Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Multi-Version Concurrency Control (Protocols)

@Andy_Pavlo // 15-721 // Spring 2019

Lecture #04

LAST CLASS

We discussed the four major design decisions for building a MVCC DBMS.

- → Concurrency Control Protocol
- → Version Storage
- → Garbage Collection
- → Index Management

TODAY'S AGENDA

Microsoft Hekaton (SQL Server)

TUM HyPer

SAP HANA

CMU Cicada

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/VoltDB) works well for some applications but terrible for others.

HEKATON MVCC

Each txn is assigned a timestamp when they begin (BeginTS) and when they commit (CommitTS).

Each tuple contains two timestamps that represents their visibility and current state:

- → **BEGIN-TS**: The BeginTS of the active txn **or** the CommitTS of the committed txn that created it.
- → **END-TS**: The BeginTS of the active txn that created the next version **or** infinity **or** the CommitTS of the committed txn that created it.

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

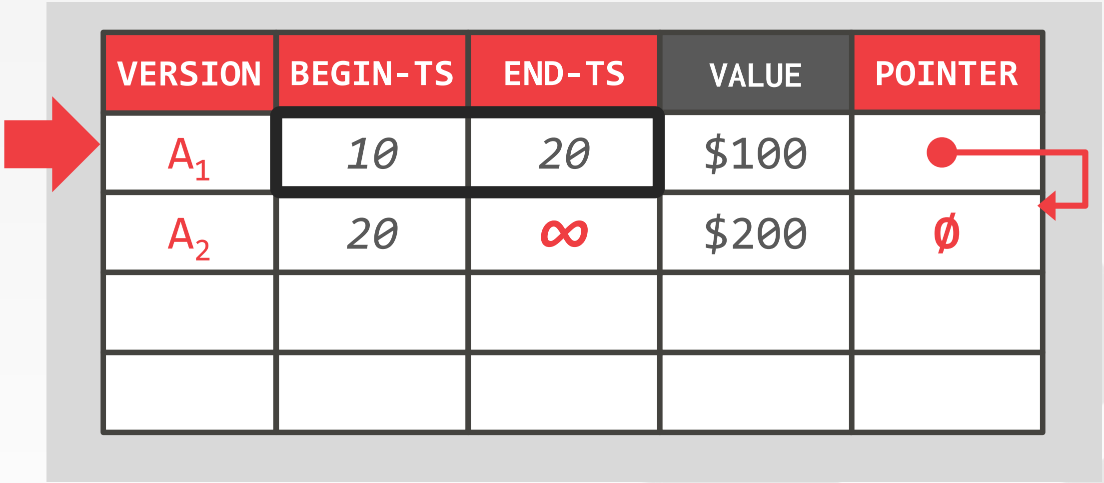
HEKATON: OPERATIONS

Thread #1

Begin @ 25

READ(A)

Main Data Table



VERSION	BEGIN-TS	END-TS	VALUE	POINTER
A_1	10	20	$100	●
A_2	20	∞	$200	∅

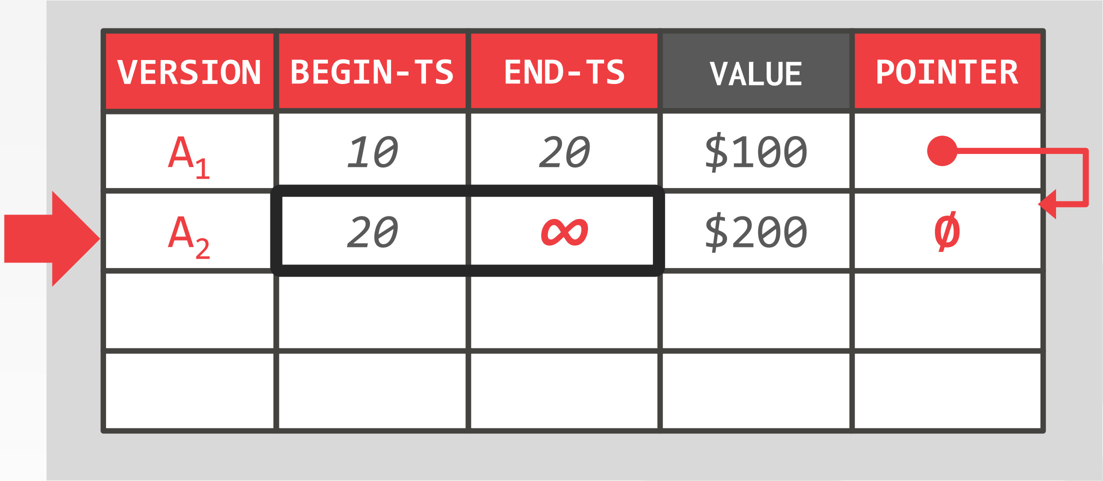
HEKATON: OPERATIONS

Thread #1

Begin @ 25

READ(A)

Main Data Table



VERSION	BEGIN-TS	END-TS	VALUE	POINTER
A_1	10	20	$100	●
A_2	20	∞	$200	Ø

HEKATON: OPERATIONS

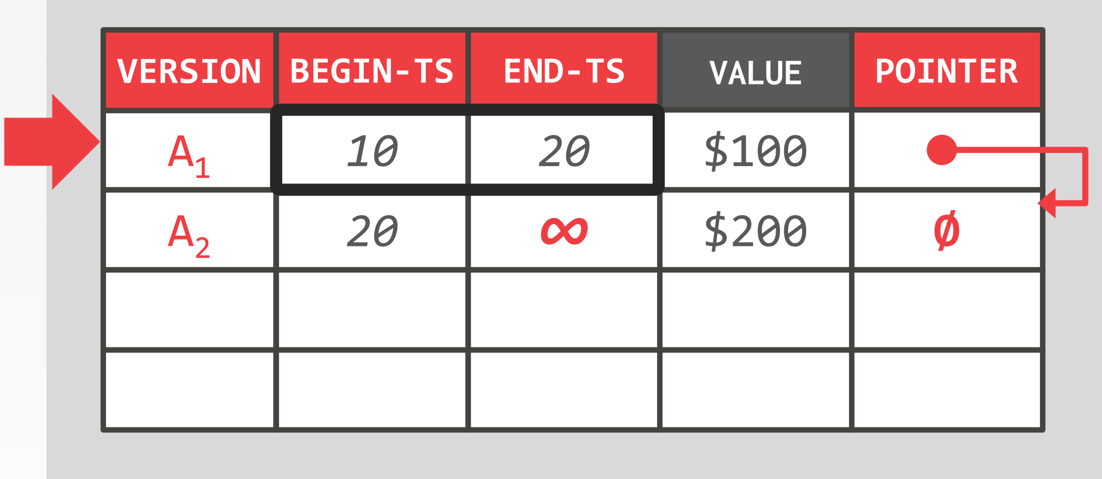
Thread #1

Begin @ 25

READ(A)

WRITE(A)

Main Data Table



VERSION	BEGIN-TS	END-TS	VALUE	POINTER
A_1	10	20	$100	●
A_2	20	∞	$200	Ø

HEKATON: OPERATIONS

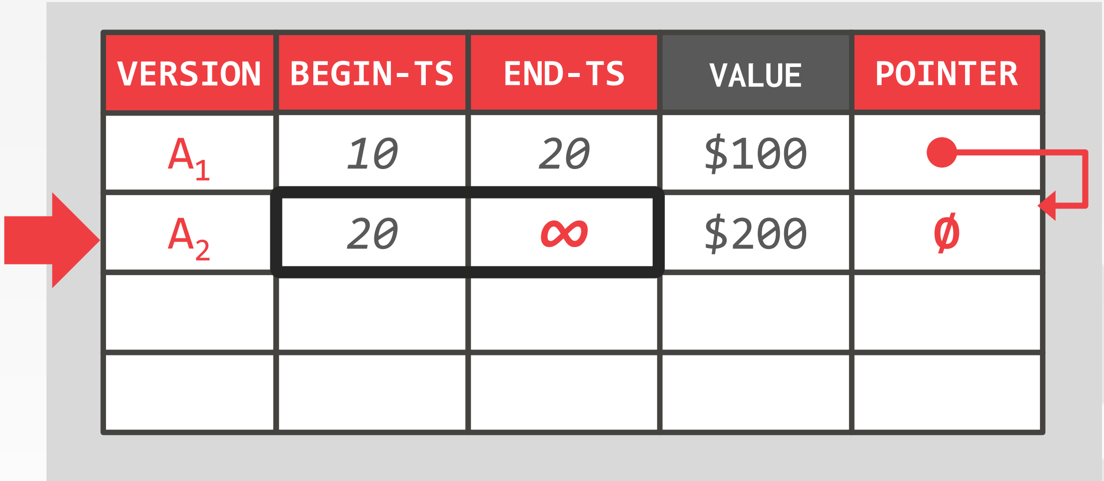
Thread #1

Begin @ 25

READ(A)

WRITE(A)

Main Data Table



VERSION	BEGIN-TS	END-TS	VALUE	POINTER
A_1	10	20	$100	●
A_2	20	∞	$200	Ø

HEKATON: OPERATIONS

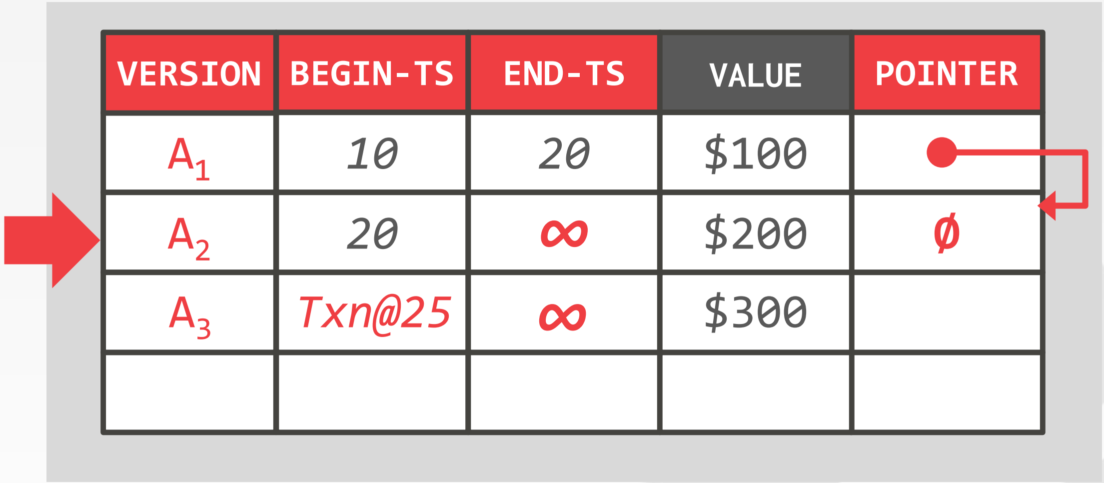
Thread #1

Begin @ 25

READ(A)

WRITE(A)

