Lecture #05

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Multi-Version Concurrency Control (Garbage Collection)

@Andy_Pavlo // 15-721 // Spring 2019

MVCC GARBAGE COLLECTION

A MVCC 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.

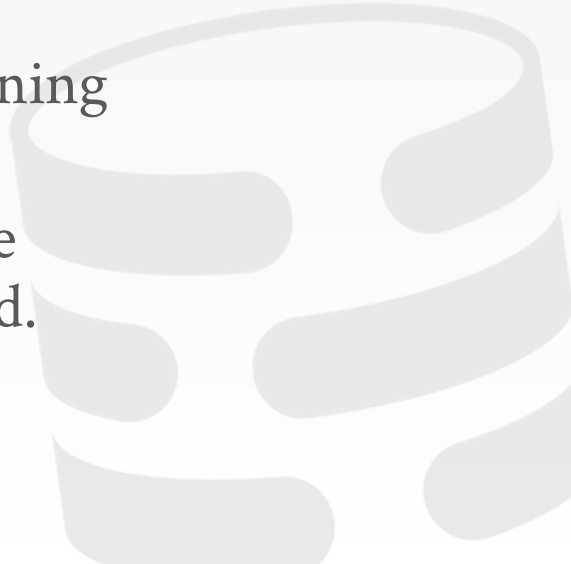
The DBMS uses the tuples' version meta-data to decide whether it is visible.

OBSERVATION

We have assumed that queries / txns will complete in a short amount of time. This means that the lifetime of an obsolete version is short as well.

But HTAP workloads may have long running queries that access old snapshots.

Such queries block the traditional garbage collection methods that we have discussed.



PROBLEMS WITH OLD VERSIONS

Increased Memory Usage

Memory Allocator Contention

Longer Version Chains

Garbage Collector CPU Spikes

Poor Time-based Version Locality

TODAY'S AGENDA

MVCC Deletes
Indexes with MVCC Tables
Garbage Collection
Block Compaction

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 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:

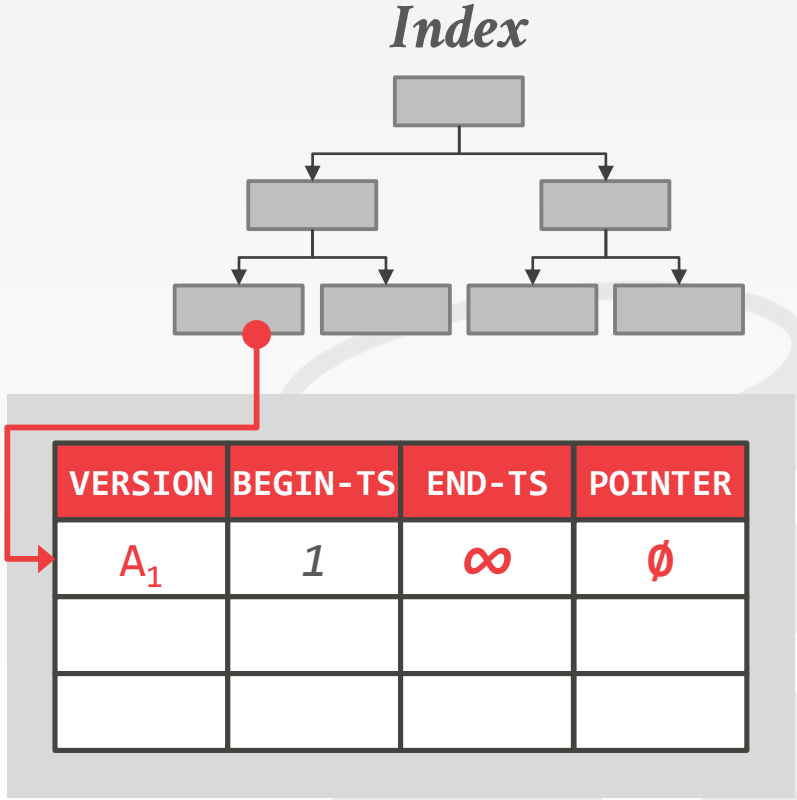
→ The same key may point to different logical tuples in different snapshots.

