Carnegie Mellon University

Intro to Database Systems (15-445/645)

Lecture #18

Multi-Version Concurrency Control

FALL 2023 » Prof. Andy Pavlo • Prof. Jignesh Patel

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

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

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	-

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

TS(T_1)=1 edule 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

Schedule

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

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

TIME

T_2 creates version A_1 and sets A_0 End-TS.

Database

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

MVCC – EXAMPLE #1

$TS(T_1)$=1

hedule

$TS(T_2)$=2

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

TIME

T_2 creates version A_1 and sets A_0 End-TS.

Database

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

MVCC – EXAMPLE #1

$TS(T_1)=1$

$TS(T_2)=2$

Schedule

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

TIME

T_2 creates version A_1 and sets A_0 End-TS.

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

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

$TS(T_1)=1$

$TS(T_2)=2$

TIME

hedule

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

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

Database

Version	Value	Begin	End
A_0	123	0	

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active

MVCC – EXAMPLE #2

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

TIME

Database

Version	Value	Begin	End
A_0	123	0	

Txn Status Table

TxnId	Timestamp	Status
T_1	1	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	
A_1	456	1	-

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active

MVCC – EXAMPLE #2

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

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	1
A_1	456	1	-

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active

MVCC – EXAMPLE #2

TS(T_1)=1

hedule

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	1
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

TIME

Schedule

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

	Timestamp	Status
		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

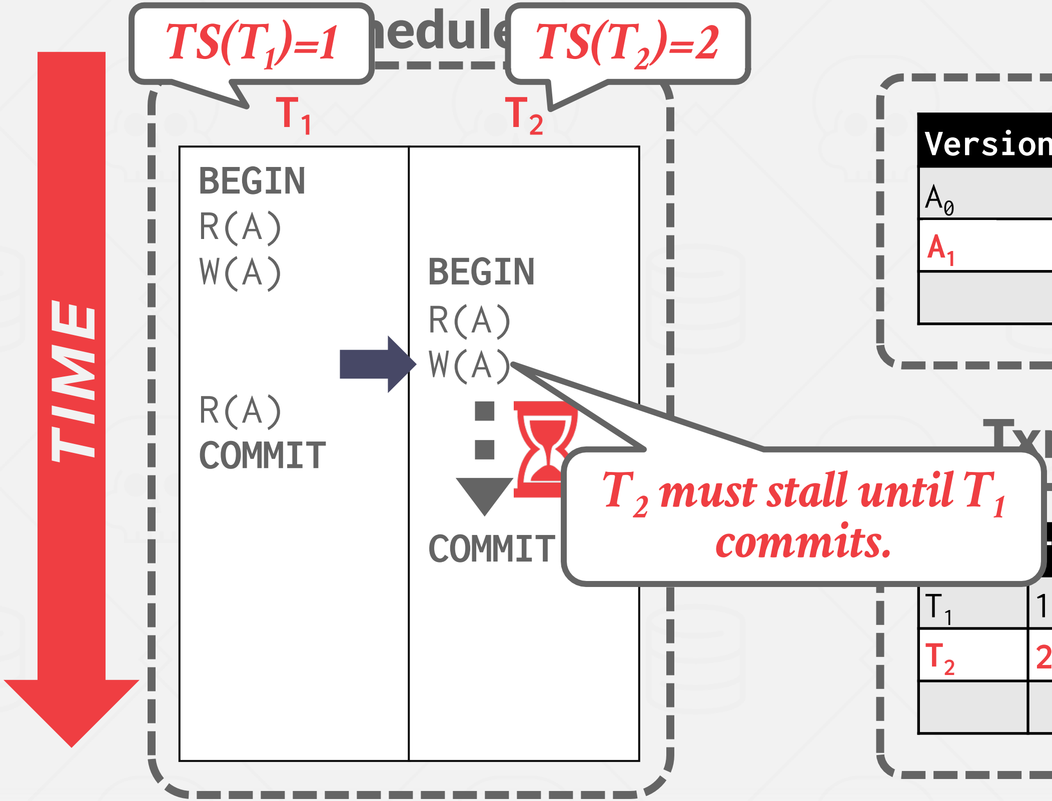
Database

Version	Value	Begin	End
A_0	123	0	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

Txn	Timestamp	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

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

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.

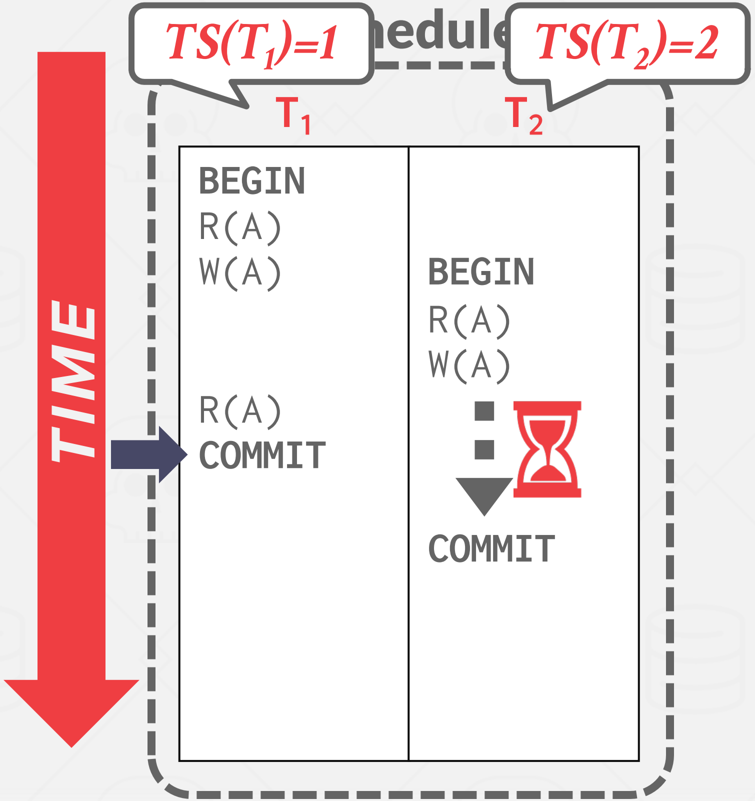
Database

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	Active
T_2	2	Active

MVCC – EXAMPLE #2

hedule

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

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	1
A_1	456	1	-

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Committed
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

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
	2	Active

SNAPSHOT ISOLATION (SI)

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

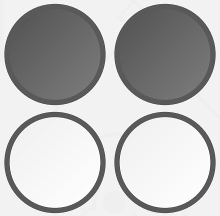
→ No torn writes from active txns.

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

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

WRITE SKEW ANOMALY

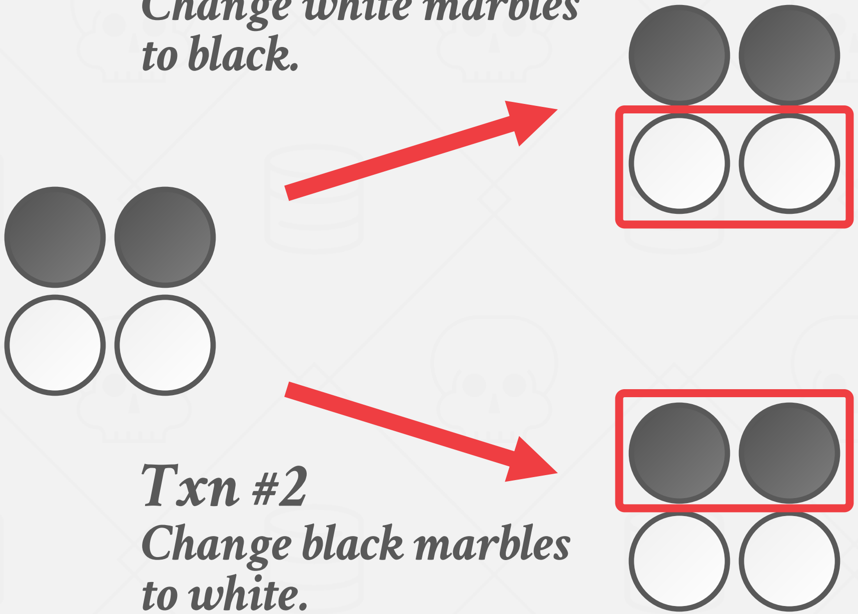
Txn #1
Change white marbles to black.



Txn #2
Change black marbles to white.