Main Data Table



VERSION	BEGIN-TS	END-TS	VALUE	POINTER
A_1	*10*	*20*	$100	●
A_2	*20*	∞	$200	Ø
A_3	*Txn@25*	∞	$300	

HEKATON: OPERATIONS

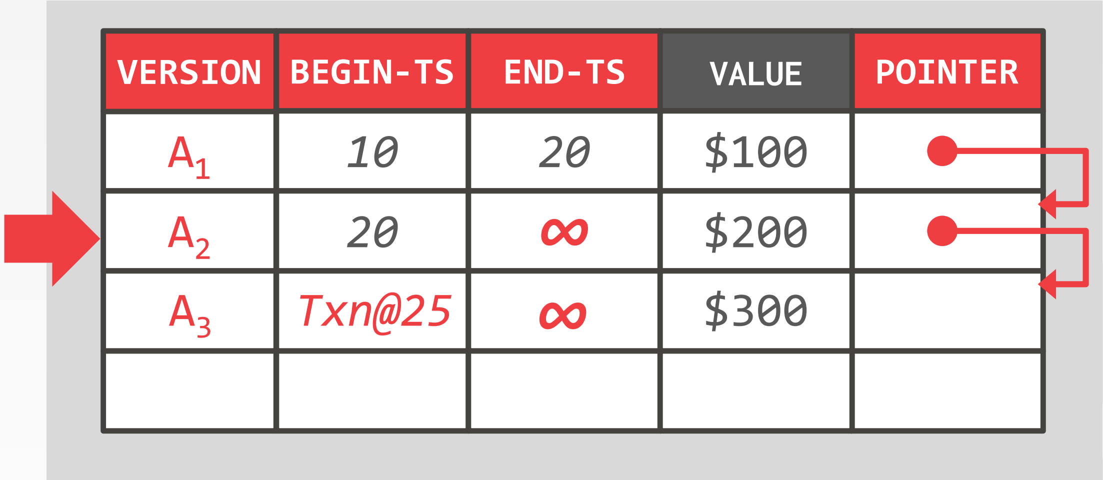
Thread #1

Begin @ 25

READ(A)

WRITE(A)

Main Data Table



VERSION	BEGIN-TS	END-TS	VALUE	POINTER
A_1	10	20	$100	●
A_2	20	∞	$200	●
A_3	*Txn@25*	∞	$300	

HEKATON: OPERATIONS

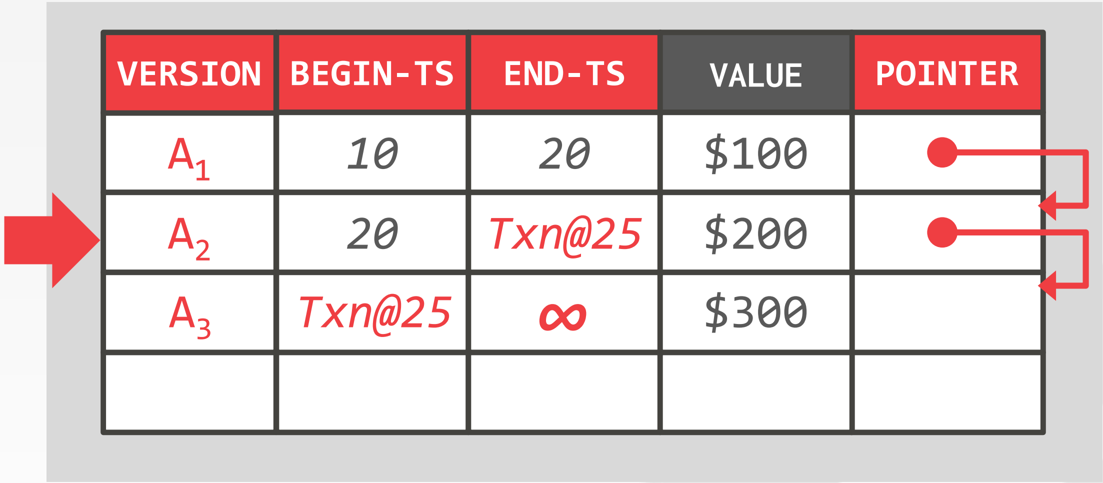
Thread #1

Begin @ 25

READ(A)

WRITE(A)

Main Data Table



VERSION	BEGIN-TS	END-TS	VALUE	POINTER
A_1	10	20	$100	●
A_2	20	Txn@25	$200	●
A_3	Txn@25	∞	$300	

HEKATON: OPERATIONS

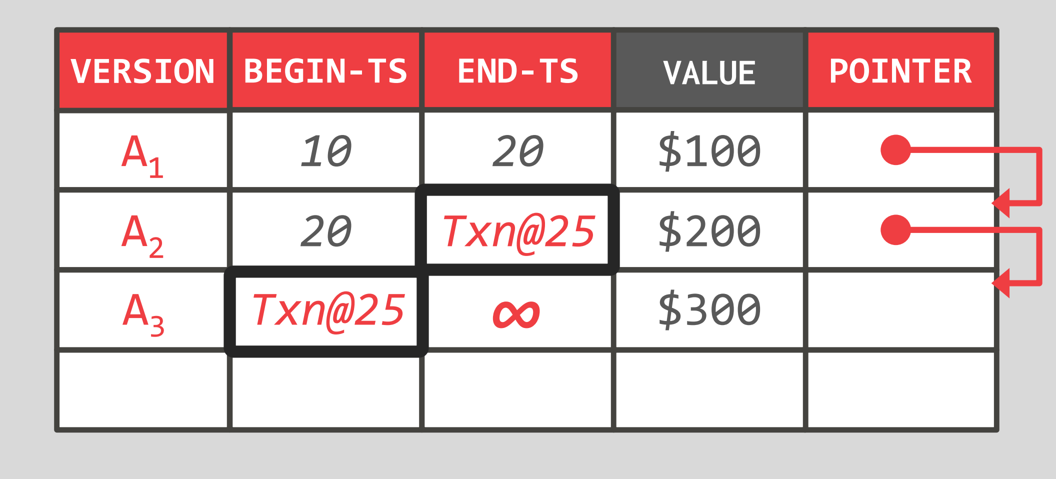
Thread #1

Begin @ 25
Commit @ 35

READ(A)

WRITE(A)

Main Data Table



VERSION	BEGIN-TS	END-TS	VALUE	POINTER
A_1	10	20	$100	●
A_2	20	Txn@25	$200	●
A_3	Txn@25	∞	$300	

HEKATON: OPERATIONS

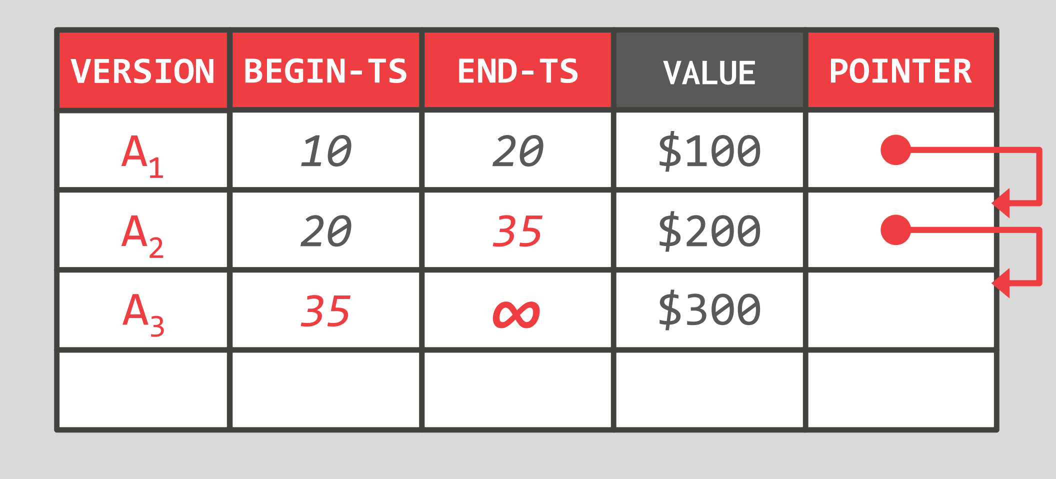
Thread #1

Begin @ 25
Commit @ 35

READ(A)

WRITE(A)

Main Data Table



VERSION	BEGIN-TS	END-TS	VALUE	POINTER
A_1	10	20	$100	●
A_2	20	35	$200	●
A_3	35	∞	$300	

HEKATON: OPERATIONS

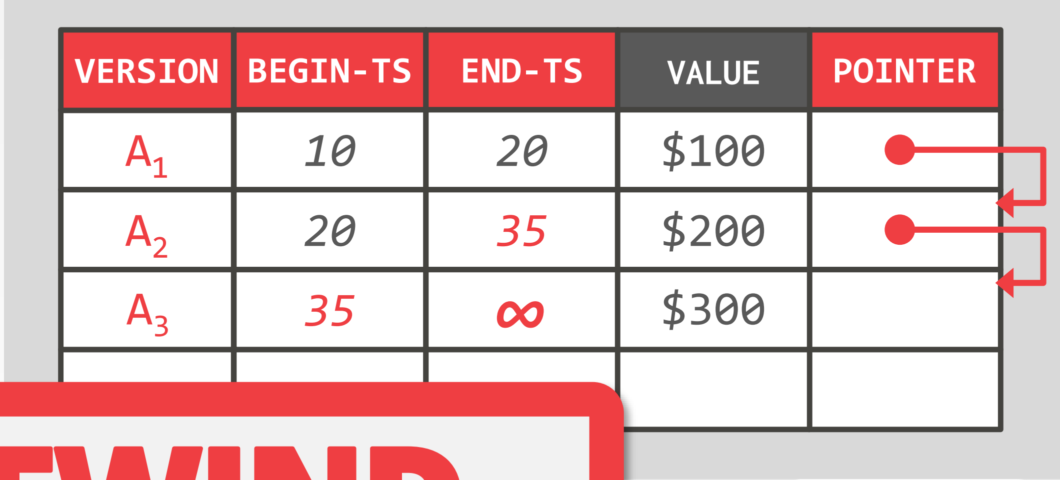
Thread #1

Begin @ 25
Commit @ 35

READ(A)

WRITE(A)