MVCC DUPLICATE KEY PROBLEM

Thread #1

Begin @ 10

READ(A)



MVCC DUPLICATE KEY PROBLEM

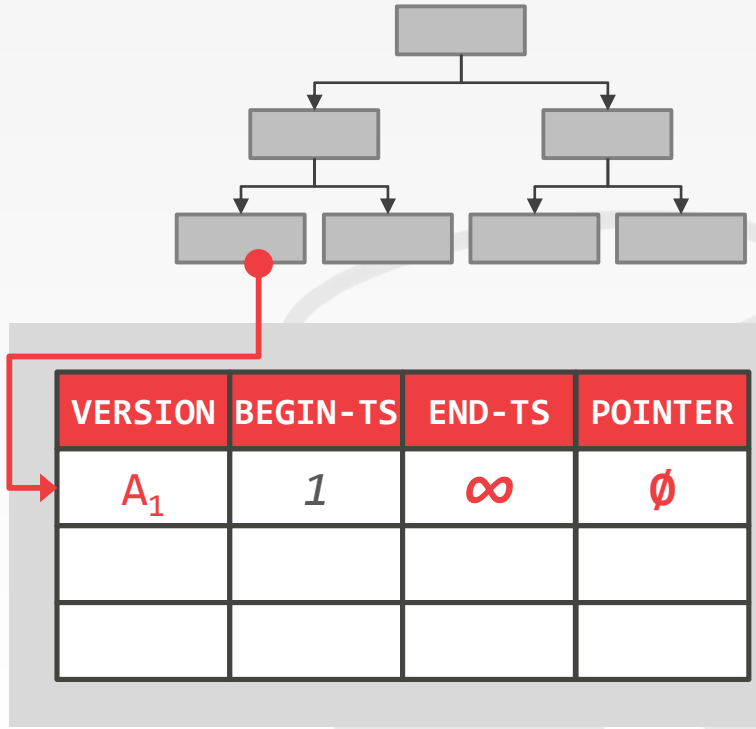
Thread #1

Begin @ 10

Thread #2

Begin @ 20

Index



MVCC DUPLICATE KEY PROBLEM

Thread #1

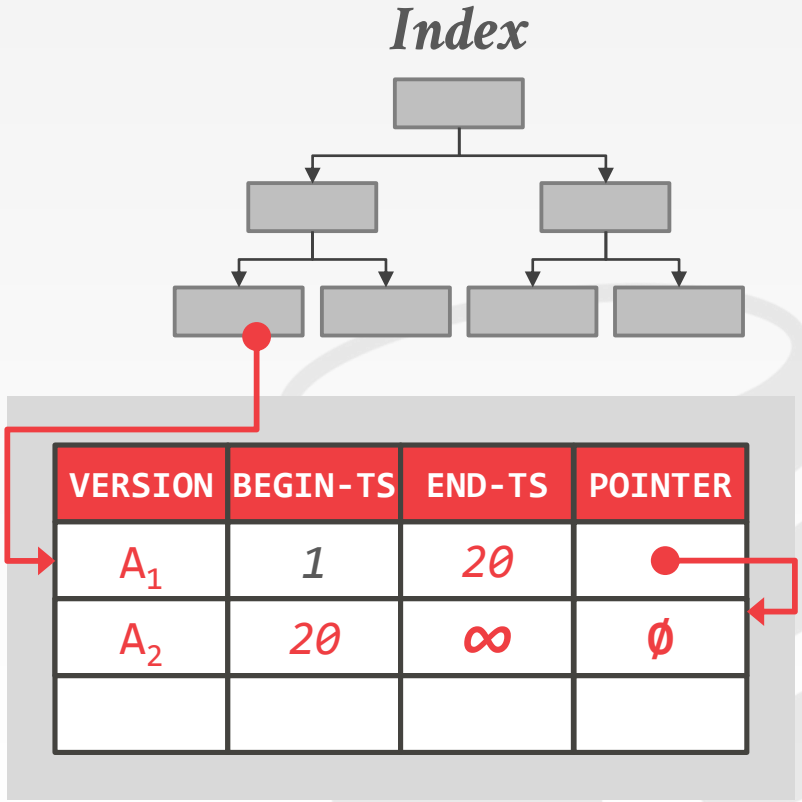
Begin @ 10

READ(A)

Thread #2

Begin @ 20

UPDATE(A)



MVCC DUPLICATE KEY PROBLEM

Thread #1

Begin @ 10

READ(A)