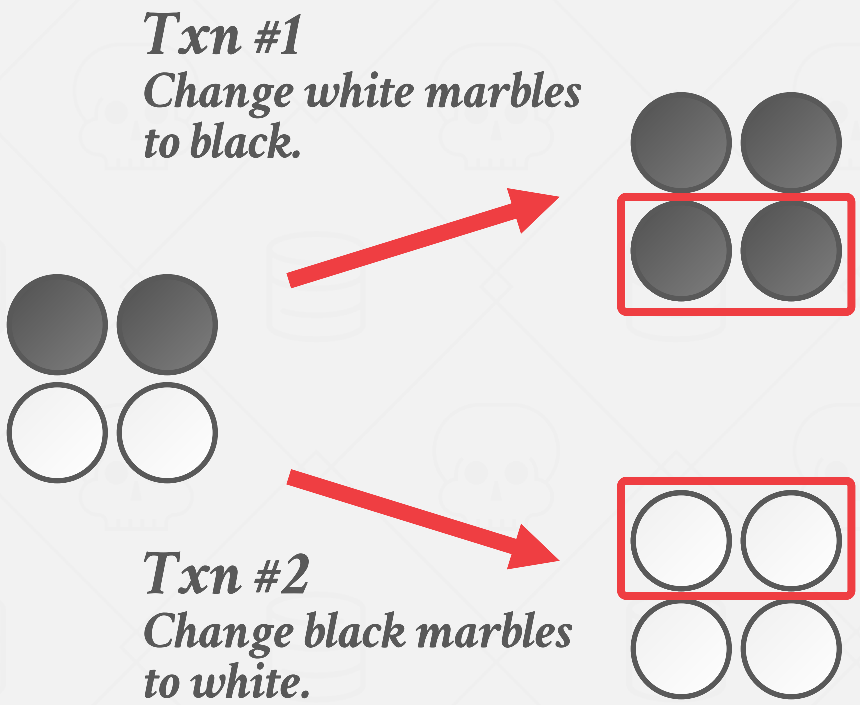
WRITE SKEW ANOMALY

Txn #1
Change white marbles to black.



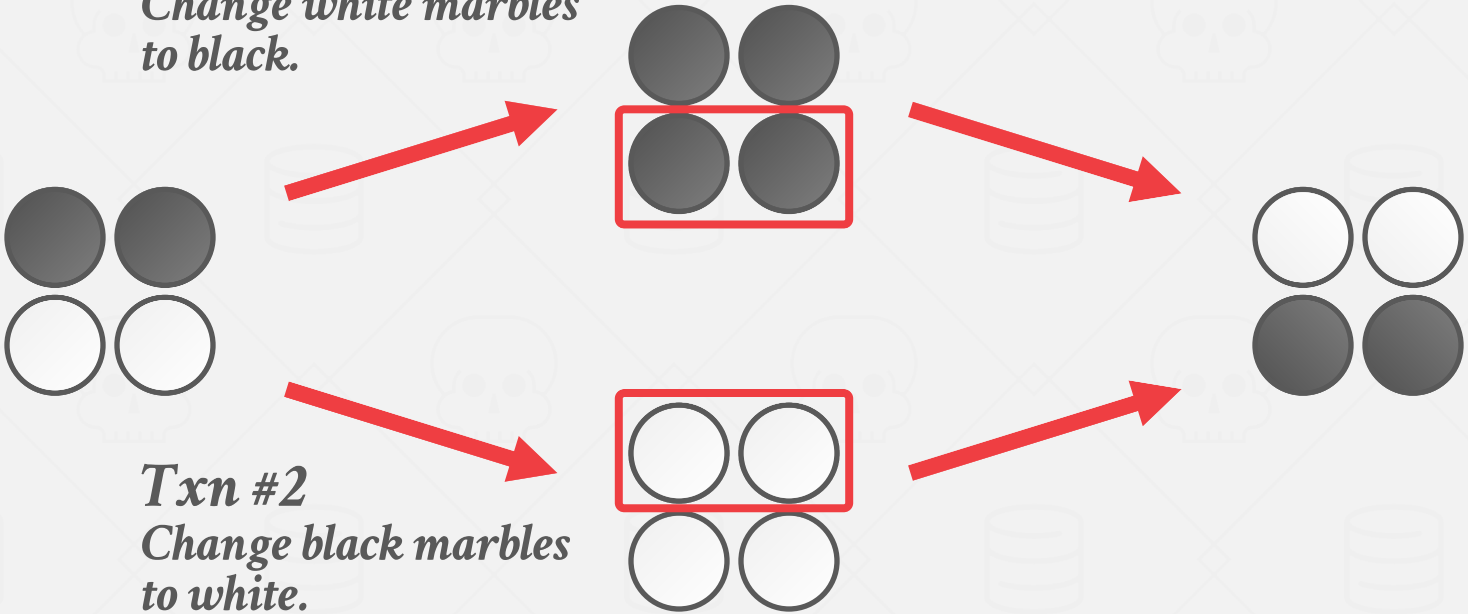
Txn #2
Change black marbles to white.

WRITE SKEW ANOMALY



WRITE SKEW ANOMALY

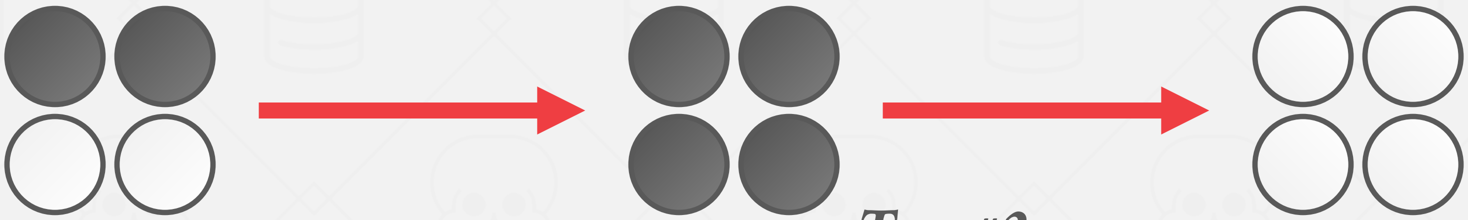
Txn #1
Change white marbles to black.



Txn #2
Change black marbles to white.

WRITE SKEW ANOMALY

Txn #1
Change white marbles to black.



Txn #2
Change black marbles to white.

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

	TUPLE	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

	TUPLE	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

	TUPLE	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

	TUPLE	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

	TUPLE	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 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

Main Table

	TUPLE	POINTER
A_2	$222	● →
B_1	$10	

Time-Travel Table

	TUPLE	POINTER
A_1	$111	Ø

TIME-TRAVEL STORAGE

Main Table

	TUPLE	POINTER
A_2	$222	
B_1	$10	

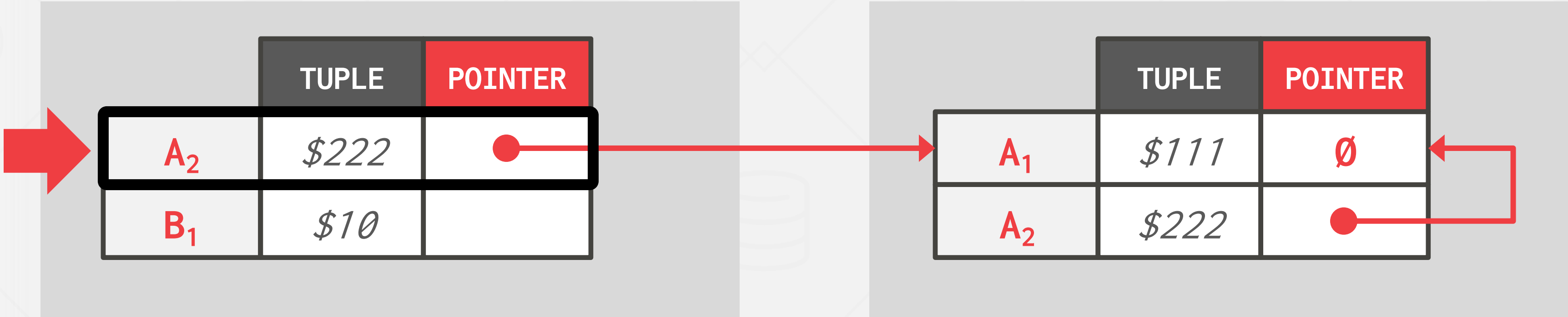
Time-Travel Table

	TUPLE	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

	TUPLE	POINTER
A_3	$333	
B_1	$10	

Time-Travel Table

	TUPLE	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

	TUPLE	POINTER
A_3	$333	
B_1	$10	

Time-Travel Table

	TUPLE	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

	TUPLE	POINTER
A_3	$333	
B_1	$10	

Time-Travel Table

	TUPLE	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 column 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 column 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 column 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 column 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 column 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*	

Delta Storage Segment

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

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

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

GARBAGE COLLECTION

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

→ No active txn in the DBMS can "see" that version (SI).

→ The version was created by an aborted txn.

Two additional design decisions:

→ How to look for expired versions?

→ How to decide when it is safe to reclaim memory?

GARBAGE COLLECTION

Approach #1: Tuple-level

→ Find old versions by examining tuples directly.