Main Data Table



VERSION	BEGIN-TS	END-TS	VALUE	POINTER
A_1	10	20	\$100	●
A_2	20	35	\$200	●
A_3	35	∞	\$300	

REWIND

HEKATON: OPERATIONS

Thread #1

Begin @ 25

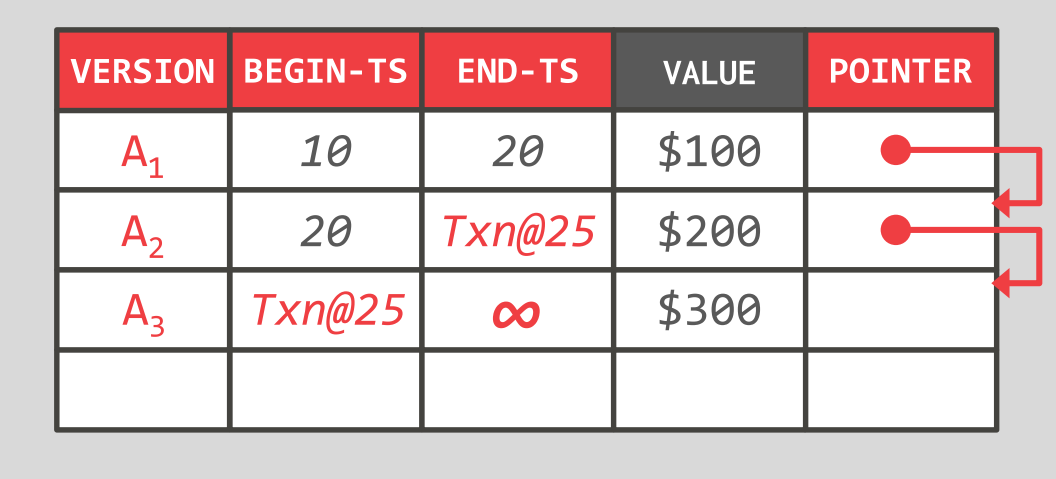
READ(A)

WRITE(A)

Thread #2

Begin @ 30

Main Data Table



VERSION	BEGIN-TS	END-TS	VALUE	POINTER
A_1	10	20	\$100	●
A_2	20	*Txn@25*	\$200	●
A_3	*Txn@25*	∞	\$300	

HEKATON: OPERATIONS

Thread #1

Begin @ 25

READ(A)

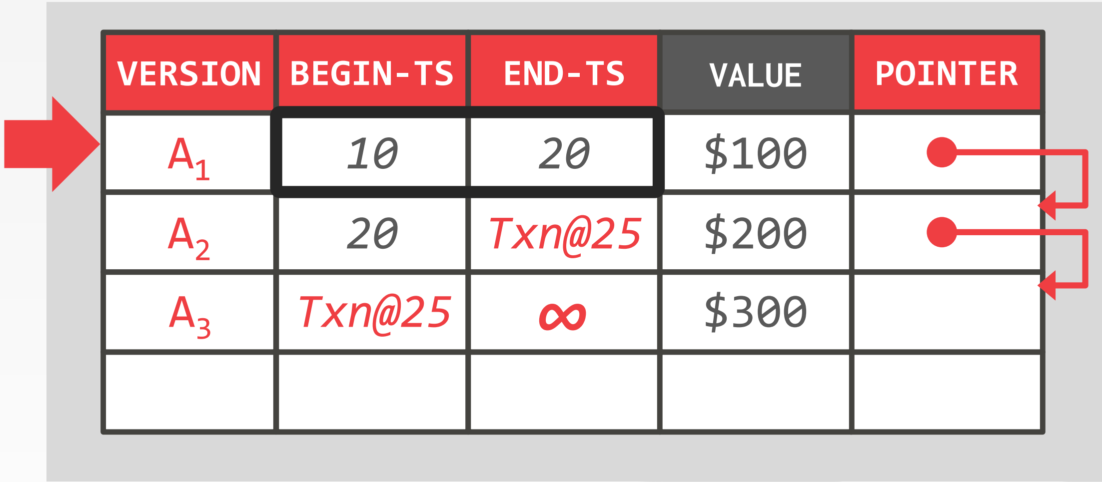
WRITE(A)

Thread #2

Begin @ 30

READ(A)

Main Data Table



VERSION	BEGIN-TS	END-TS	VALUE	POINTER
A_1	10	20	\$100	●
A_2	20	Txn@25	\$200	●
A_3	Txn@25	∞	\$300	

HEKATON: OPERATIONS

Thread #1

Begin @ 25

READ(A)

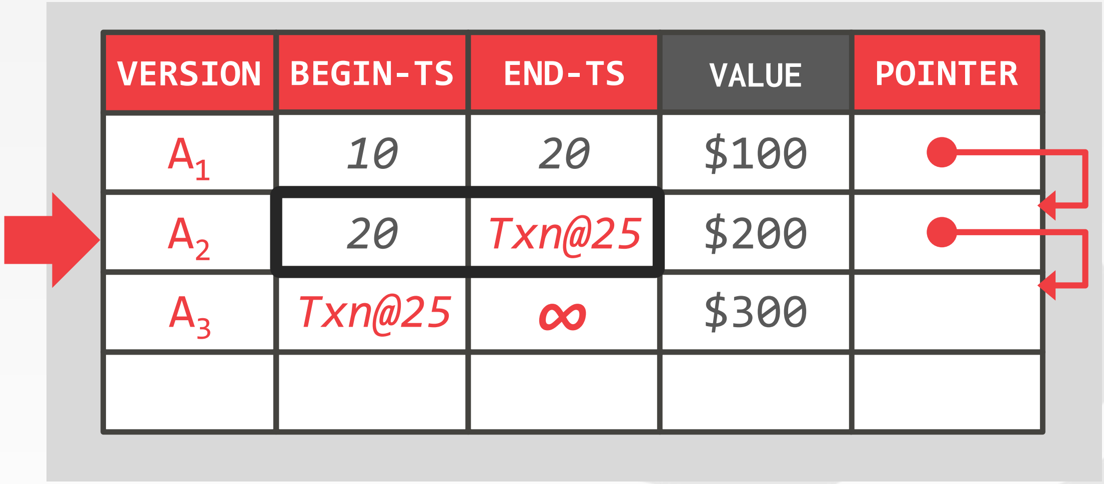
WRITE(A)

Thread #2

Begin @ 30

READ(A)

Main Data Table



VERSION	BEGIN-TS	END-TS	VALUE	POINTER
A_1	10	20	\$100	●
A_2	20	Txn@25	\$200	●
A_3	Txn@25	∞	\$300	

HEKATON: OPERATIONS

Thread #1

Begin @ 25

READ(A)

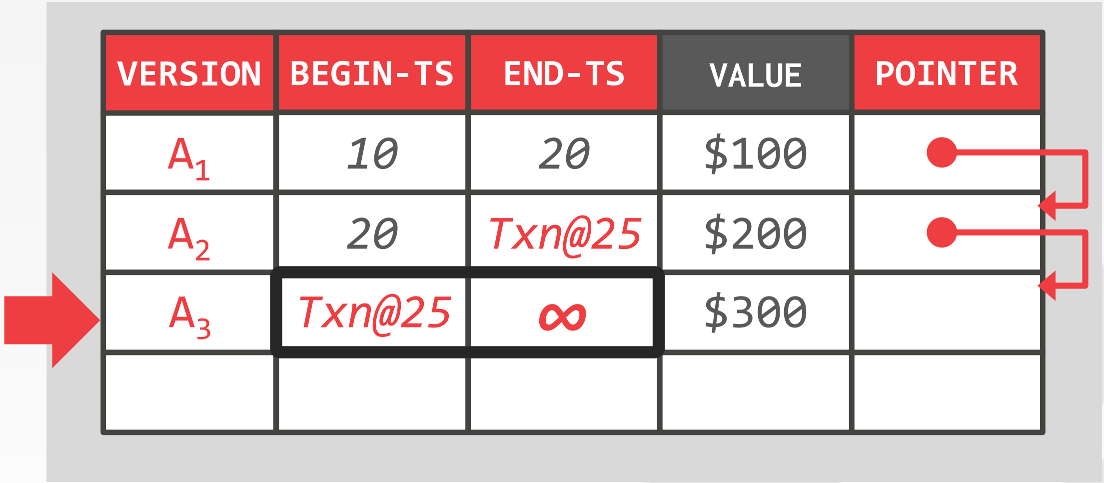
WRITE(A)

Thread #2

Begin @ 30

READ(A)

Main Data Table



VERSION	BEGIN-TS	END-TS	VALUE	POINTER
A_1	10	20	$100	●
A_2	20	Txn@25	$200	●
A_3	Txn@25	∞	$300	

HEKATON: OPERATIONS

Thread #1

Begin @ 25

READ(A)

WRITE(A)

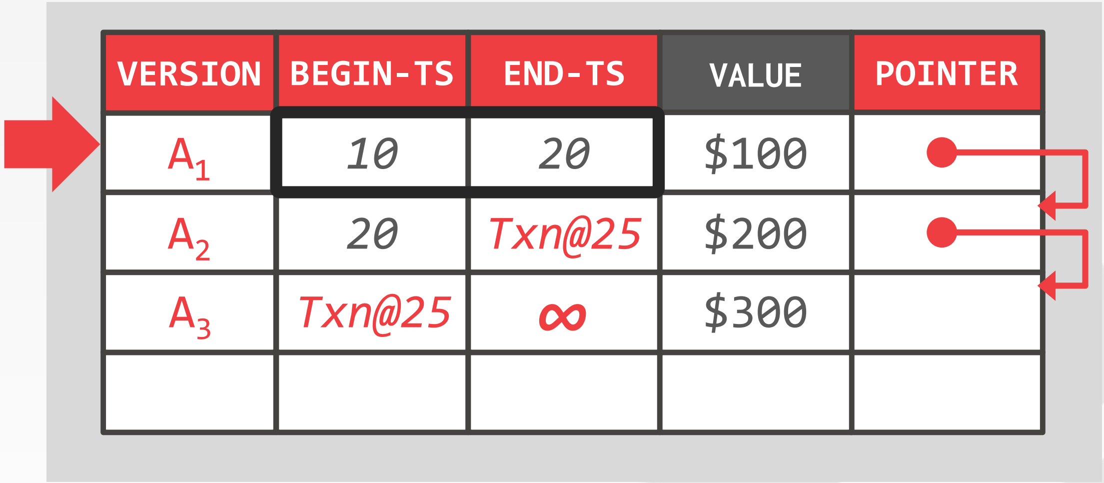
Thread #2

Begin @ 30

READ(A)

WRITE(A)

Main Data Table



VERSION	BEGIN-TS	END-TS	VALUE	POINTER
A_1	10	20	\$100	●
A_2	20	Txn@25	\$200	●
A_3	Txn@25	∞	\$300	

HEKATON: OPERATIONS

Thread #1

Begin @ 25

READ(A)

WRITE(A)