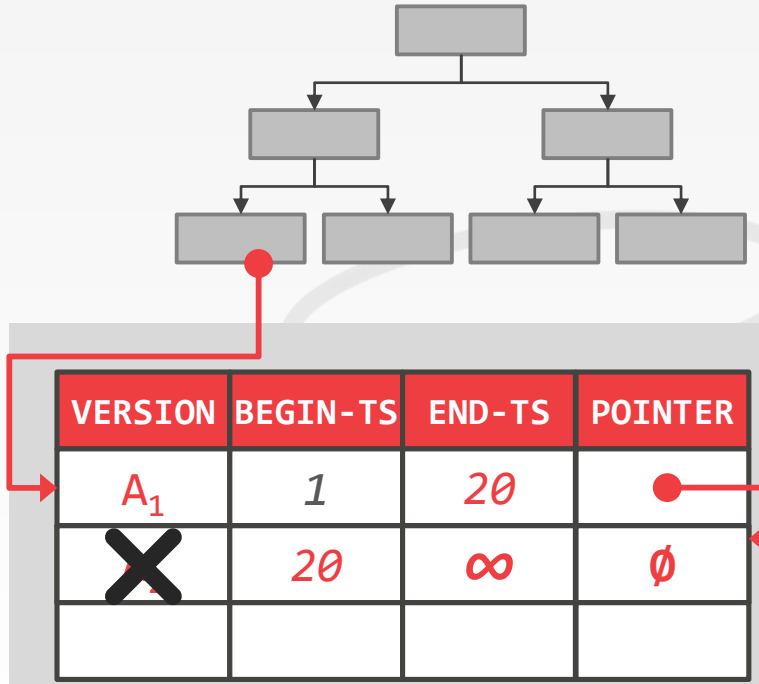
Thread #2

Begin @ 20

UPDATE(A)

DELETE(A)

Index



MVCC DUPLICATE KEY PROBLEM

Thread #1

Begin @ 10

READ(A)

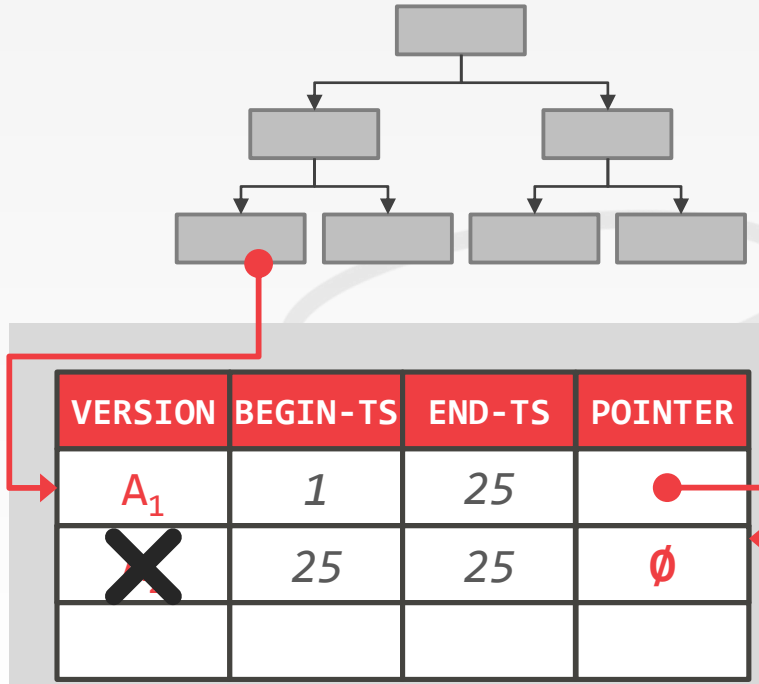
Thread #2

Begin @ 20
Commit @ 25

UPDATE(A)

DELETE(A)

Index



MVCC DUPLICATE KEY PROBLEM

Thread #1

Begin @ 10

READ(A)

Thread #2

Begin @ 20

Commit @ 25

UPDATE(A)

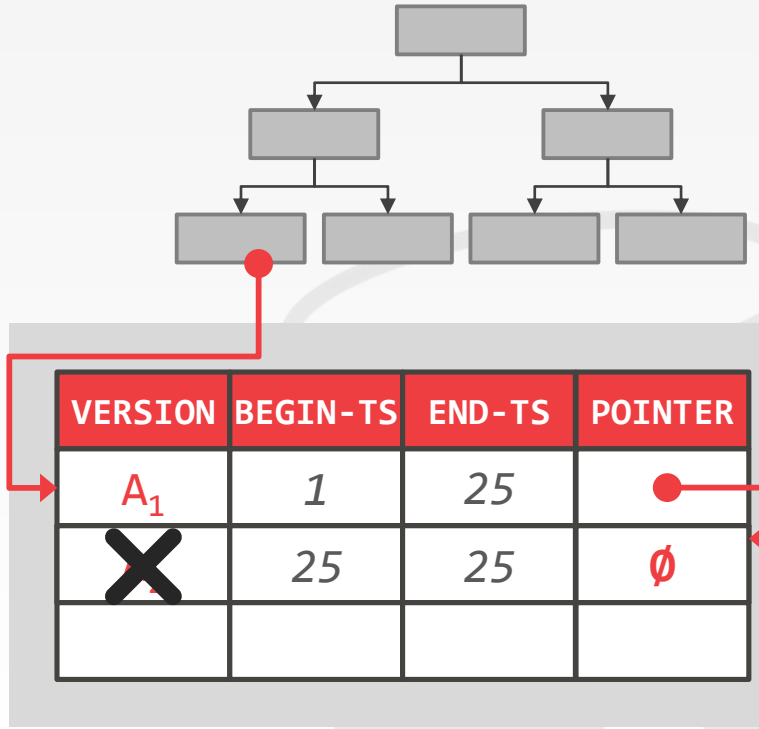
DELETE(A)

