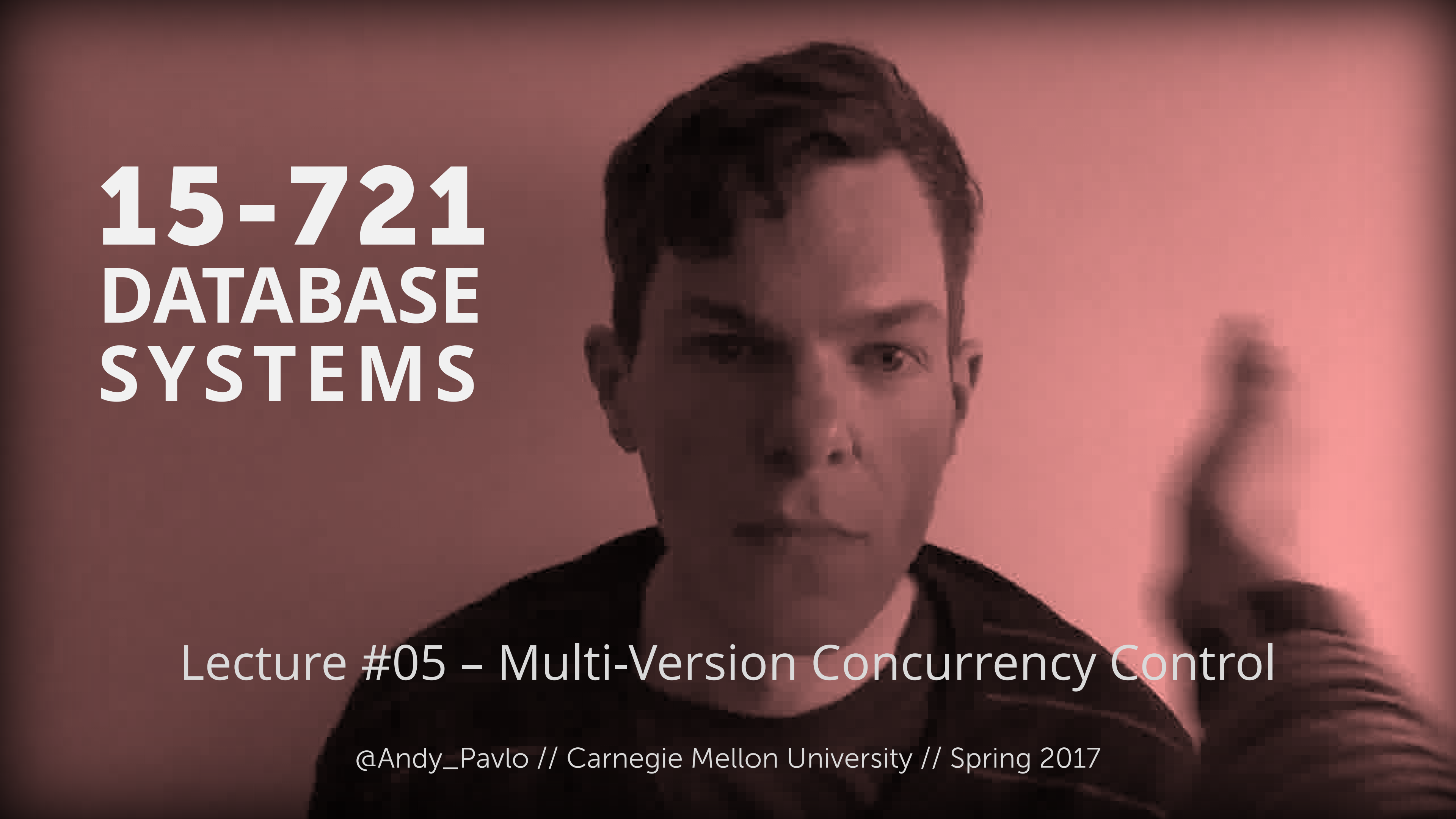


15-721
DATABASE SYSTEMS

Lecture #05 – Multi-Version Concurrency Control

@Andy_Pavlo // Carnegie Mellon University // Spring 2017

TODAY'S AGENDA

Compare-and-Swap (CAS)
MVCC Overview
Design Decisions
Modern MVCC Implementations
Project #2

COMPARE-AND-SWAP

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

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

→ Otherwise operation fails

M

20

Address *New Value*

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

Compare Value

COMPARE-AND-SWAP

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

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

→ Otherwise operation fails

M

30

Address *New Value*

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

Compare Value

COMPARE-AND-SWAP

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

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

→ Otherwise operation fails

M

30

Address

New Value

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

X

Compare Value

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.
Used in almost every new DBMS in last 10 years.

MULTI-VERSION CONCURRENCY CONTROL

Main benefits:

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

MVCC is more than just a "concurrency control protocol". It completely affects how the DBMS manages transactions and the database.

MVCC DESIGN DECISIONS

Concurrency Control Protocol

Version Storage

Garbage Collection

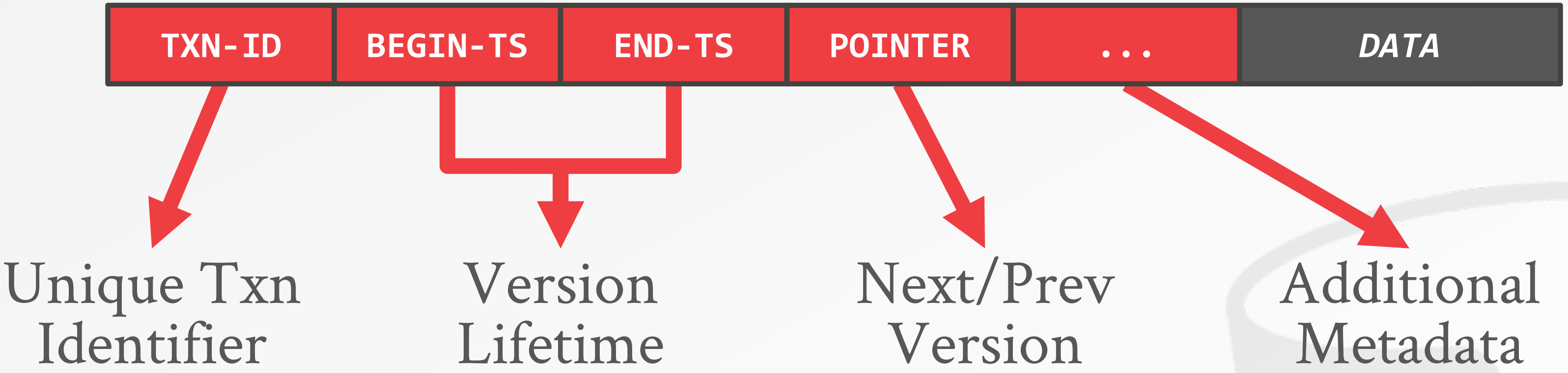
Index Management

WE STILL NEED TO THINK OF A TITLE BUT TRUST ME
THIS IS A REALLY GOOD PAPER ON IN-MEMORY MVCC
VLDB 2017

MVCC IMPLEMENTATIONS

	Protocol	*Version Storage*	*Garbage Collection*	*Indexes*
Oracle	MV2PL	Delta	Vacuum	Logical
Postgres	MV-2PL/MV-TO	Append-Only	Vacuum	Physical
MySQL-InnoDB	MV-2PL	Delta	Vacuum	Logical
HYRISE	MV-OCC	Append-Only	–	Physical
Hekaton	MV-OCC	Append-Only	Cooperative	Physical
MemSQL	MV-OCC	Append-Only	Vacuum	Physical
SAP HANA	MV-2PL	Time-travel	Hybrid	Logical
NuoDB	MV-2PL	Append-Only	Vacuum	Logical
HyPer	MV-OCC	Delta	Txn-level	Logical

TUPLE FORMAT



CONCURRENCY CONTROL PROTOCOL

Approach #1: Timestamp Ordering

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

Approach #2: Optimistic Concurrency Control

→ Three-phase protocol from last class.
→ Use private workspace for new versions.

Approach #3: Two-Phase Locking

→ Txns acquire appropriate lock on physical version before they can read/write a logical tuple.

TIMESTAMP ORDERING (MVTO)

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_x	0	1	1	∞
B_x	0	0	1	∞

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

TIMESTAMP ORDERING (MVTO)

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_x	0	1	1	∞
B_x	0	0	1	∞

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

TIMESTAMP ORDERING (MVTO)

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_x	0	1	1	∞
B_x	0	0	1	∞

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

TIMESTAMP ORDERING (MVTO)

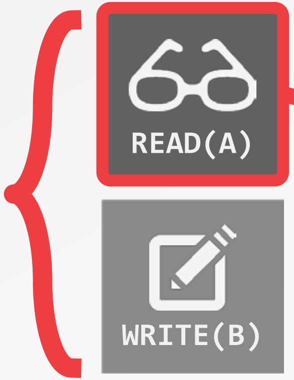
T_{id}=10

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_x	0	1	1	∞
B_x	0	0	1	∞

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

TIMESTAMP ORDERING (MVTO)

$T_{id}=10$



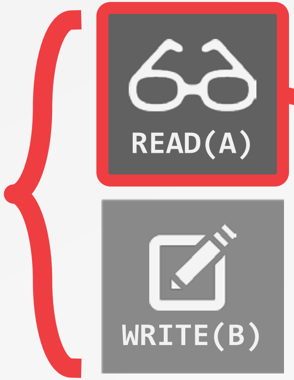
	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_x	0	1	1	∞
B_x	0	0	1	∞

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

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

TIMESTAMP ORDERING (MVTO)

T_{id}=10

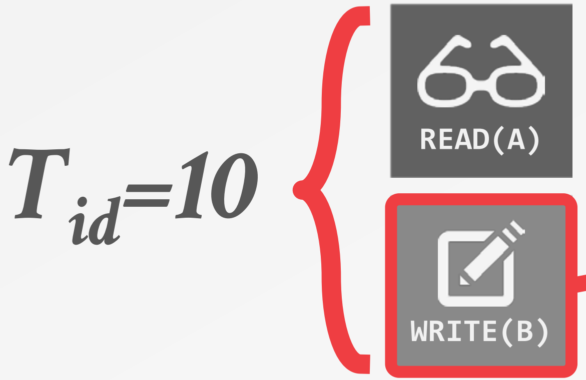


	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_x	0	10	1	∞
B_x	0	0	1	∞

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

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

TIMESTAMP ORDERING (MVTO)



	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_x	0	10	1	∞
B_x	0	0	1	∞

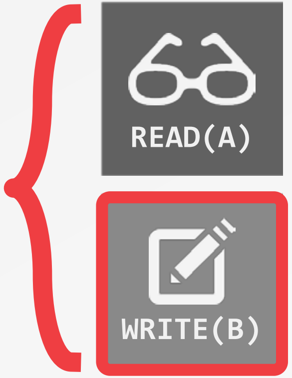
Use “read-ts” field in the header to keep track of the timestamp of the last txn that read it.

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

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

TIMESTAMP ORDERING (MVTO)

T_{id}=10



	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_x	0	10	1	∞
B_x	10	0	1	∞

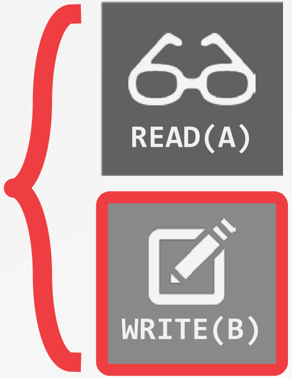
Use "read-ts" field in the header to keep track of the timestamp of the last txn that read it.