→ Background Vacuuming vs. Cooperative Cleaning

Approach #2: Transaction-level

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

TUPLE-LEVEL GC

Txn #1

T_{id}=12

Txn #2

T_{id}=25

	BEGIN-TS	END-TS
A_{100}	1	9
B_{100}	1	9
B_{101}	10	20

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

TUPLE-LEVEL GC

Txn #1

T_{id}=12

Txn #2

T_{id}=25

	BEGIN-TS	END-TS
A_{100}	1	9
B_{100}	1	9
B_{101}	10	20

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

TUPLE-LEVEL GC

Txn #1

T_{id}=12

Txn #2

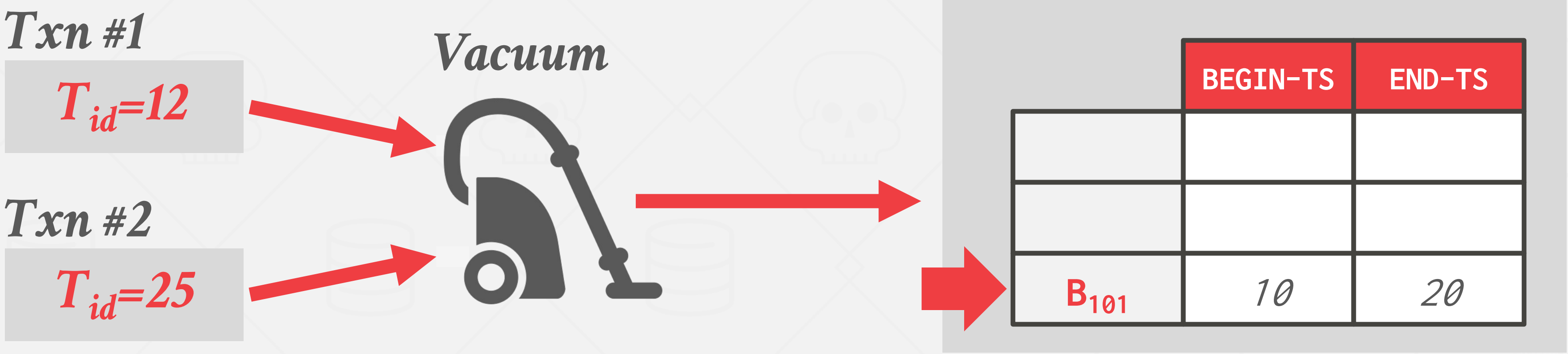
T_{id}=25

Vacuum

	BEGIN-TS	END-TS
A_{100}	1	9
B_{100}	1	9
B_{101}	10	20

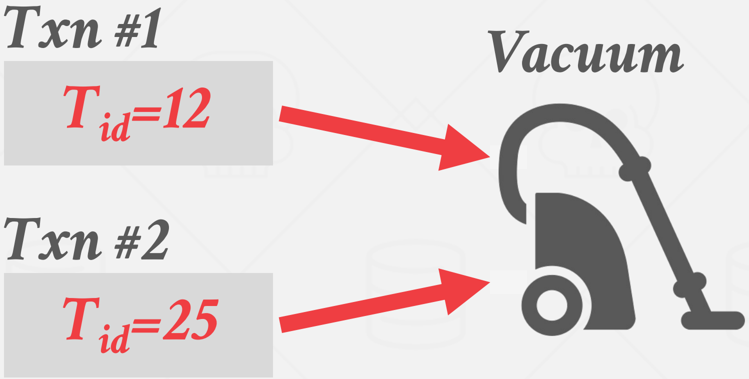
Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

TUPLE-LEVEL GC



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

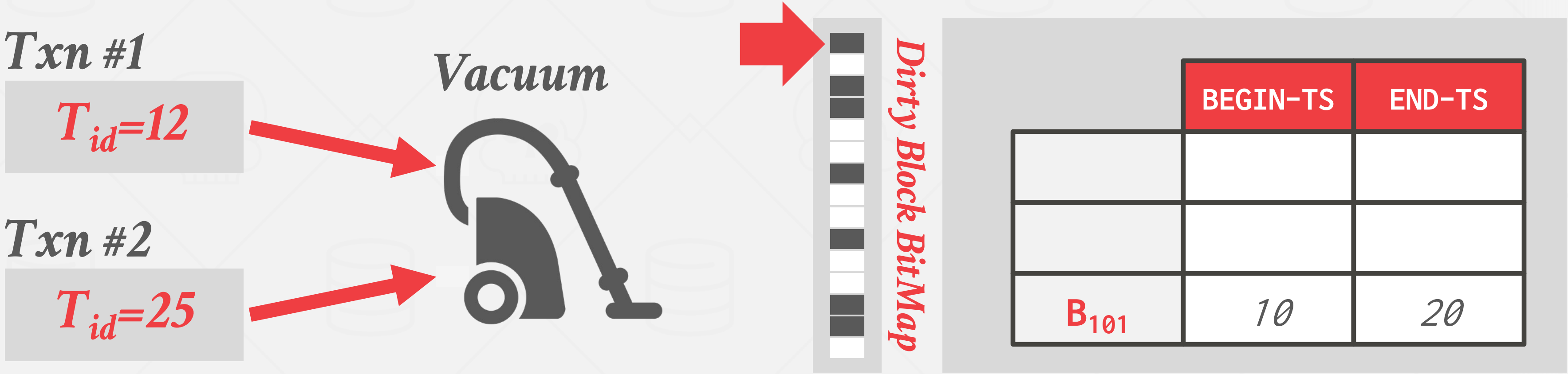
TUPLE-LEVEL GC



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

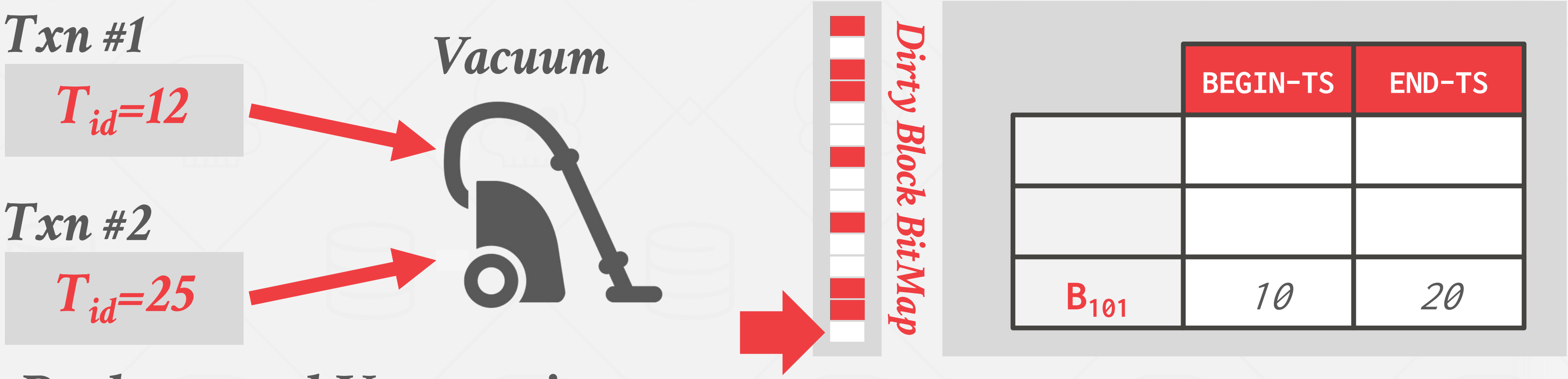
Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

TUPLE-LEVEL GC



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

TUPLE-LEVEL GC



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

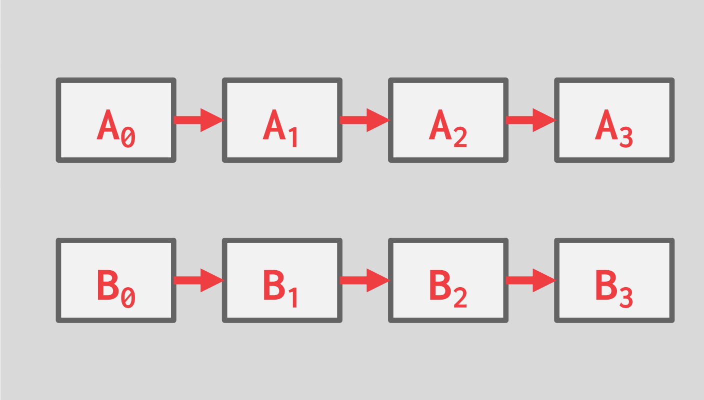
TUPLE-LEVEL GC

Txn #1

T_{id}=12

Txn #2

T_{id}=25



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

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

TUPLE-LEVEL GC

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

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

TUPLE-LEVEL GC

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

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

TUPLE-LEVEL GC

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

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

TUPLE-LEVEL GC

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

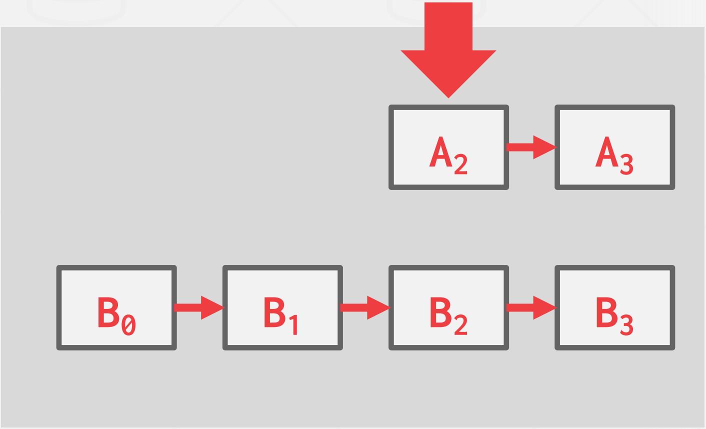
Cooperative Cleaning:
Worker threads identify reclaimable versions as they traverse version chain. Only works with O2N.

