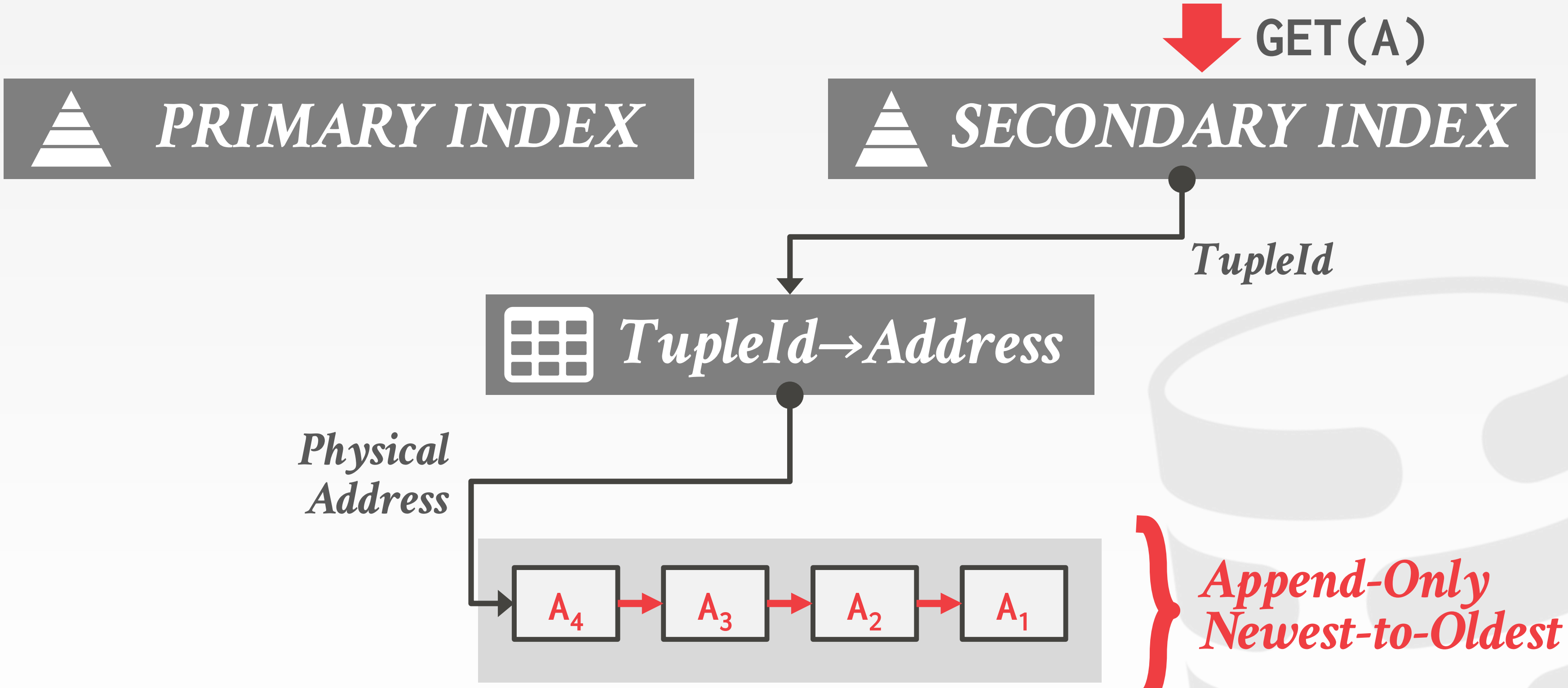
INDEX POINTERS



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.

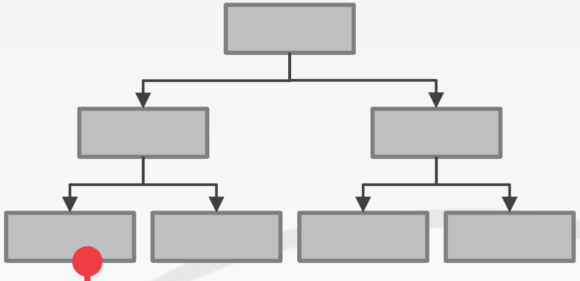
MVCC DUPLICATE KEY PROBLEM

Thread #1

Begin @ 10

READ(A)