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

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

TIMESTAMP ORDERING (MVTO)

T_{id}=10



	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_X	0	10	1	∞
B_X	10	0	1	∞
B_{X+1}	10	0	10	∞

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

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

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

TIMESTAMP ORDERING (MVTO)

T_{id}=10

- READ(A)
- WRITE(B)

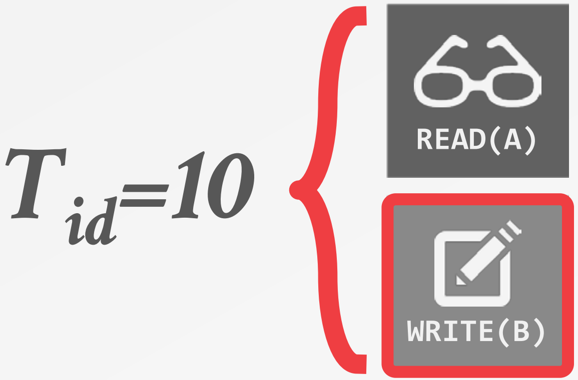
	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_x	0	10	1	∞
B_x	10	0	1	10
B_{X+1}	10	0	10	∞

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

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

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

TIMESTAMP ORDERING (MVTO)



	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_x	0	10	1	∞
B_x	0	0	1	10
B_{X+1}	0	0	10	∞

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

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

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

VERSION STORAGE

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

→ This allows the DBMS to find the version that is visible to a particular txn at runtime.
→ Indexes always point to the "head" of the chain.

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

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

VERSION STORAGE

Approach #1: Append-Only Storage

→ New versions are appended to the same table space.

Approach #2: Time-Travel Storage

→ Old versions are copied to separate table space.

Approach #3: Delta Storage

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

APPEND-ONLY STORAGE

Main Table

	KEY	VALUE	POINTER
A_x	XXX	$111	●
A_{x+1}	XXX	$222	Ø
B_x	YYY	$10	Ø

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

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

APPEND-ONLY STORAGE

Main Table

	KEY	VALUE	POINTER
A_x	XXX	$111	● → A_{x+1}
A_{x+1}	XXX	$222	Ø
B_x	YYY	$10	Ø
A_{x+2}	XXX	$333	Ø

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

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

APPEND-ONLY STORAGE

Main Table

	KEY	VALUE	POINTER
A_x	XXX	$111	•
A_{x+1}	XXX	$222	•
B_x	YYY	$10	Ø
A_{x+2}	XXX	$333	Ø

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

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

VERSION CHAIN ORDERING

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

→ Just append new version to end of the chain.
→ Have to traverse chain on look-ups.

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

→ Have to update index pointers for every new version.
→ Don't have to traverse chain on look ups.

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

TIME-TRAVEL STORAGE

Main Table

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

Time-Travel Table

	KEY	VALUE	POINTER
A_1	XXX	$111	Ø

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

TIME-TRAVEL STORAGE

Main Table

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

Time-Travel Table

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

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

TIME-TRAVEL STORAGE

Main Table

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

Time-Travel Table

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

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

Overwrite master version in the main table. Update pointers.

TIME-TRAVEL STORAGE

Main Table

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

Time-Travel Table

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

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

Overwrite master version in the main table. Update pointers.

DELTA STORAGE

Main Table

	KEY	VALUE	POINTER
A_1	XXX	$111	
B_1	YYY	$10	

Delta Storage Segment

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

DELTA STORAGE

Main Table

	KEY	VALUE	POINTER
A_2	XXX	$222	• →
B_1	YYY	$10	

Delta Storage Segment

	DELTA	POINTER
A_1	(VALUE→$111)	Ø

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

DELTA STORAGE

Main Table

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

Delta Storage Segment

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

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

DELTA STORAGE

Main Table

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

Delta Storage Segment

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

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

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

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

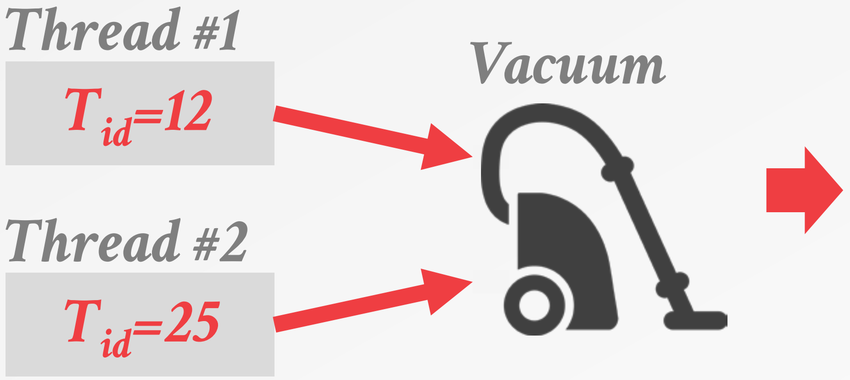
→ Find old versions by examining tuples directly.

→ 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.

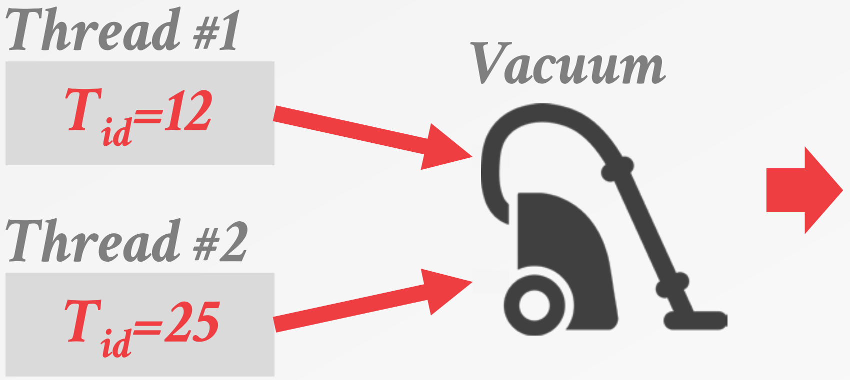
TUPLE-LEVEL GC



	TXN-ID	BEGIN-TS	END-TS
A_x	0	1	9
B_x	0	1	9
B_{x+1}	0	10	20

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

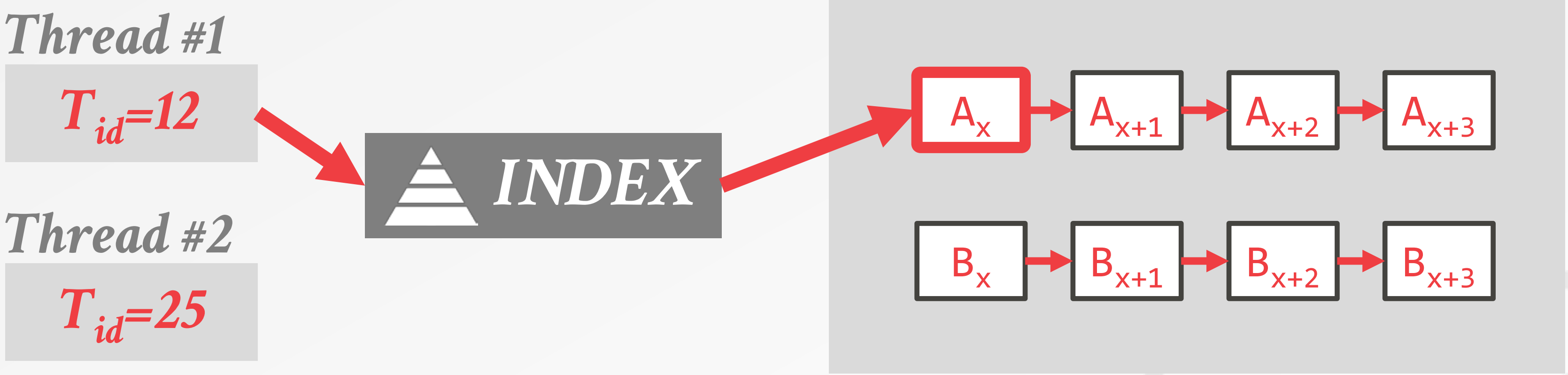
TUPLE-LEVEL GC



Dirty?	TXN-ID	BEGIN-TS	END-TS
B_{x+1}	0	10	20

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

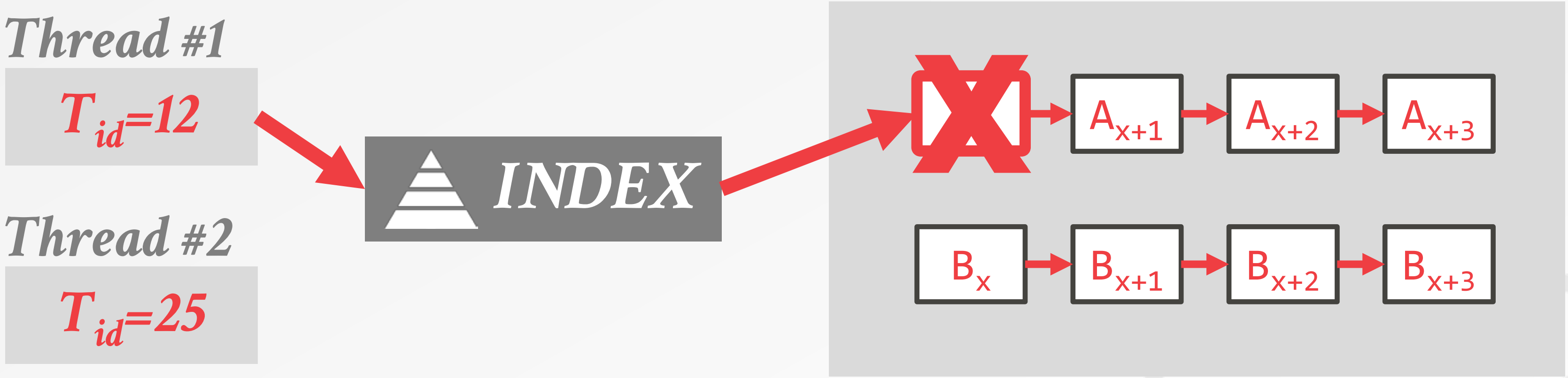
TUPLE-LEVEL GC



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

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

TUPLE-LEVEL GC



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

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

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.

INDEX MANAGEMENT

PKey indexes always point to version chain head.