Thread #3

Begin @ 30

INSERT(A)

Index



MVCC DUPLICATE KEY PROBLEM

Thread #1

Begin @ 10

READ(A)

Thread #2

Begin @ 20

Commit @ 25

UPDATE(A)

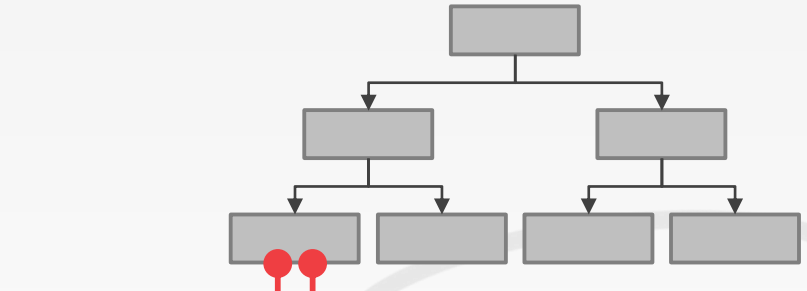
DELETE(A)

Thread #3

Begin @ 30

INSERT(A)

Index



VERSION	BEGIN-TS	END-TS	POINTER
A_1	1	25	●
✕	25	25	∅
A_1	30	∞	∅

MVCC DUPLICATE KEY PROBLEM

Thread #1
Begin @ 10

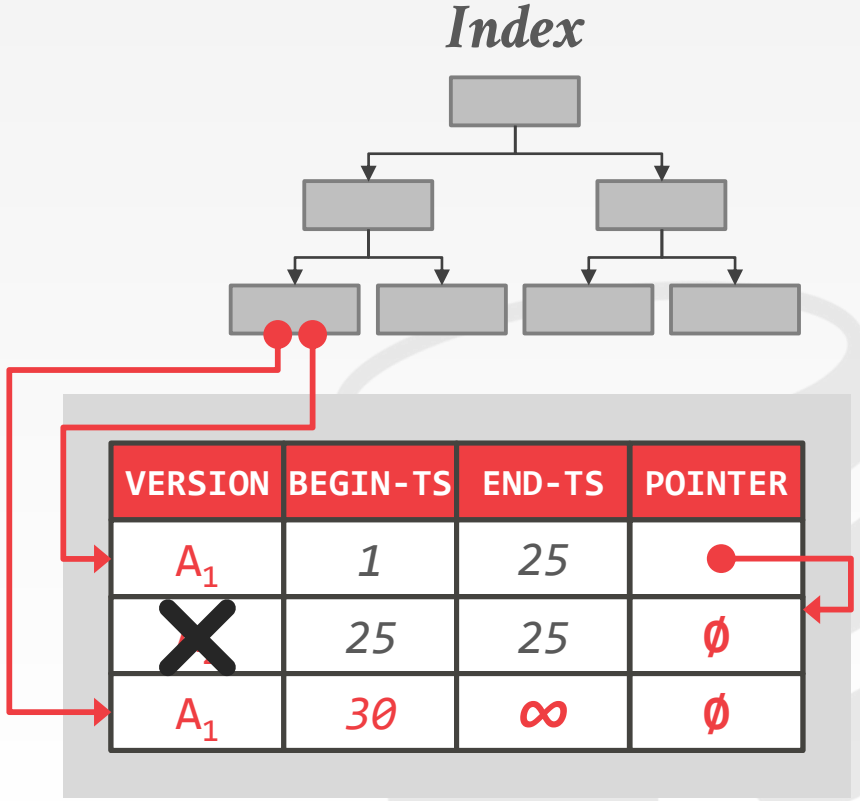
READ(A) READ(A)

Thread #2
Begin @ 20
Commit @ 25

UPDATE(A) DELETE(A)

Thread #3
Begin @ 30

INSERT(A)



MVCC INDEXES

Each index's underlying data structure has to 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 have to follow the pointers to find the proper physical version.

