Carnegie Mellon University

Database Systems

15-445/645 SPRING 2026

ANDY PAVLO
JIGNESH PATEL

Lecture #21

Concurrency Control: Multi-Versioning – Part II

ADMINISTRIVIA

Homework #5 is due Sunday Apr 12th @ 11:59pm

Project #4 is due Sunday Apr 26th @ 11:59pm

Final Exam is on Tuesday Apr 28th @ 5:30-8:30pm

UPCOMING DATABASE TALKS

SpacetimeDB (DB Seminar)
→ Monday April 6th @ 4:30pm ET
→ Zoom

Multigres (DB Seminar)
→ Monday April 13th @ 4:30pm ET
→ Zoom

VillageSQL (DB Seminar)
→ Monday April 20th @ 4:30pm ET
→ Zoom

LAST CLASS

We introduced the concept of multi-versioning in a DBMS and how it effects concurrency control.
→ **Writers don't block readers.**
→ **Readers don't block writers.**

We also introduced Snapshot Isolation and how it is susceptible to the **write-skew anomaly**.
→ Read-only txns read a consistent snapshot without acquiring locks or txn ids using timestamps to determine visibility.

TODAY'S AGENDA

Version Storage
Garbage Collection
Compaction
Index Management
Deletes

VERSION STORAGE

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

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

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

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

VERSION STORAGE

Approach #1: Append-Only Storage

→ New versions are appended to the same table space.

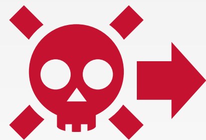
Approach #2: Time-Travel Storage

→ Old versions are copied to separate table space.

Approach #3: Delta Storage

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

VERSION STORAGE



Don't Do This!

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

→ New versions are appended to the same table space.

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

→ Old versions are copied to separate table space.

Approach #3: Delta Storage ← *Common*

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

VERSION STORAGE

Don't Do This!

Approach #1: Append-Onl

→ New versions are appended to

Approach #2: Time-Trave

→ Old versions are copied to sep

Approach #3: Delta Storag

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

Andy Pavlo

The Part of PostgreSQL We Hate the Most

Posted on April 26, 2023

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

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

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

APPEND-ONLY STORAGE

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

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

Main Table

	value	*pointer*
A_0	\$111	→ A_1
A_1	\$222	Ø
B_1	\$10	Ø

APPEND-ONLY STORAGE

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

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

Main Table

	value	*pointer*
A_0	$111	●
A_1	$222	Ø
B_1	$10	Ø
A_2	$333	Ø

APPEND-ONLY STORAGE

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

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

Main Table

	value	*pointer*
A_0	\$111	•
A_1	\$222	•
B_1	\$10	Ø
A_2	\$333	Ø

VERSION CHAIN ORDERING

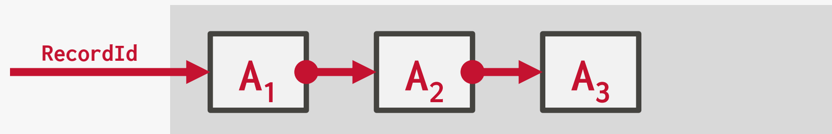
Approach #1: Oldest-to-Newest (O2N)

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

VERSION CHAIN ORDERING

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

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

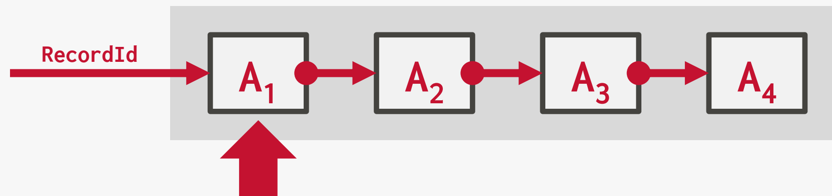


VERSION CHAIN ORDERING

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

→ Append new version to end of the chain.

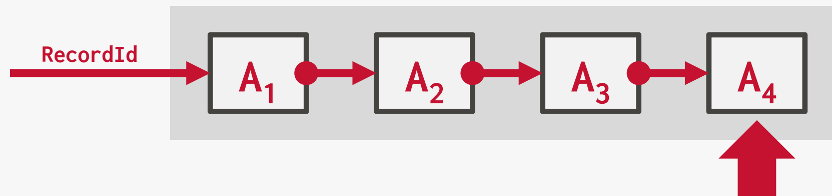
→ Must traverse chain on look-ups.



VERSION CHAIN ORDERING

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

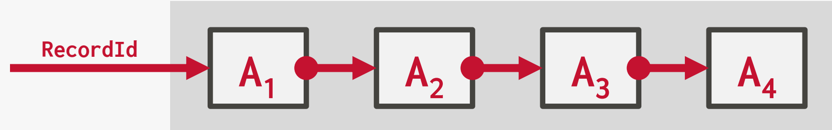
→ Append new version to end of the chain.
→ Must traverse chain on look-ups.



VERSION CHAIN ORDERING

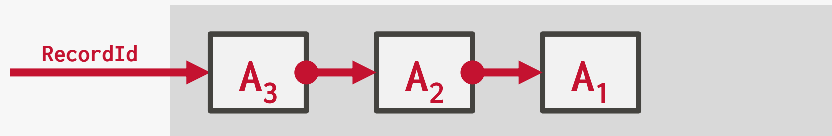
Approach #1: Oldest-to-Newest (O2N)

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



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

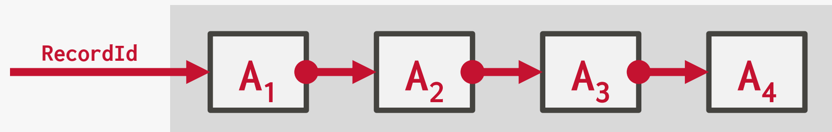
→ Must update index pointers for every new version.
→ Do not have to traverse chain on look-ups.
→ Better approach if most txns only want the newest version.



VERSION CHAIN ORDERING

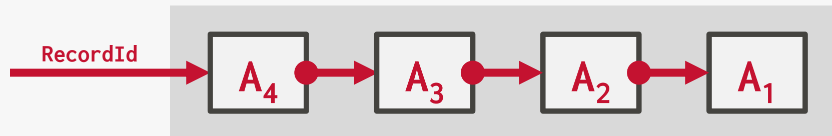
Approach #1: Oldest-to-Newest (O2N)

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



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

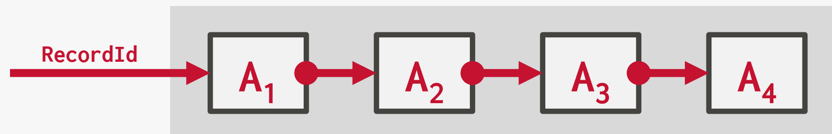
→ Must update index pointers for every new version.
→ Do not have to traverse chain on look-ups.
→ Better approach if most txns only want the newest version.