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

Secondary indexes are more complicated…

UBER Engineering

JOIN THE TEAM

MEET THE PEOPLE

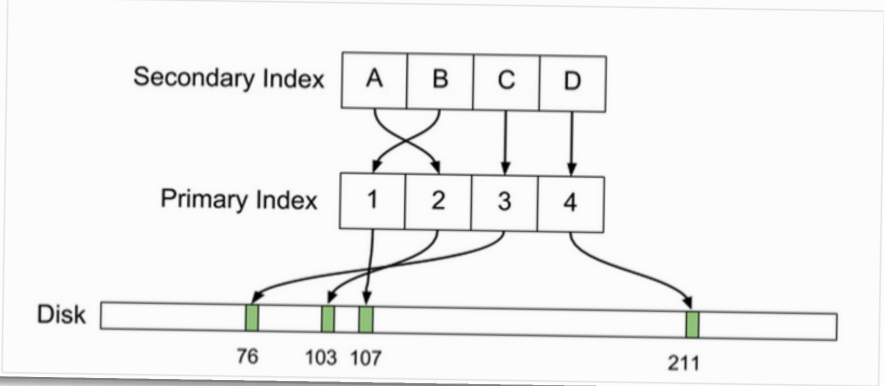
SEARCH

ARCHITECTURE

WHY UBER ENGINEERING SWITCHED FROM POSTGRES TO MYSQL

JULY 26, 2016

BY EVAN KLITZKE



SECONDARY INDEXES

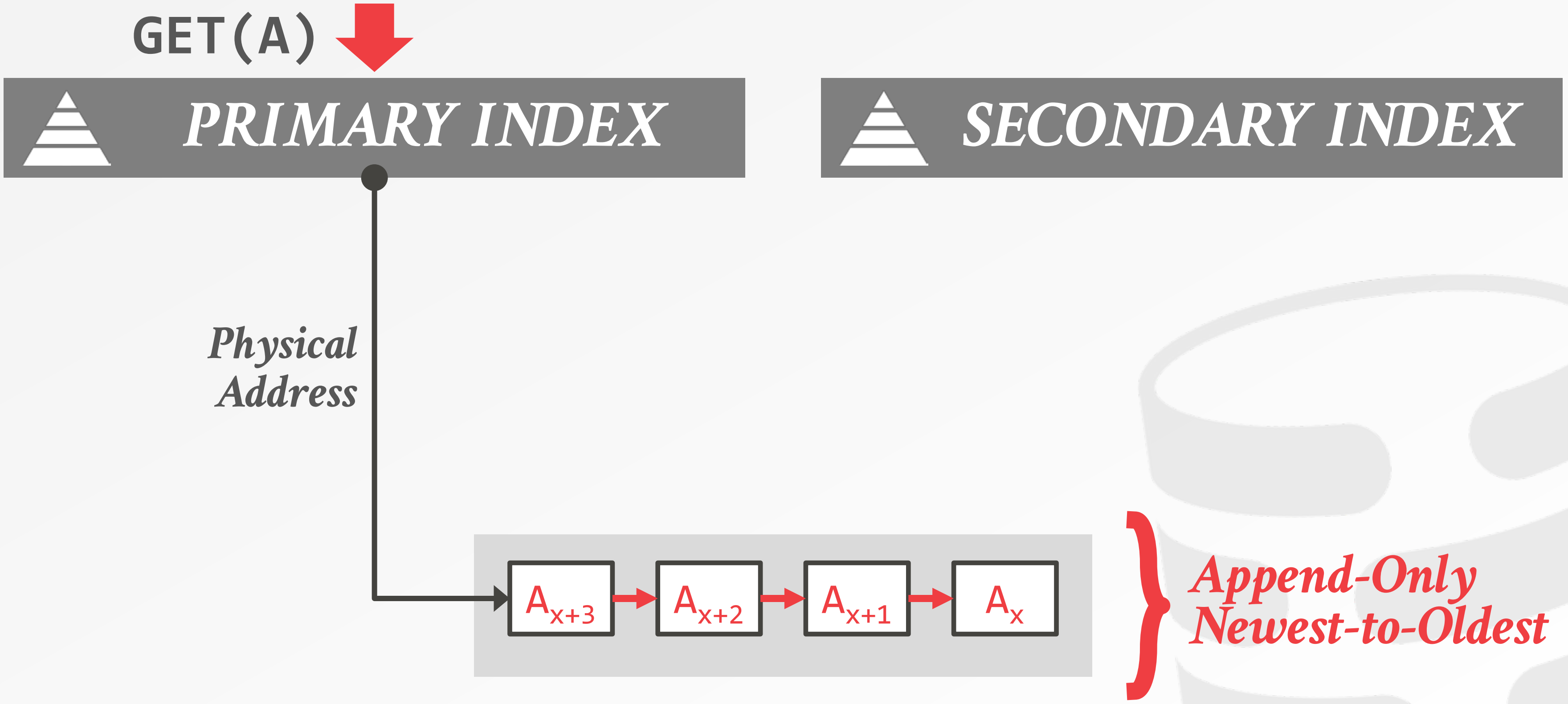
Approach #1: Logical Pointers

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

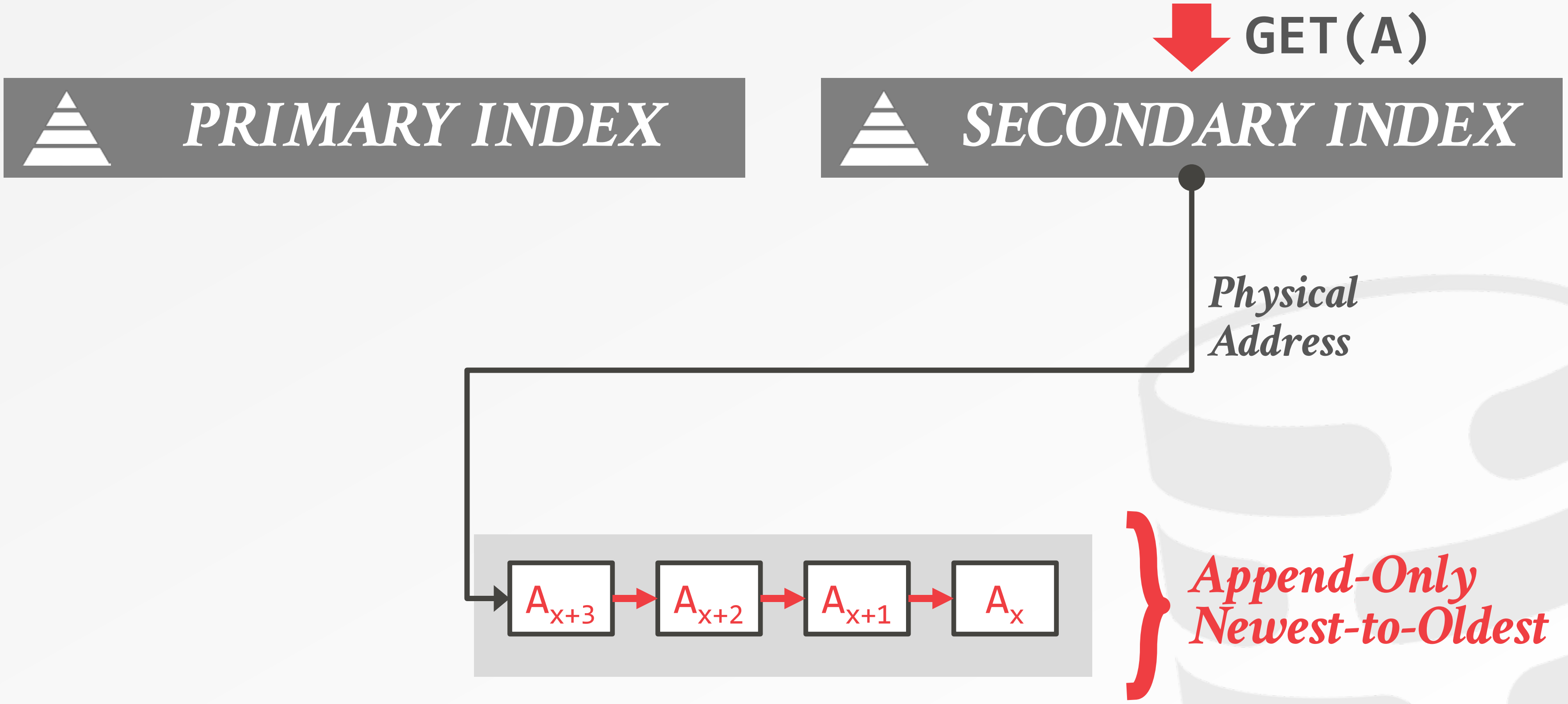
Approach #2: Physical Pointers

→ Use the physical address to the version chain head.

