

Lecture #06

Carnegie Mellon University

# ADVANCED DATABASE SYSTEMS

Multi-Version  
Concurrency Control (Part II)

@Andy\_Pavlo // 15-721 // Spring 2018

# TODAY'S AGENDA

---

Microsoft Hekaton (SQL Server)

TUM HyPer

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) works well for some applications but terrible for others.

# HEKATON MVCC

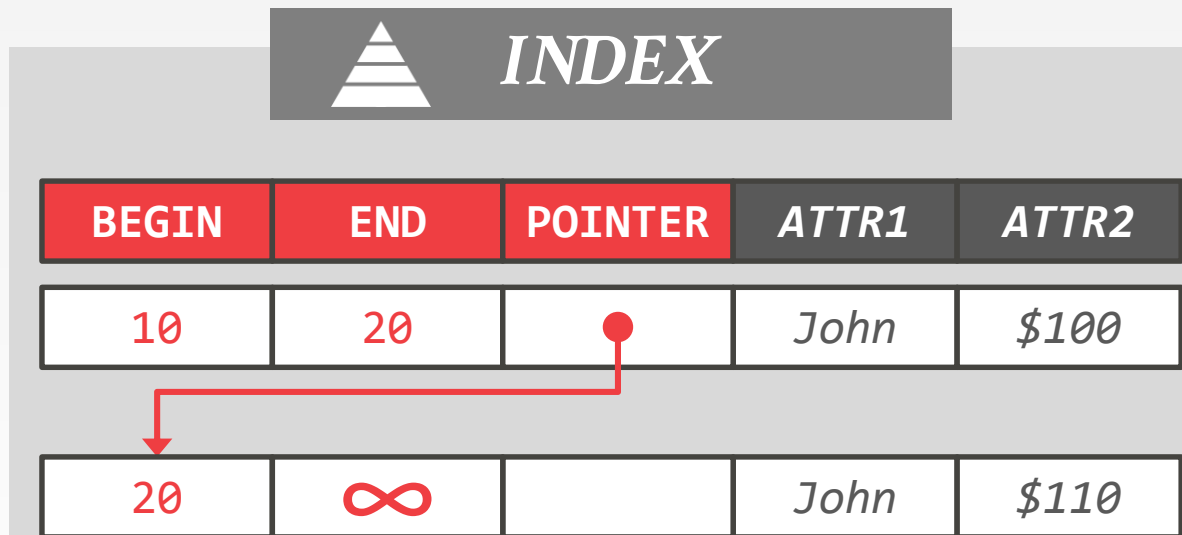
---

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

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

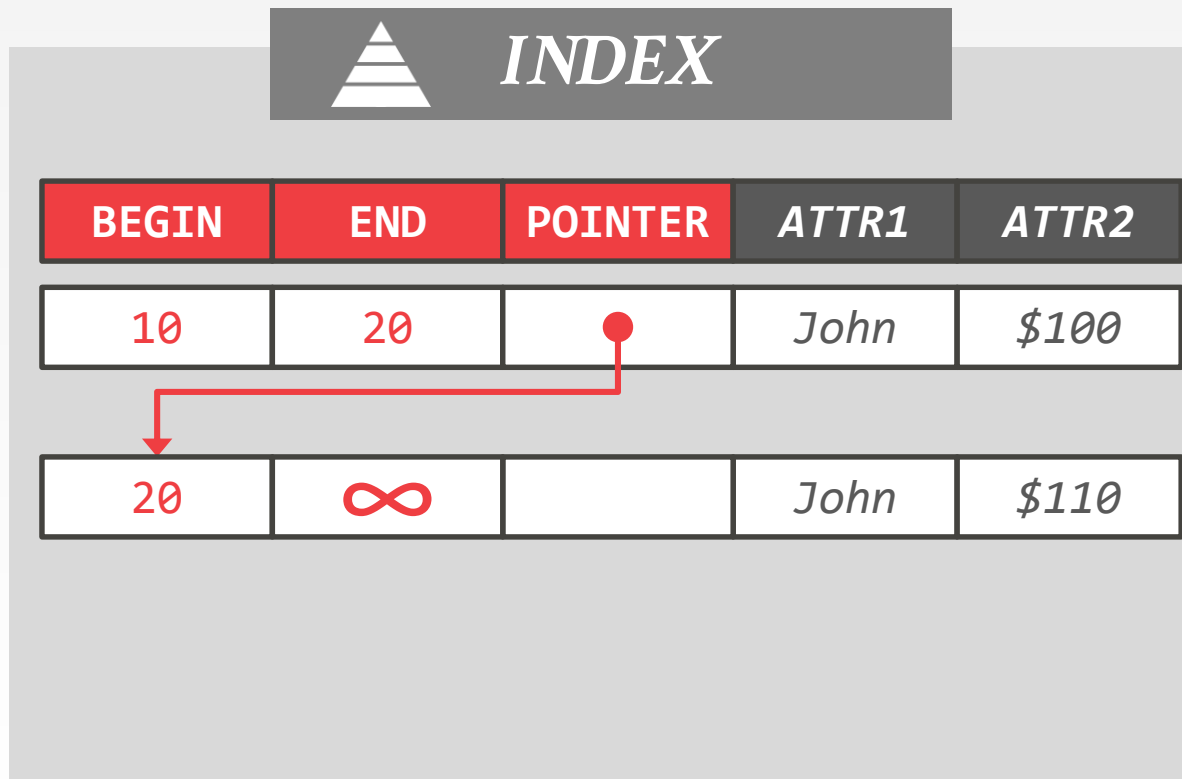
- **BEGIN**: The BeginTS of the active txn or the EndTS of the committed txn that created it.
- **END**: The BeginTS of the active txn that created the next version or infinity or the EndTS of the committed txn that created it.