VERSION CHAIN ORDERING

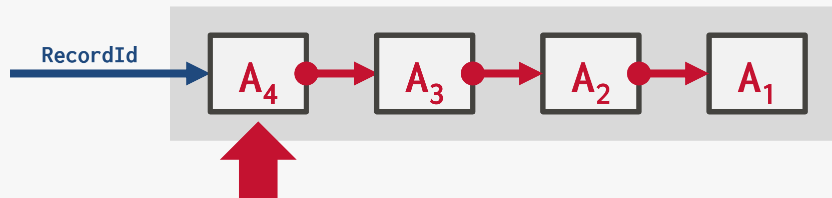
Approach #1: Oldest-to-Newest (O2N)

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



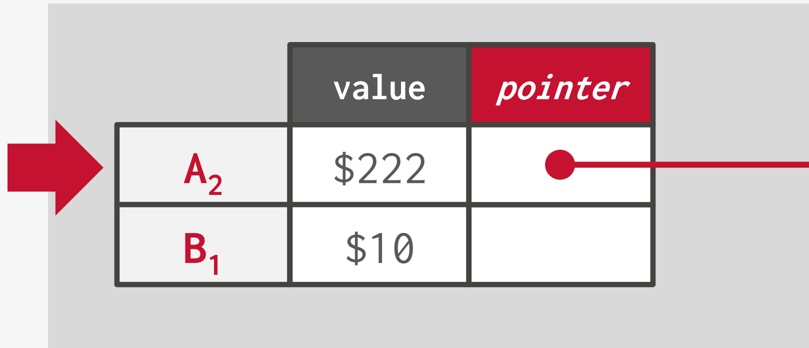
Approach #2: Newest-to-Oldest (N2O)

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

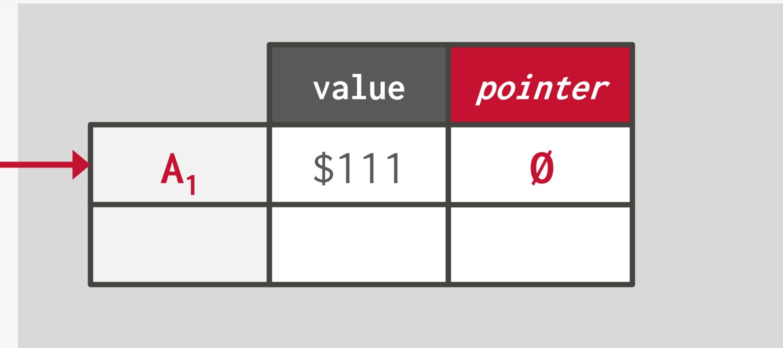


TIME-TRAVEL STORAGE

Main Table

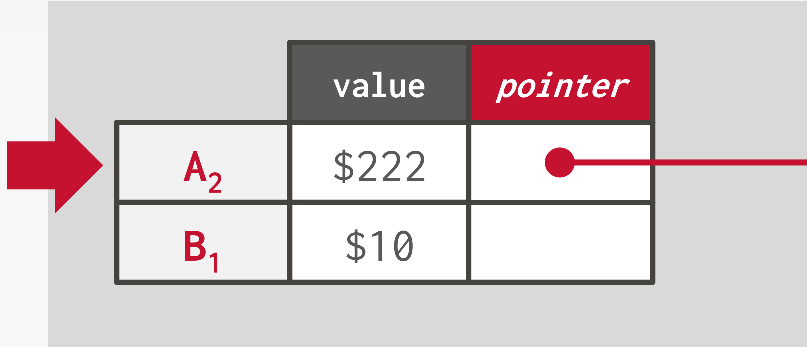


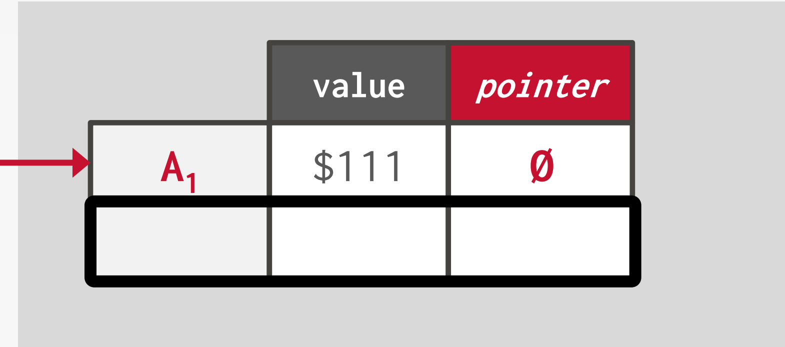
Time-Travel Table



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

TIME-TRAVEL STORAGE

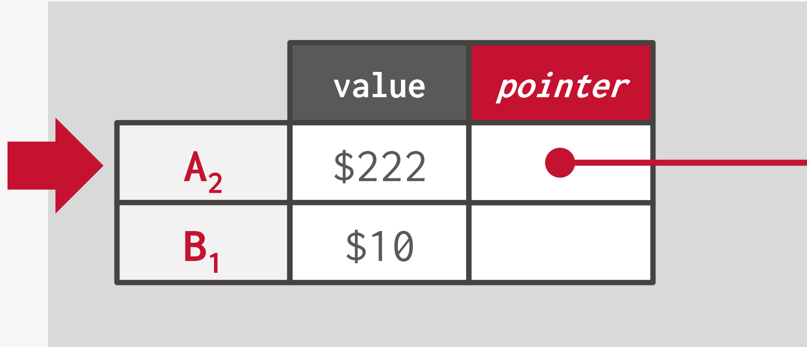




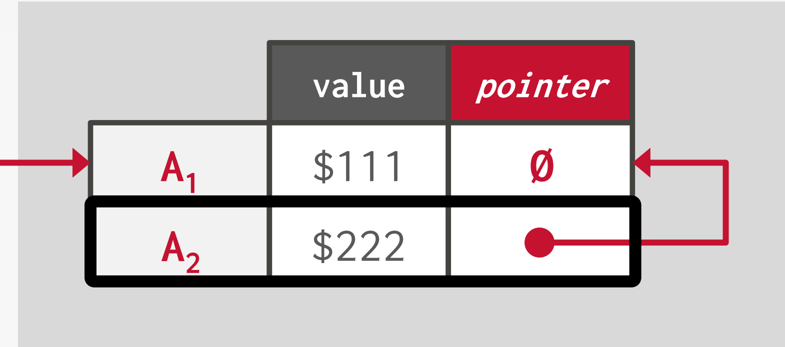
On every update, copy the current version to the time-travel table. Update pointers.

TIME-TRAVEL STORAGE

Main Table



Time-Travel Table



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

TIME-TRAVEL STORAGE

Main Table

	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	• → A_1
B_1	$10	

Time-Travel Table

	value	*pointer*
A_1	$111	Ø
A_2	$222	• → A_1

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	• (→ A_2)
B_1	$10	

Time-Travel Table

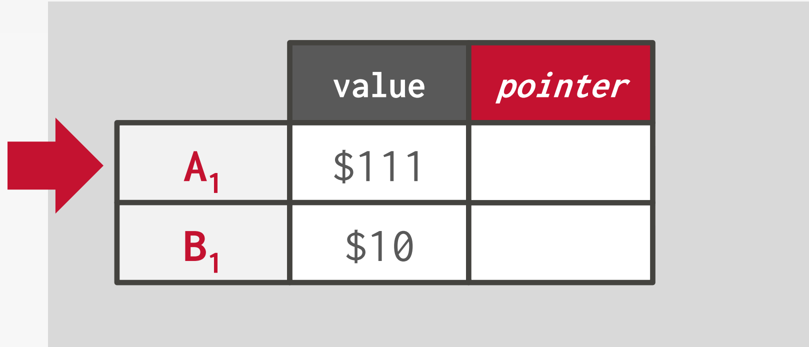
	value	*pointer*
A_1	$111	Ø
A_2	$222	• (→ A_1)

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

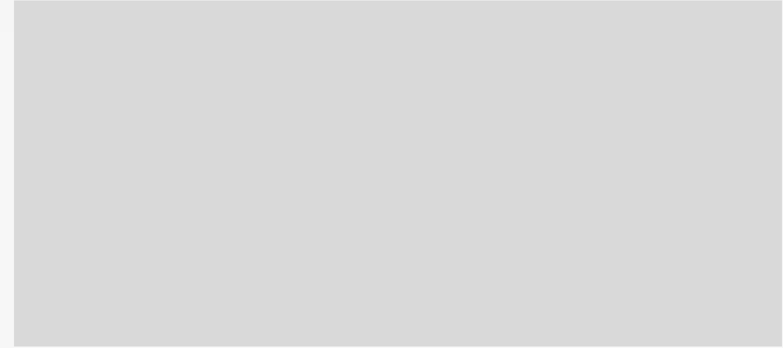
Overwrite master version in the main table and update pointers.