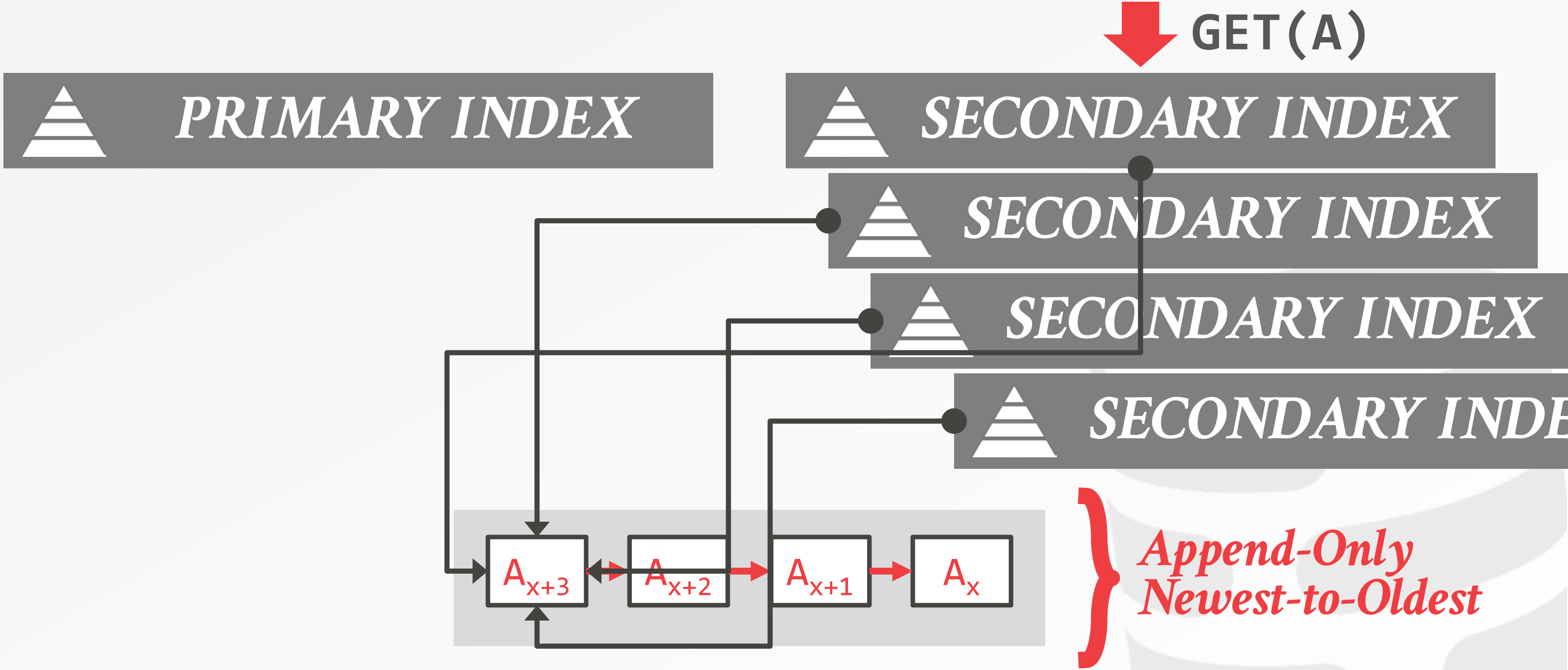
INDEX POINTERS



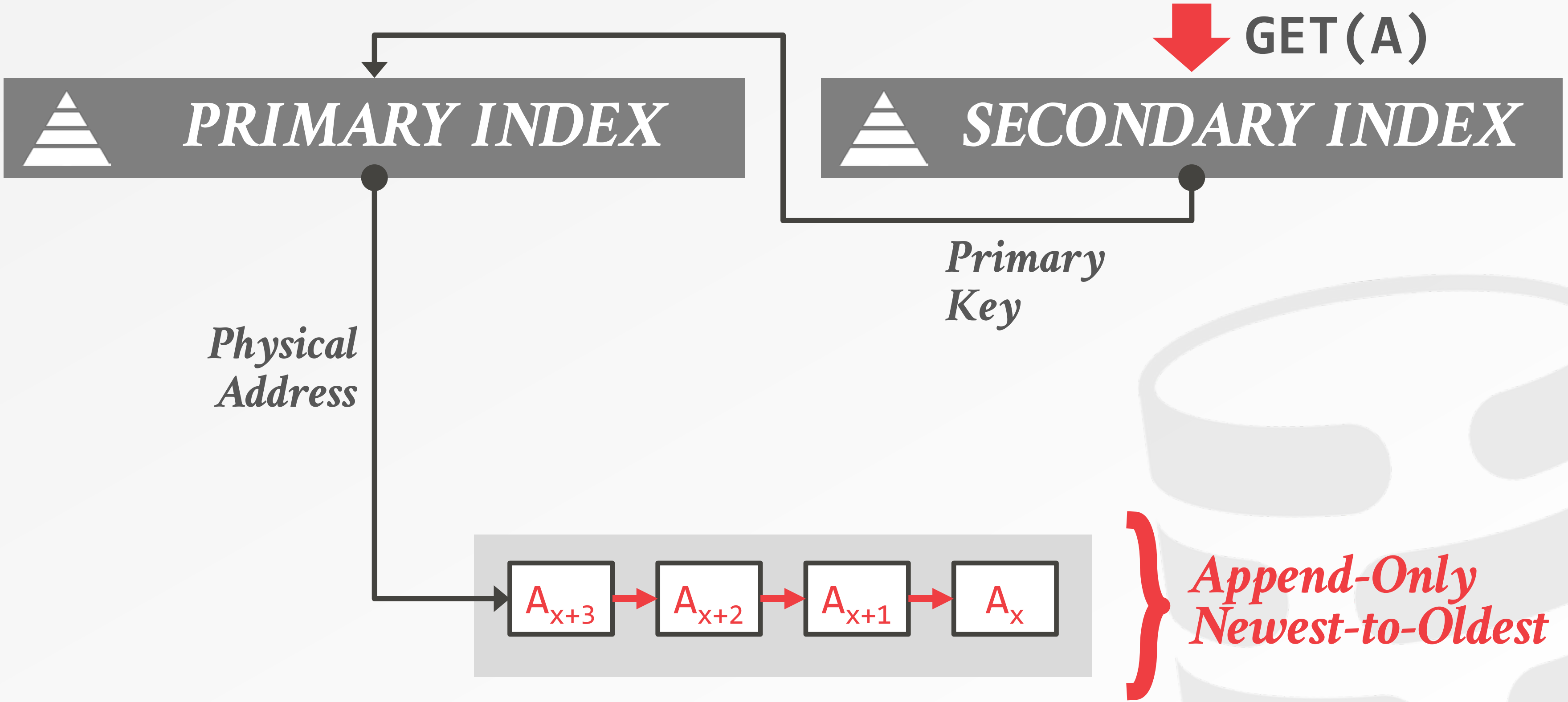
INDEX POINTERS



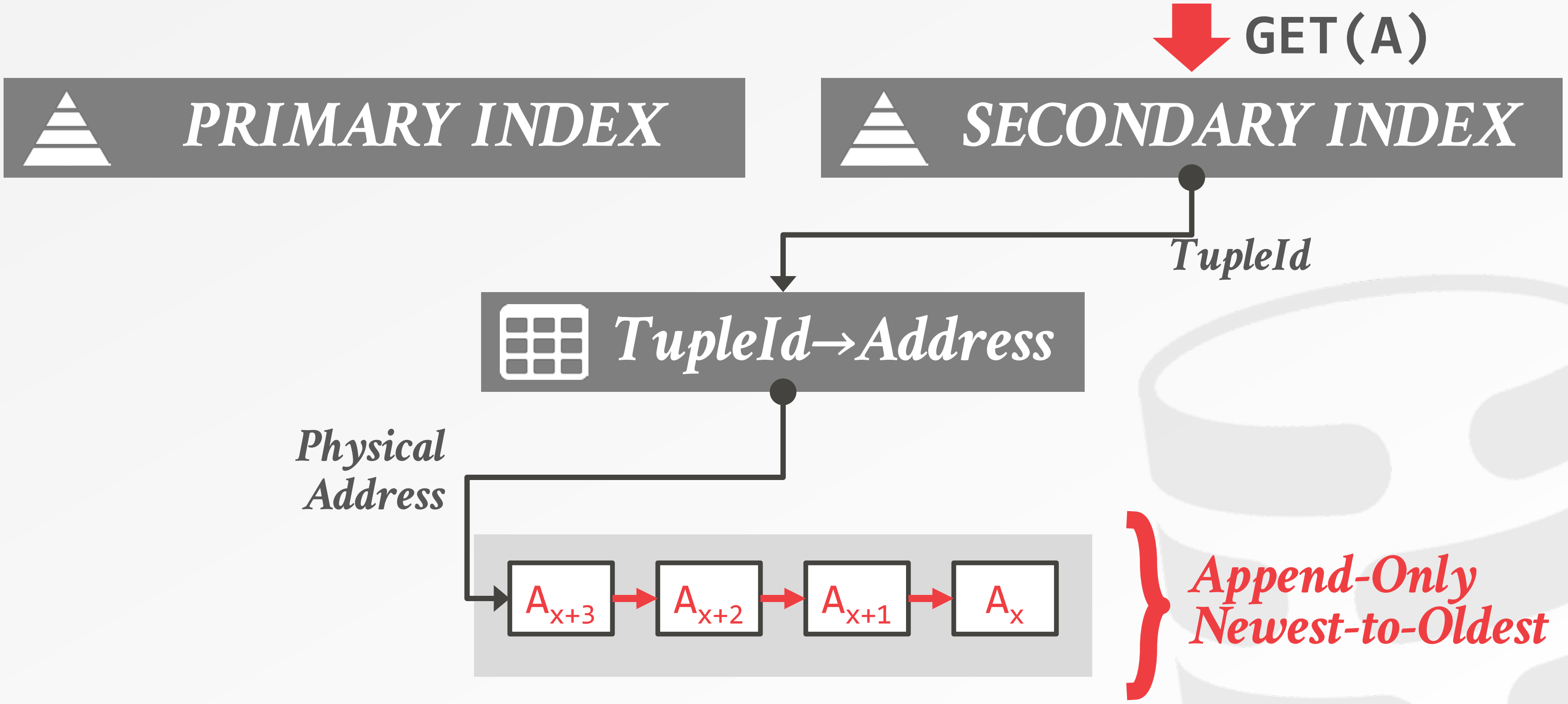
INDEX POINTERS



INDEX POINTERS

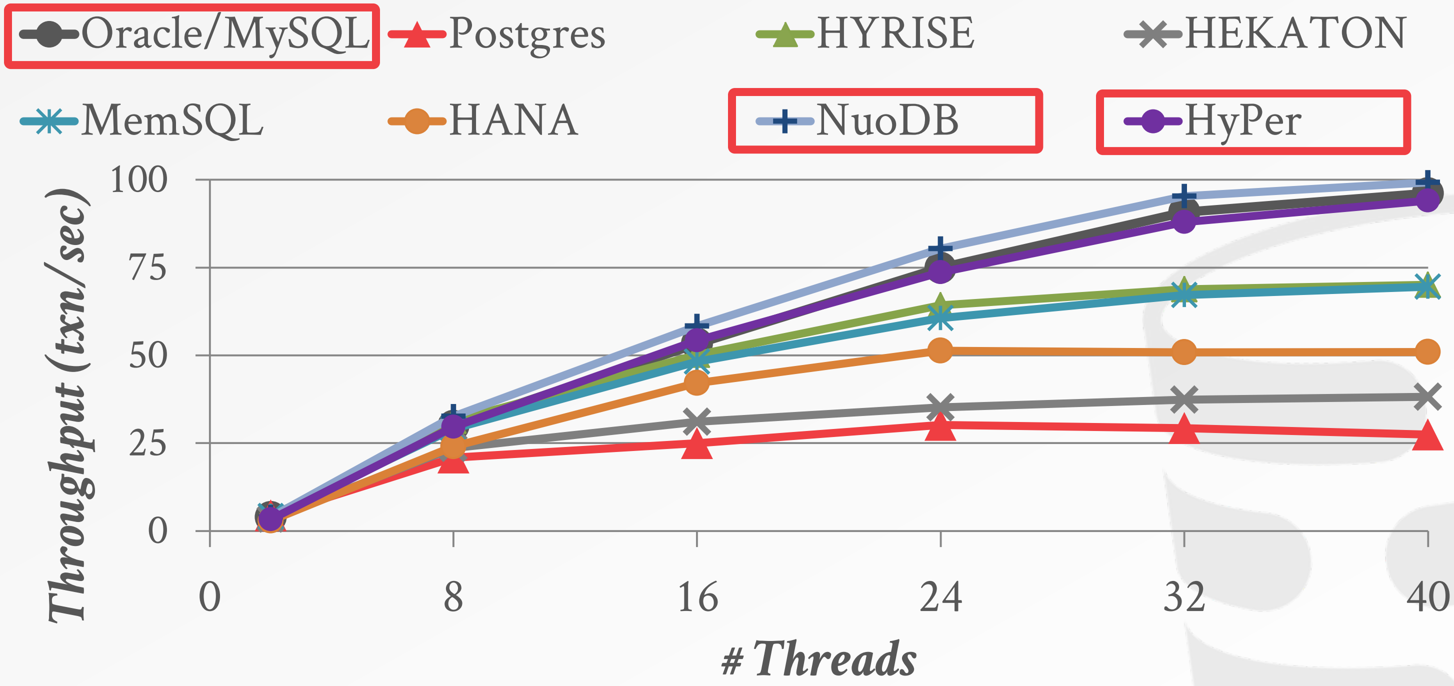


INDEX POINTERS



MVCC CONFIGURATION EVALUATION

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



MODERN MVCC

Microsoft Hekaton (SQL Server)

TUM HyPer

SAP HANA

MICROSOFT HEKATON

Incubator project started in 2008 to create new OLTP engine for MSFT SQL Server (MSSQL).

→ Led by DB ballers Paul Larson and Mike Zwilling

Had to integrate with MSSQL ecosystem.

Had to support all possible OLTP workloads with predictable performance.

→ Single-threaded partitioning (e.g., H-Store) works well for some applications but terrible for others.

HEKATON MVCC

Every txn is assigned a timestamp (TS) when they **begin** and when they **commit**.

DBMS maintains "chain" of versions per tuple:

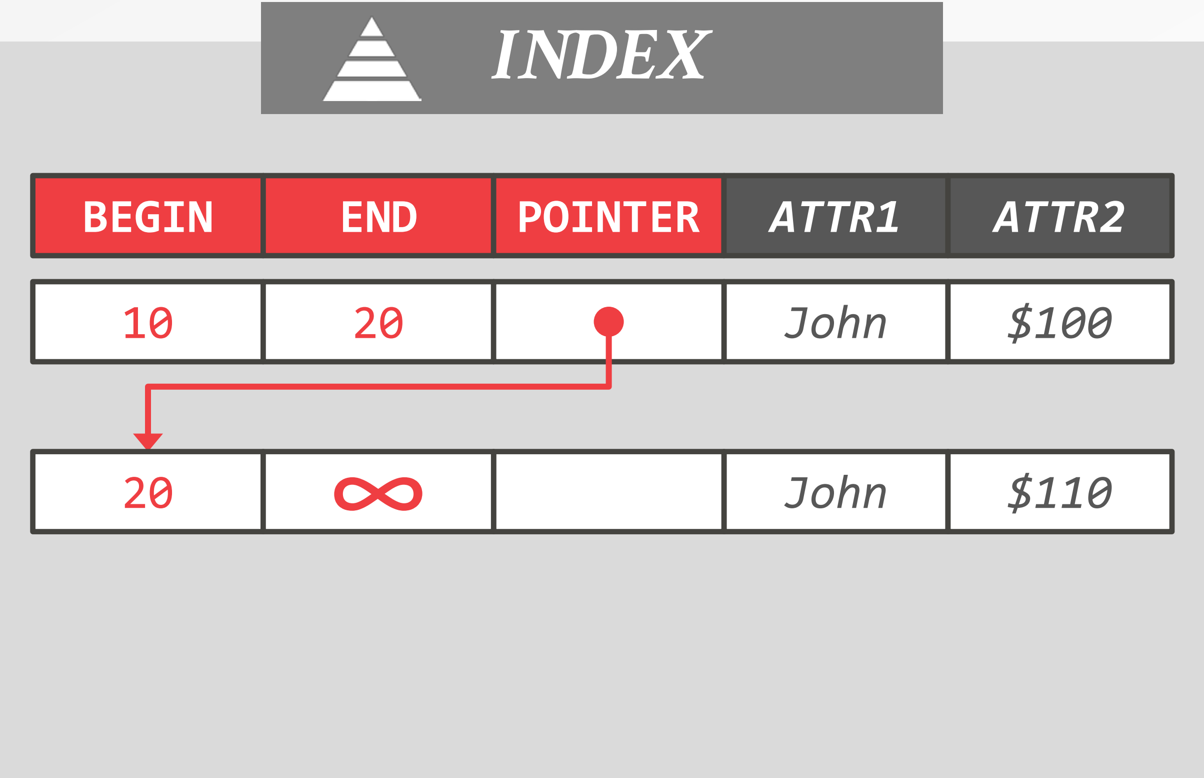
→ **BEGIN**: The BeginTS of the active txn **or** the EndTS of the committed txn that created it.

→ **END**: The BeginTS of the active txn that created the next version **or** infinity **or** the EndTS of the committed txn that created it.

→ **POINTER**: Location of the next version in the chain.