# HEKATON: OPERATIONS



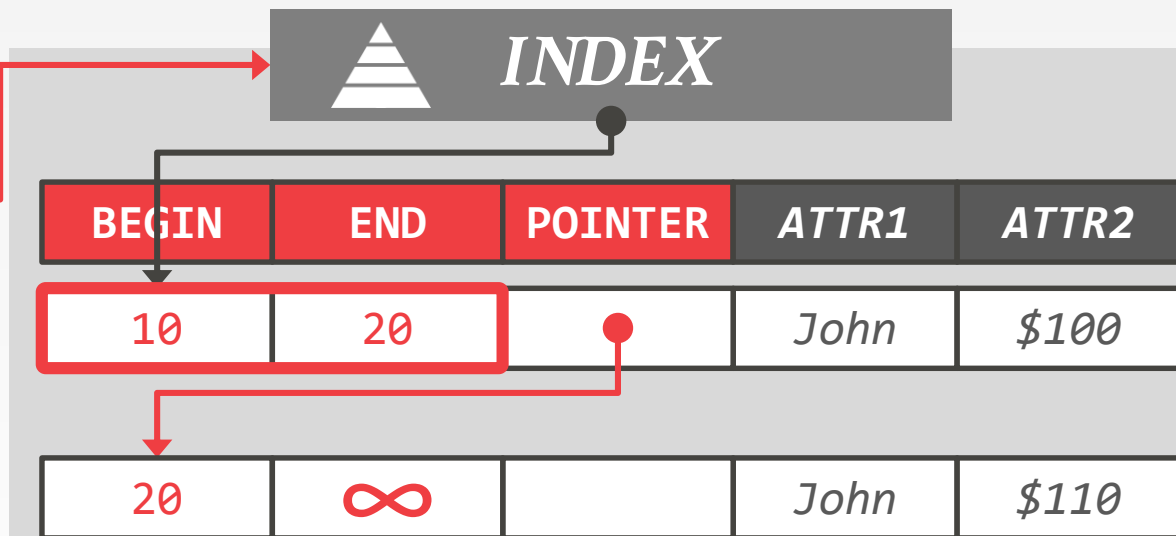
# HEKATON: OPERATIONS

BEGIN @ 25



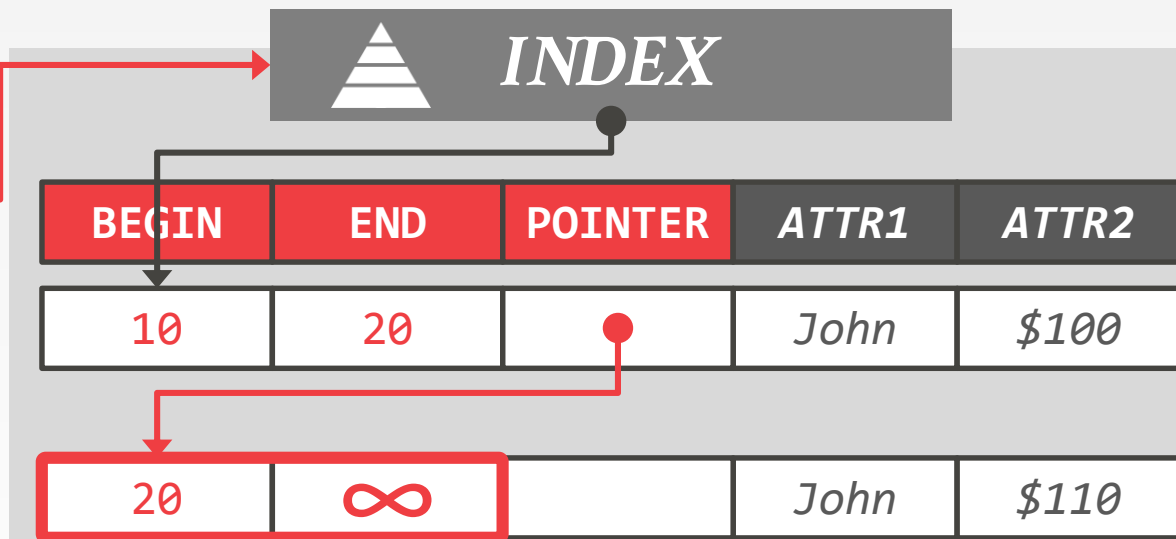
# HEKATON: OPERATIONS

BEGIN @ 25  
Read "John"



# HEKATON: OPERATIONS

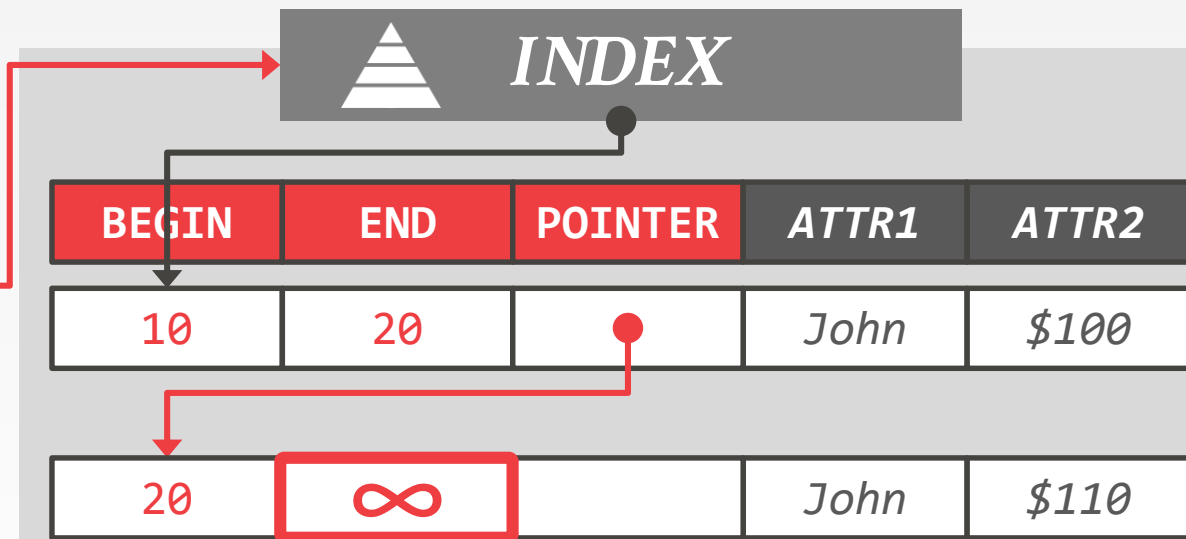
BEGIN @ 25  
Read "John"