GC DESIGN DECISIONS

Index Clean-up

Version Tracking / Identification

Granularity

Comparison Unit

HYBRID GARBAGE COLLECTION FOR MULTI-VERSION CONCURRENCY CONTROL IN SAP HANA
SIGMOD 2016

GC – INDEX CLEAN-UP

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

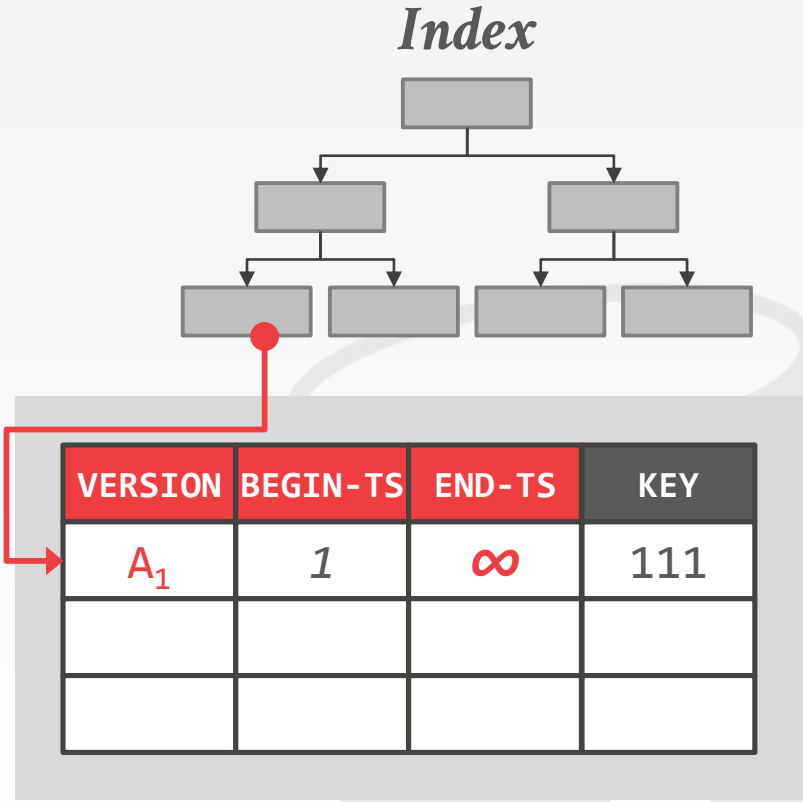
Track the txn's modifications to individual indexes to support GC of older versions on commit and removal modifications on abort.