Index



	BEGIN-TS	END-TS	POINTER
A_1	1	∞	Ø

MVCC DUPLICATE KEY PROBLEM

Thread #1

Begin @ 10

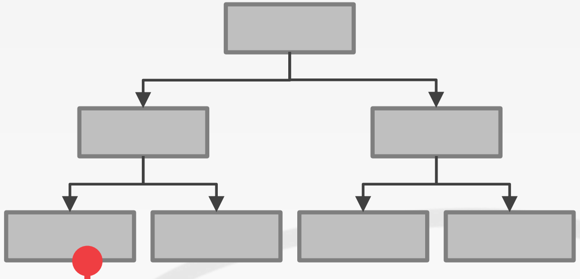
READ(A)

Thread #2

Begin @ 20

UPDATE(A)

Index



	BEGIN-TS	END-TS	POINTER
A_1	1	∞	Ø

MVCC DUPLICATE KEY PROBLEM

Thread #1

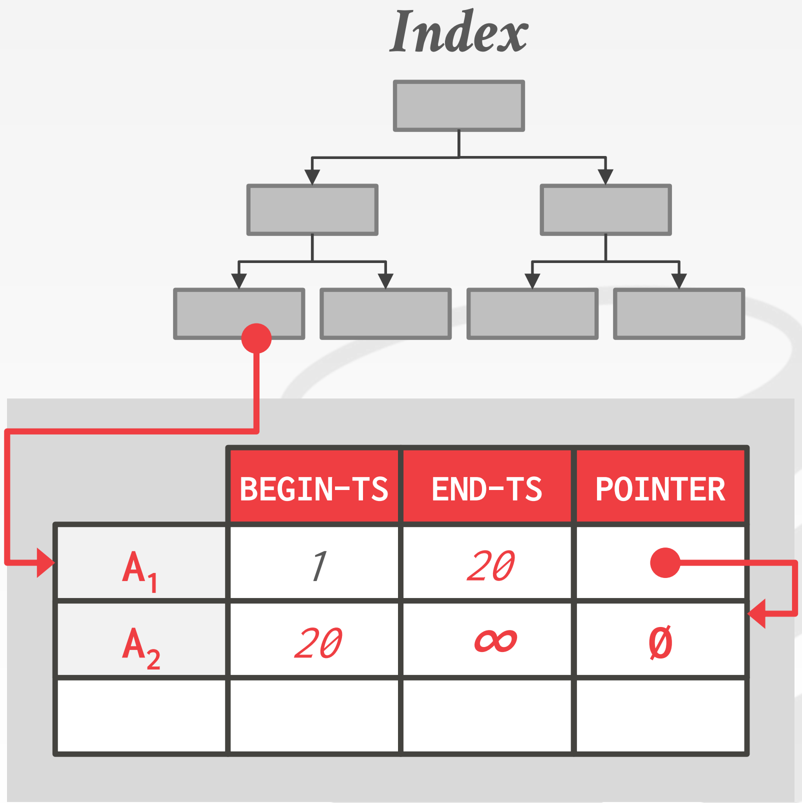
Begin @ 10

READ(A)

Thread #2

Begin @ 20

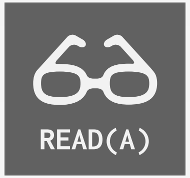
UPDATE(A)