# HEKATON: OPERATIONS

BEGIN @ 25  
 Read "John"  
 Update "John"

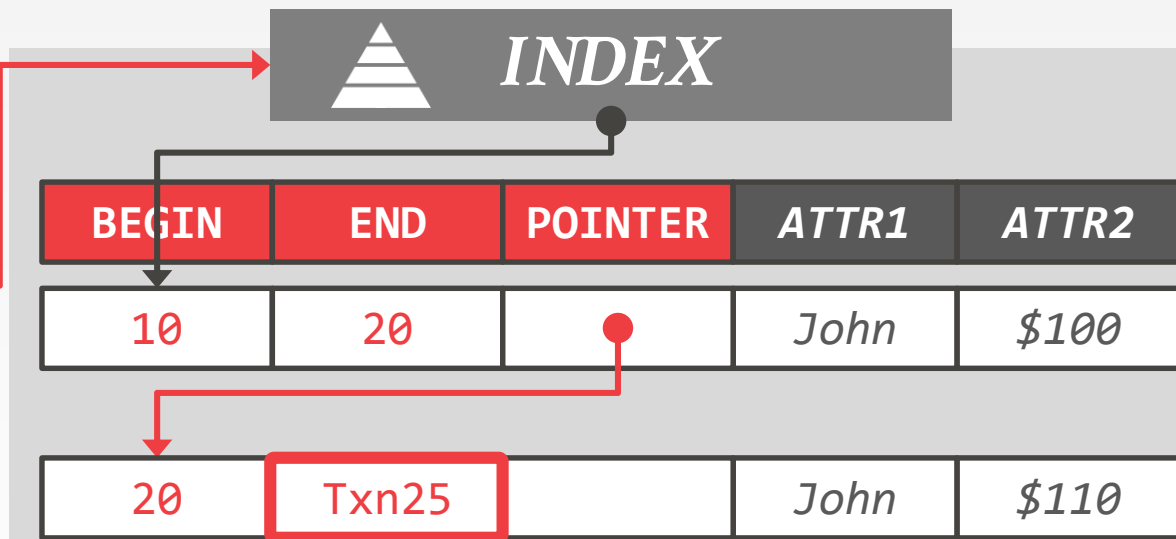


# HEKATON: OPERATIONS

**BEGIN @ 25**

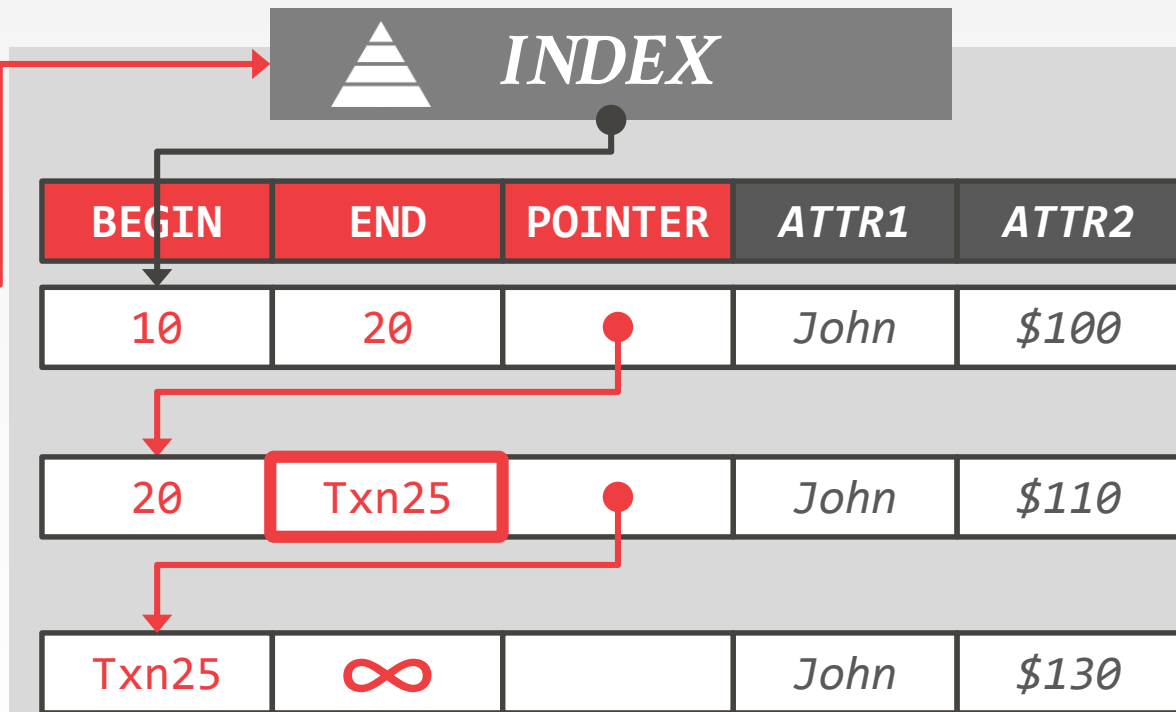
Read "John"

Update "John"