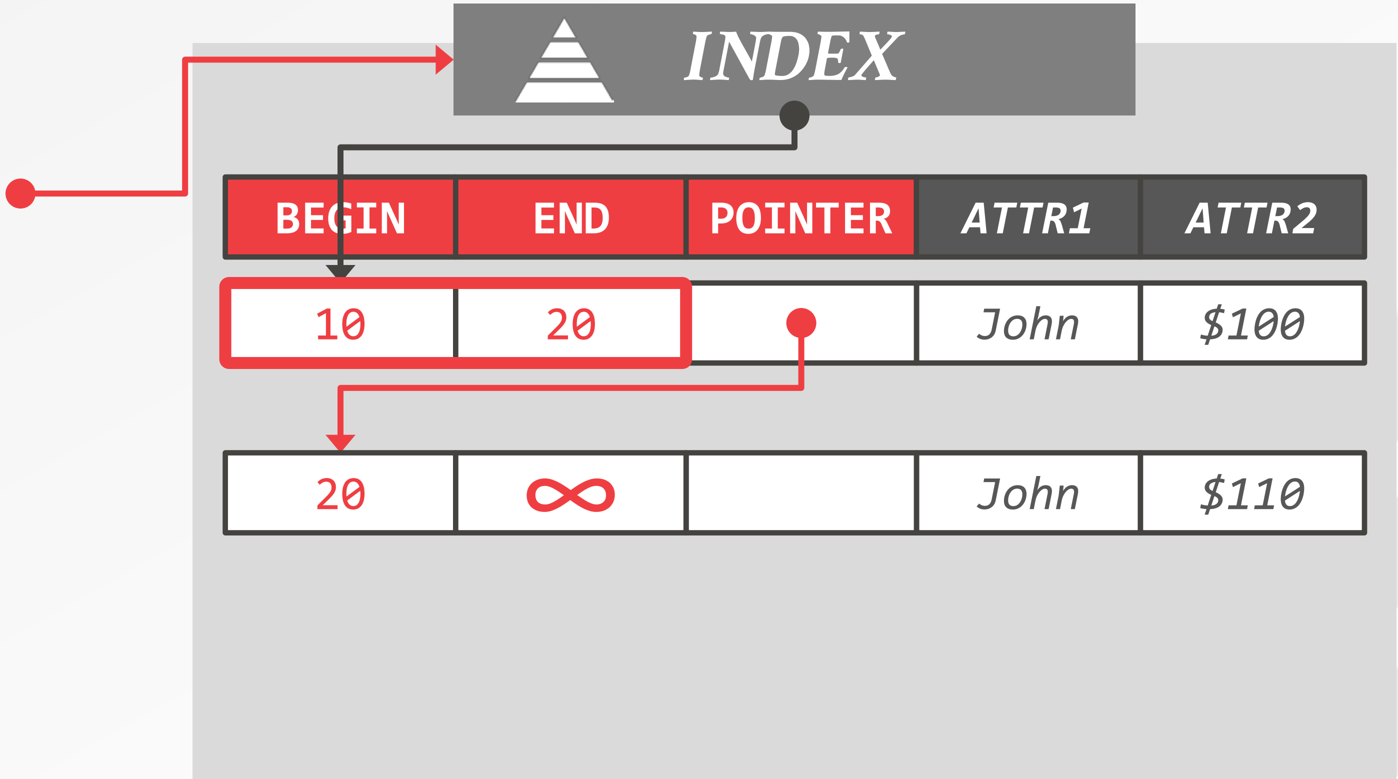
HIGH-PERFORMANCE CONCURRENCY CONTROL MECHANISMS FOR MAIN-MEMORY DATABASES
VLDB 2011

HEKATON: OPERATIONS



HEKATON: OPERATIONS

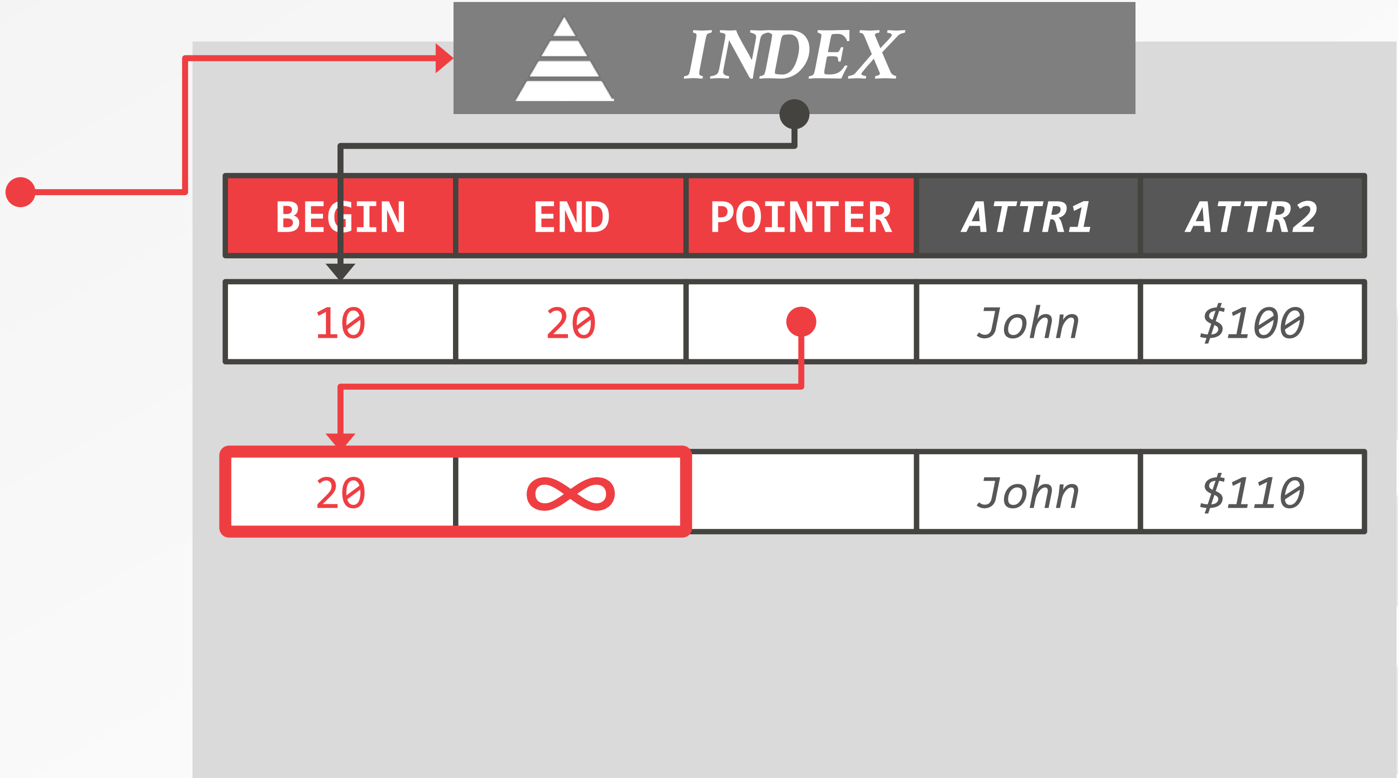
BEGIN @ 25
Read "John"



HEKATON: OPERATIONS

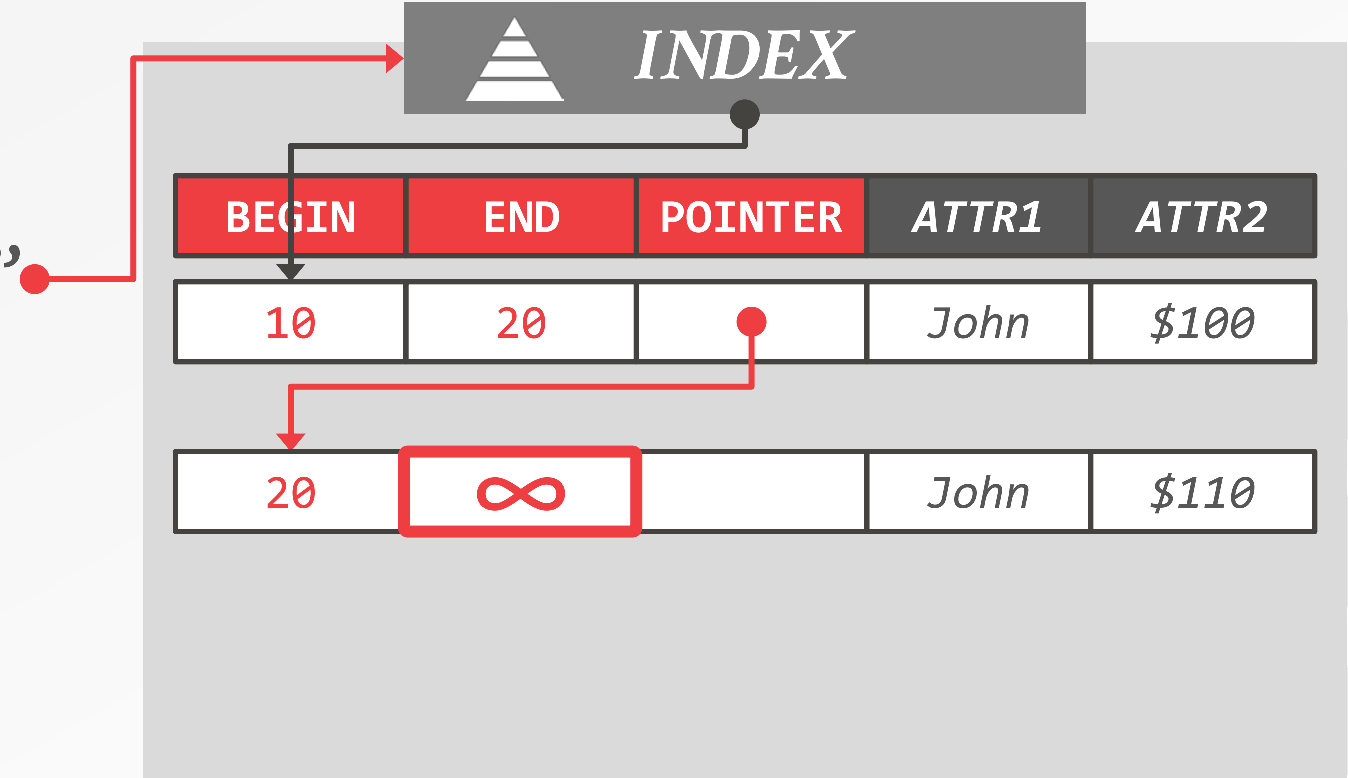
BEGIN @ 25

Read "John"



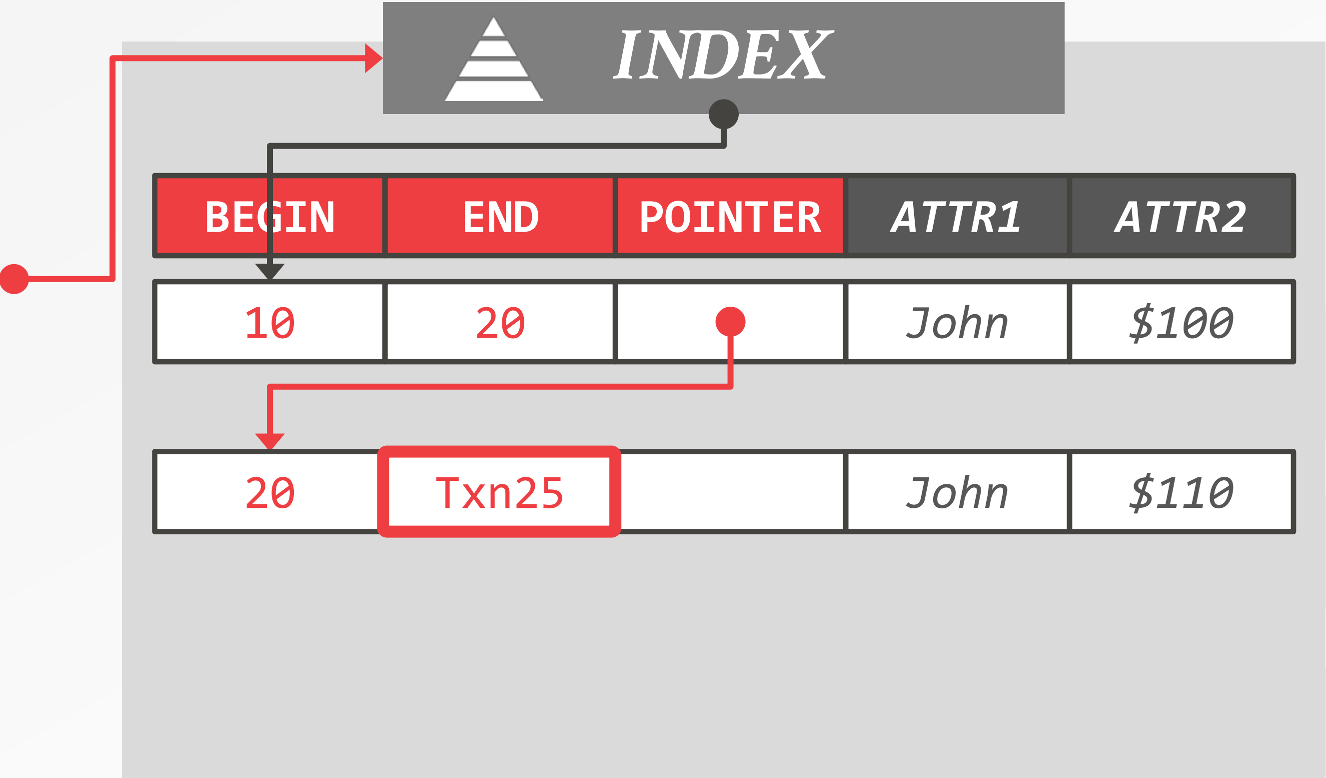
HEKATON: OPERATIONS

BEGIN @ 25
Read "John"
Update "John"