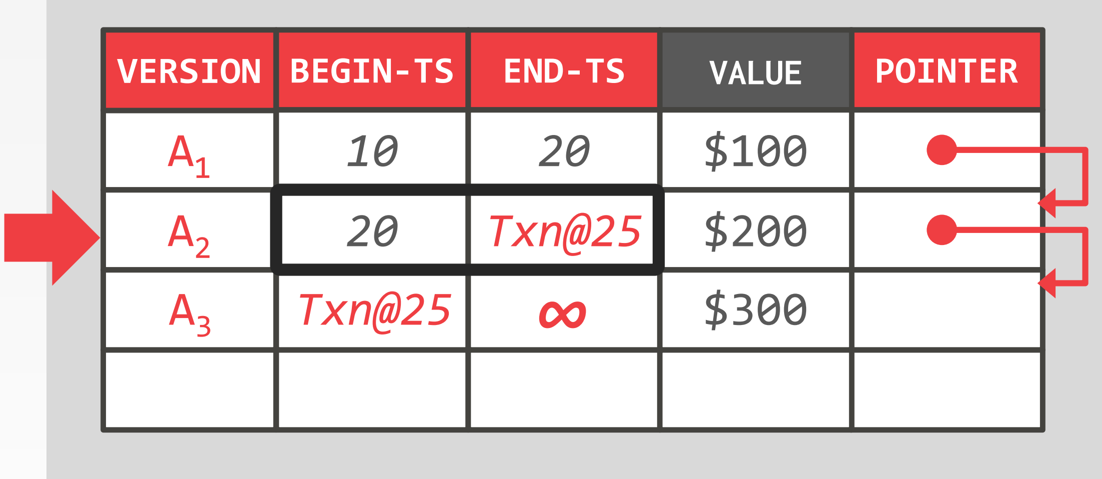
Thread #2

Begin @ 30

READ(A)

WRITE(A)

Main Data Table



VERSION	BEGIN-TS	END-TS	VALUE	POINTER
A_1	*10*	*20*	$100	●
A_2	*20*	*Txn@25*	$200	●
A_3	*Txn@25*	∞	$300	

HEKATON: OPERATIONS

Thread #1

Begin @ 25

READ(A)

WRITE(A)

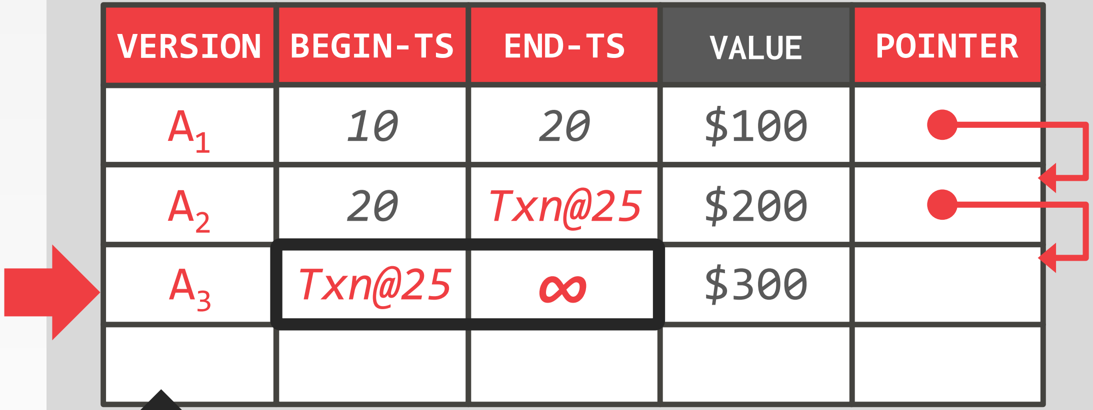
Thread #2

Begin @ 30

READ(A)

WRITE(A)

Main Data Table



VERSION	BEGIN-TS	END-TS	VALUE	POINTER
A_1	10	20	$100	●
A_2	20	Txn@25	$200	●
A_3	Txn@25	∞	$300	

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 LIFECYCLE

Txn Events	*Txn Phases*	
BEGIN		Get BeginTS, set state to **ACTIVE**
	Normal processing	Track txn's read set, scan set, and write set.
PRECOMMIT		Get CommitTS, set state to **VALIDATING**
	Validation	Validate reads and scans → If validation OK, write new versions to redo log
COMMIT		Set txn state to **COMMITTED**
	Post-processing	Update version timestamps → BeginTS in new versions, CommitTS in old versions
TERMINATE		Set txn state to **TERMINATED**
		Remove from txn map

Source: Paul Larson

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: OPTIMISTIC VS. PESSIMISTIC

Optimistic Txns:

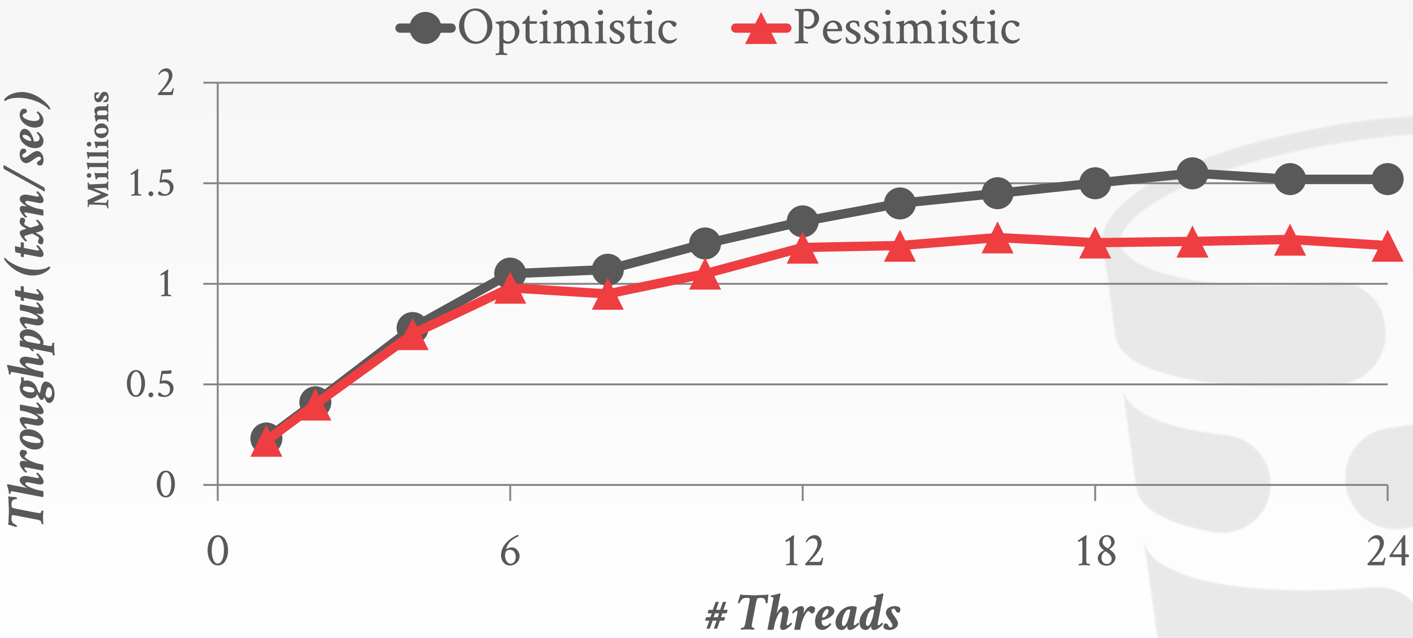
→ Check whether a version read is still visible at the end of the txn.
→ Repeat all index scans to check for phantoms.

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: LESSONS

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

Column-store with delta record versioning.

→ In-Place updates for non-indexed attributes
→ Delete/Insert updates for indexed attributes.
→ Newest-to-Oldest Version Chains
→ No Predicate Locks / No Scan Checks

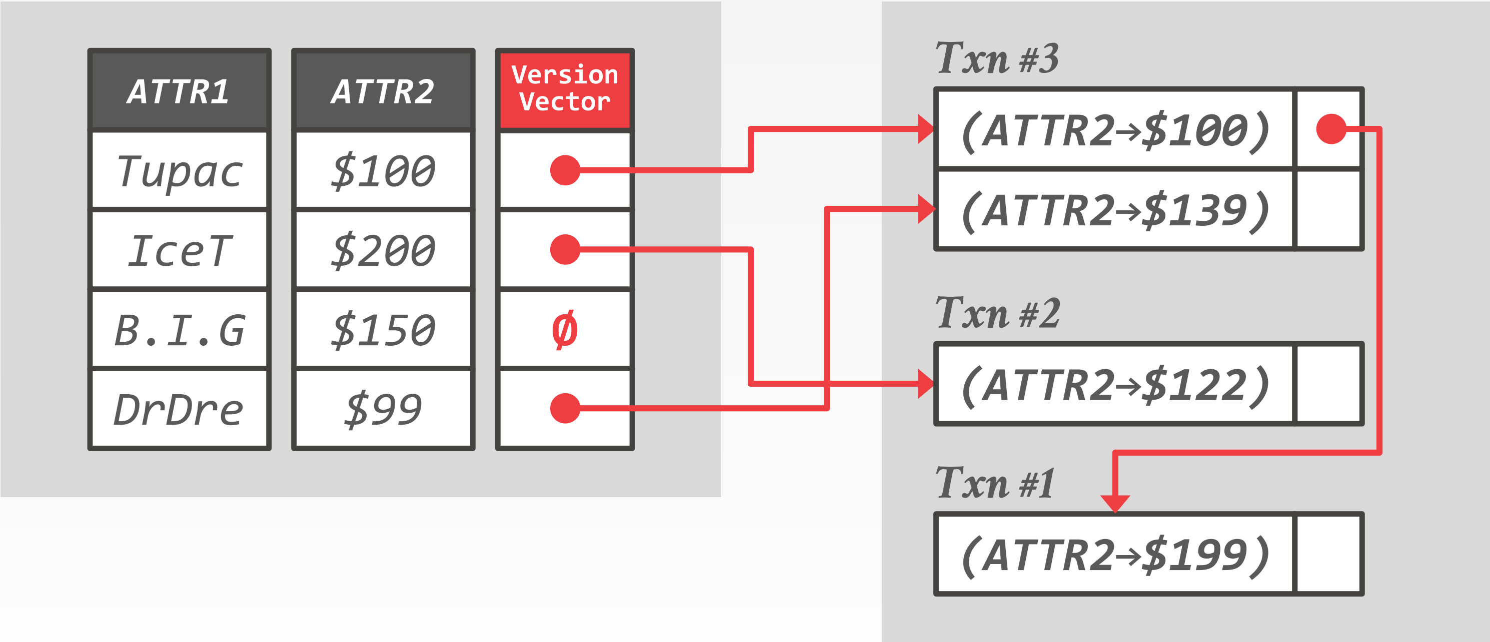
Avoids write-write conflicts by aborting txns that try to update an uncommitted object.