TUPLE-LEVEL GC

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

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

TUPLE-LEVEL GC



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

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

TUPLE-LEVEL GC

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

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

TRANSACTION-LEVEL GC

Each txn keeps track of its read/write set.

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

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

TRANSACTION-LEVEL GC

Txn #1

```
BEGIN @ 10
```

	BEGIN-TS	END-TS	DATA
A_2	*1*	∞	-
B_6	*8*	∞	-

TRANSACTION-LEVEL GC

Txn #1

BEGIN @ 10

UPDATE(A)

	BEGIN-TS	END-TS	DATA
A_2	1	∞	-
B_6	8	∞	-

TRANSACTION-LEVEL GC

Txn #1

```
BEGIN @ 10
```

UPDATE(A)

	BEGIN-TS	END-TS	DATA
A_2	1	∞	-
B_6	8	∞	-

TRANSACTION-LEVEL GC

Txn #1

```
BEGIN @ 10
```

UPDATE(A)

	BEGIN-TS	END-TS	DATA
A_2	1	10	-
B_6	8	∞	-
A_3	10	∞	-

TRANSACTION-LEVEL GC

Txn #1

BEGIN @ 10

Old Versions

A_2

UPDATE(A)

	BEGIN-TS	END-TS	DATA
A_2	*1*	*10*	-
B_6	*8*	∞	-
A_3	*10*	∞	-

TRANSACTION-LEVEL GC

Txn #1

BEGIN @ 10

Old Versions

A_2

UPDATE(A)

	BEGIN-TS	END-TS	DATA
A_2	1	10	-
B_6	8	∞	-
A_3	10	∞	-

TRANSACTION-LEVEL GC

Txn #1

BEGIN @ 10

Old Versions

A_2

UPDATE(A)

UPDATE(B)

	BEGIN-TS	END-TS	DATA
A_2	1	10	-
B_6	8	∞	-
A_3	10	∞	-

TRANSACTION-LEVEL GC

Txn #1

BEGIN @ 10

Old Versions

A_2

UPDATE(A)

UPDATE(B)

	BEGIN-TS	END-TS	DATA
A_2	1	10	-
B_6	8	10	-
A_3	10	∞	-
B_7	10	∞	-

TRANSACTION-LEVEL GC

Txn #1

BEGIN @ 10

Old Versions

A_2

B_6

UPDATE(A)

UPDATE(B)

	BEGIN-TS	END-TS	DATA
A_2	1	10	-
B_6	8	10	-
A_3	10	∞	-
B_7	10	∞	-

TRANSACTION-LEVEL GC

Txn #1

BEGIN @ 10
COMMIT @ 15

Old Versions

A_2

B_6

UPDATE(A)

UPDATE(B)

	BEGIN-TS	END-TS	DATA
A_2	1	10	-
B_6	8	10	-
A_3	10	∞	-
B_7	10	∞	-

TRANSACTION-LEVEL GC

Txn #1

BEGIN @ 10
COMMIT @ 15

Old Versions

UPDATE(A)

UPDATE(B)

	BEGIN-TS	END-TS	DATA
A_2	1	10	-
B_6	8	10	-
A_3	10	∞	-
B_7	10	∞	-

Vacuum

TS<10 $\left\{ \begin{array}{l} A_2 \\ B_6 \end{array} \right.$

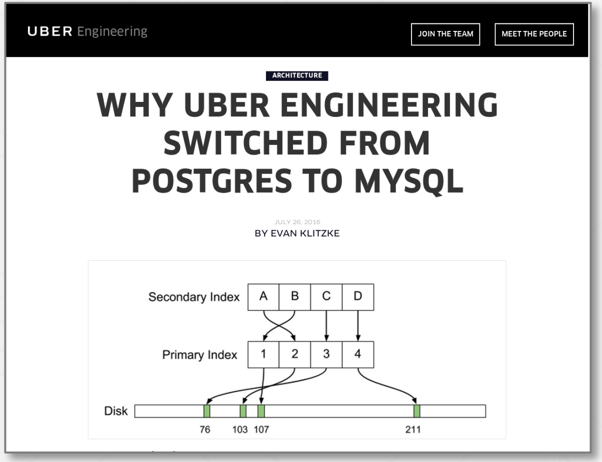
INDEX MANAGEMENT

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…



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

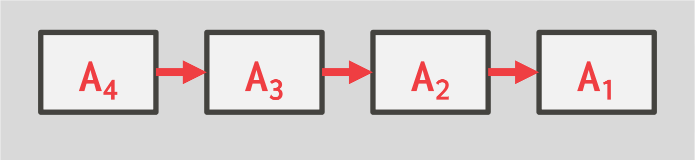
→ Use the physical address to the version chain head.

INDEX POINTERS

GET(A)