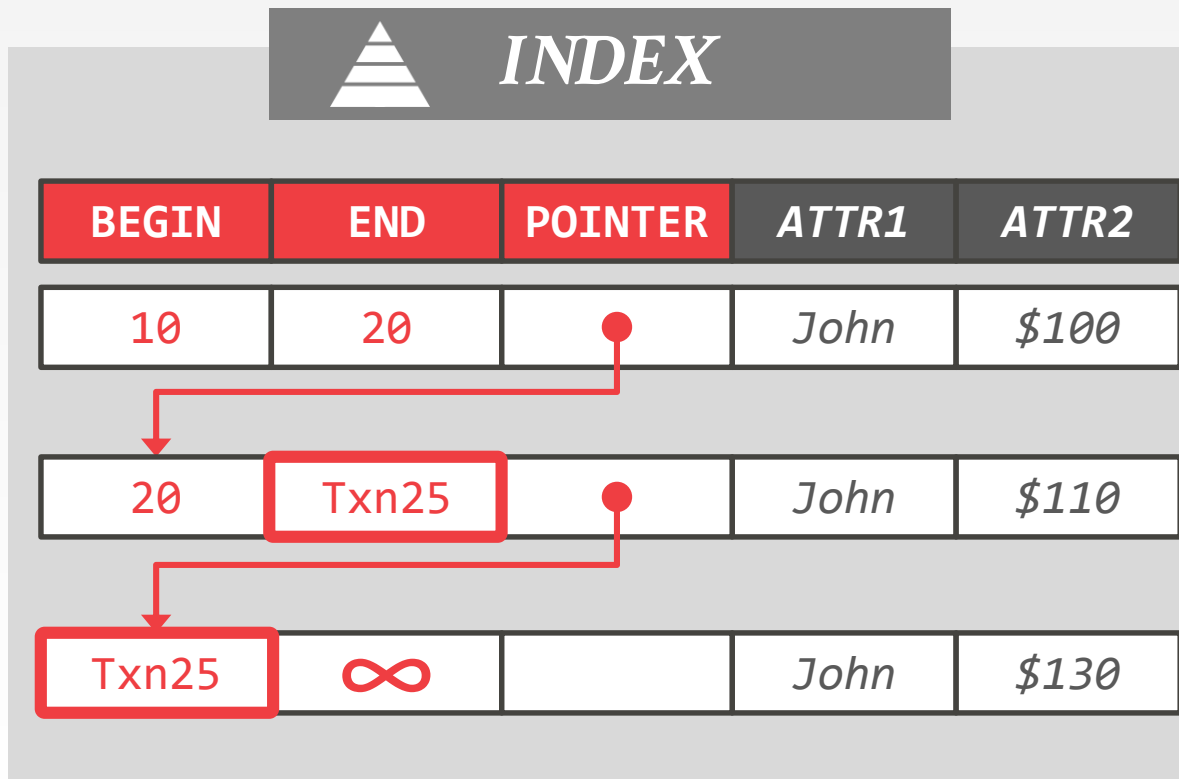
# HEKATON: OPERATIONS

BEGIN @ 25  
 Read "John"  
 Update "John"