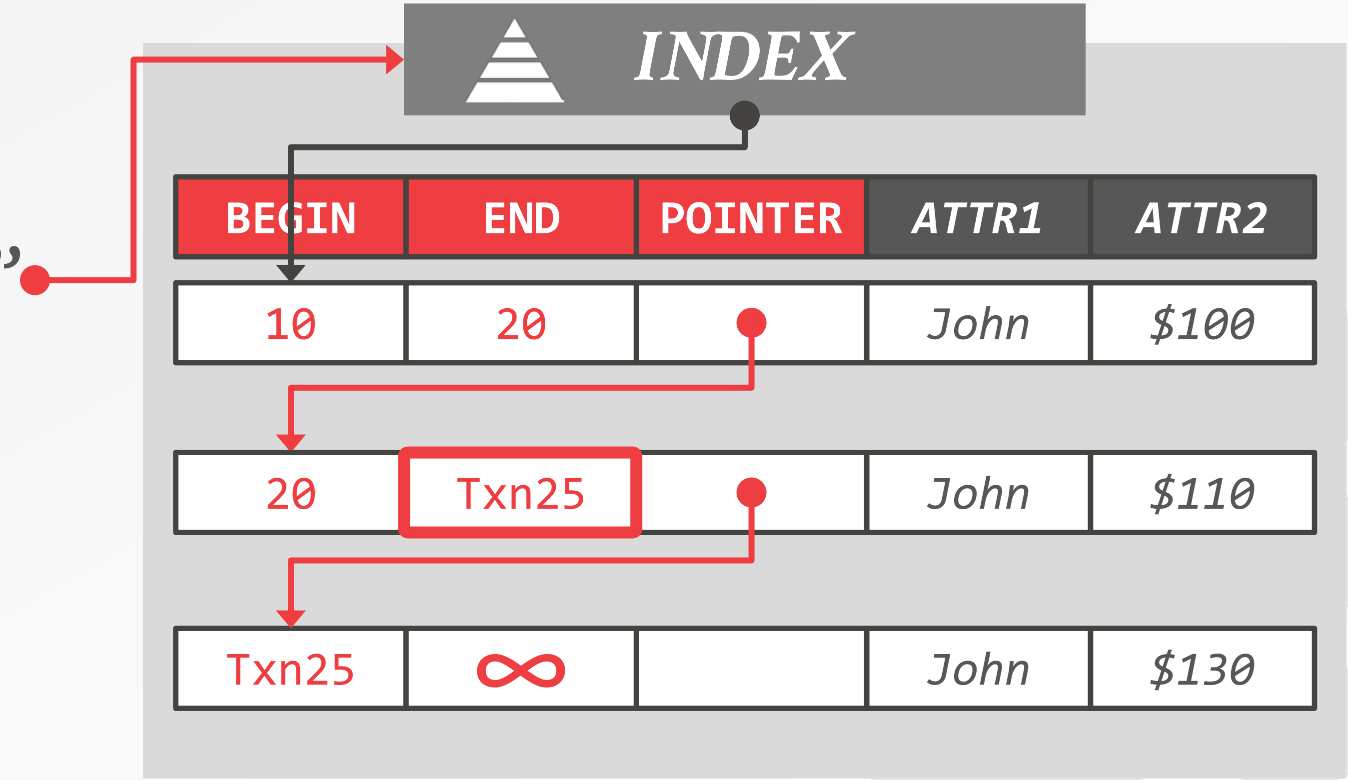
HEKATON: OPERATIONS

BEGIN @ 25
Read "John"
Update "John"



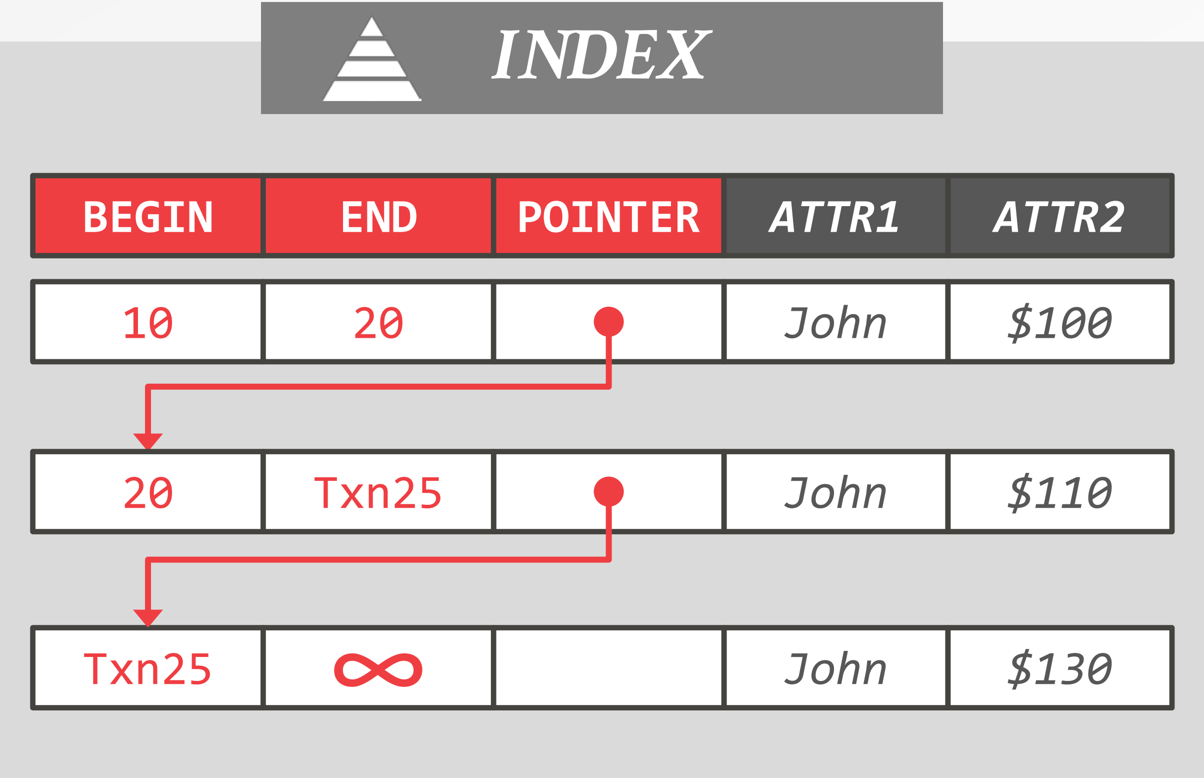
HEKATON: OPERATIONS

BEGIN @ 25
Read “John”
Update “John”



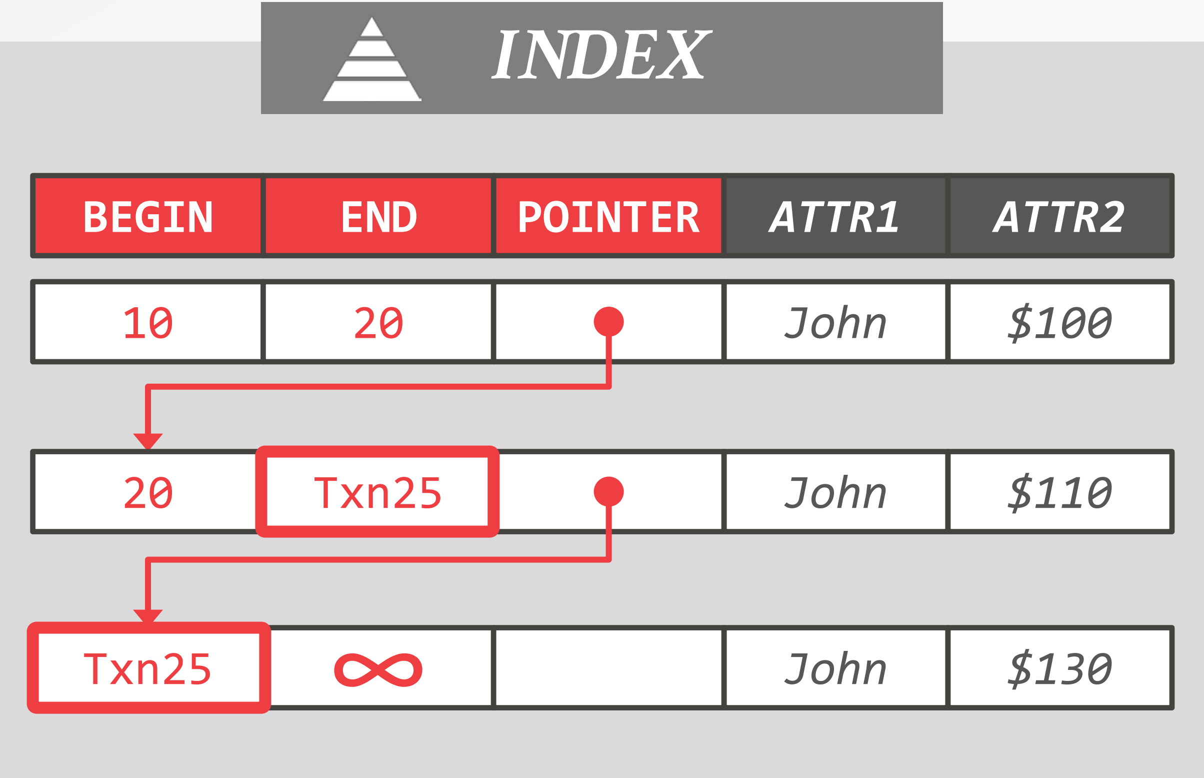
HEKATON: OPERATIONS

BEGIN @ 25
Read "John"
Update "John"
COMMIT @ 35



HEKATON: OPERATIONS

BEGIN @ 25
Read "John"
Update "John"
COMMIT @ 35



HEKATON: OPERATIONS

BEGIN @ 25
Read “John”
Update “John”
COMMIT @ 35

INDEX

BEGIN	END	POINTER	*ATTR1*	*ATTR2*
10	20	●	*John*	*$100*
20	35	●	*John*	*$110*
35	∞		*John*	*$130*