MVCC DUPLICATE KEY PROBLEM

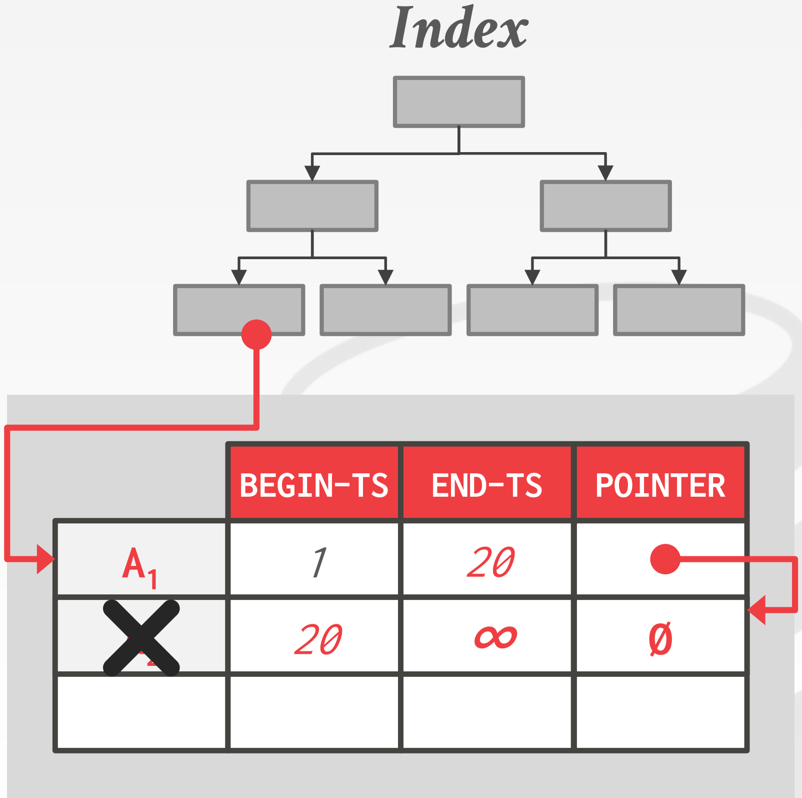
Thread #1

Begin @ 10



Thread #2

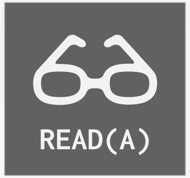
Begin @ 20



MVCC DUPLICATE KEY PROBLEM

Thread #1

Begin @ 10

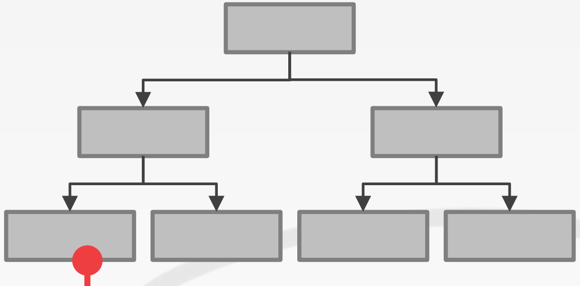


Thread #2

Begin @ 20

Commit @ 25

Index



	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)

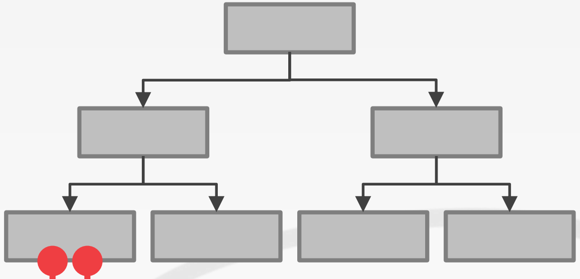
DELETE(A)

Thread #3

Begin @ 30

INSERT(A)

Index



	BEGIN-TS	END-TS	POINTER
A_1	1	25	
~~A_2~~	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

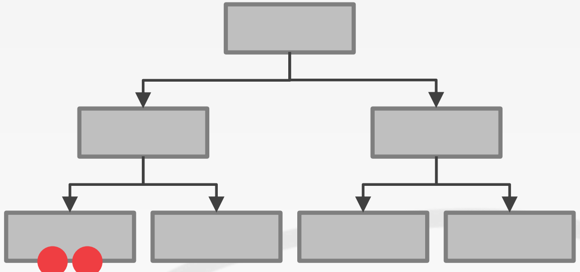
UPDATE(A) DELETE(A)

Thread #3

Begin @ 30

INSERT(A)

Index



	BEGIN-TS	END-TS	POINTER
A_1	1	25	•
$\cancel{A_2}$	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 EVALUATION PAPER

We implemented all the design decisions in the Peloton DBMS as part of 15-721 in Spring 2016.

Two categories of experiments:

→ Evaluate each of the design decisions in isolation to determine their trade-offs.

→ Compare configurations of real-world MVCC systems.