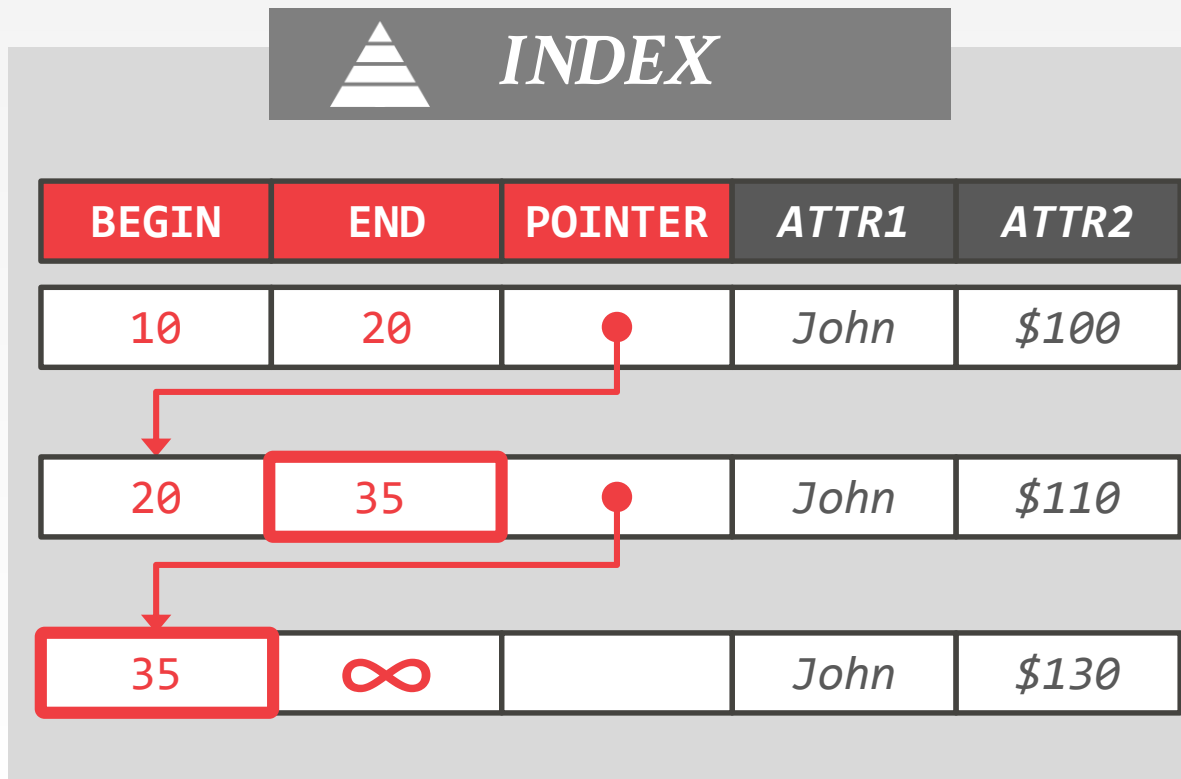
# HEKATON: OPERATIONS

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*



**REWIND**

*ATTR1*

*ATTR2*

*John*

*\$100*

20

35

*John*

*\$110*

35

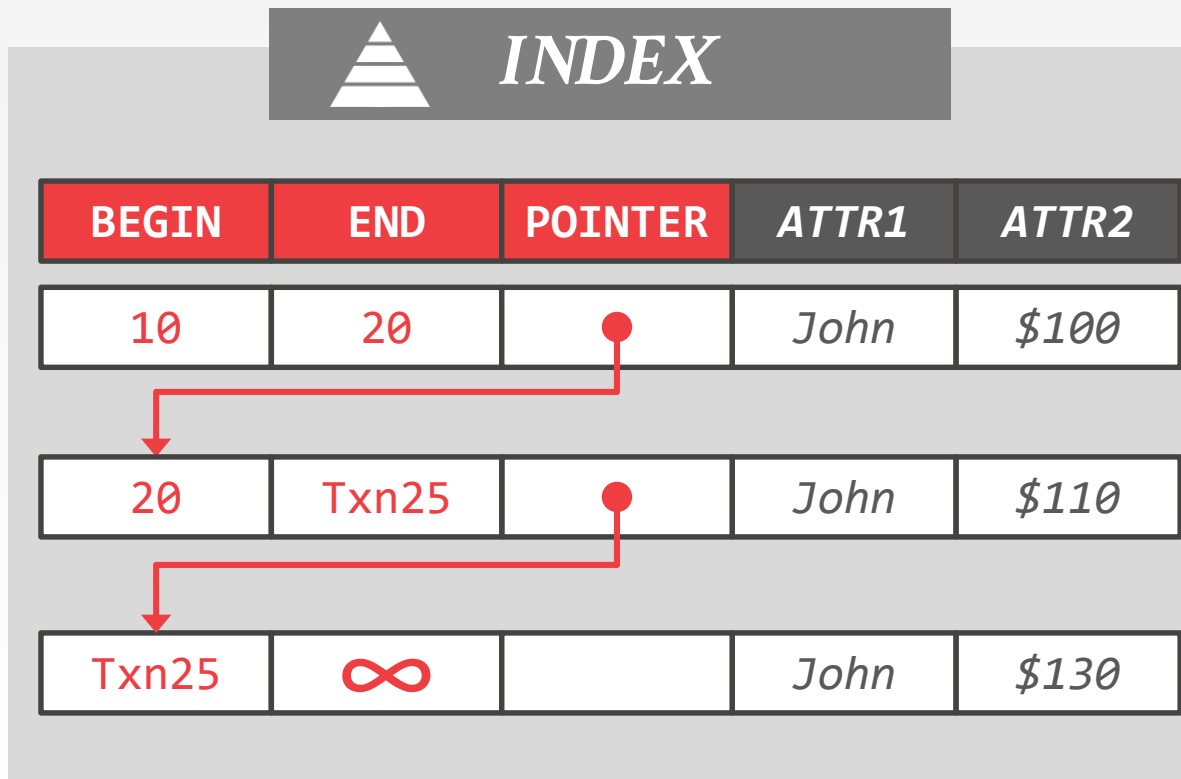
$\infty$

*John*

*\$130*

# HEKATON: OPERATIONS

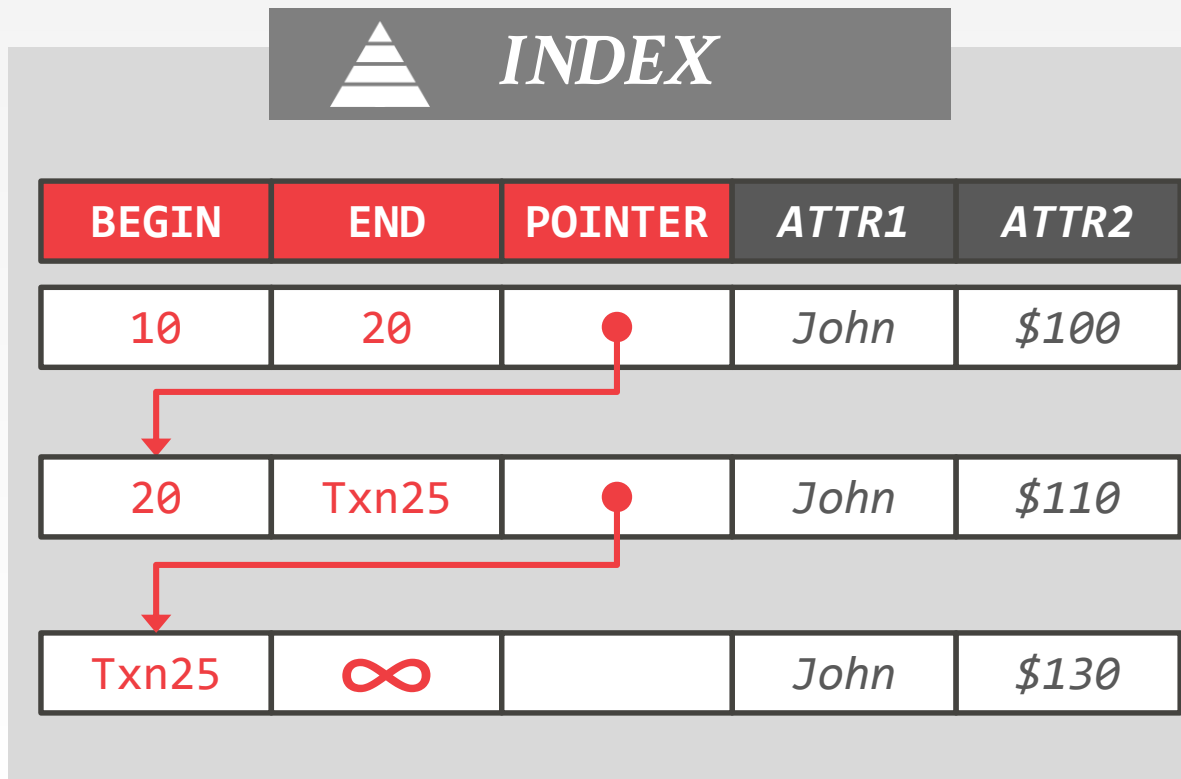
**BEGIN @ 25**  
 Read "John"  
 Update "John"