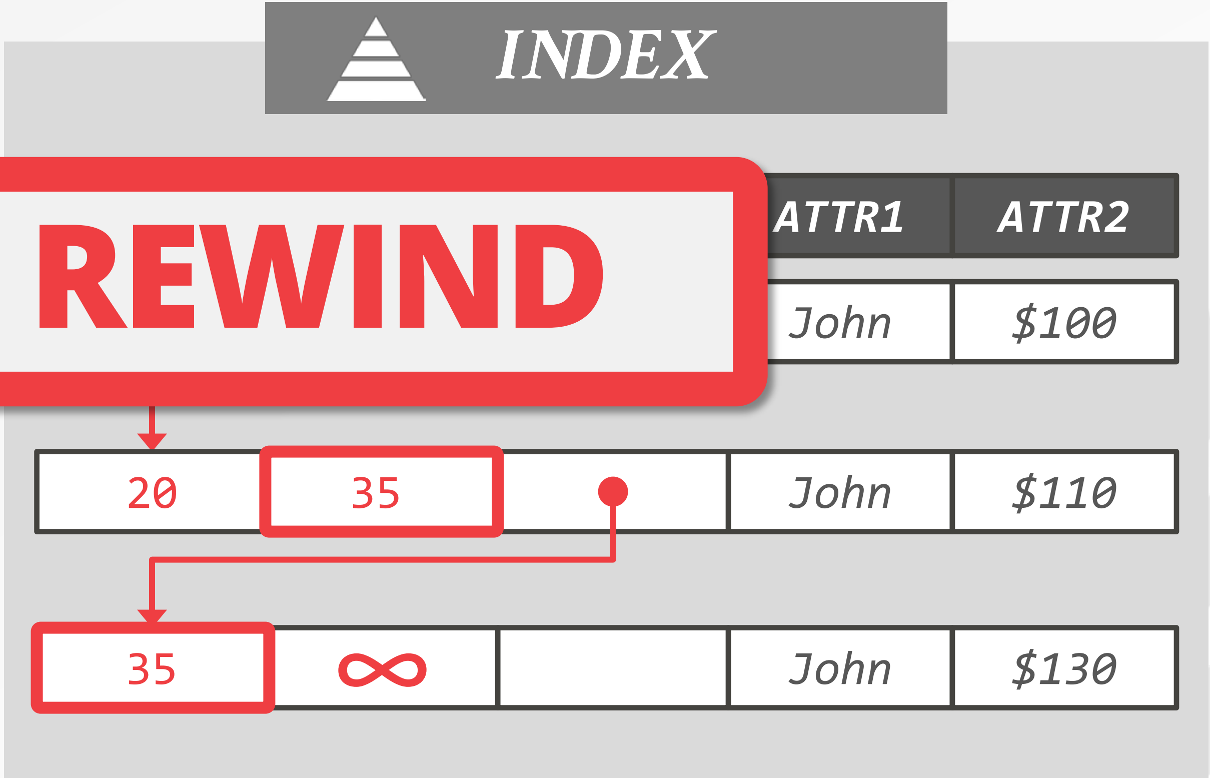
HEKATON: OPERATIONS

BEGIN @ 25

Read "John"

Update "Joh

COMMIT @ 35



HEKATON: OPERATIONS

BEGIN @ 25
Read "John"
Update "John"

INDEX

BEGIN	END	POINTER	ATTR1	ATTR2
10	20	●	John	$100
20	Txn25	●	John	$110
Txn25	∞		John	$130

HEKATON: OPERATIONS

BEGIN @ 25
Read "John"
Update "John"

BEGIN @ 30

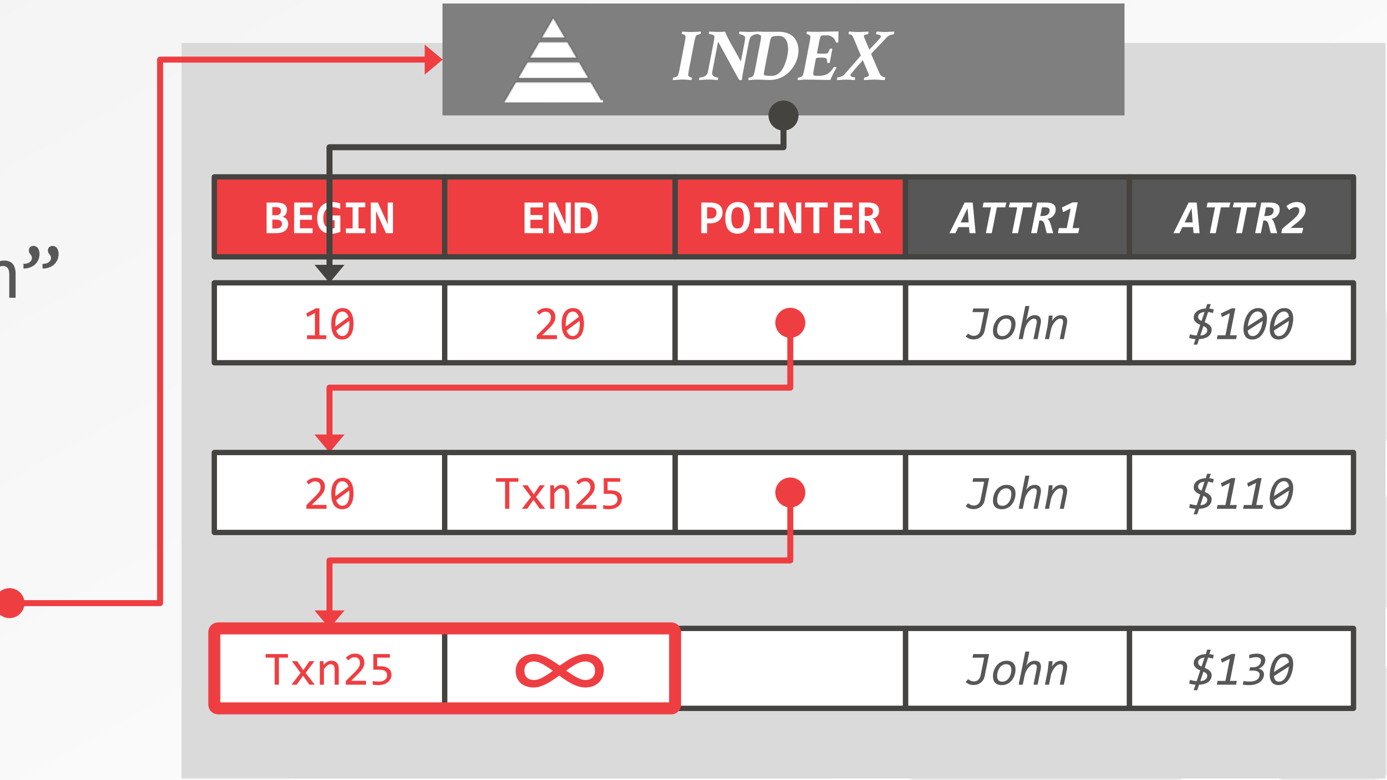
INDEX

BEGIN	END	POINTER	ATTR1	ATTR2
10	20	●	John	$100
20	Txn25	●	John	$110
Txn25	∞		John	$130

HEKATON: OPERATIONS

BEGIN @ 25
Read "John"
Update "John"

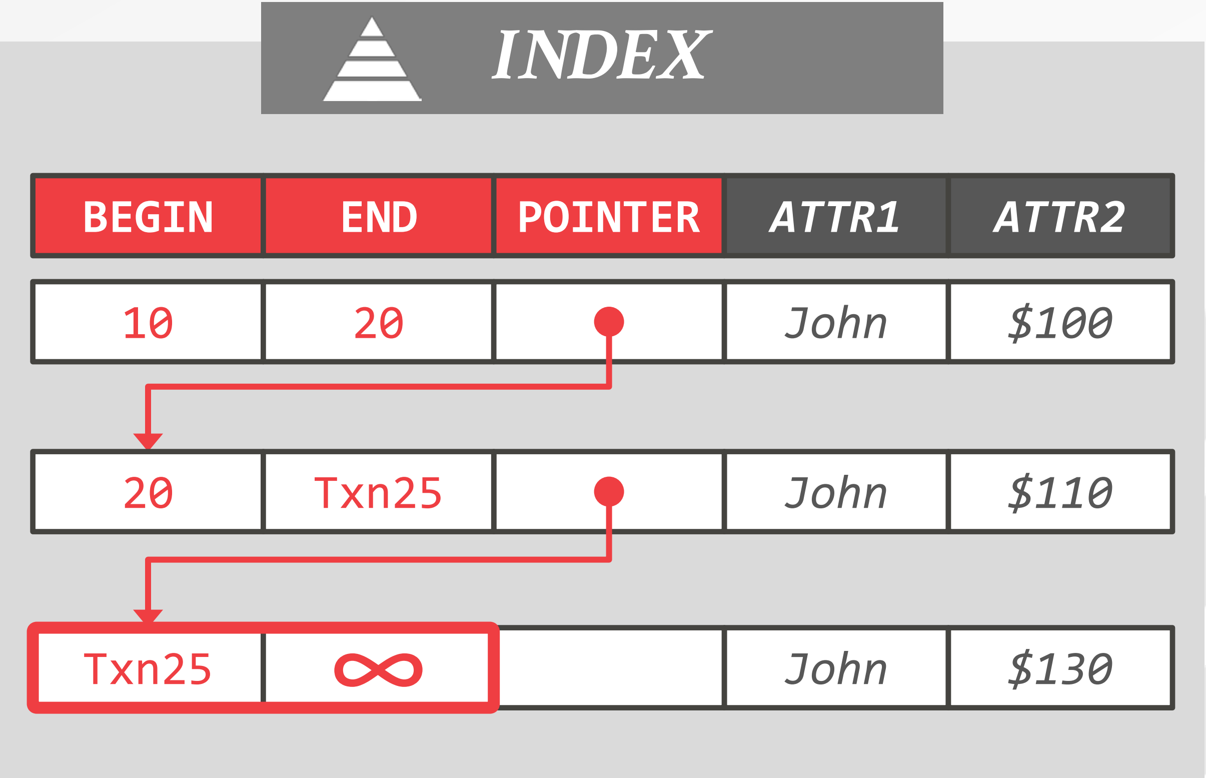
BEGIN @ 30
Read "John"



HEKATON: OPERATIONS

BEGIN @ 25
Read “John”
Update “John”

BEGIN @ 30
Read “John”
Update “John”



HEKATON: OPERATIONS

BEGIN @ 25
Read "John"
Update "John"

BEGIN @ 30
Read "John"
Update "John"

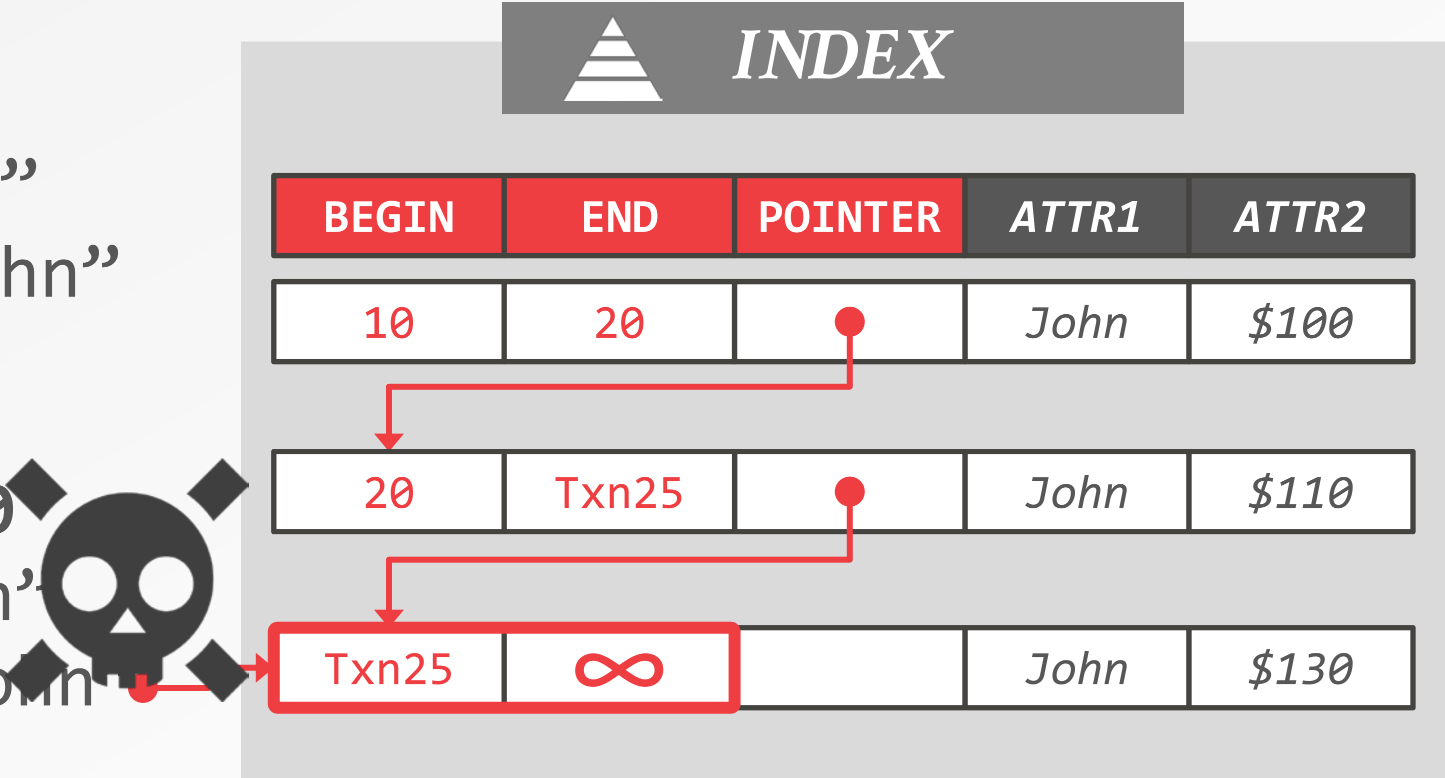
INDEX

BEGIN	END	POINTER	ATTR1	ATTR2
10	20	•	John	$100
20	Txn25	•	John	$110
Txn25	∞		John	$130

HEKATON: OPERATIONS

BEGIN @ 25
Read “John”
Update “John”

BEGIN @ 30
Read “John”
Update “John”



HEKATON: TRANSACTION STATE MAP

Global map of all txns' states in the system:

→ **ACTIVE**: The txn is executing read/write operations.