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

HYPER: STORAGE ARCHITECTURE

Main Data Table

Delta Storage (Per Txn)



ATTR1	ATTR2	Version Vector
Tupac	$100	●
IceT	$200	●
B.I.G	$150	Ø
DrDre	$99	●

Txn #3

(ATTR2→$100)

(ATTR2→$139)

Txn #2

(ATTR2→$122)

Txn #1

(ATTR2→$199)

HYPER: VALIDATION

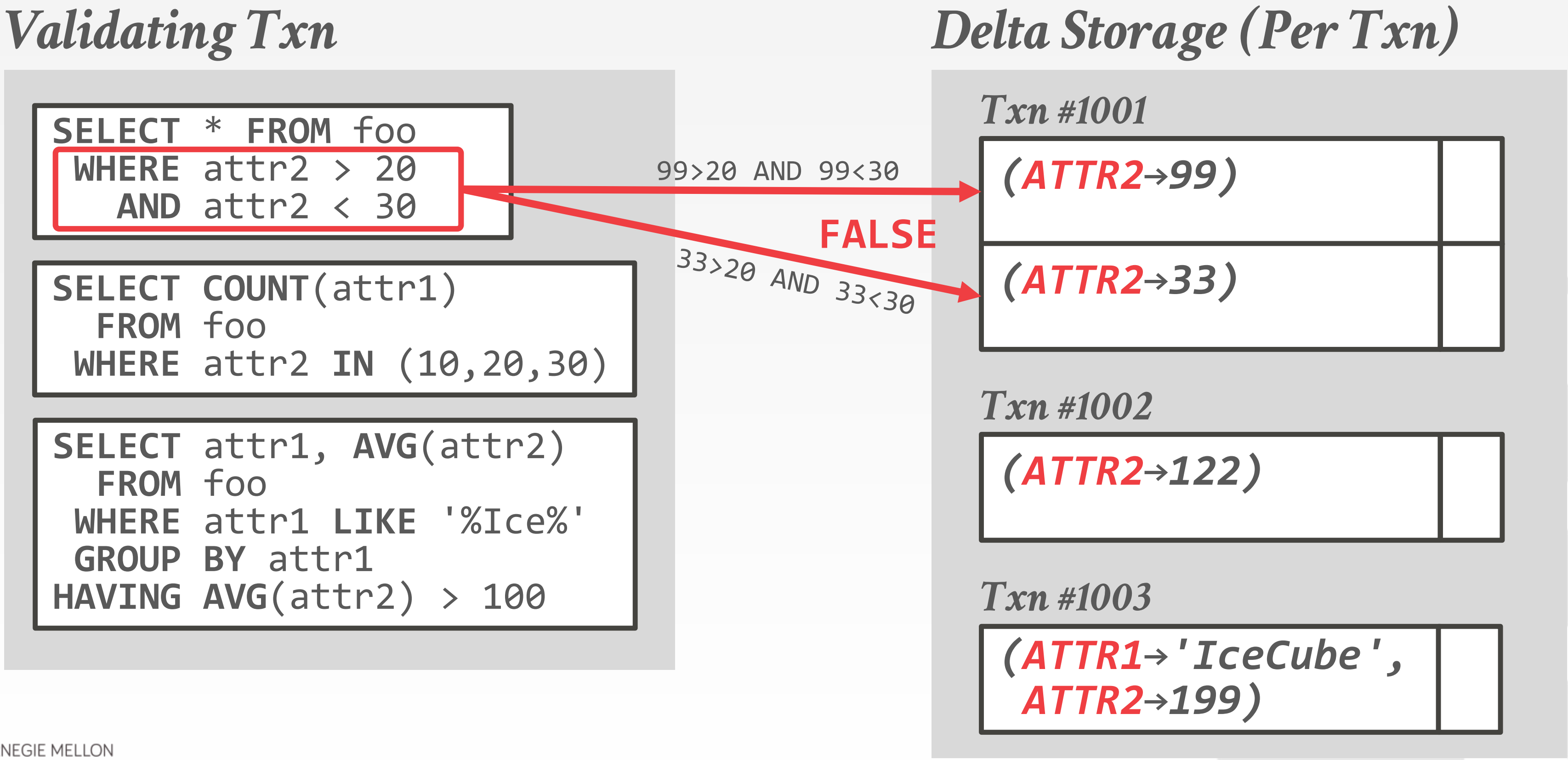
First-Writer Wins

- → The version vector always points to the last committed version.
- → Do not need to check whether write-sets overlap.

Check the undo buffers (i.e., delta records) of txns that committed **after** the validating txn started.

- → Compare the committed txn's write set for phantoms using **Precision Locking**.
- → Only need to store the txn's read predicates and not its entire read set.

HYPER: PRECISION LOCKING



Validating Txn

```
SELECT * FROM foo
 WHERE attr2 > 20
   AND attr2 < 30
```

```
SELECT COUNT(attr1)
  FROM foo
 WHERE attr2 IN (10,20,30)
```

```
SELECT attr1, AVG(attr2)
  FROM foo
 WHERE attr1 LIKE '%Ice%'
 GROUP BY attr1
HAVING AVG(attr2) > 100
```

Delta Storage (Per Txn)

Txn #1001

(ATTR2→99)

(ATTR2→33)

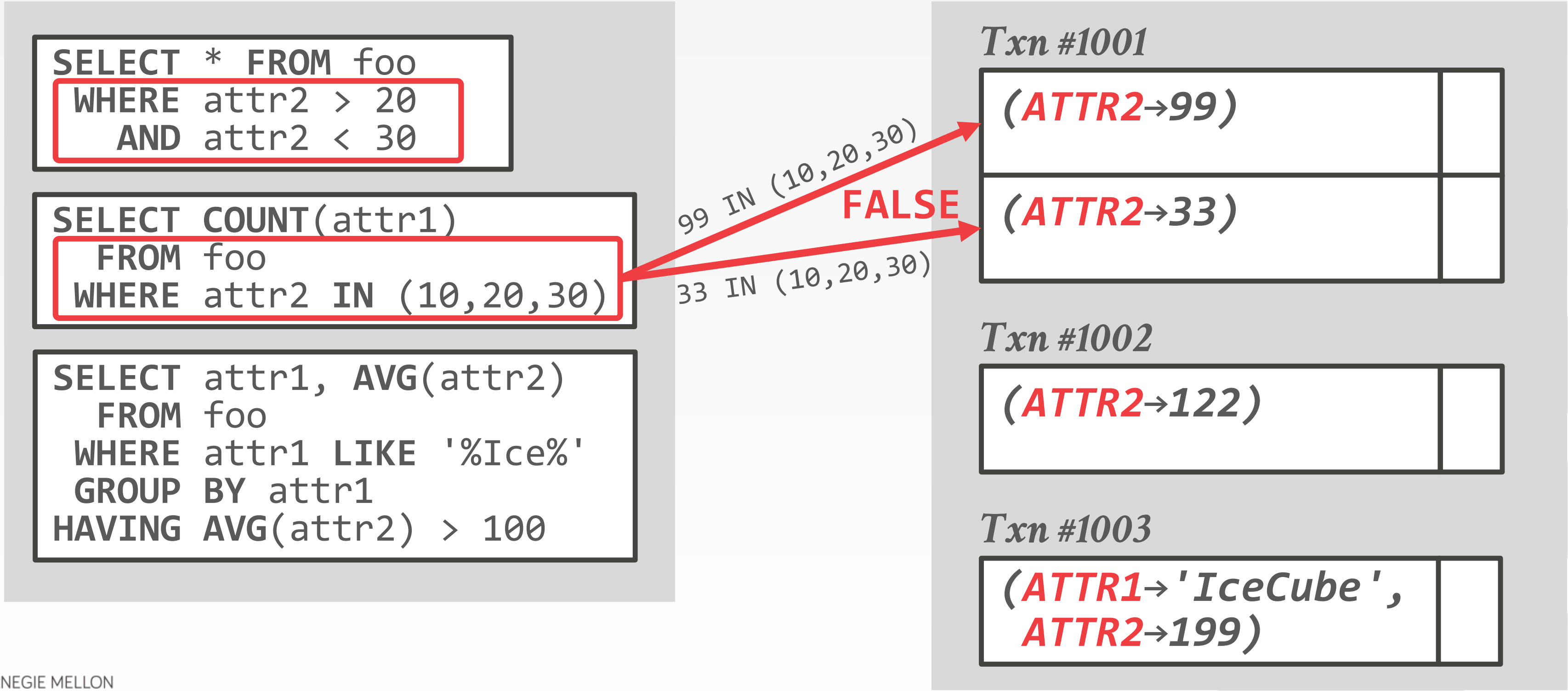
Txn #1002

(ATTR2→122)

Txn #1003

(ATTR1→'IceCube',
ATTR2→199)

HYPER: PRECISION LOCKING



Validating Txn

```
SELECT * FROM foo
 WHERE attr2 > 20
   AND attr2 < 30
```

```
SELECT COUNT(attr1)
  FROM foo
 WHERE attr2 IN (10,20,30)
```

```
SELECT attr1, AVG(attr2)
  FROM foo
 WHERE attr1 LIKE '%Ice%'
 GROUP BY attr1
HAVING AVG(attr2) > 100
```

Delta Storage (Per Txn)

Txn #1001

(ATTR2→99)

(ATTR2→33)

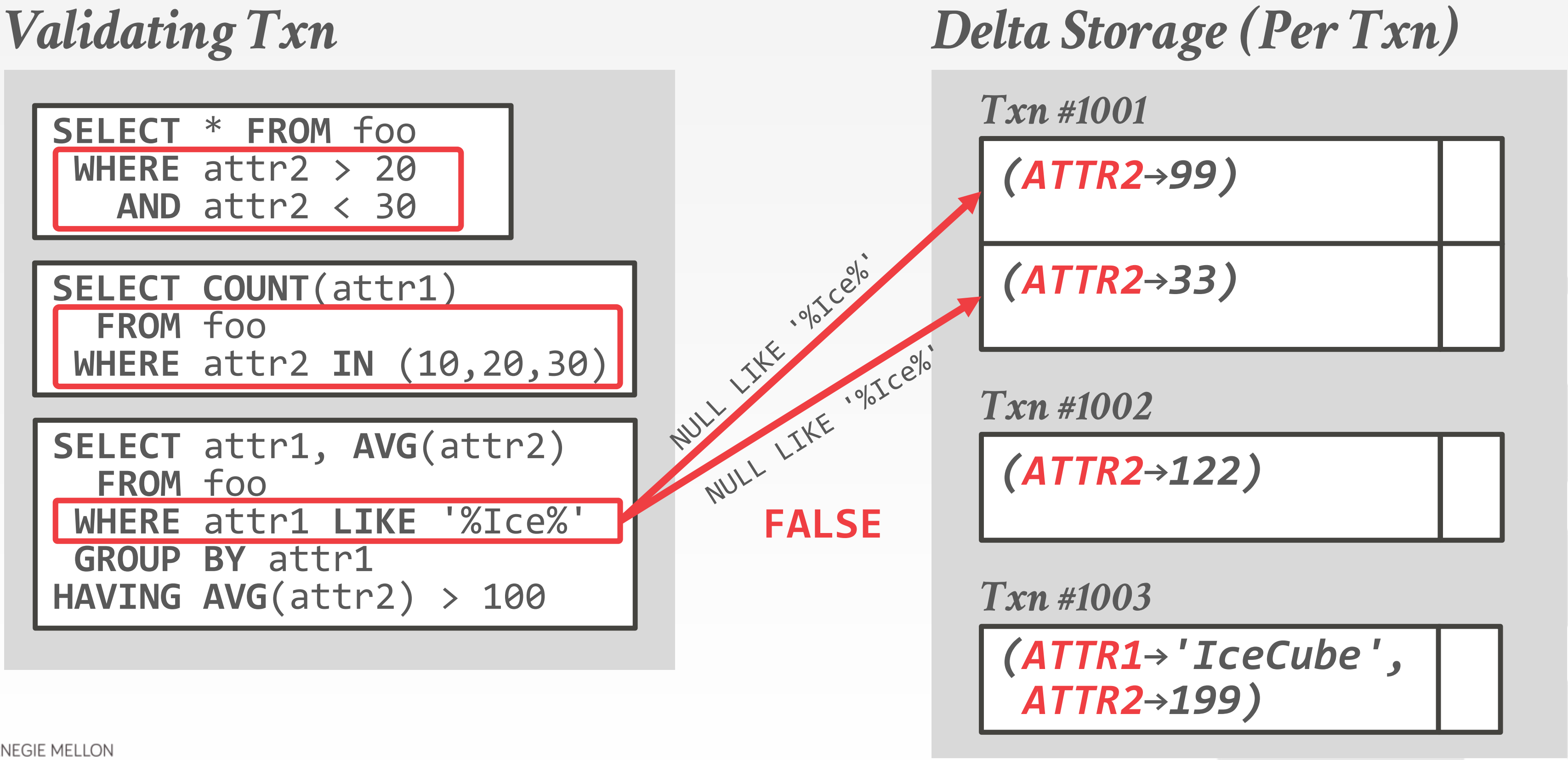
Txn #1002

(ATTR2→122)