# HEKATON: OPERATIONS

**BEGIN @ 25**  
 Read "John"  
 Update "John"

**BEGIN @ 30**

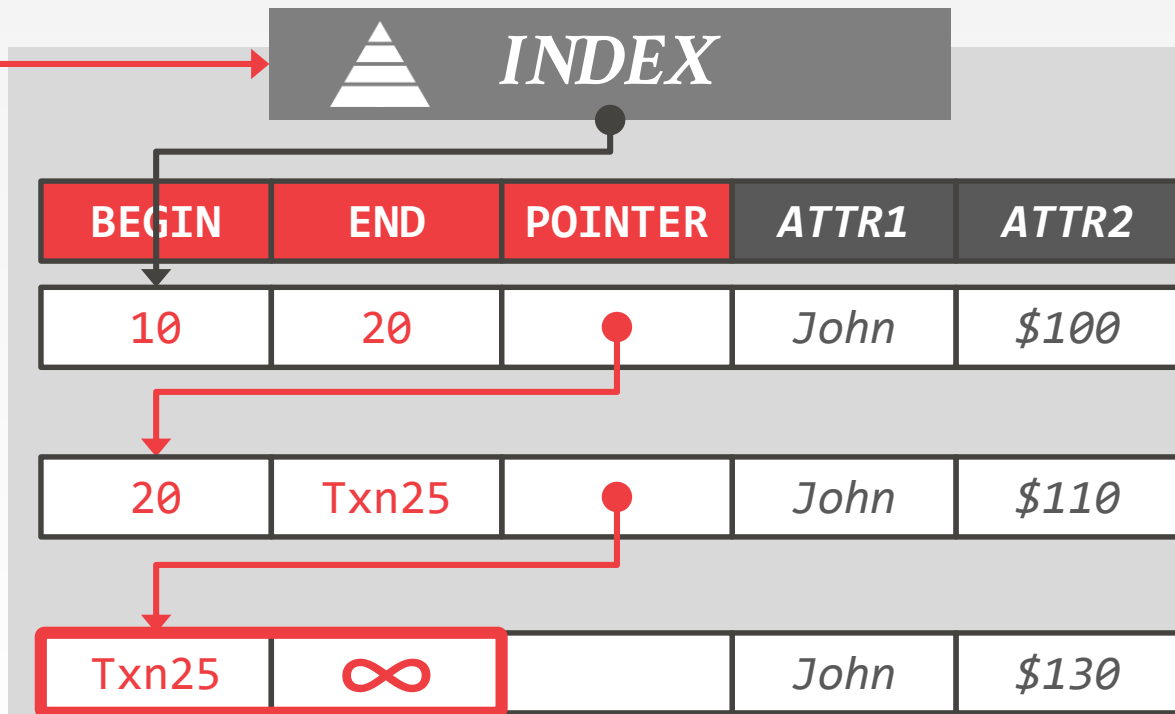




# HEKATON: OPERATIONS

**BEGIN @ 25**  
 Read "John"  
 Update "John"

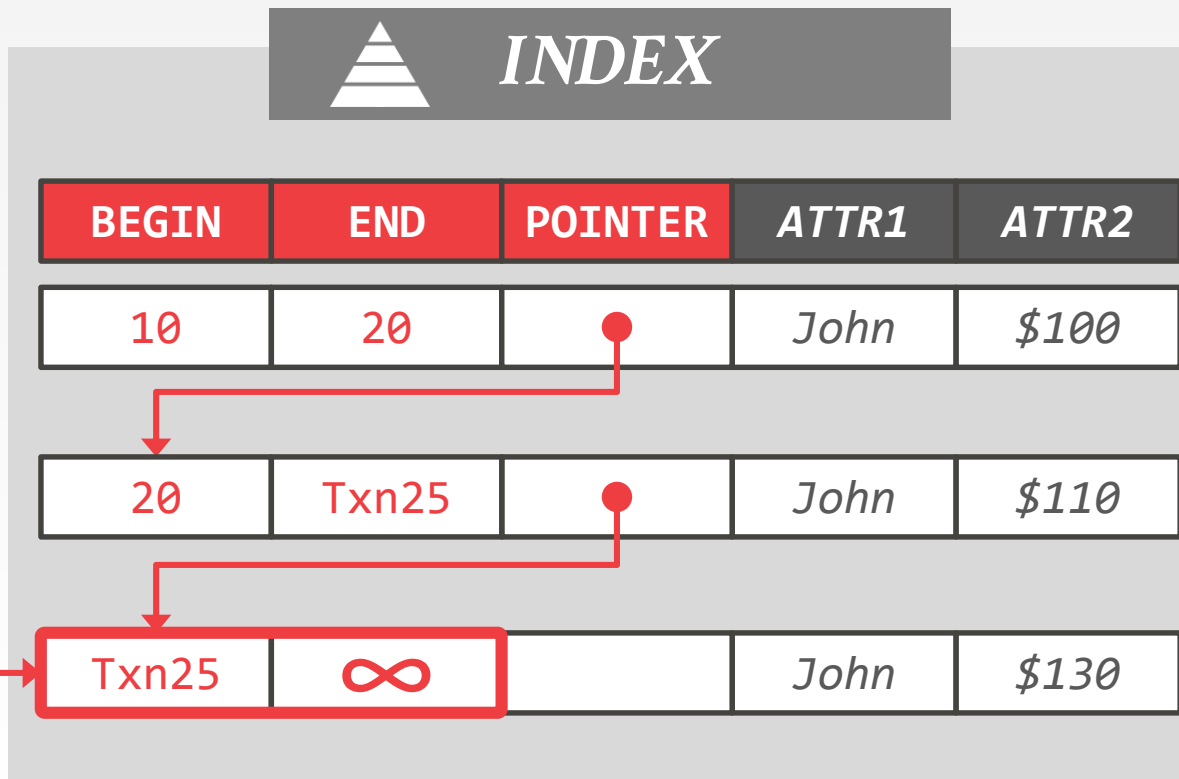
**BEGIN @ 30**  
 Read "John"