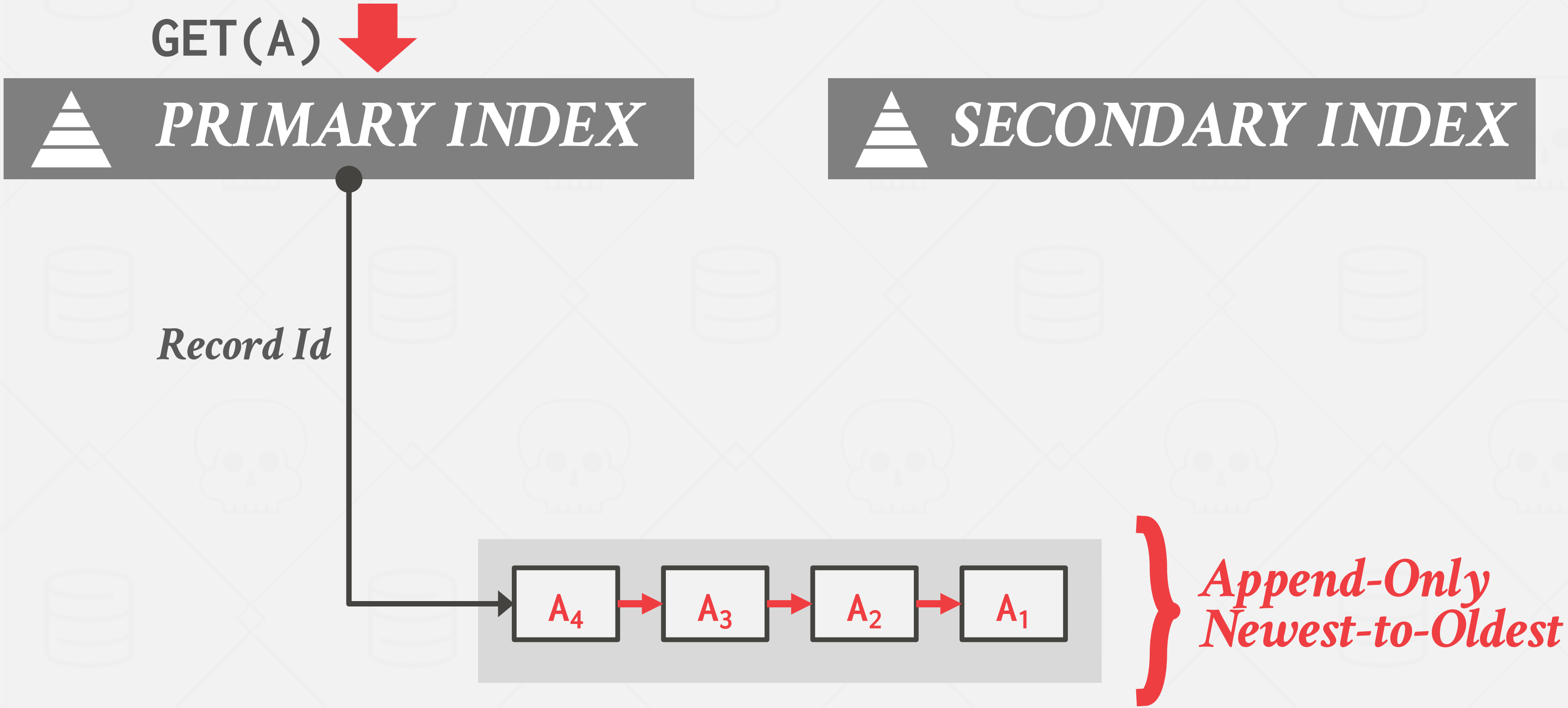
PRIMARY INDEX

SECONDARY INDEX

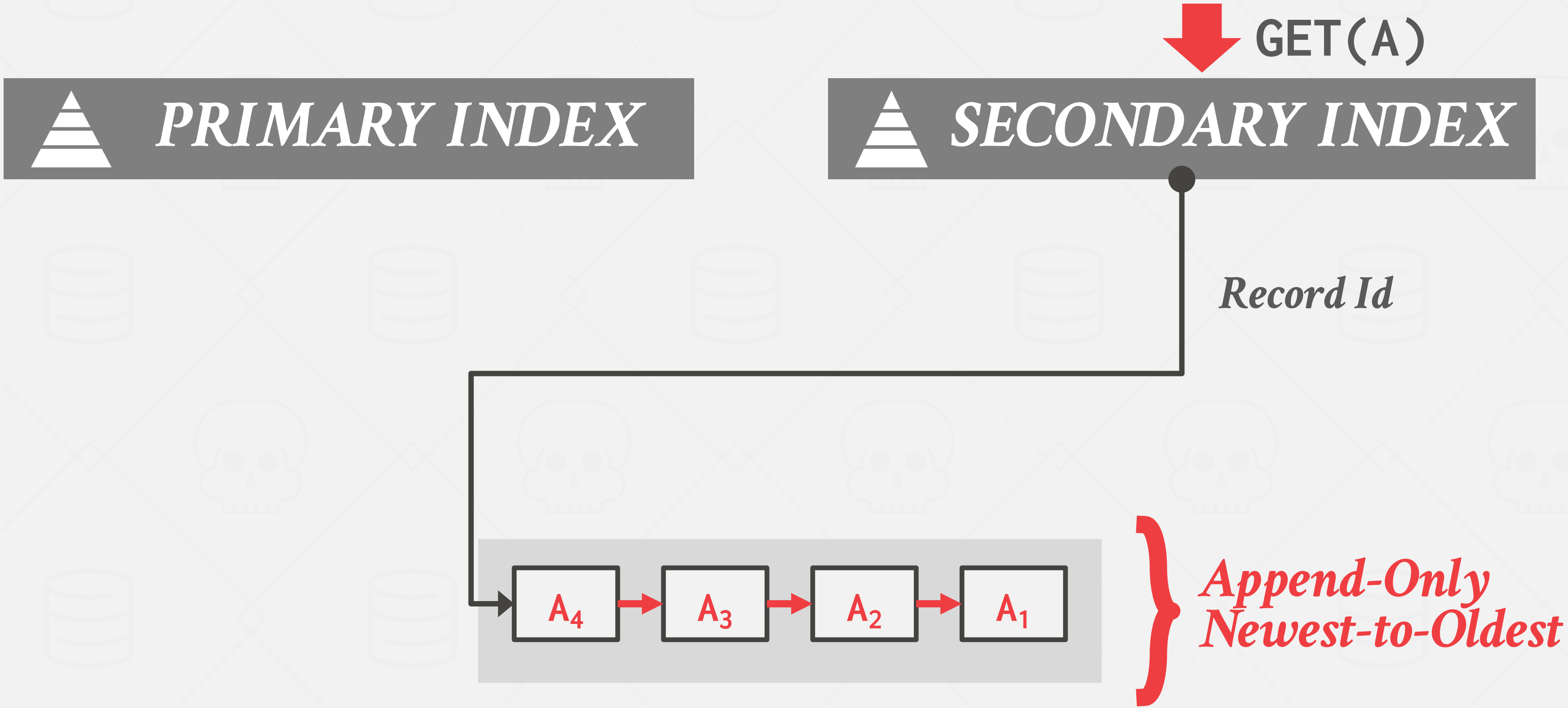


Append-Only
Newest-to-Oldest

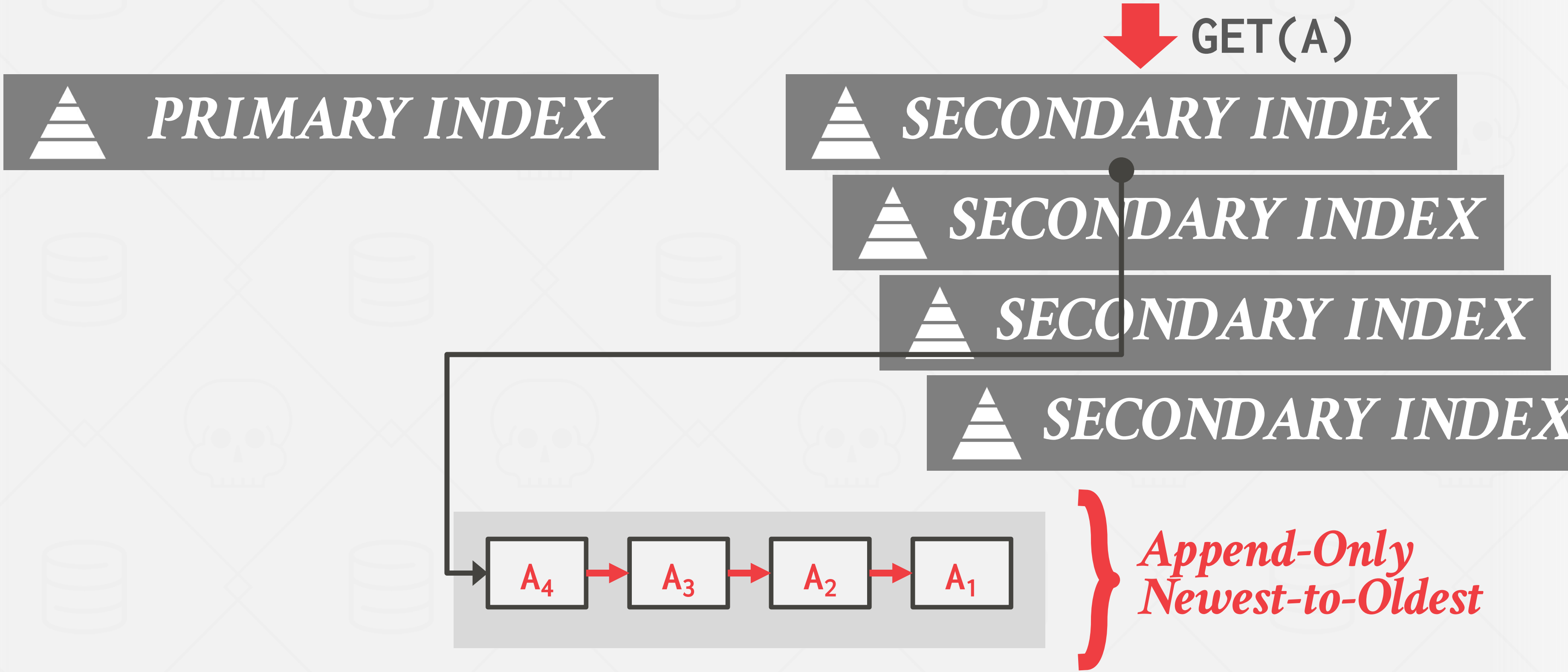
INDEX POINTERS



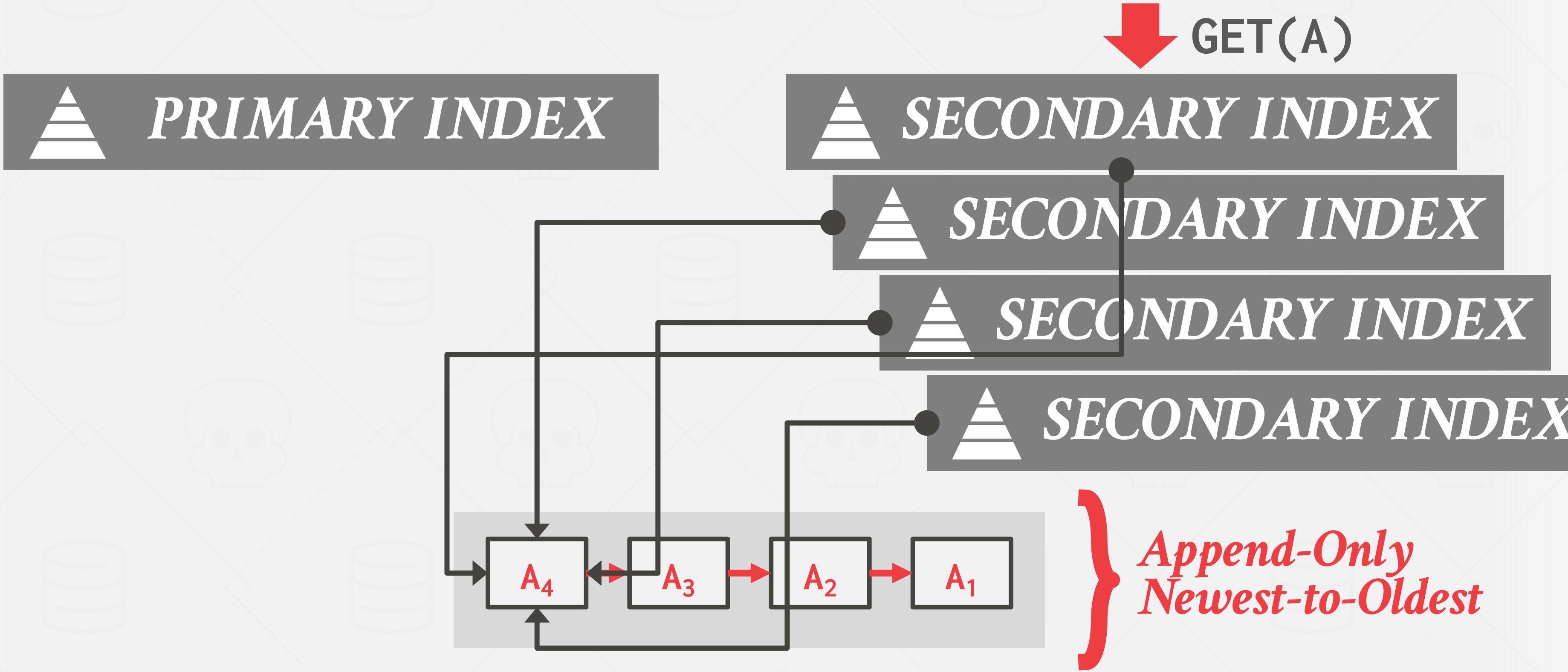
INDEX POINTERS



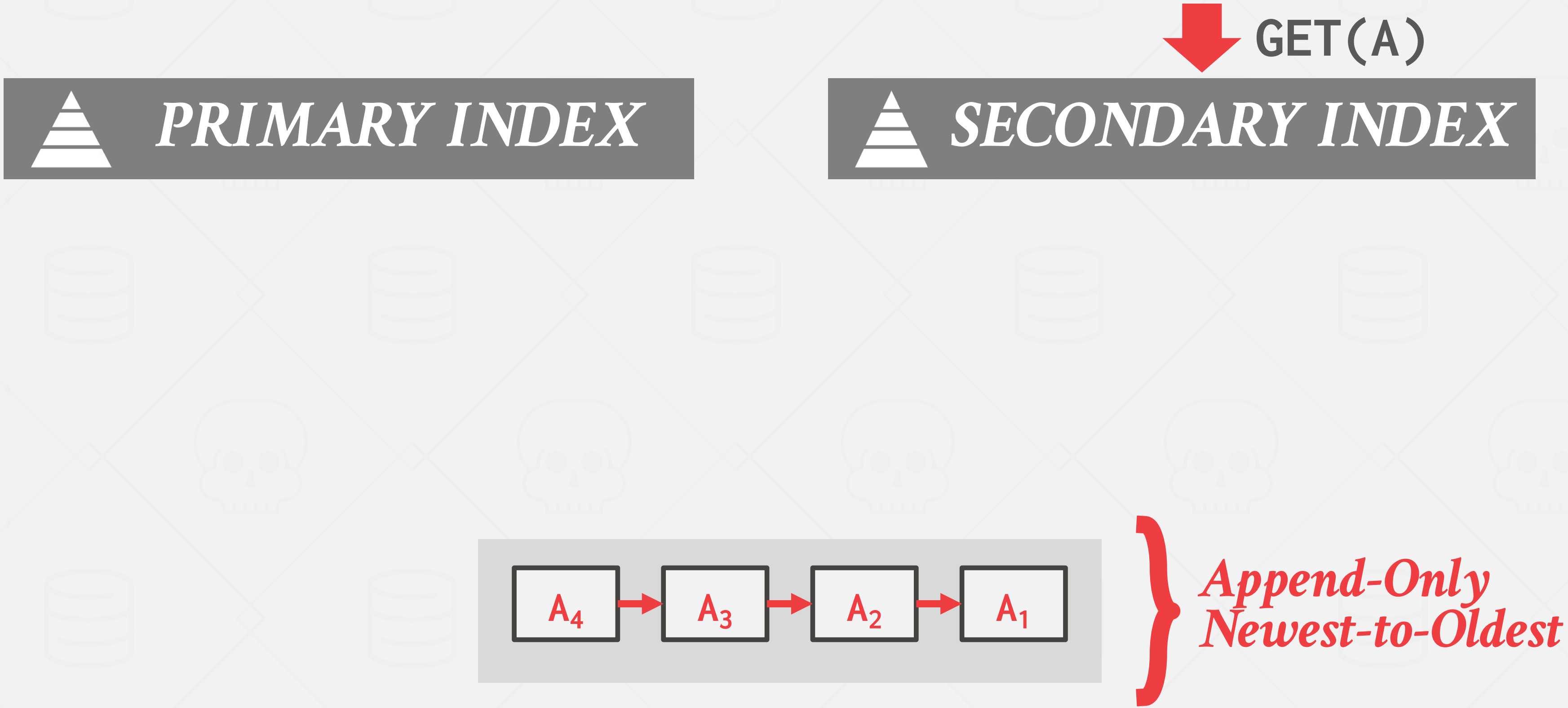
INDEX POINTERS



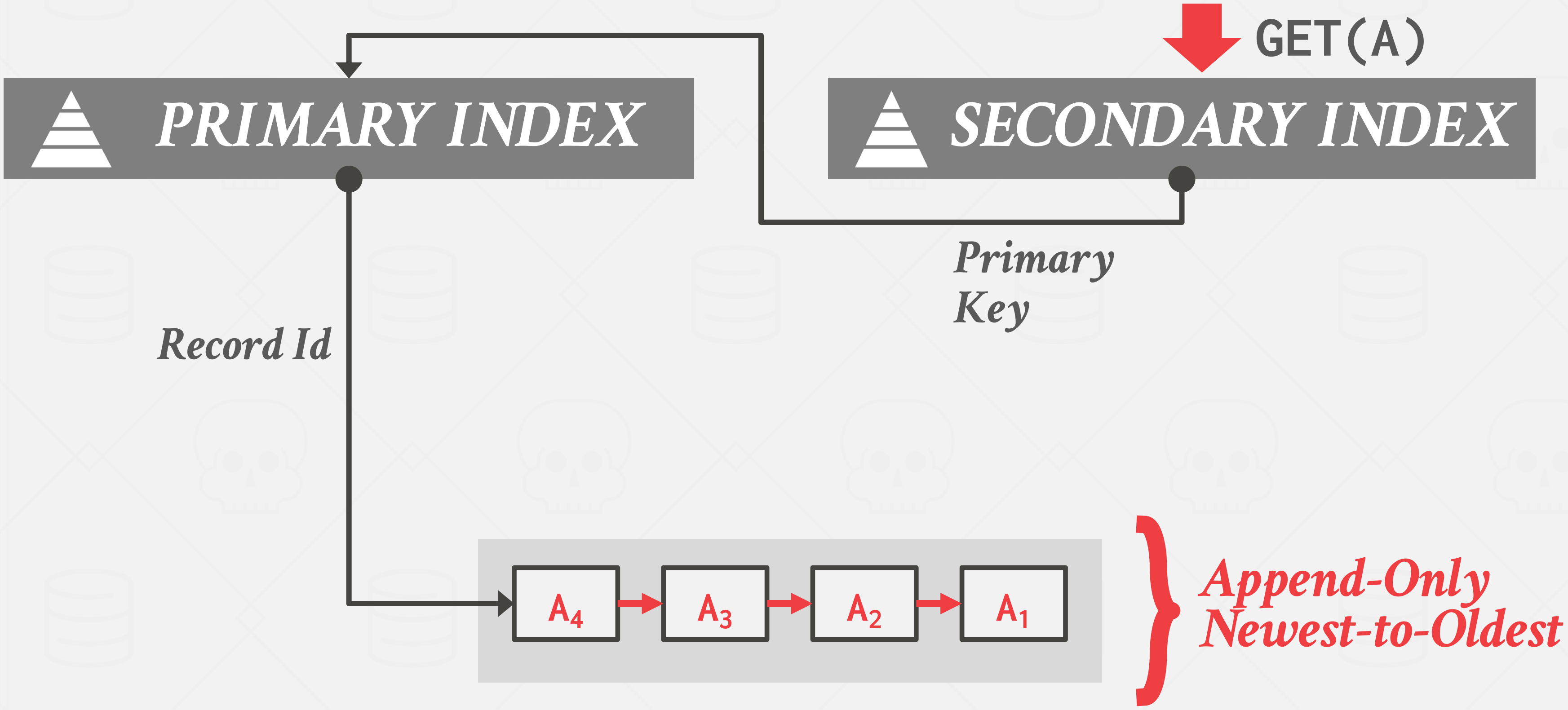
INDEX POINTERS



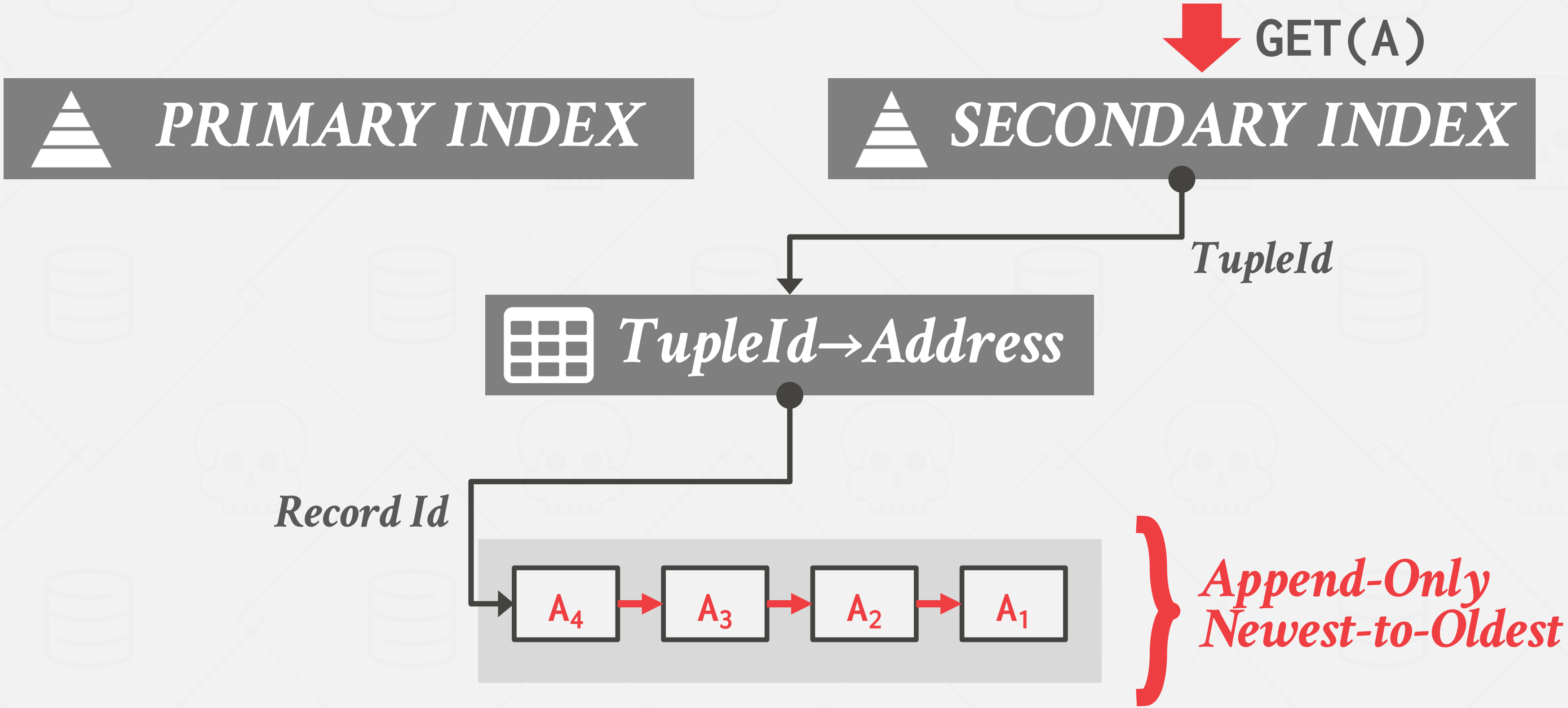
INDEX POINTERS



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)