DELTA STORAGE

Main Table



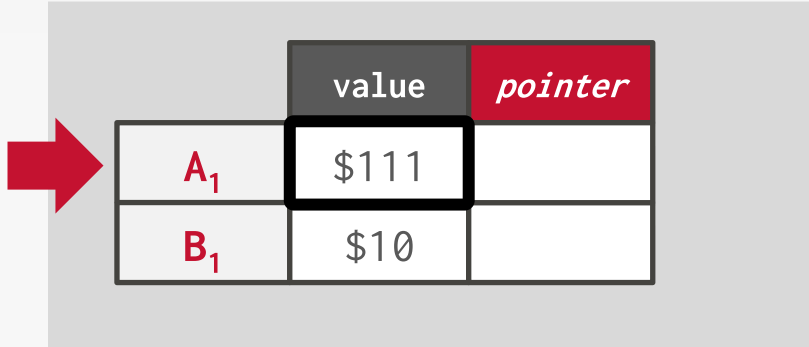
Delta Storage Segment



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

DELTA STORAGE

Main Table



Delta Storage Segment

	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	• → A_1
B_1	$10	

Delta Storage Segment

	delta	*pointer*
A_1	*(VALUE→$111)*	Ø
A_2	*(VALUE→$222)*	• → A_1

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

DELTA STORAGE

Main Table

	value	*pointer*
A_2	\$222	• → A_1
B_1	\$10	

Delta Storage Segment

	delta	*pointer*
A_1	*(VALUE→\$111)*	Ø
A_2	*(VALUE→\$222)*	• → A_1

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

DELTA STORAGE

Main Table

	value	*pointer*
A_3	$333	● → A_2
B_1	$10	

Delta Storage Segment

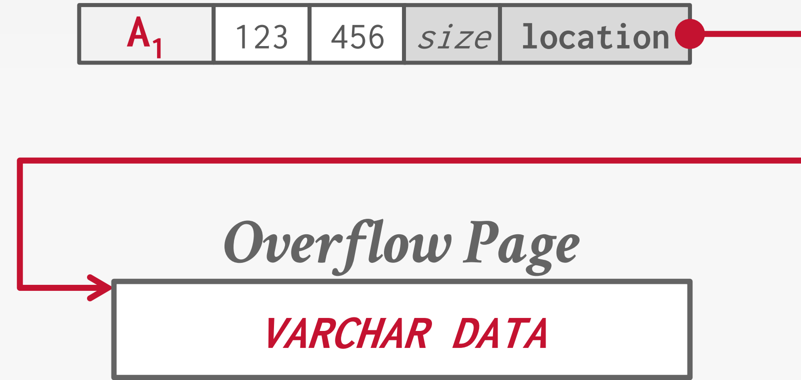
	delta	*pointer*
A_1	(VALUE→$111)	Ø
A_2	(VALUE→$222)	● → A_1

On every update, copy only the 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.

VERSIONING WITH LARGE VALUES

If a txn does not update data stored in an overflow page, then the DBMS will reuse the overflow pointer instead of copying the data.



Approach #1: Reference Counting

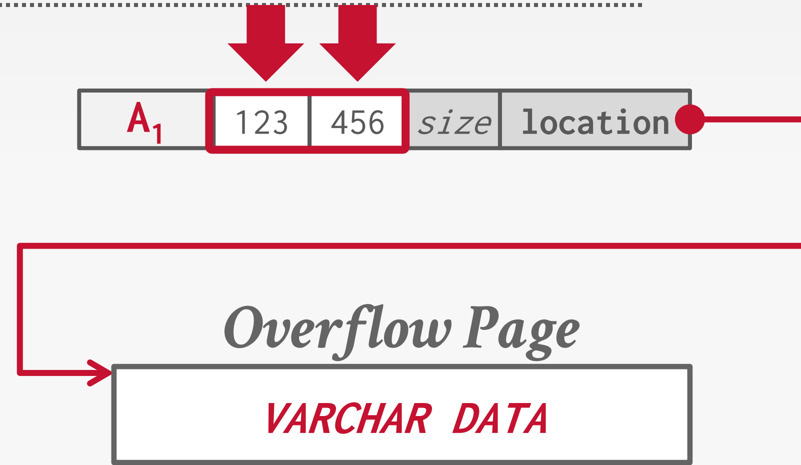
→ Use internal table to track which physical versions point to an overflow page.

Approach #2: Garbage Collection

→ When removing a physical version, check if any other version also points to the same overflow page.

VERSIONING WITH LARGE VALUES

If a txn does not update data stored in an overflow page, then the DBMS will reuse the overflow pointer instead of copying the data.



Approach #1: Reference Counting

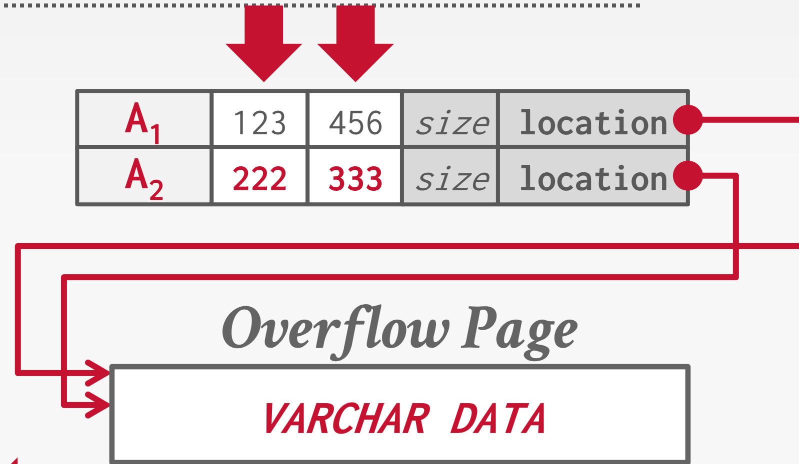
→ Use internal table to track which physical versions point to an overflow page.

Approach #2: Garbage Collection

→ When removing a physical version, check if any other version also points to the same overflow page.

VERSIONING WITH LARGE VALUES

If a txn does not update data stored in an overflow page, then the DBMS will reuse the overflow pointer instead of copying the data.



Approach #1: Reference Counting — *Rare*

→ Use internal table to track which physical versions point to an overflow page.

Approach #2: Garbage Collection — *Common*

→ When removing a physical version, check if any other version also points to the same overflow page.

OBSERVATION

Maintaining old versions causes will cause performance issues for txns in the DBMS over time.

→ Increased Memory Usage
→ Longer Version Chains
→ Garbage Collector CPU Spikes
→ Poor Time-based Version Locality

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 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