PELOTON MISTAKE

Thread #1

Begin @ 10

key=222



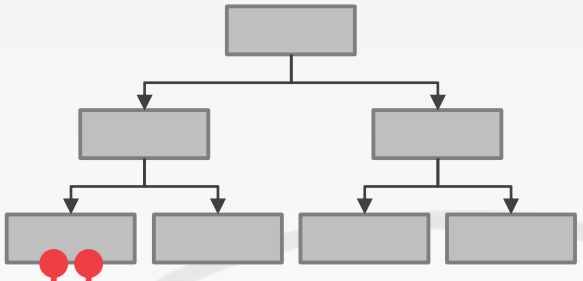
PELOTON MISTAKE

Thread #1

Begin @ 10

key=222

Index



VERSION	BEGIN-TS	END-TS	KEY
A_1	*1*	*10*	111
A_2	*10*	*∞*	222

PELOTON MISTAKE

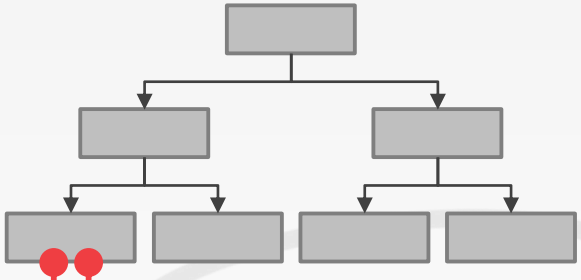
Thread #1

Begin @ 10

key=222

key=333

Index



VERSION	BEGIN-TS	END-TS	KEY
A_1	*1*	*10*	111
A_2	*10*	*∞*	222

PELOTON MISTAKE

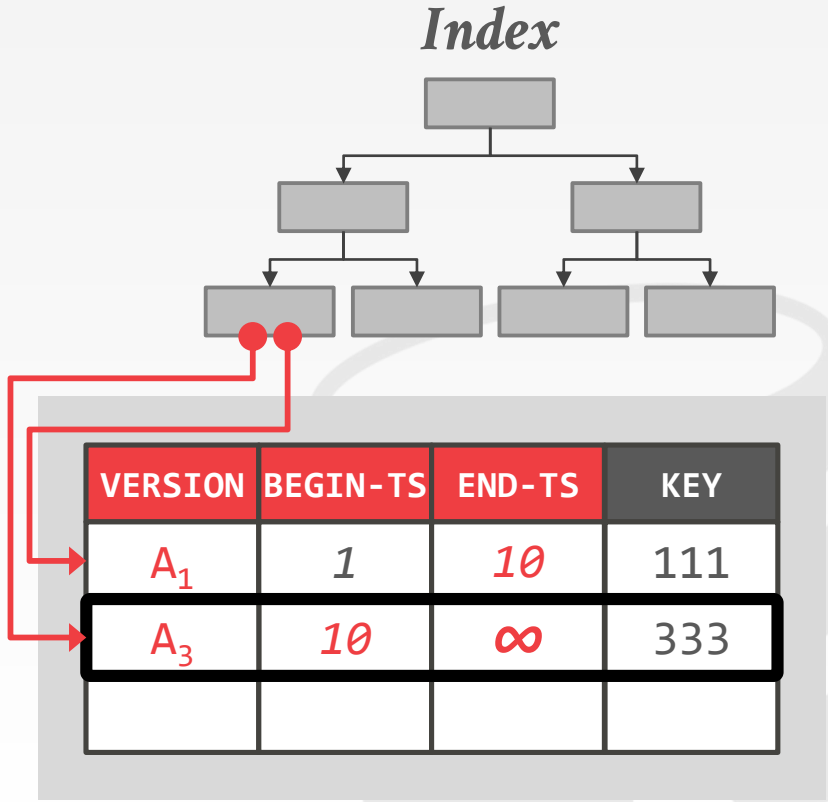
Thread #1

Begin @ 10

UPDATE(A) *key=222*

UPDATE(A) *key=333*

If a txn writes to same tuple more than once, then it just overwrites its previous version.



PELOTON MISTAKE

Thread #1

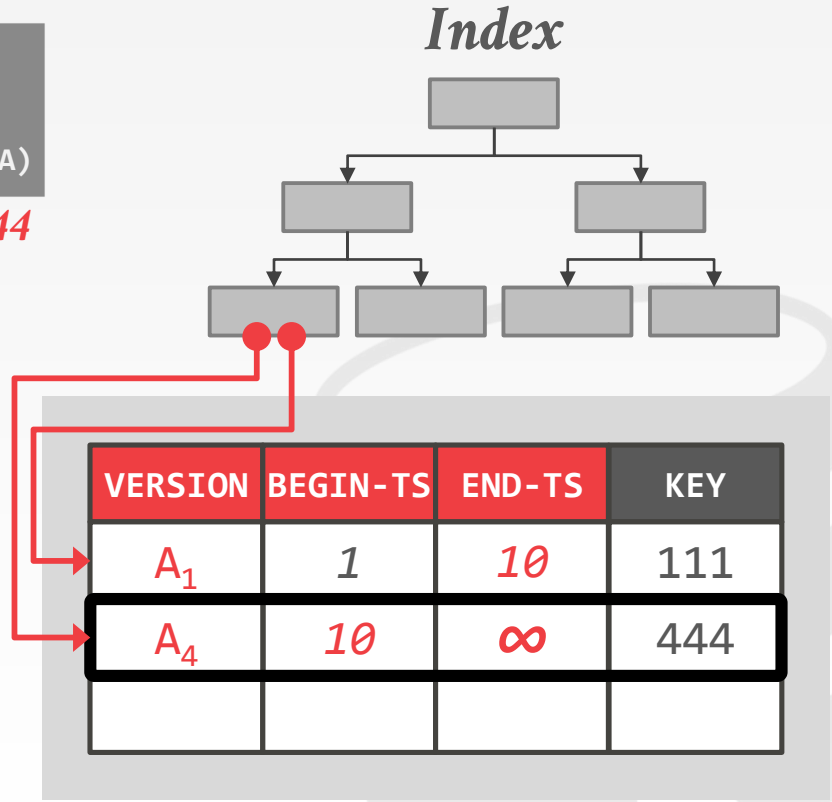
Begin @ 10

UPDATE(A) *key=222*

UPDATE(A) *key=333*

UPDATE(A) *key=444*

If a txn writes to same tuple more than once, then it just overwrites its previous version.