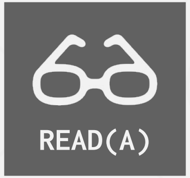
Every index must support duplicate keys from different snapshots:

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

MVCC DUPLICATE KEY PROBLEM

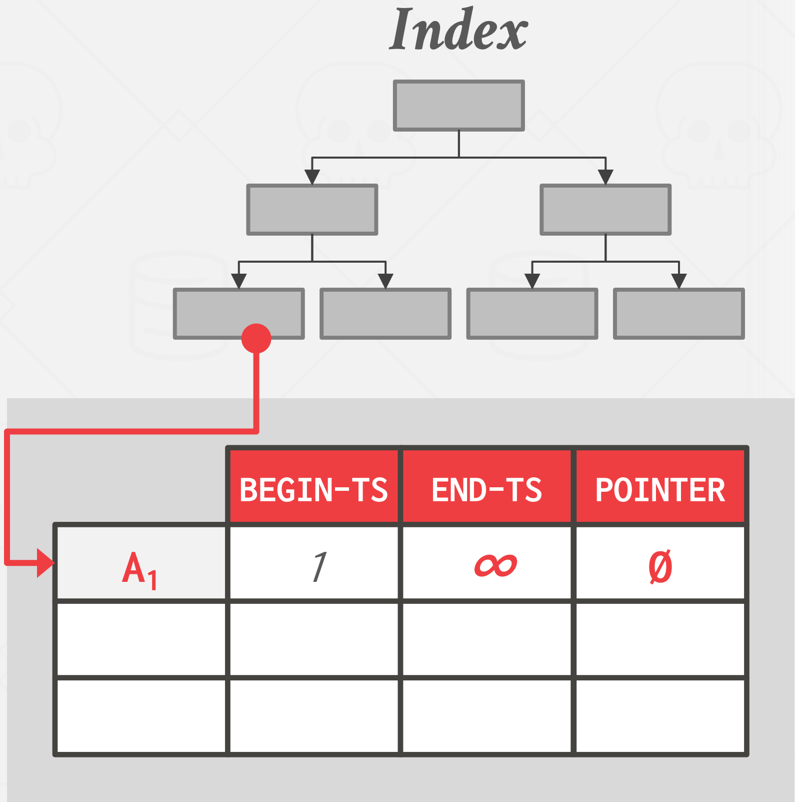
Txn #1

```
BEGIN @ 10
```



Txn #2

```
BEGIN @ 20
```



MVCC DUPLICATE KEY PROBLEM

Txn #1

```
BEGIN @ 10
```

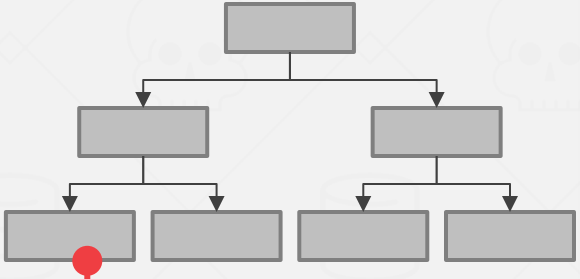
READ(A)

Txn #2

```
BEGIN @ 20
```

UPDATE(A)

Index



	BEGIN-TS	END-TS	POINTER