# 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	$\infty$		John	\$130

# 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

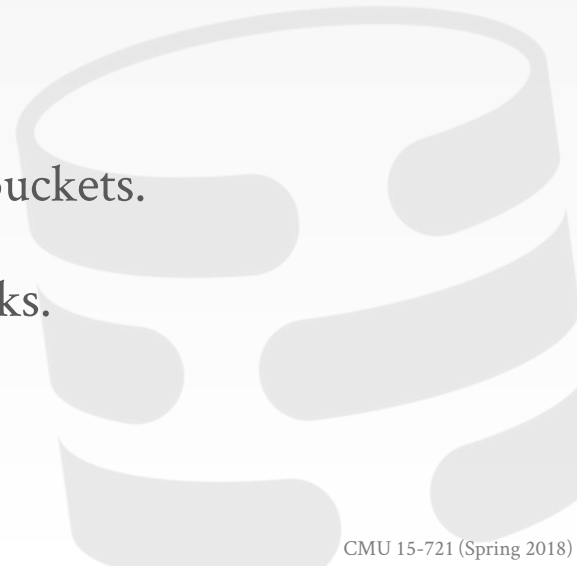
---

## **Optimistic Txns:**

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

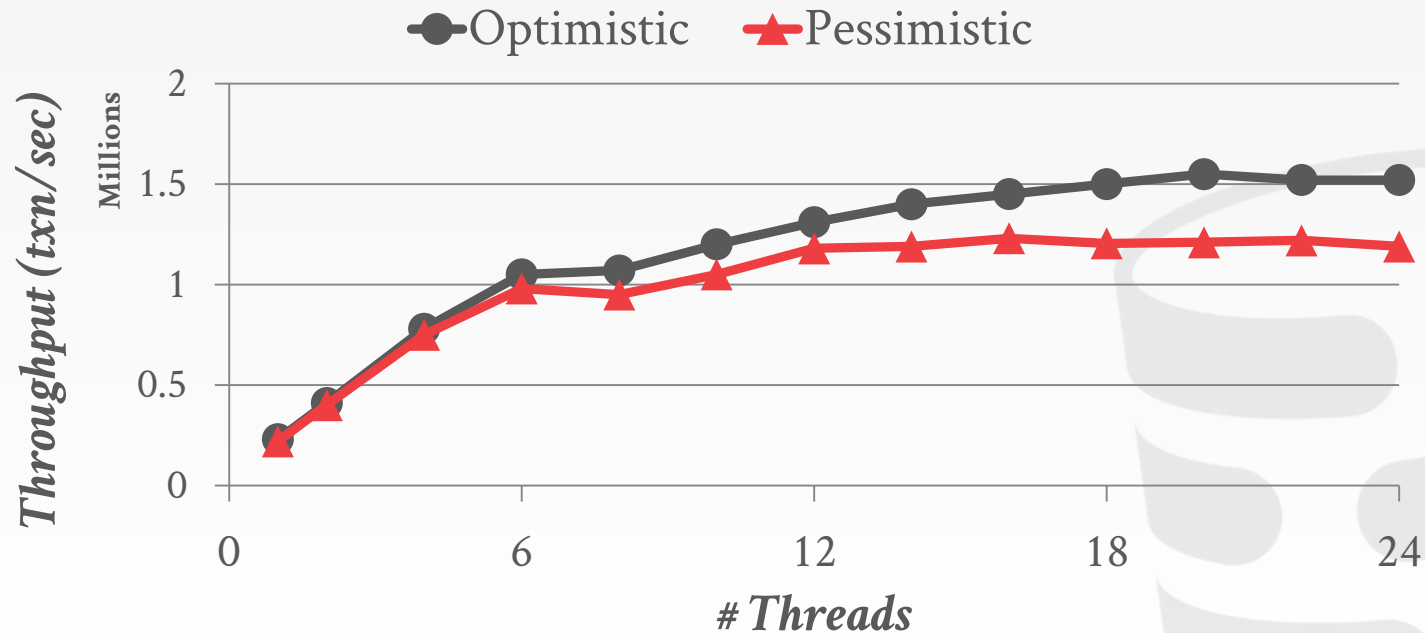
## **Pessimistic Txns:**

- 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





# 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)

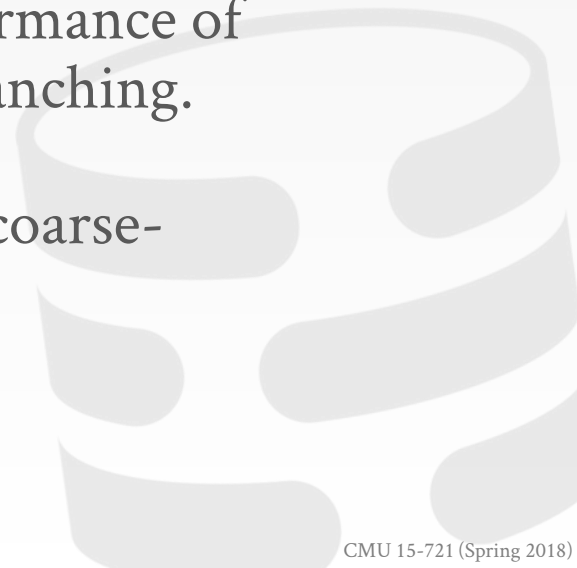
# OBSERVATIONS

---

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.

Designed for HTAP workloads.

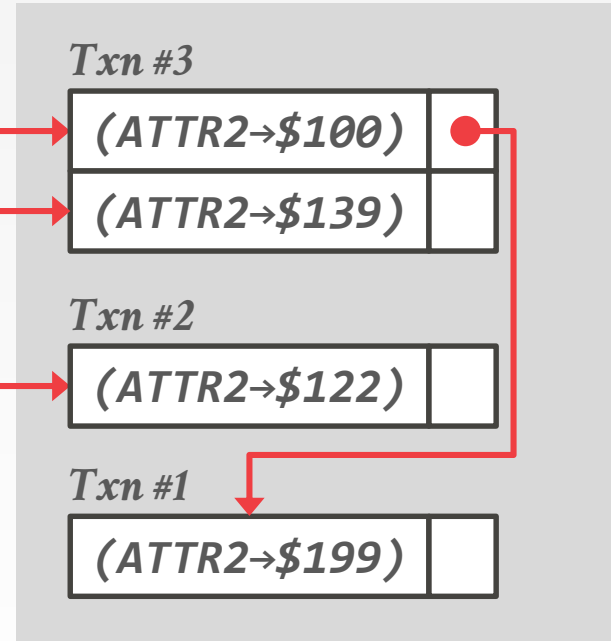


# HYPER: STORAGE ARCHITECTURE

## Main Data Table

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

## Delta Storage (Per Txn)



# 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
```

99 > 20 AND 99 < 30

**FALSE**

33 > 20 AND 33 < 30

## 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
```