Txn #1003

(ATTR1→'IceCube',
 ATTR2→199)

HYPER: PRECISION LOCKING



Validating Txn

```
SELECT * FROM foo
 WHERE attr2 > 20
   AND attr2 < 30
```

```
SELECT COUNT(attr1)
  FROM foo
 WHERE attr2 IN (10,20,30)
```

```
SELECT attr1, AVG(attr2)
  FROM foo
 WHERE attr1 LIKE '%Ice%'
 GROUP BY attr1
HAVING AVG(attr2) > 100
```

Delta Storage (Per Txn)

Txn #1001

(ATTR2→99)

(ATTR2→33)

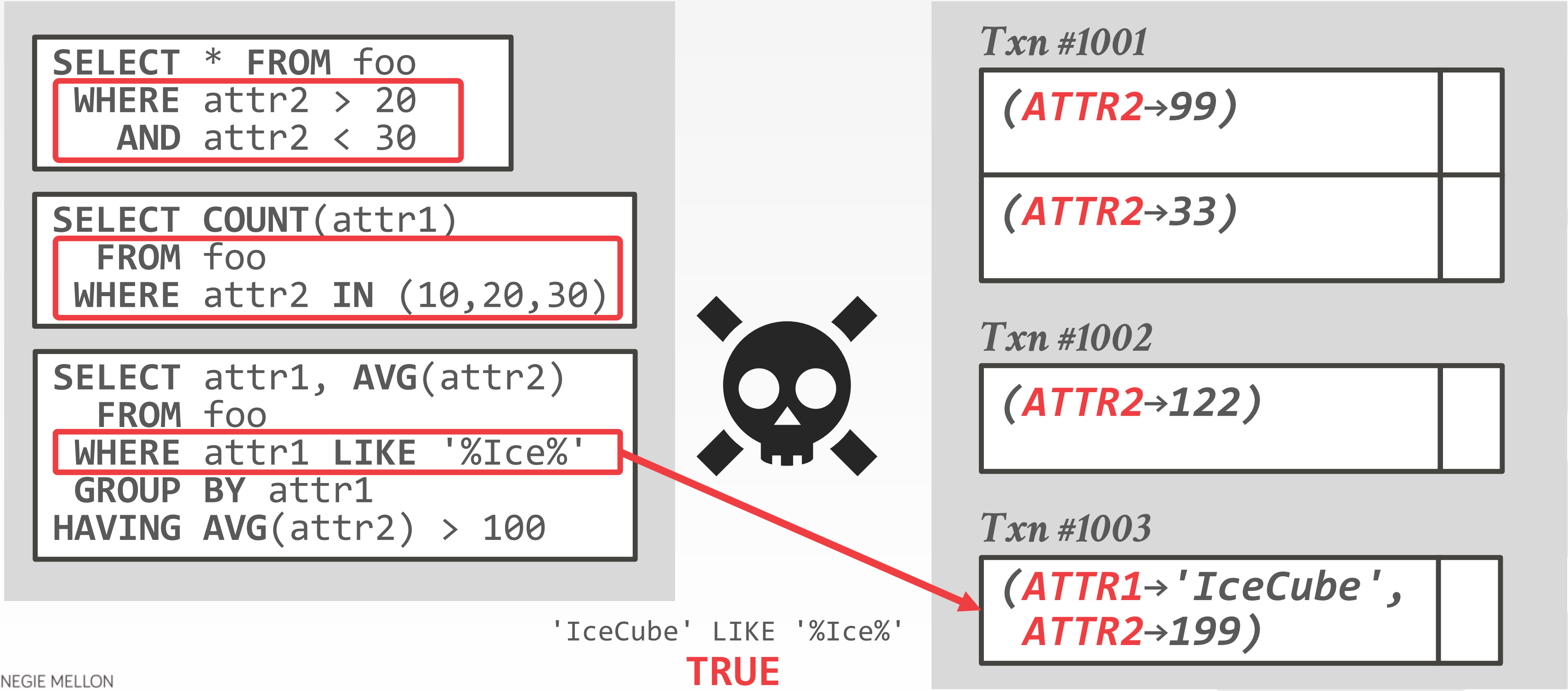
Txn #1002

(ATTR2→122)

Txn #1003

(ATTR1→'IceCube',
ATTR2→199)

HYPER: PRECISION LOCKING



Validating Txn

```
SELECT * FROM foo
 WHERE attr2 > 20
   AND attr2 < 30
```

```
SELECT COUNT(attr1)
  FROM foo
 WHERE attr2 IN (10,20,30)
```

```
SELECT attr1, AVG(attr2)
  FROM foo
 WHERE attr1 LIKE '%Ice%'
 GROUP BY attr1
HAVING AVG(attr2) > 100
```

Delta Storage (Per Txn)

Txn #1001

(ATTR2→99)

(ATTR2→33)

Txn #1002

(ATTR2→122)

Txn #1003

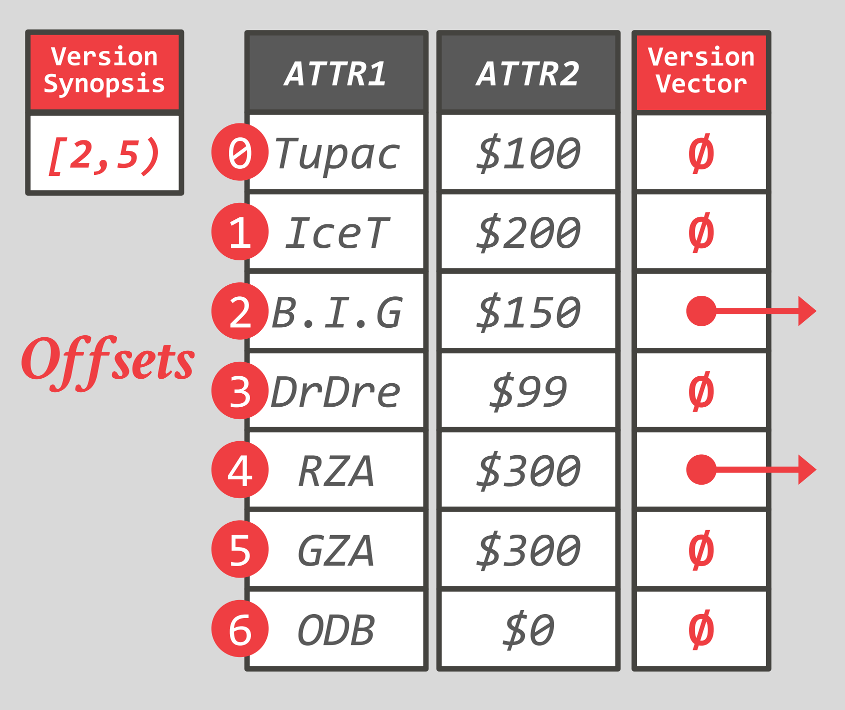
(ATTR1→'IceCube',
ATTR2→199)

`'IceCube' LIKE '%Ice%'`

TRUE

HYPER: VERSION SYNOPSES

Main Data Table



Version Synopsis
[2,5)

Offsets

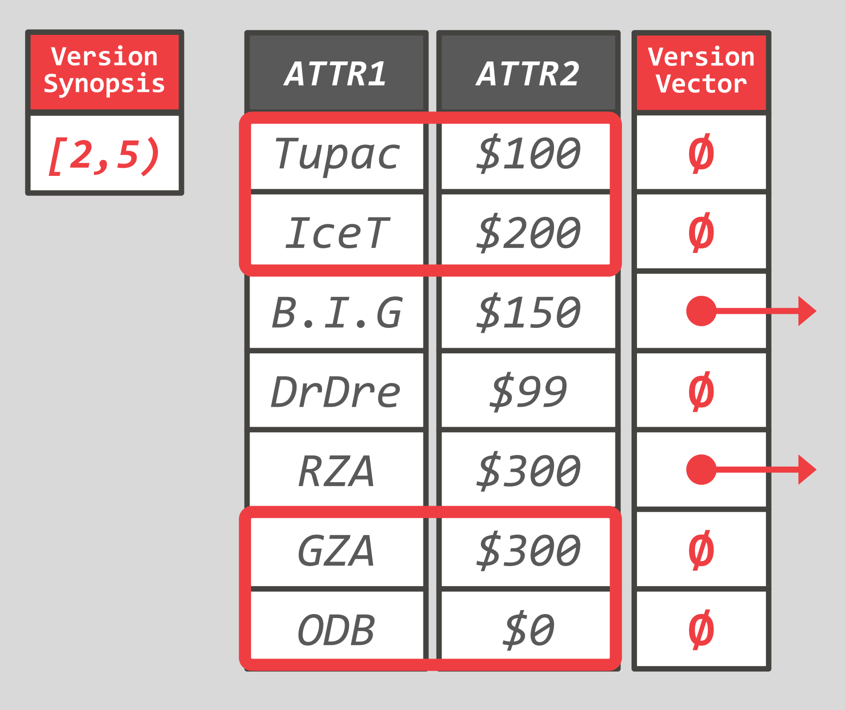
	ATTR1	ATTR2	Version Vector
0	Tupac	$100	Ø
1	IceT	$200	Ø
2	B.I.G	$150	●→
3	DrDre	$99	Ø
4	RZA	$300	●→
5	GZA	$300	Ø
6	ODB	$0	Ø

Store a separate column that tracks the position of the first and last versioned tuple in a block of tuples.

When scanning tuples, the DBMS can check for strides of tuples without older versions and execute more efficiently.