PELOTON MISTAKE

Thread #1

Begin @ 10
ABORT

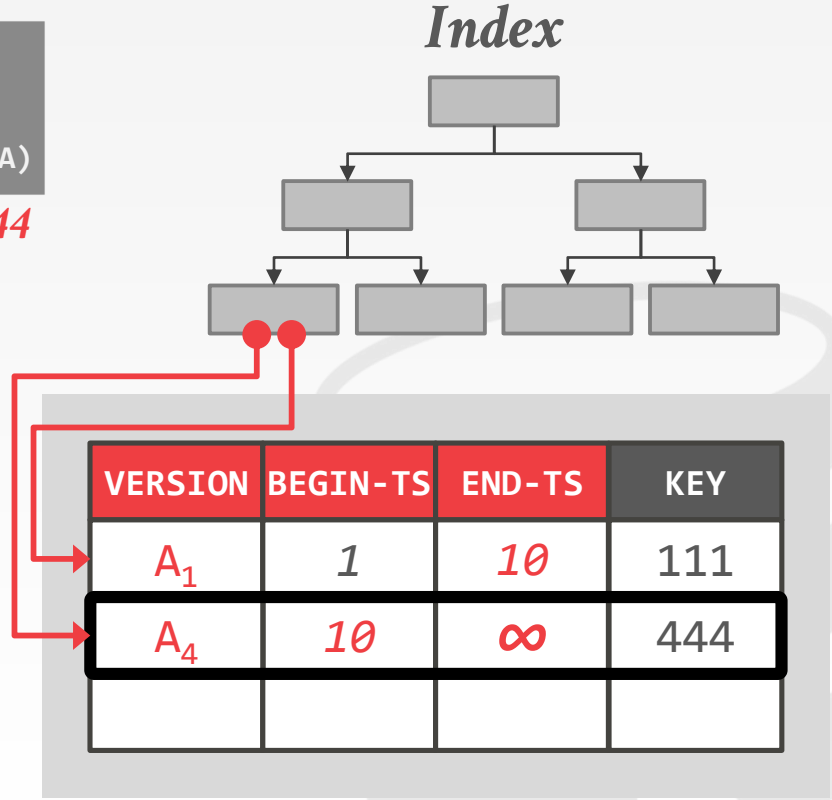
UPDATE(A) *key=222*

UPDATE(A) *key=333*

UPDATE(A) *key=444*

If a txn writes to same tuple more than once, then it just overwrites its previous version.

Upon rollback, the DBMS did not know what keys it added to the index in previous versions.



GC – VERSION TRACKING

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.

GC – VERSION TRACKING

Thread #1

Begin @ 10

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

GC – VERSION TRACKING

Thread #1

Begin @ 10

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

GC – VERSION TRACKING

Thread #1

Begin @ 10

Old Versions

A_2

UPDATE(A)