Txn #1

TS=12

Txn #2

TS=25

Vacuum

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

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

TUPLE-LEVEL GC

Txn #1

TS=12

Txn #2

TS=25

Vacuum

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

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

TUPLE-LEVEL GC

Txn #1

TS=12

Txn #2

TS=25

Vacuum

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

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

TUPLE-LEVEL GC

Txn #1

TS=12

Txn #2

TS=25

Vacuum

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

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

TUPLE-LEVEL GC

Txn #1

TS=12

Txn #2

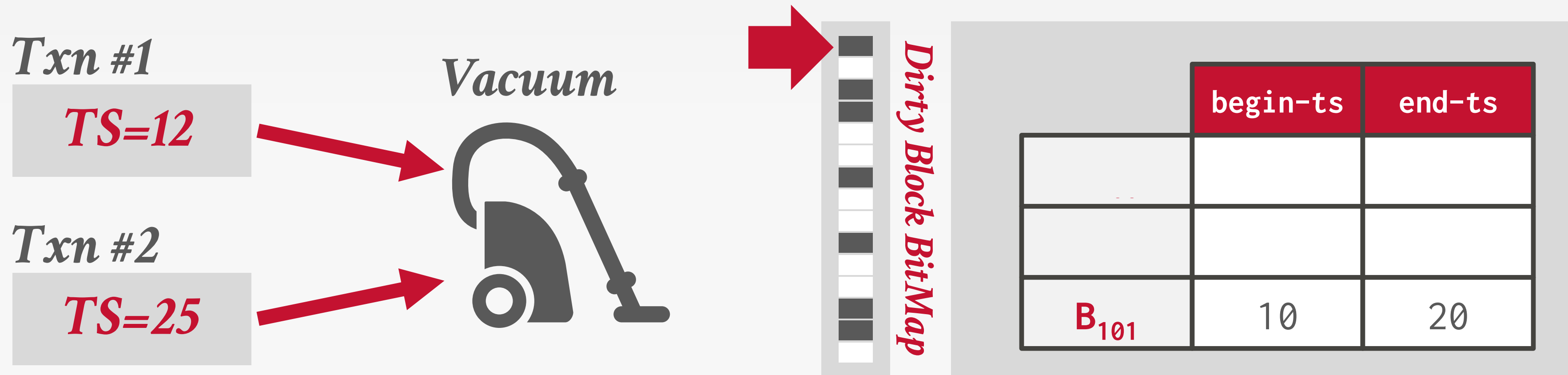
TS=25

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



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

TUPLE-LEVEL GC

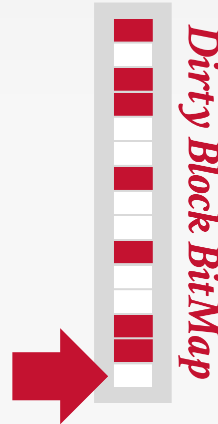
Txn #1

TS=12

Txn #2

TS=25

Vacuum



	begin-ts	end-ts
B_{101}	10	20

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

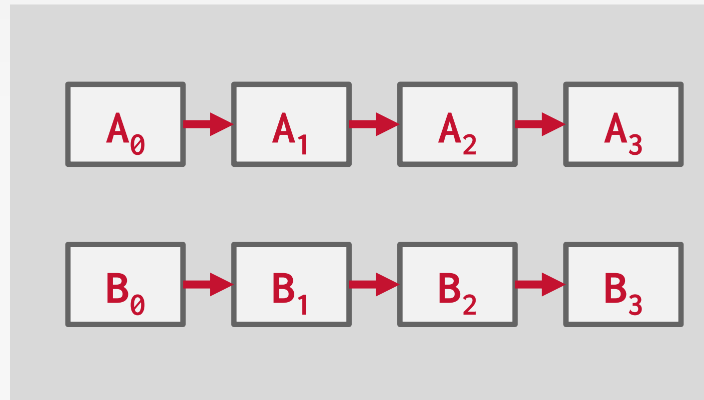
TUPLE-LEVEL GC

Txn #1

TS=12

Txn #2

TS=25



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

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

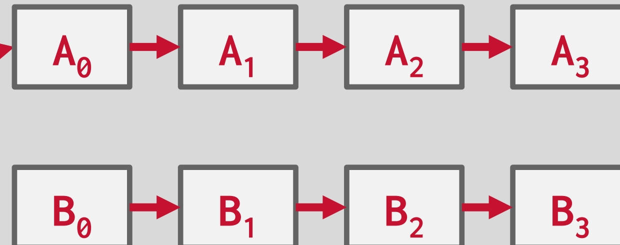
TUPLE-LEVEL GC

Txn #1

TS=12

Txn #2

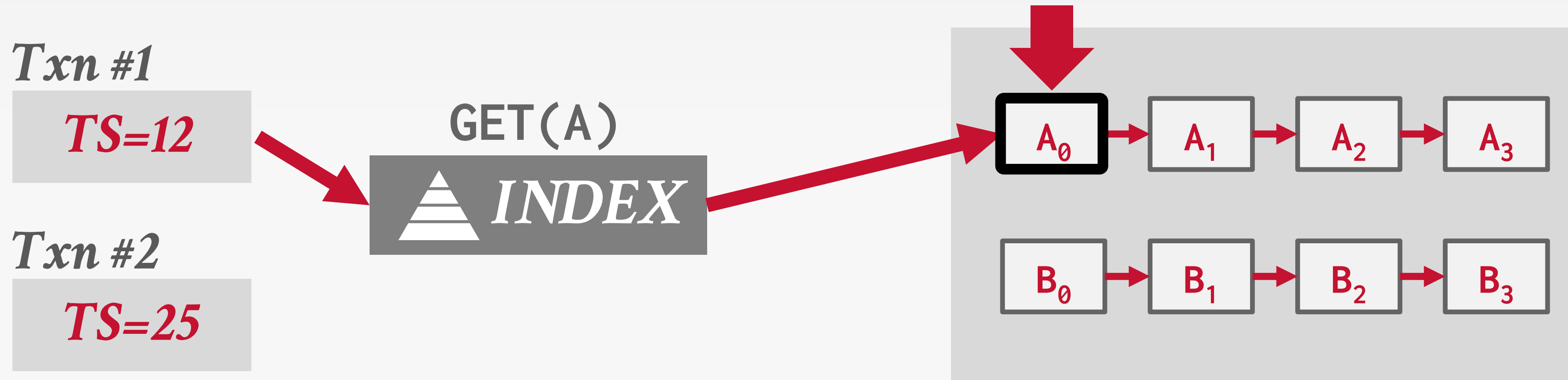
TS=25



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

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

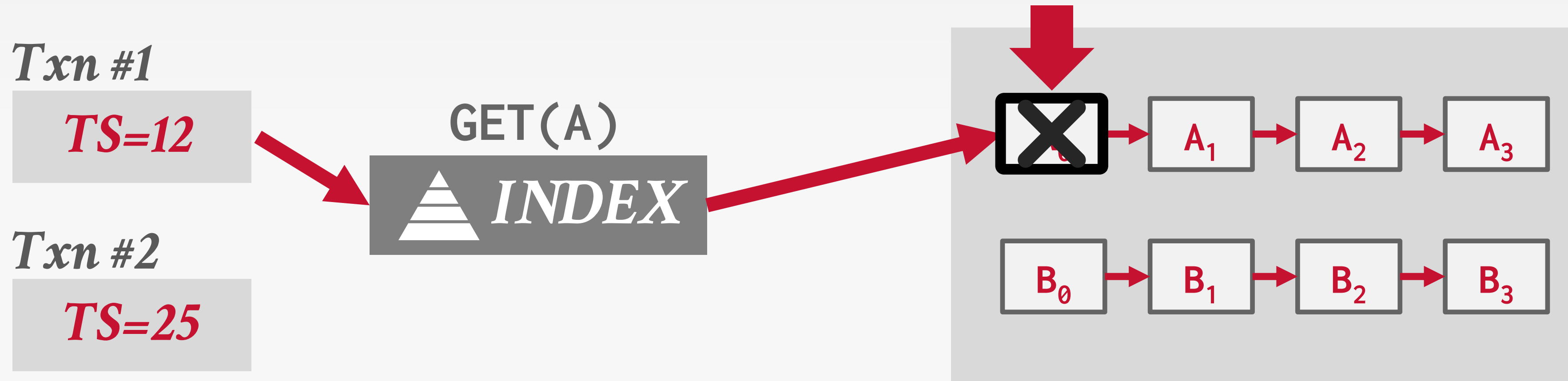
TUPLE-LEVEL GC



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

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

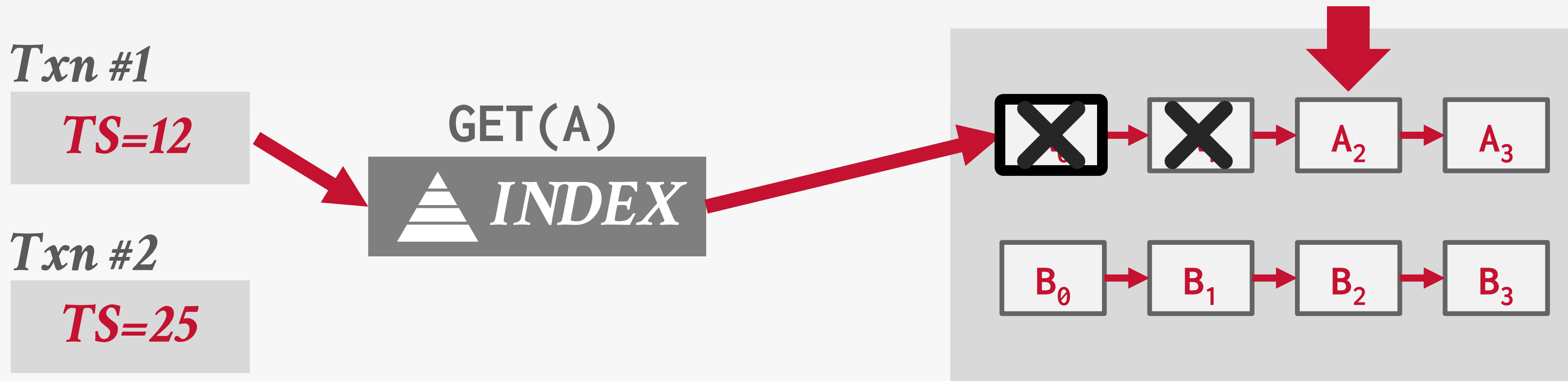
TUPLE-LEVEL GC



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

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

TUPLE-LEVEL GC



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

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

TUPLE-LEVEL GC

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

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

TRANSACTION-LEVEL GC

Each txn keeps track of its read/write set.