HYPER: VERSION SYNOPSES

Main Data Table



Version Synopsis
[2,5)

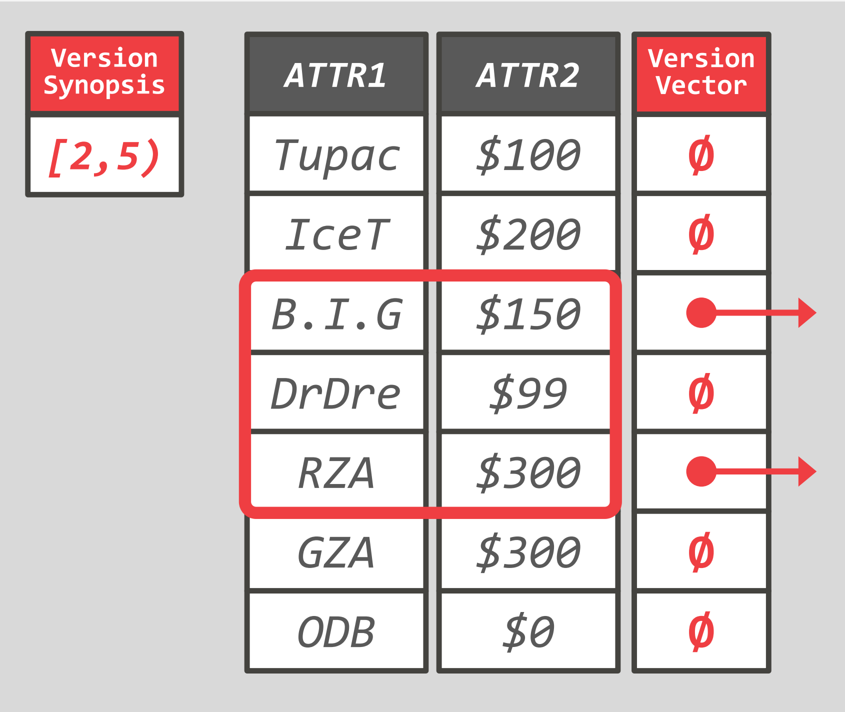
ATTR1	ATTR2	Version Vector
Tupac	$100	Ø
IceT	$200	Ø
B.I.G	$150	●
DrDre	$99	Ø
RZA	$300	●
GZA	$300	Ø
ODB	$0	Ø

Store a separate column that tracks the position of the first and last versioned tuple in a block of tuples.

When scanning tuples, the DBMS can check for strides of tuples without older versions and execute more efficiently.

HYPER: VERSION SYNOPSES

Main Data Table



Version Synopsis
[2,5)

ATTR1	ATTR2	Version Vector
Tupac	$100	∅
IceT	$200	∅
B.I.G	$150	●
DrDre	$99	∅
RZA	$300	●
GZA	$300	∅
ODB	$0	∅

Store a separate column that tracks the position of the first and last versioned tuple in a block of tuples.

When scanning tuples, the DBMS can check for strides of tuples without older versions and execute more efficiently.

SAP HANA

In-memory HTAP DBMS with time-travel version storage (**N2O**).

- → Supports both optimistic and pessimistic MVCC.
- → Latest versions are stored in time-travel space.
- → Hybrid storage layout (row + columnar).

Based on P*TIME, TREX, and MaxDB.

First released in 2012.

EFFICIENT TRANSACTION PROCESSING IN SAP HANA DATABASE: THE END OF A COLUMN-STORE MYTH
SIGMOD 2012

SAP HANA: VERSION STORAGE

Store the oldest version in the main data table.

Each tuple maintains a flag to denote whether there exists newer versions in the version space.

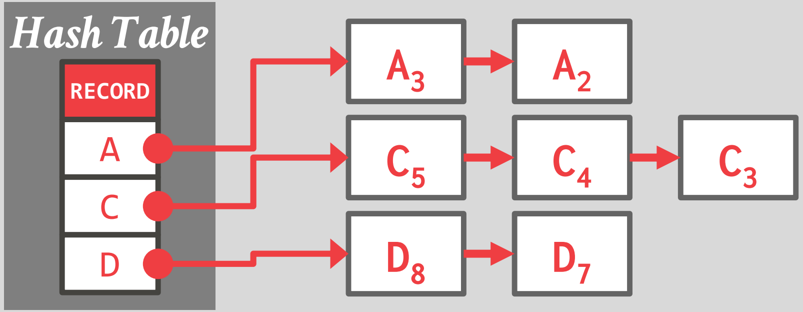
Maintain a separate hash table that maps record identifiers to the head of version chain.

SAP HANA: VERSION STORAGE

Main Data Table

RID	VERS?	VERSION	DATA
A	*True*	A_1	–
B	*False*	B_3	–
C	*True*	C_2	–
D	*True*	D_6	–

Version Storage



SAP HANA: TRANSACTIONS

Instead of embedding meta-data about the txn that created a version with the data, store a pointer to a context object.

→ Reads are slower because you have to follow pointers.
→ Large updates are faster because it's a single write to update the status of all tuples.

Store meta-data about whether a txn has committed in a separate object as well.

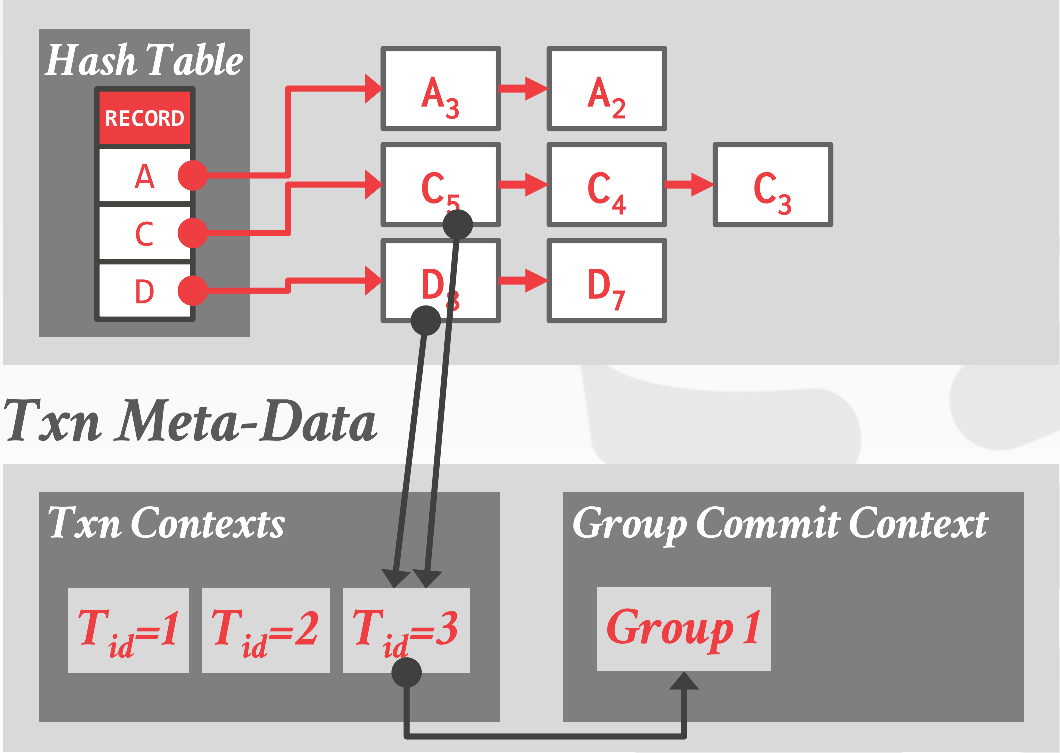
SAP HANA: VERSION STORAGE

Main Data Table

RID	VERS?	VERSION	DATA
A	*True*	A_1	-
B	*False*	B_3	-
C	*True*	C_2	-
D	*True*	D_6	-

Version Storage

Txn Meta-Data



Thread #1

$T_{id} = 3$

WRITE(C)

WRITE(D)

MVCC LIMITATIONS

Computation & Storage Overhead

→ Most MVCC schemes use indirection to search a tuple's version chain. This increases CPU cache misses.

→ Also requires frequent garbage collection to minimize the number versions that a thread has to evaluate.

Shared Memory Writes

→ Most MVCC schemes store versions in "global" memory in the heap without considering locality.

Timestamp Allocation

→ All threads access single shared counter.

Source: Hyeontaek Lim

OCC LIMITATIONS

Frequent Aborts

→ Txns will abort too quickly under high contention, causing high churn.

Extra Reads & Writes

→ Each txn has to copy tuples into their private workspace to ensure repeatable reads. It then has to check whether it read consistent data when it commits.

Index Contention

→ Txns install "virtual" index entries to ensure unique-key invariants.

Source: Hyeontaek Lim

CMU CICADA

In-memory OLTP engine based on optimistic MVCC with append-only storage (N2O).

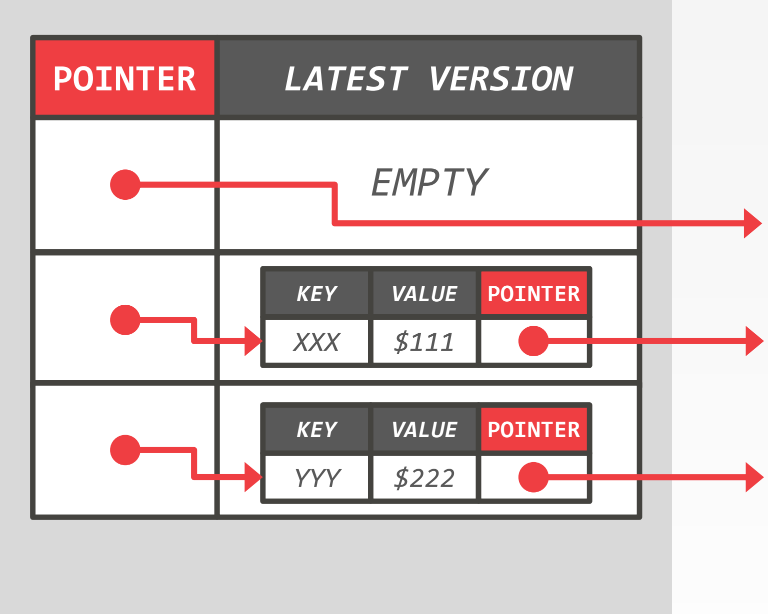
→ Best-effort Inlining
→ Loosely Synchronized Clocks
→ Contention-Aware Validation
→ Index Nodes Stored in Tables

Designed to be scalable for both low- and high-contention workloads.

CICADA: DEPENDABLY FAST MULTI-CORE IN-MEMORY TRANSACTIONS
SIGMOD 2017

CICADA: BEST-EFFORT INLINING

Record Meta-data



Record meta-data is stored in a fixed location.

Threads will attempt to inline read-mostly version within this meta-data to reduce version chain traversals.

CICADA: FAST VALIDATION

Contention-aware Validation
→ Validate access to recently modified records first.

Early Consistency Check
→ Pre-validate access set before making global writes.

Skip if all recent txns committed successfully.