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

MVCC DESIGN DECISIONS

CC Protocol: Inconclusive results…

Version Storage: Deltas

Garbage Collection: Tuple-Level Vacuuming

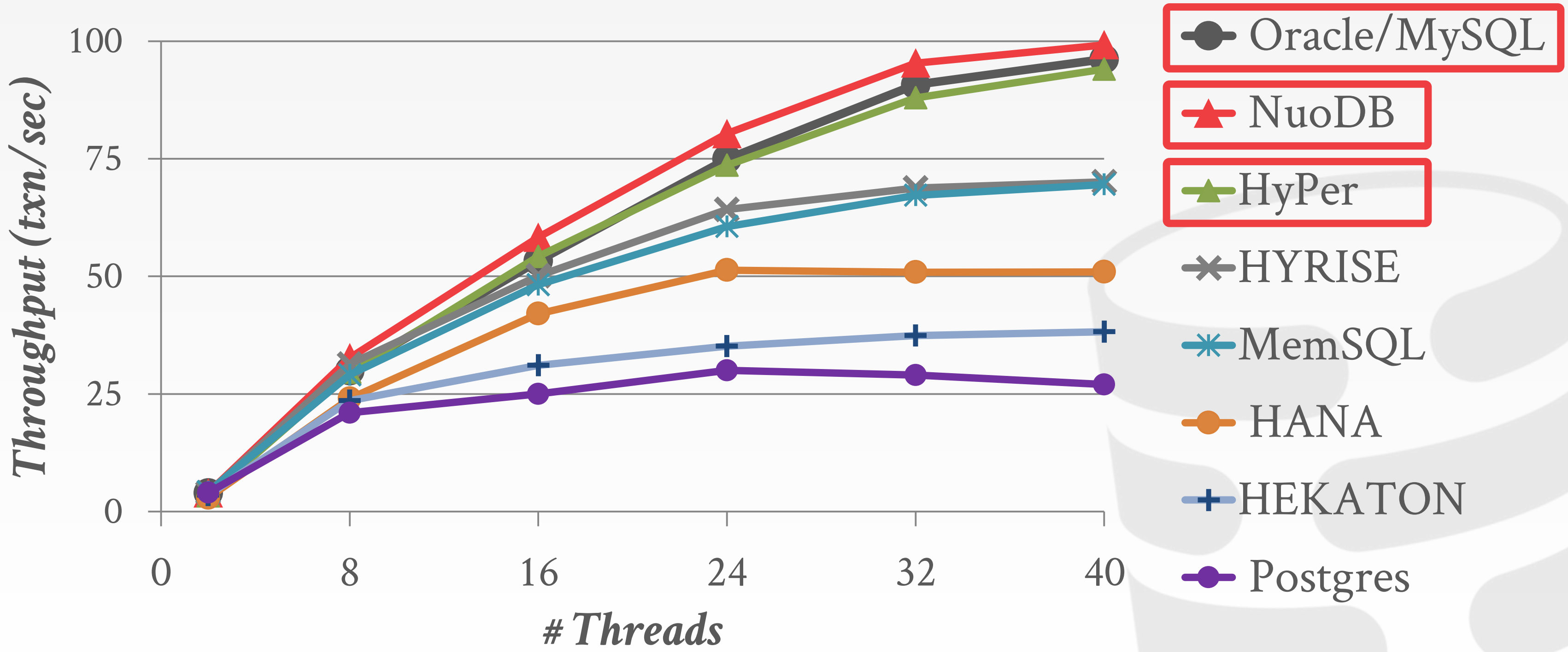
Indexes: Logical Pointers

MVCC CONFIGURATION EVALUATION

	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

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

G+ 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)

ySQL

N

PROJECT #1

Identify bottlenecks in the DBMS's sequential scan implementation using profiling tools and refactor the system to remove it.

This project is meant to teach you how to work in a highly concurrent system.

YET-TO-BE-NAMED DBMS

CMU's new in-memory hybrid relational DBMS

- → HyPer-style MVCC column store
- → Multi-threaded architecture
- → Latch-free Bw-Tree Index
- → Native support for Apache Arrow format
- → Vectorized Execution Engine
- → MemSQL-style LLVM-based Query Compilation
- → Cascades-style Query Optimizer
- → Postgres Wire Protocol / Catalog Compatible

Long term vision is to build a "self-driving" system

PROJECT #1 – TESTING

We are providing you with a suite of C++ benchmarks for you check your implementation.

→ Focus on the ***ConcurrentSlotIterators*** microbenchmark but you will want to run all of them to make sure your code works.

We strongly encourage you to do your own additional testing.

→ Different workloads

→ Different # of threads

→ Different access patterns

PROJECT #1 – GRADING

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

We will also use Google's Sanitizers when testing your code.

All source code must pass ClangFormat + ClangTidy syntax formatting checker.

→ See documentation for formatting guidelines

DEVELOPMENT ENVIRONMENT

The DBMS builds on Ubuntu 18.04+ and OSX.

→ You can also do development on docker or VM.

This is CMU so I'm going to assume that each of you can get access to a machine.

Important: You will <u>not</u> be able to identify the bottleneck on a machine with less than 8 cores.

TESTING ENVIRONMENT

Every student will receive $50 of Amazon AWS credits to run experiments on EC2.

→ Setup monitoring + alerts to prevent yourself from burning through your credits.

→ Use spot instances whenever possible.

Target EC2 Instance: **c5.9xlarge**

→ On Demand: $1.53/hr

→ Spot Instance: $0.34/hr (as of Jan 2020)

PROJECT #1

Due Date: February 16th @ 11:59pm

Source code + final report will be turned in using Gradescope but graded using a different machine.

Full description and instructions:

https://15721.courses.cs.cmu.edu/spring2020/project1.html

PARTING THOUGHTS

MVCC is the best approach for supporting txns in mixed workloads.

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

NEXT CLASS

Modern MVCC Implementations
→ TUM HyPer
→ CMU Cicada
→ Microsoft Hekaton