A_1	1	20	●
A_2	20	∞	Ø

MVCC DUPLICATE KEY PROBLEM

Txn #1

```
BEGIN @ 10
```

READ(A)

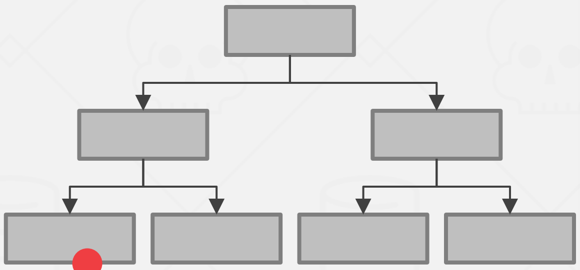
Txn #2

```
BEGIN @ 20
```

UPDATE(A)

DELETE(A)

Index



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

MVCC DUPLICATE KEY PROBLEM

Txn #1

```
BEGIN @ 10
```

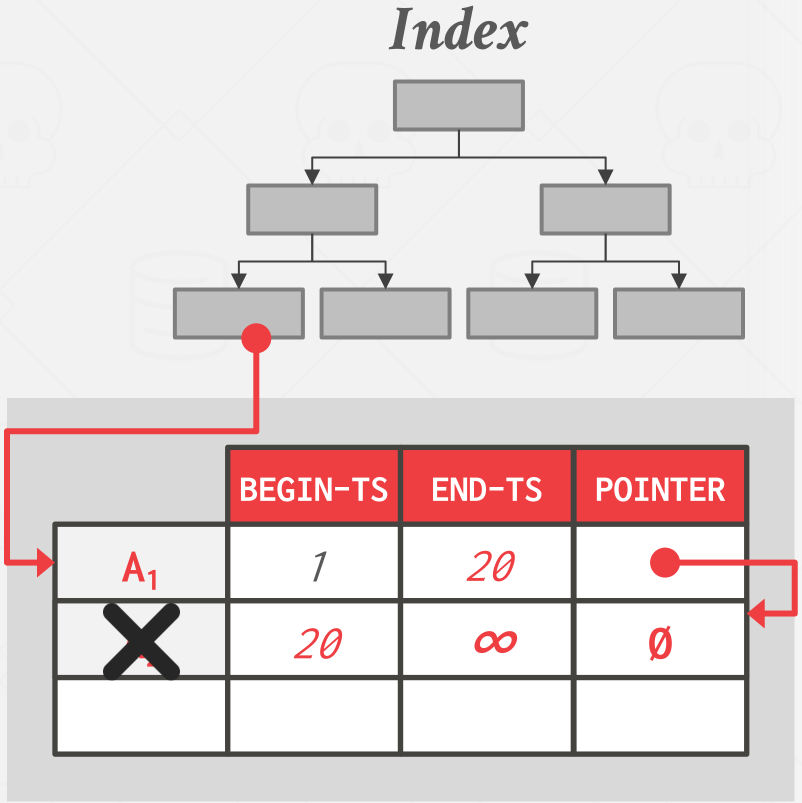
READ(A)

Txn #2

```
BEGIN @ 20
COMMIT @ 25
```

UPDATE(A)

DELETE(A)



MVCC DUPLICATE KEY PROBLEM

Txn #1

BEGIN @ 10

READ(A)