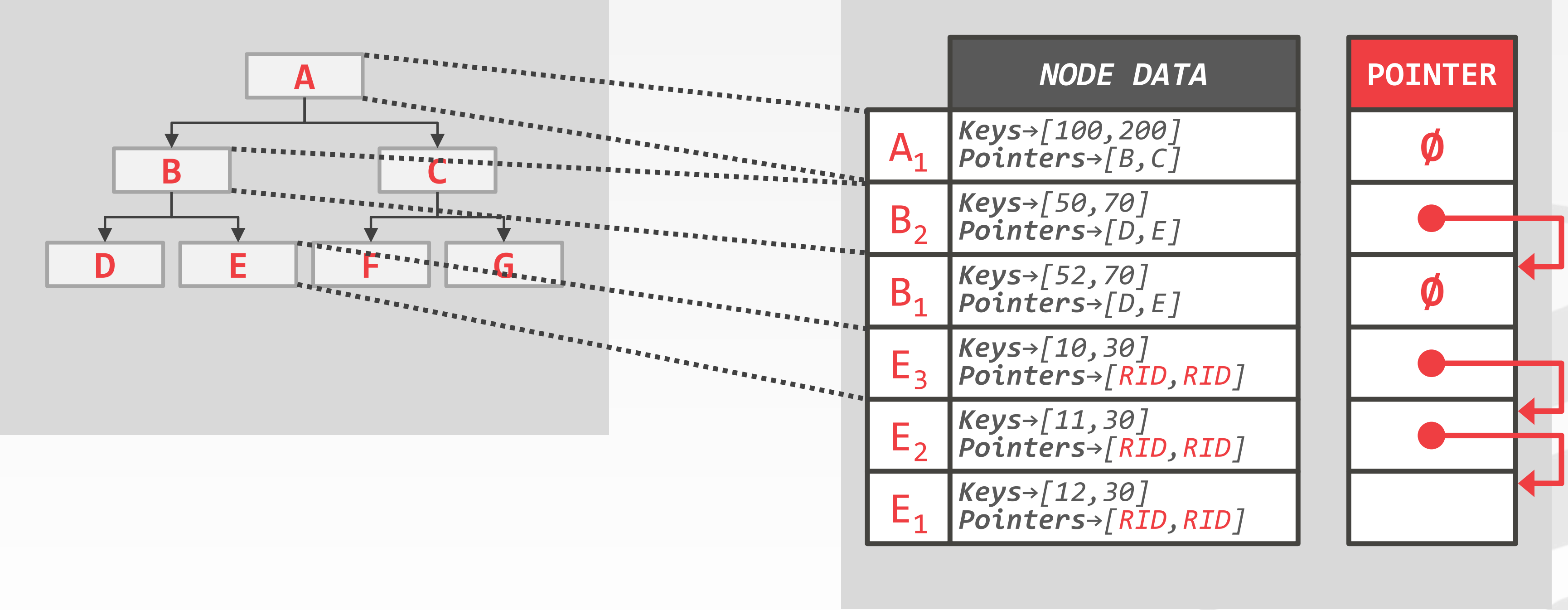
Incremental Version Search
→ Resume from last search location in version list.

Source: Hyeontaek Lim

CICADA: INDEX STORAGE

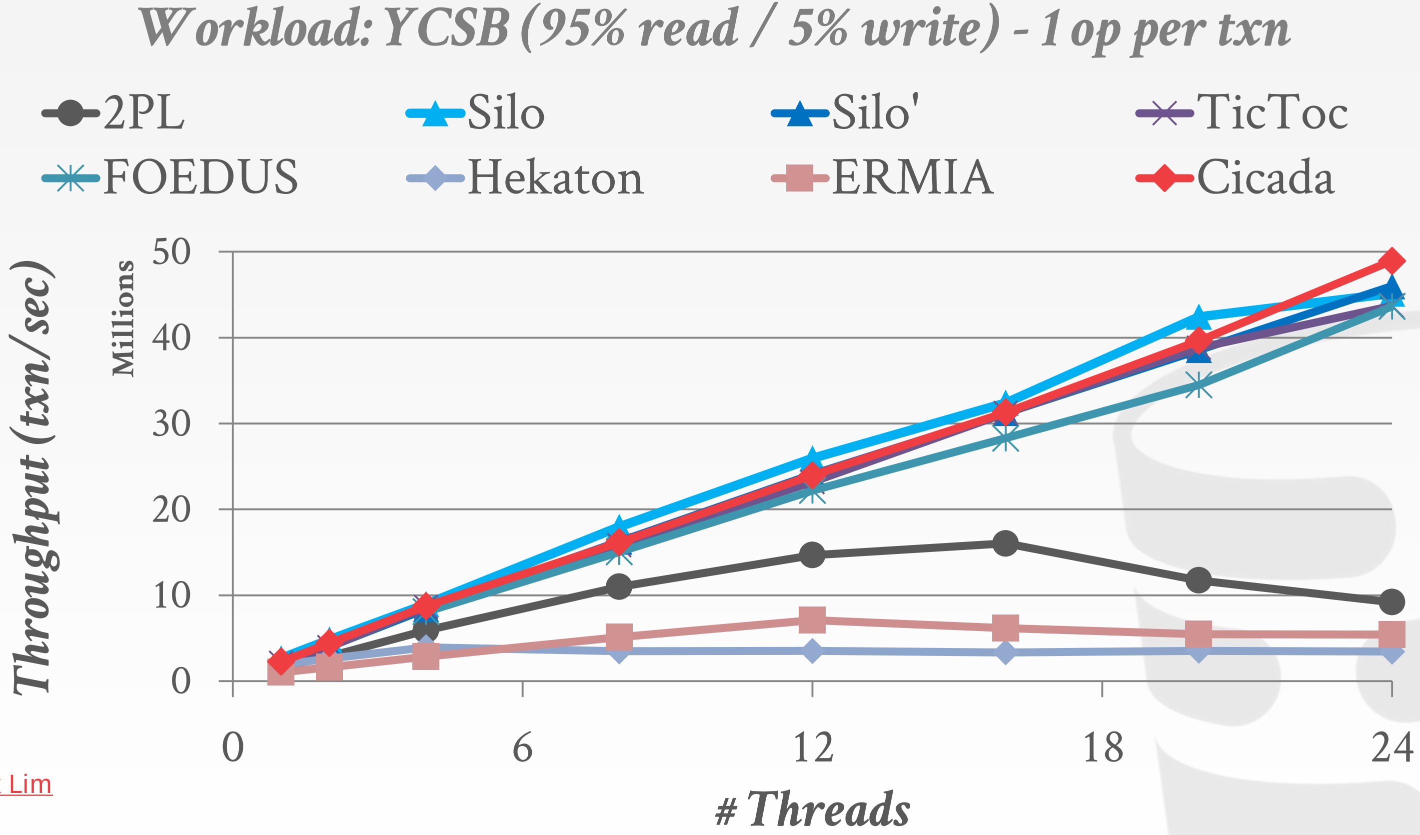
Index

Index Node Table



	NODE DATA	POINTER
A_1	*Keys→[100,200]* *Pointers→[B,C]*	Ø
B_2	*Keys→[50,70]* *Pointers→[D,E]*	●
B_1	*Keys→[52,70]* *Pointers→[D,E]*	Ø
E_3	*Keys→[10,30]* *Pointers→[RID,RID]*	●
E_2	*Keys→[11,30]* *Pointers→[RID,RID]*	●
E_1	*Keys→[12,30]* *Pointers→[RID,RID]*	

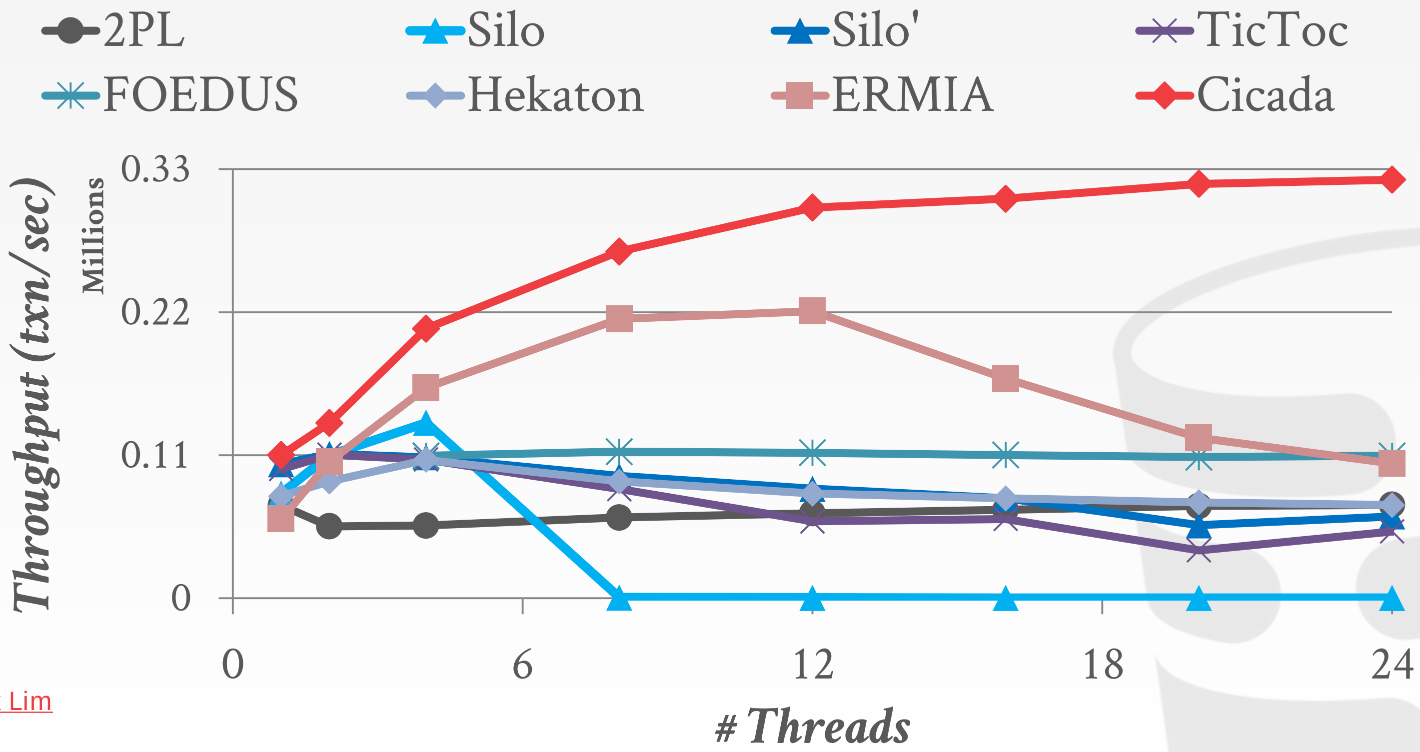
CICADA: LOW CONTENTION



Source: Hyeontaek Lim

CICADA: HIGH CONTENTION

Workload: TPC-C (1 Warehouse)



Source: Hyeontaek Lim

PARTING THOUGHTS

There are different ways to check for phantoms in MVCC. We will see more "traditional" ways next week.

NEXT CLASS

MVCC Garbage Collection