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

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

TRANSACTION-LEVEL GC

Txn #1

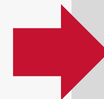
BEGIN TS=10

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

TRANSACTION-LEVEL GC

Txn #1

BEGIN TS=10

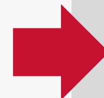


	begin-ts	end-ts	value
A_2	1	∞	-
B_6	8	∞	-

TRANSACTION-LEVEL GC

Txn #1

BEGIN TS=10



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

TRANSACTION-LEVEL GC

Txn #1

BEGIN TS=10

Old Versions

A_2

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

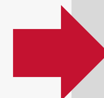
TRANSACTION-LEVEL GC

Txn #1

BEGIN TS=10

Old Versions

A_2



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

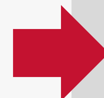
TRANSACTION-LEVEL GC

Txn #1

BEGIN TS=10

Old Versions

A_2



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

TRANSACTION-LEVEL GC

Txn #1

BEGIN TS=10

Old Versions

A_2

B_6

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

TRANSACTION-LEVEL GC

Txn #1

BEGIN TS=10
COMMIT TS=15

Old Versions

A_2

B_6

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

TRANSACTION-LEVEL GC

Txn #1

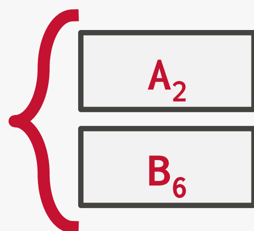
BEGIN TS=10
COMMIT TS=15

Old Versions

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

Vacuum

TS<10



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.

Run GC before the DBMS gets close to this upper limit. Otherwise, it must stop accepting new commands when the system gets close to the max txn id.

OBSERVATION

If the application deletes a tuple, then what should the DBMS do with the slots occupied by that tuple's versions?

→ Always reuse variable-length data slots.
→ More nuanced for fixed-length data slots.

What if the application deletes many (but not all) tuples in a table in a short amount of time?

MVCC DELETED TUPLES

Approach #1: Reuse Slot
- → Allow workers to insert new tuples in the empty slots.
- → Obvious choice for append-only storage since there is no distinction between versions.
- → Destroys temporal locality of tuples in delta storage.

Approach #2: Leave Slot Unoccupied
- → Workers can only insert new tuples in slots that were not previously occupied.
- → Ensures that tuples in the same block were inserted into the database at around the same time.
- → Need an extra mechanism to fill holes.

BLOCK COMPACTION

Consolidating less-than-full blocks into fewer blocks and then returning empty blocks to storage (e.g., OS).

- → Move data using **DELETE** + **INSERT** to ensure transactional guarantees during consolidation.
- → Example: PostgreSQL's **VACUUM FULL**

Ideally the DBMS will want to store tuples that are likely to be accessed together within a window of time together in the same block.

BLOCK COMPACTION: TARGETS

Approach #1: Time Since Last Update

→ Leverage the **BEGIN-TS** in each tuple.

Approach #2: Time Since Last Access

→ Expensive to maintain unless tuple has **READ-TS**.

Approach #3: Application-level Semantics

→ Tuples from the same table that are related to each other according to some higher-level construct.

→ Difficult to figure out automatically.

BLOCK COMPACTION: TRUNCATE

TRUNCATE operation removes all tuples in a table.
→ Think of it like a **DELETE** without a **WHERE** clause.

Fastest way to execute is to drop the table and then create it again.
→ Do not need to track the visibility of individual tuples.
→ GC will free all storage when there are no active txns that exist before the drop operation.
→ If the catalog is transactional, then this easy to do.

INDEX MANAGEMENT

Primary key indexes point to version chain head.

- → How often the DBMS must update the pkey index depends on whether the system creates new versions when a tuple is updated.
- → If a txn updates a tuple's pkey attribute(s), then this is treated as a **DELETE** followed by an **INSERT**.

Secondary indexes are more complicated…

IND

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

Secondary inde

UBER Engineering

JOIN THE TEAM

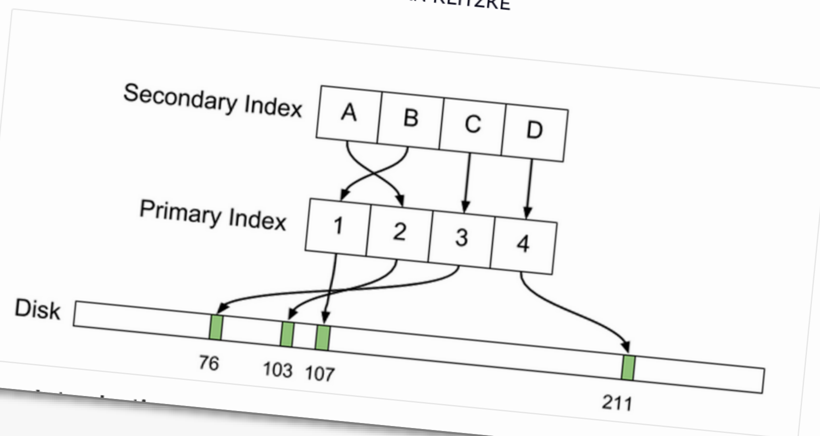
MEET THE PEOPLE

ARCHITECTURE

WHY UBER ENGINEERING SWITCHED FROM POSTGRES TO MYSQL

JULY 26, 2016

BY EVAN KLITZKE



SECONDARY INDEXES

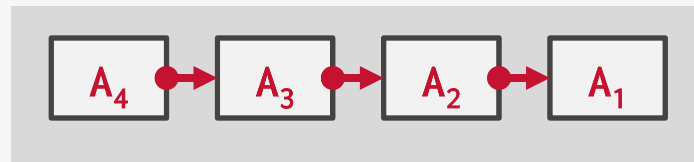
Approach #1: Logical Pointers
- → Use a fixed identifier per tuple that does not change.
- → Requires an extra indirection layer.
- → Primary Key vs. Tuple Id

Approach #2: Physical Pointers
- → Use the physical address to the version chain head.