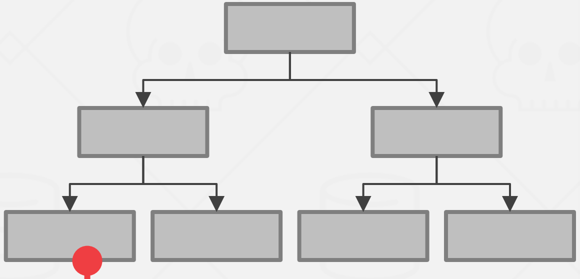
Txn #2

BEGIN @ 20
COMMIT @ 25

UPDATE(A)

DELETE(A)

Index



	BEGIN-TS	END-TS	POINTER
A_1	1	20	
~~A_2~~	20	20	Ø

MVCC DUPLICATE KEY PROBLEM

Txn #1

BEGIN @ 10

READ(A)

Txn #2

BEGIN @ 20
COMMIT @ 25

UPDATE(A)

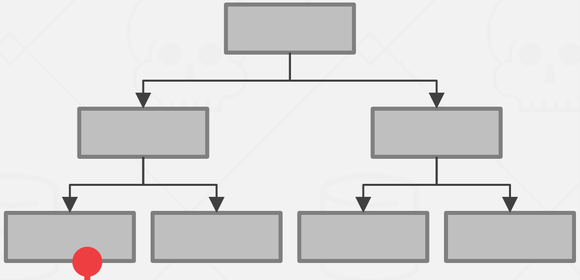
DELETE(A)

Txn #3

BEGIN @ 30

INSERT(A)

Index



	BEGIN-TS	END-TS	POINTER
A_1	1	20	
	20	20	Ø

MVCC DUPLICATE KEY PROBLEM

Txn #1

BEGIN @ 10

READ(A)

Txn #2

BEGIN @ 20
COMMIT @ 25

UPDATE(A)

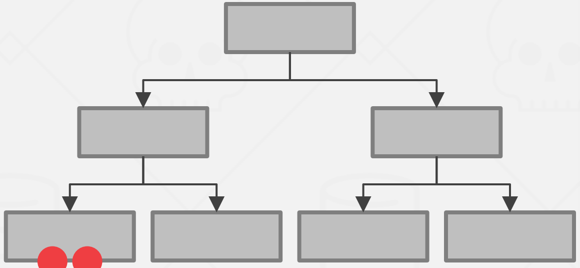
DELETE(A)

Txn #3

BEGIN @ 30

INSERT(A)

Index



	BEGIN-TS	END-TS	POINTER
A_1	1	20	•
~~A_2~~	20	20	Ø
A_1	30	∞	Ø

MVCC DUPLICATE KEY PROBLEM

Txn #1

BEGIN @ 10

READ(A) READ(A)

Txn #2

BEGIN @ 20
COMMIT @ 25

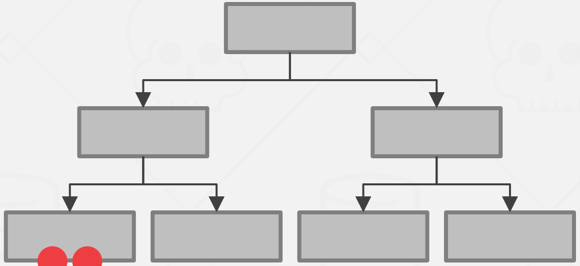
UPDATE(A) DELETE(A)

Txn #3

BEGIN @ 30

INSERT(A)

Index



	BEGIN-TS	END-TS	POINTER
A_1	1	20	•
	20	20	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 (2015)	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
CockroachDB	MV-2PL	Delta (LSM)	Compaction	Logical

CONCLUSION

MVCC is the widely used scheme in DBMSs.

Even systems that do not support multi-statement txns (e.g., NoSQL) use it.

IN-MEMORY MVCC

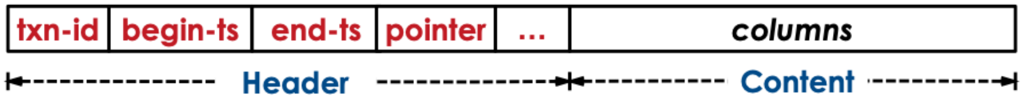
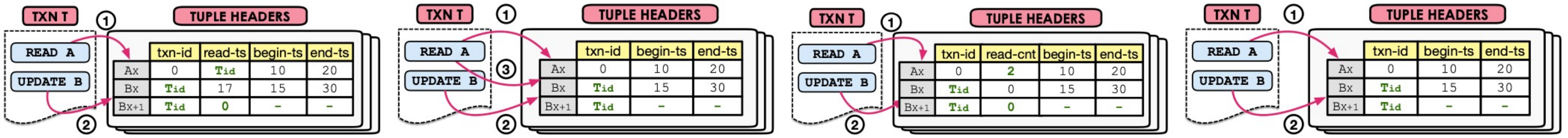


Figure 1: Tuple Format – The basic layout of a physical version of a tuple.



(a) Timestamp Ordering (b) Optimistic Concurrency Control (c) Two-phase Locking (d) Serialization Certifier

Figure 2: Concurrency Control Protocols – Examples of how the protocols process a transaction that executes a READ followed by an UPDATE.

The Hekaton Memory-Optimized OLTP Engine

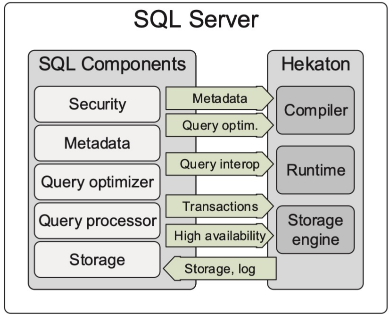
Per-Ake Larson
palarson@microsoft.com

Mike Zwilling
mikezw@microsoft.com

Kevin Farlee
kfarlee@microsoft.com

Abstract

Hekaton is a new OLTP engine optimized for memory resident data and fully integrated into SQL Server; a database can contain both regular disk-based tables and in-memory tables. In-memory (a.k.a.Hekaton) tables are fully durable and accessed using standard T-SQL. A query can reference both Hekaton tables and regular tables and a transaction can update data in both types of tables. T-SQL stored procedures that reference only Hekaton tables are compiled into machine code for further performance improvements. To allow for high concurrency the engine uses latch-free data structures and optimistic, multi-version concurrency control. This paper gives an overview of the design of the Hekaton engine and reports some initial results.



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

Computer architecture advancements has led to the rise of multi-core, in-memory DBMSs that employ efficient transaction management mechanisms to maximize parallelism without sacrificing serializability. The most popular scheme used in DBMSs developed in the last decade is *multi-version concurrency control* (MVCC). The basic idea of MVCC is that the DBMS maintains multiple physical versions of each logical object in the database to allow operations on the same object to proceed in parallel. These objects can be at any granularity, but almost every MVCC DBMS uses tuples because it provides a good balance between parallelism versus the overhead of version tracking. Multi-versioning allows read-only transactions to access older versions of tuples without preventing read-write transactions from simultaneously generating newer versions. Contrast this with a single-version system where transactions always overwrite a tuple with new information whenever they update it.

What is interesting about this trend of recent DBMSs using MVCC is that the scheme is not new. The first mention of it appeared

Proceedings of the VLDB Endowment, Vol. 10, No. 7

...and the first implementation started in ...DBMS (now open-sourced as Firebird). ...e of the most widely deployed disk-...uding Oracle (since 1984 [4]), Postgres ...QL's InnoDB engine (since 2001). But ...ntemporaries to these older systems ...eme (e.g., IBM DB2, Sybase), almost ...MS eschews this approach in favor of ...oth commercial (e.g., Microsoft Heka-...emSQL [1], NuoDB [3]) and academic ...36]) systems.

...ystems using MVCC, there is no one ... There are several design choices that ...d performance behaviors. Until now, ...hensive evaluation of MVCC in a mod-...onment. The last extensive study was ...ed simulated workloads running in a ... single CPU core. The design choices ...MSs are inappropriate for in-memory ...ne with a large number of CPU cores. ... does not reflect recent trends in latch-free [27] and serializable [20] concurrency control, as well as in-memory storage [36] and hybrid workloads [40].

In this paper, we perform such a study for key transaction management design decisions in of MVCC DBMSs: (1) concurrency control protocol, (2) version storage, (3) garbage collection, and (4) index management. For each of these topics, we describe the state-of-the-art implementations for in-memory DBMSs and discuss their trade-offs. We also highlight the issues that prevent them from scaling to support larger thread counts and more complex workloads. As part of this investigation, we implemented all of the approaches in the **Peloton** [5] in-memory MVCC DBMS. This provides us with a uniform platform to compare implementations that is not encumbered by other architecture facets. We deployed Peloton on a machine with 40 cores and evaluate it using two OLTP benchmarks. Our analysis identifies the scenarios that stress the implementations and discuss ways to mitigate them (if it all possible).

2. BACKGROUND

We first provide an overview of the high-level concepts of MVCC. We then discuss the meta-data that the DBMS uses to track transactions and maintain versioning information.

2.1 MVCC Overview

A transaction management scheme permits end-users to access a database in a multi-programmed fashion while preserving the illu-

NEXT CLASS

Logging and recovery!