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

GC – VERSION TRACKING

Thread #1

Begin @ 10

Old Versions

A_2

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

GC – VERSION TRACKING

Thread #1

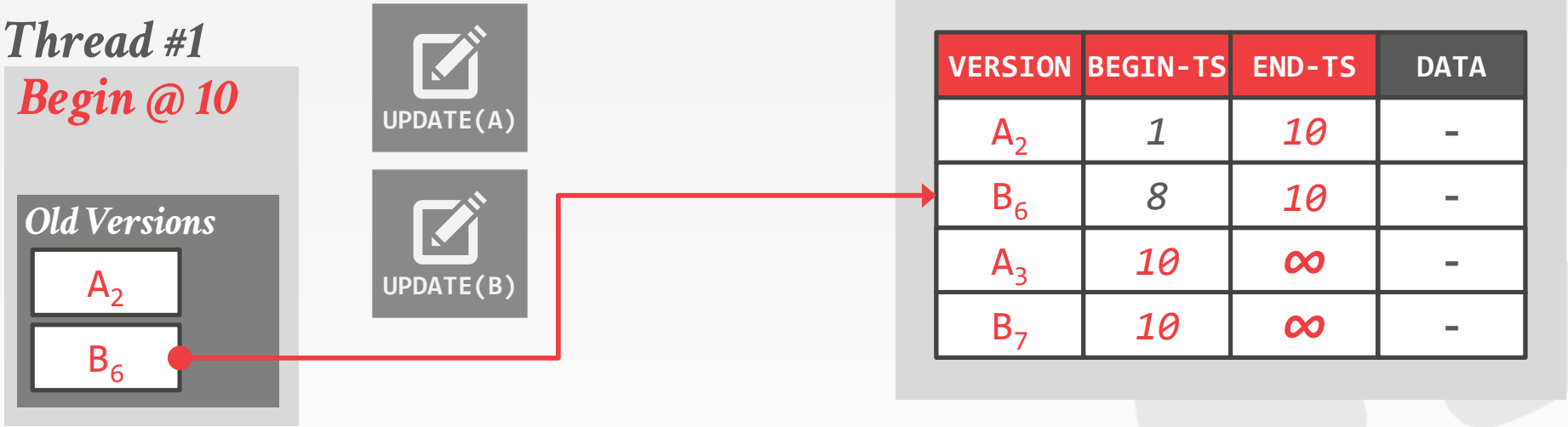
Begin @ 10

Old Versions

A_2

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

GC – VERSION TRACKING



GC – VERSION TRACKING

Thread #1

Begin @ 10
Commit @ 15

Old Versions

A_2

B_6

VERSION	BEGIN-TS	END-TS	DATA
A_2	1	15	-
B_6	8	15	-
A_3	15	∞	-
B_7	15	∞	-

GC – VERSION TRACKING

Thread #1

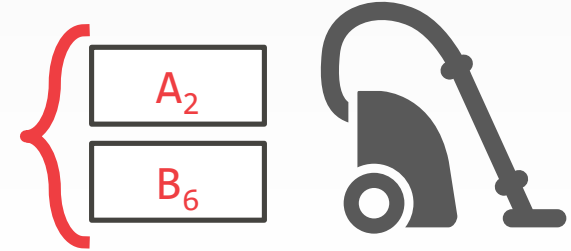
Begin @ 10
Commit @ 15

Old Versions

VERSION	BEGIN-TS	END-TS	DATA
A_2	1	15	-
B_6	8	15	-
A_3	15	∞	-
B_7	15	∞	-

Vacuum

TS<15



GC – GRANULARITY

How should the DBMS internally organize the expired versions that it needs to check to determine whether they are reclaimable.

Trade-off between the ability to reclaim versions sooner versus computational overhead.

GC – GRANULARITY

Approach #1: Single Version

→ Track the visibility of individual versions and reclaim them separately.
→ More fine-grained control, but higher overhead.

Approach #2: Group Version

→ Organize versions into groups and reclaim all of them together.
→ Less overhead, but may delay reclamations.

GC – GRANULARITY

Approach #3: Tables

→ Reclaim all versions from a table if the DBMS determines that active txns will never access it.

→ Special case for stored procedures and prepared statements since it requires the DBMS knowing what tables a txn will access in advance.

GC – COMPARISON UNIT

How should the DBMS determine whether version(s) are reclaimable.

Examining the list of active txns and reclaimable versions should be latch-free.

→ It is okay if the GC misses a recently committed txn. It will find it in the next round.

GC – COMPARISON UNIT

Approach #1: Timestamp

→ Use a global minimum timestamp to determine whether versions are safe to reclaim.
→ Easiest to implement and execute.

Approach #2: Interval

→ Excise timestamp ranges that are not visible.
→ More difficult to identify ranges.

GC – COMPARISON UNIT

Thread #1

Begin @ 10

VERSION	BEGIN-TS	END-TS	DATA
A_1	*1*	∞	-

GC – COMPARISON UNIT

Thread #1

Begin @ 10

Thread #2

Begin @ 20

VERSION	BEGIN-TS	END-TS	DATA
A_1	1	20	-
A_2	20	∞	-

GC – COMPARISON UNIT

Thread #1

Begin @ 10

Thread #2

Begin @ 20
Commit @ 25

VERSION	BEGIN-TS	END-TS	DATA
A_1	1	25	-
A_2	25	∞	-

GC – COMPARISON UNIT

Thread #1

Begin @ 10

Thread #2

Begin @ 20
Commit @ 25

Thread #3

Begin @ 30

VERSION	BEGIN-TS	END-TS	DATA
A_1	1	25	-
A_2	25	30	-
A_3	30	∞	-

GC – COMPARISON UNIT

Thread #1

Begin @ 10

Thread #2

Begin @ 20
Commit @ 25

Thread #3

Begin @ 30
Commit @ 35

VERSION	BEGIN-TS	END-TS	DATA
A_1	1	25	-
A_2	25	35	-
A_3	35	∞	-

GC – COMPARISON UNIT

Thread #1

Begin @ 10

Thread #2

Begin @ 20
Commit @ 25

Thread #3

Begin @ 30
Commit @ 35

VERSION	BEGIN-TS	END-TS	DATA
A_1	1	25	-
A_2	25	35	-
A_3	35	∞	-

Timestamp

→ GC cannot reclaim A_2 because the lowest active txn TS (10) is less than END-TS.

Interval

→ GC can reclaim A_2 because no active txn TS intersects the interval [25,35].

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 memory to the OS.

→ Move data using **DELETE** + **INSERT** to ensure transactional guarantees during consolidation.

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.

→ This will matter more when we talk about compression and moving cold data out to disk.

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.
→ The GC will free all memory when there are no active txns that exist before the drop operation.
→ If the catalog is transactional, then this easy to do.

PARTING THOUGHTS

Classic storage vs. compute trade-off.

My impression is that people want to reduce the memory footprint of the DBMS and are willing to pay a (small) computational overhead for more aggressive GC.

PROJECT #1

Identify bottlenecks in the DBMS's transaction manager 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 ***concurrent-read*** 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 only builds on Ubuntu 18.04 and OSX.
→ You can also do development on a VM.

This is CMU so I'm going to assume that each of you are capable of getting access to a machine.

Important: You will not be able to identify the bottleneck on a machine with less than 20 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: $0.45/hr (as of Jan 2019)

PROJECT #1

Due Date: February 27th @ 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/spring2019/project1.html

NEXT CLASS

Index Locking + Latching