INDEX POINTERS: APPEND-ONLY

PRIMARY INDEX

SECONDARY INDEX



Append-Only
Newest-to-Oldest

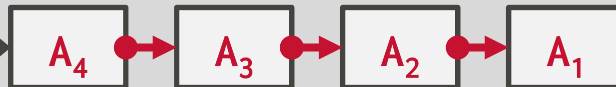
INDEX POINTERS: APPEND-ONLY

GET(A)

PRIMARY INDEX

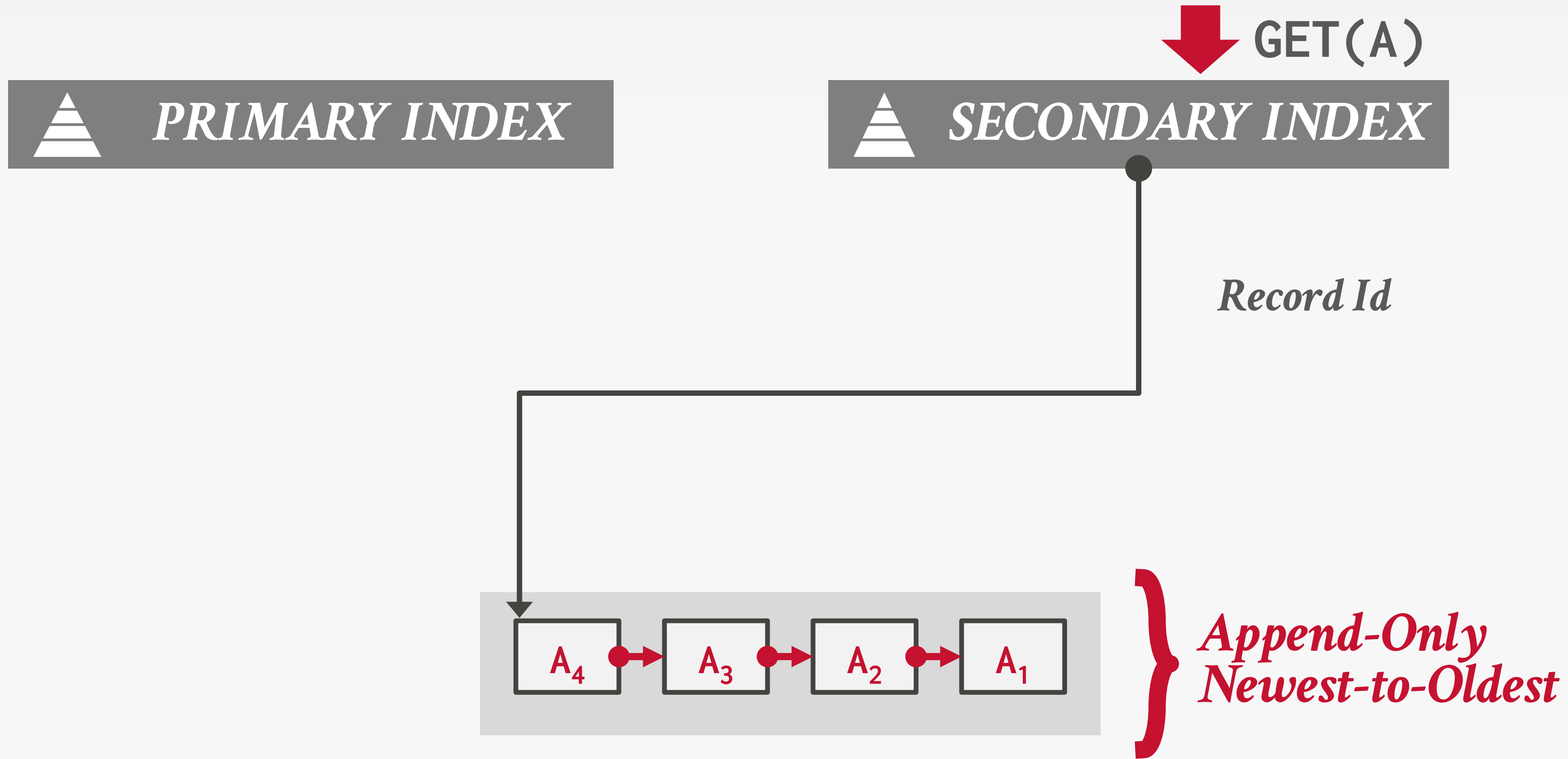
SECONDARY INDEX

Record Id

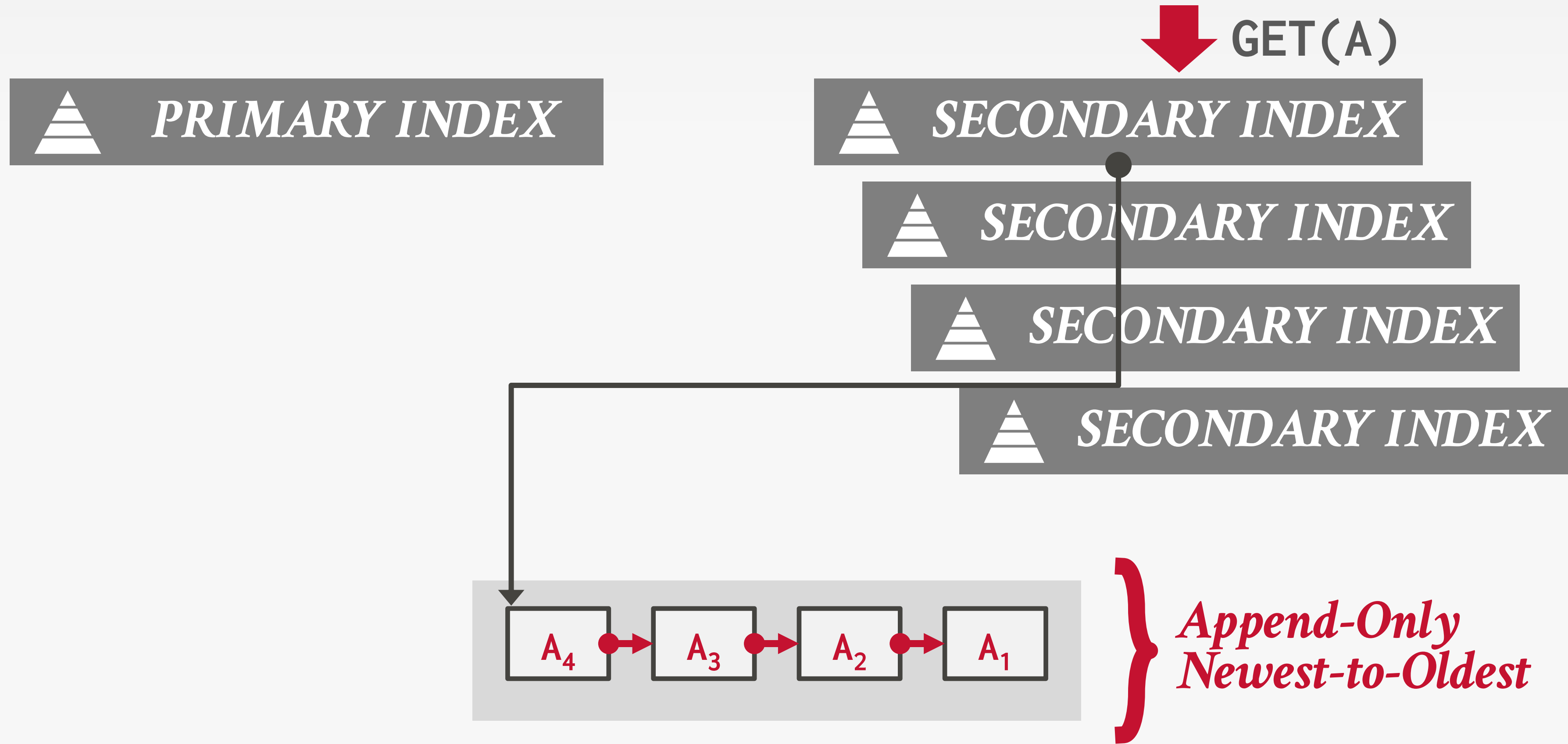


Append-Only
Newest-to-Oldest

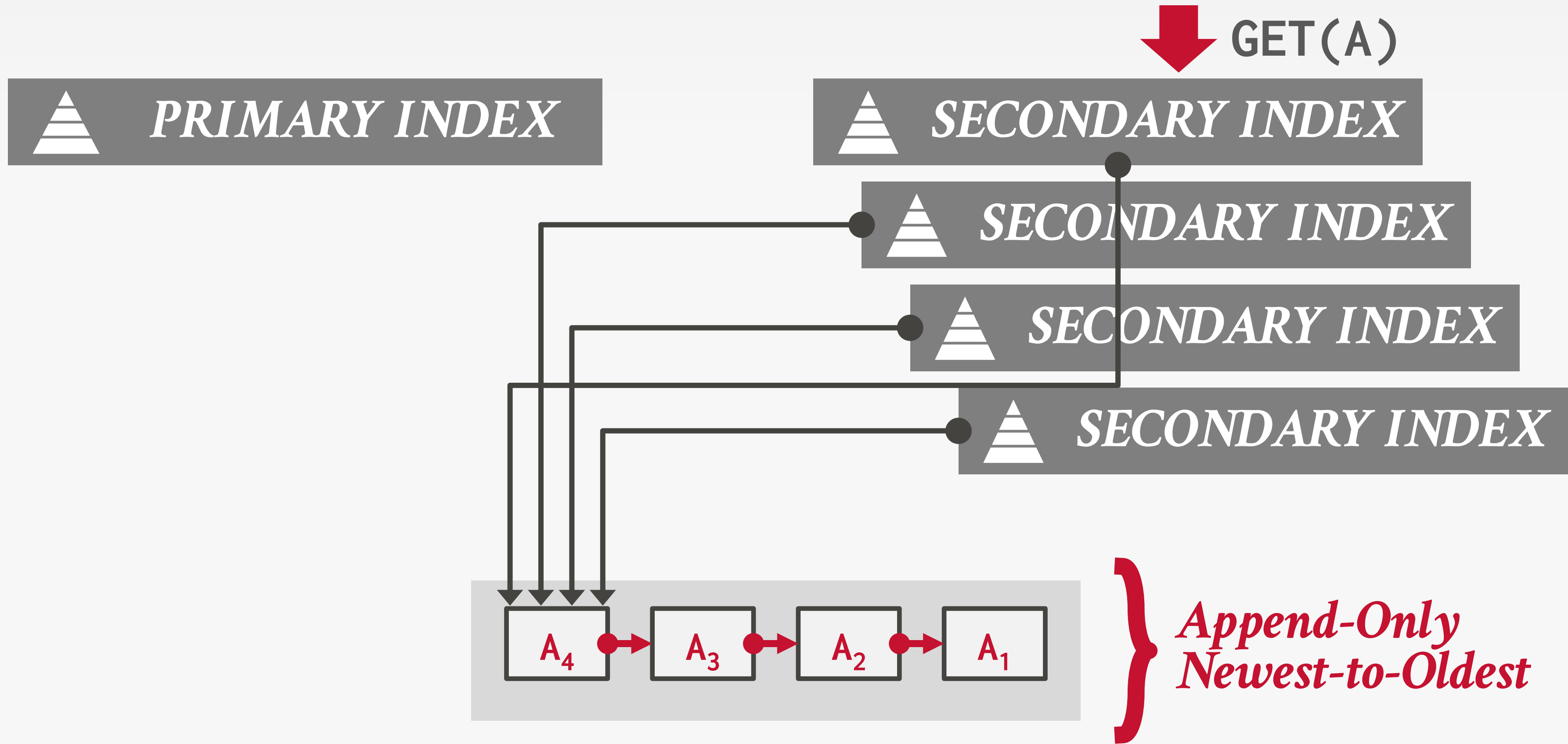
INDEX POINTERS: APPEND-ONLY



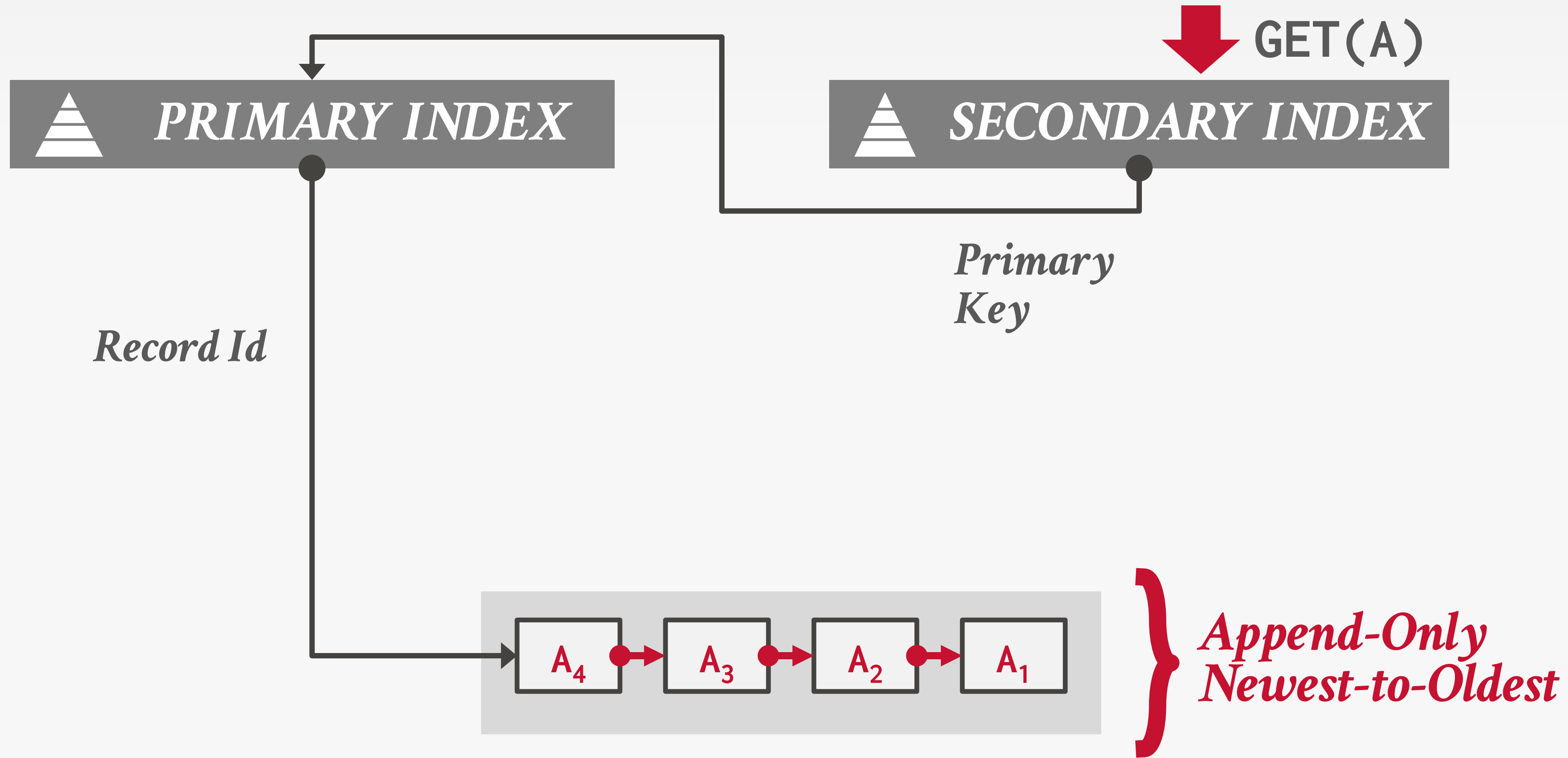
INDEX POINTERS: APPEND-ONLY



INDEX POINTERS: APPEND-ONLY



INDEX POINTERS: APPEND-ONLY

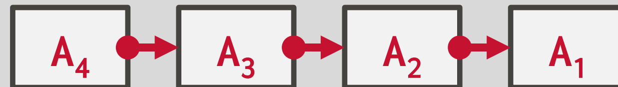


INDEX POINTERS: APPEND-ONLY

GET(A)

PRIMARY INDEX

SECONDARY INDEX



Append-Only
Newest-to-Oldest

MVCC INDEXES

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

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

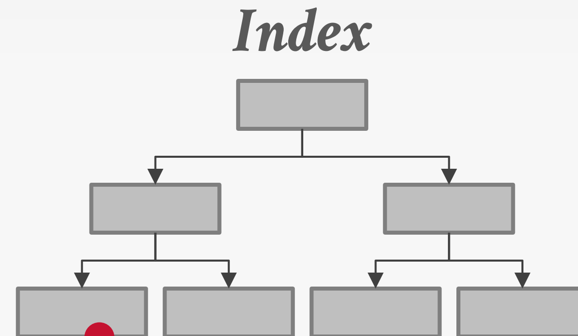
Every index must support duplicate keys from different snapshots:

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

MVCC DUPLICATE KEY PROBLEM

Txn #1

BEGIN TS=10



	begin-ts	end-ts	pointer
A_1	1	∞	Ø

MVCC DUPLICATE KEY PROBLEM

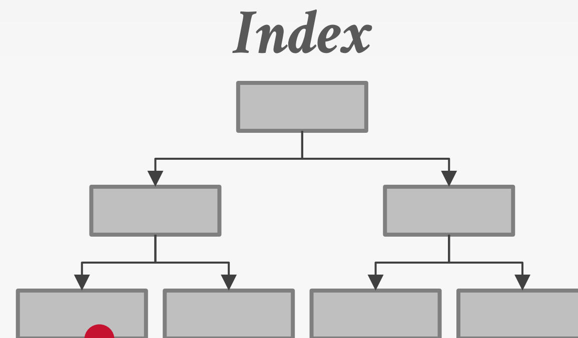
Txn #1

```
BEGIN TS=10
```

Txn #2

```
BEGIN TS=20
```

Index



	begin-ts	end-ts	pointer
A_1	1	∞	Ø

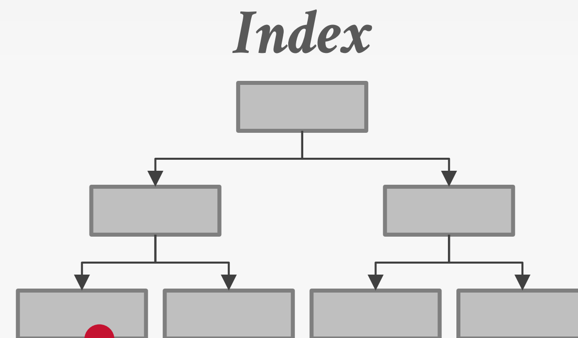
MVCC DUPLICATE KEY PROBLEM

Txn #1

BEGIN TS=10

Txn #2

BEGIN TS=20



	begin-ts	end-ts	pointer
A_1	1	∞	•
A_2	20	∞	Ø

MVCC DUPLICATE KEY PROBLEM

Txn #1

BEGIN TS=10

READ(A)

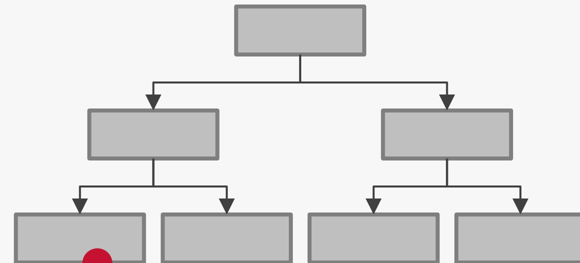
Txn #2

BEGIN TS=20

UPDATE(A)

DELETE(A)

Index



	begin-ts	end-ts	pointer
A_1	1	∞	●
[crossed out]	20	∞	Ø

MVCC DUPLICATE KEY PROBLEM

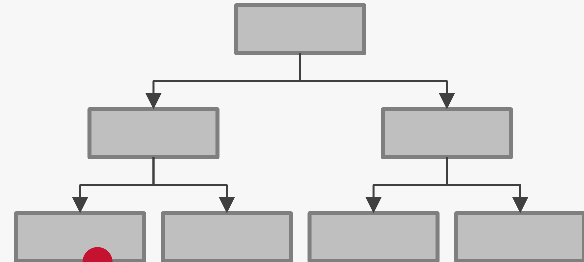
Txn #1

BEGIN TS=10

Txn #2

BEGIN TS=20

COMMIT TS=25



	begin-ts	end-ts	pointer
A_1	1	∞	●
	20	∞	Ø

MVCC DUPLICATE KEY PROBLEM

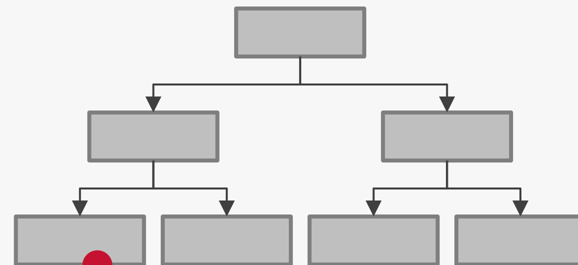
Txn #1

BEGIN TS=10

Txn #2

BEGIN TS=20

COMMIT TS=25



	begin-ts	end-ts	pointer
A_1	1	20	•
	20	20	Ø

MVCC DUPLICATE KEY PROBLEM

Txn #1

BEGIN TS=10

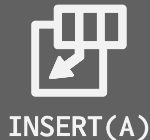
Txn #2

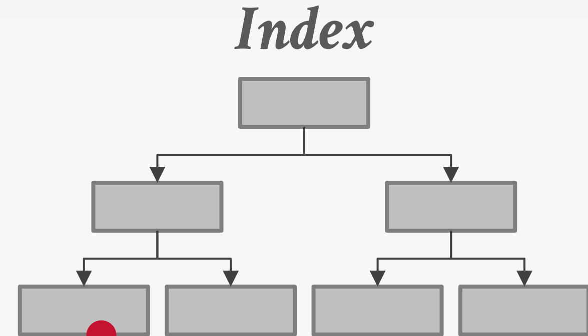
BEGIN TS=20

COMMIT TS=25

Txn #3

BEGIN TS=30





	begin-ts	end-ts	pointer
A_1	1	20	•
~~A_2~~	20	20	Ø

MVCC DUPLICATE KEY PROBLEM

Txn #1

BEGIN TS=10

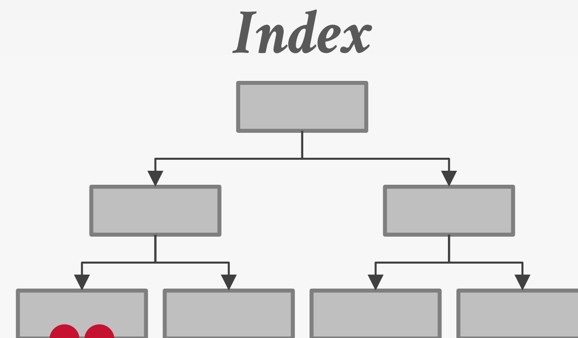
Txn #2

BEGIN TS=20

COMMIT TS=25

Txn #3

BEGIN TS=30



	begin-ts	end-ts	pointer
A_1	1	20	•
~~A_2~~	20	20	Ø
A_3	30	∞	Ø

MVCC DUPLICATE KEY PROBLEM

Txn #1

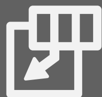
```
BEGIN TS=10
```

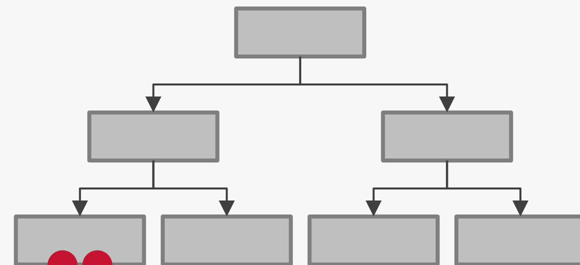
Txn #2

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

Txn #3

```
BEGIN TS=30
```





	begin-ts	end-ts	pointer
A_1	1	20	•
~~A_2~~	20	20	Ø
A_3	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: MARKERS

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.

MVV DELETES: INDEX CLEAN-UP

The DBMS must remove a tuples' keys from indexes when their corresponding versions are no longer visible to active txns.

Track the txn's modifications to individual indexes to support GC of older versions on commit and removal modifications on abort.

MVCC IMPLEMENTATIONS

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

CONCLUSION

There are other implementation factors for an MVCC DBMS beyond the design decisions we discussed today.

Need to balance the trade-offs between indirection and performance.

NEXT CLASS

Logging and recovery!