→ **VALIDATING**: The txn has invoked commit and the DBMS is checking whether it is valid.

→ **COMMITTED**: The txn is finished, but may have not updated its versions' TS.

→ **TERMINATED**: The txn has updated the TS for all of the versions that it created.

HEKATON: TRANSACTION META-DATA

Read Set

→ Pointers to every version read.

Write Set

→ Pointers to versions updated (old and new), versions deleted (old), and version inserted (new).

Scan Set

→ Stores enough information needed to perform each scan operation.

Commit Dependencies

→ List of txns that are waiting for this txn to finish.

HEKATON: TRANSACTION VALIDATION

Read Stability

→ Check that each version read is still visible as of the end of the txn.

Phantom Avoidance

→ Repeat each scan to check whether new versions have become visible since the txn began.

Extent of validation depends on isolation level:

→ **SERIALIZABLE**: Read Stability + Phantom Avoidance
→ **REPEATABLE READS**: Read Stability
→ **SNAPSHOT ISOLATION**: None
→ **READ COMMITTED**: None

HEKATON: OPTIMISTIC VS. PESSIMISTIC

Optimistic Txns:

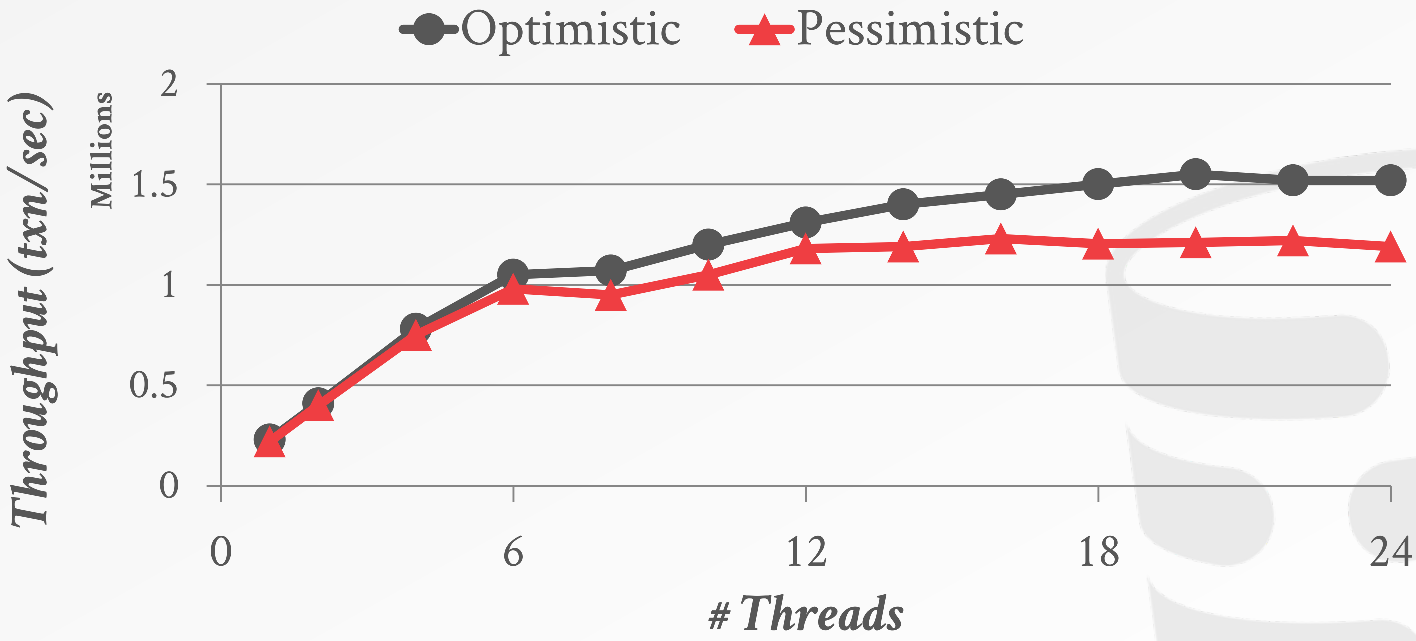
→ Check whether a version read is still visible at the end of the txn.
→ Repeat all index scans to check for phantoms.

Pessimistic Txns:

→ Use shared & exclusive locks on records and buckets.
→ No validation is needed.
→ Separate background thread to detect deadlocks.

HEKATON: OPTIMISTIC VS. PESSIMISTIC

Database: Single table with 1000 tuples
Workload: 80% read-only txns + 20% update txns
Processor: 2 sockets, 12 cores



Source: Paul Larson

HEKATON: PERFORMANCE

Bwin – Large online betting company
→ Before: 15,000 requests/sec
→ Hekaton: 250,000 requests/sec

EdgeNet – Up-to-date inventory status
→ Before: 7,450 rows/sec (ingestion rate)
→ Hekaton: 126,665 rows/sec

SBI Liquidity Market – FOREX broker
→ Before: 2,812 txn/sec with 4 sec latency
→ Hekaton: 5,313 txn/sec with <1 sec latency

Source: Paul Larson

HEKATON: IMPLEMENTATION

Use only lock-free data structures
→ No latches, spin locks, or critical sections
→ Indexes, txn map, memory alloc, garbage collector
→ We will discuss Bw-Trees + Skip Lists later…

Only one single serialization point in the DBMS to get the txn's begin and commit timestamp
→ Atomic Addition (CAS)

OBSERVATIONS

Read/scan set validations are expensive if the txns access a lot of data.

Appending new versions hurts the performance of OLAP scans due to pointer chasing & branching.

Record-level conflict checks may be too coarse-grained and incur false positives.

HYPER MVCC

Rollback Segment with Deltas
→ In-Place updates for non-indexed attributes
→ Delete/Insert updates for indexed attributes.

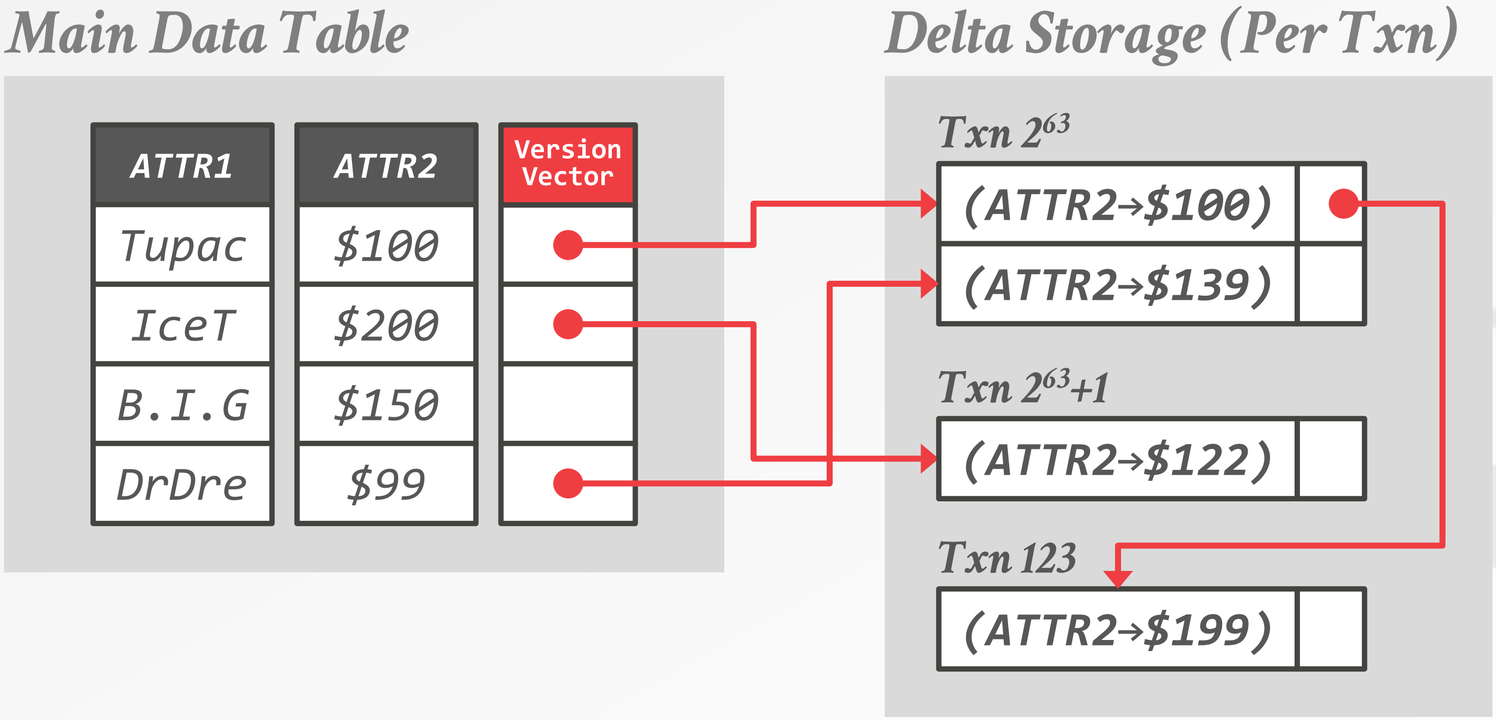
Newest-to-Oldest Version Chains

No Predicate Locks

Avoids write-write conflicts by aborting txns that try to update an uncommitted object.

FAST SERIALIZABLE MULTI-VERSION CONCURRENCY CONTROL FOR MAIN-MEMORY DATABASE SYSTEMS
SIGMOD 2015

HYPER MVCC


Main Data Table
Delta Storage (Per Txn)
ATTR1
ATTR2
Version Vector
Tupac
$100
IceT
$200
B.I.G
$150
DrDre
$99
Txn 2^{63}
(ATTR2→$100)
(ATTR2→$139)
Txn 2^{63}+1
(ATTR2→$122)
Txn 123
(ATTR2→$199)

PARTING THOUGHTS

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

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

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

PROJECT #2

Implement a latch-free Skip List in Peloton.
→ Forward / Reverse Iteration
→ Garbage Collection

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

PROJECT #2 – DESIGN

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

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

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

→ There is no right answer.

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

PROJECT #2 – TESTING

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

We also have a BwTree implementation to compare against.

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

PROJECT #2 – DOCUMENTATION

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

We will inspect the submissions manually.

PROJECT #2 – GRADING

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

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

→ We will use Valgrind when testing your code.

All source code must pass ClangFormat syntax formatting checker.

→ See Peloton documentation for formatting guidelines.

PROJECT #2 – GROUPS

This is a group project.
→ Everyone should contribute equally.
→ I will review commit history.

Email me if you do not have a group.

PROJECT #2

Due Date: March 2nd, 2017 @ 11:59pm

Projects will be turned in using Autolab.

Full description and instructions:

http://15721.courses.cs.cmu.edu/spring2017/project2.html

NEXT CLASS

Index Locking + Latching