99 IN (10,20,30)

FALSE

33 IN (10,20,30)

## 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
```

NULL LIKE '%Ice%'

NULL LIKE '%Ice%'

**FALSE**

## 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
```



'Ice Cube' LIKE '%Ice%'  
**TRUE**

## Delta Storage (Per Txn)

Txn #1001

(ATTR2→99)

(ATTR2→33)

Txn #1002

(ATTR2→122)

Txn #1003

(ATTR1→'IceCube',  
ATTR2→199)

# HYPER: VERSION SYNOPSSES

## Main Data Table

Version Synopsis	ATTR1	ATTR2	Version Vector
[2, 5)	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.

# HYPER: VERSION SYNOPSSES

## Main Data Table

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

*Offsets*

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 SYNOPSSES

## Main Data Table

Version Synopsis	ATTR1	ATTR2	Version Vector
[2, 5)	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.

# HYPER: VERSION SYNOPSSES

## Main Data Table

Version Synopsis	ATTR1	ATTR2	Version Vector
[2, 5)	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.

# 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

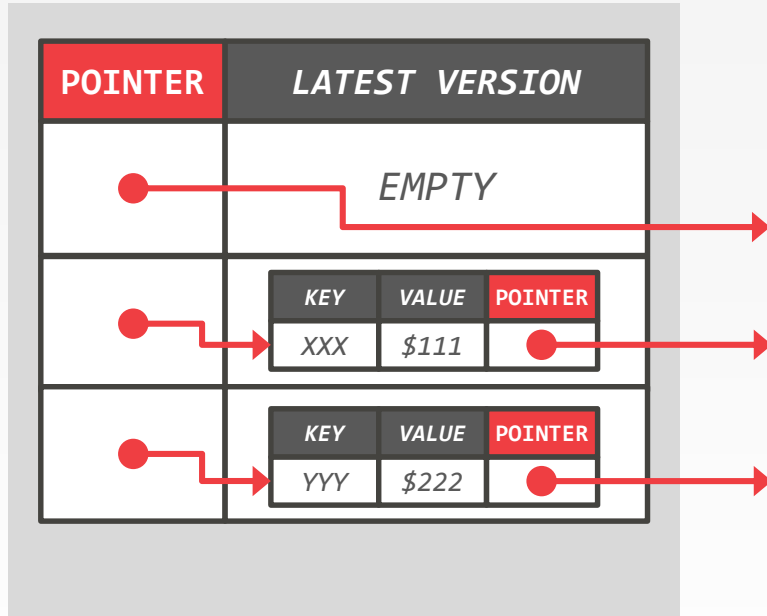
Designed to be scalable for both low- and high-contention workloads.



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.

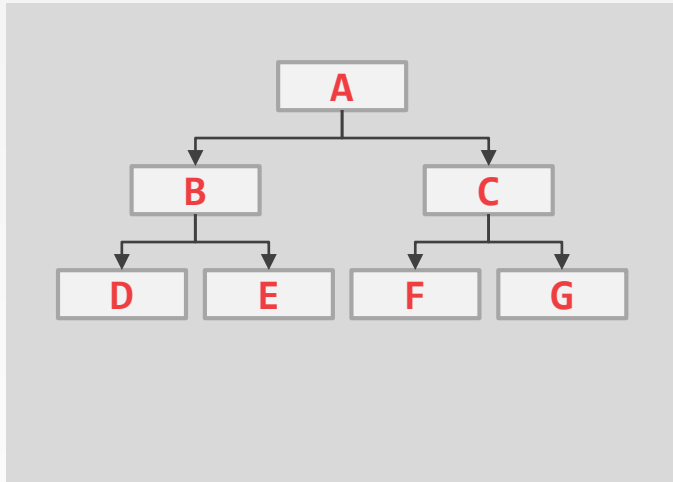
*Skip if all  
recent txns  
committed  
successfully.*

## Incremental Version Search

→ Resume from last search location in version list.

# CICADA: INDEX STORAGE

## Index



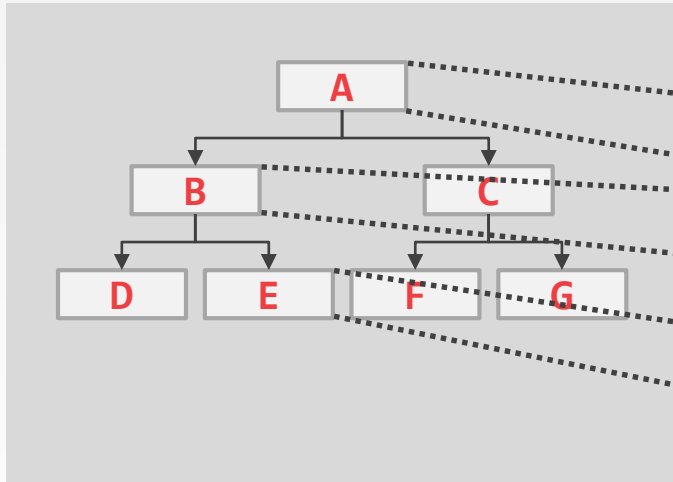
## Index Node Table

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

Diagram illustrating the Index Node Table structure. The table contains six rows representing index nodes. The first row is the root node  $A_1$ . The second and third rows are leaf nodes  $B_2$  and  $B_1$ . The last three rows are leaf nodes  $E_3$ ,  $E_2$ , and  $E_1$ . Red arrows indicate pointers from the pointer column to the corresponding nodes in the index structure.

# CICADA: INDEX STORAGE

## Index



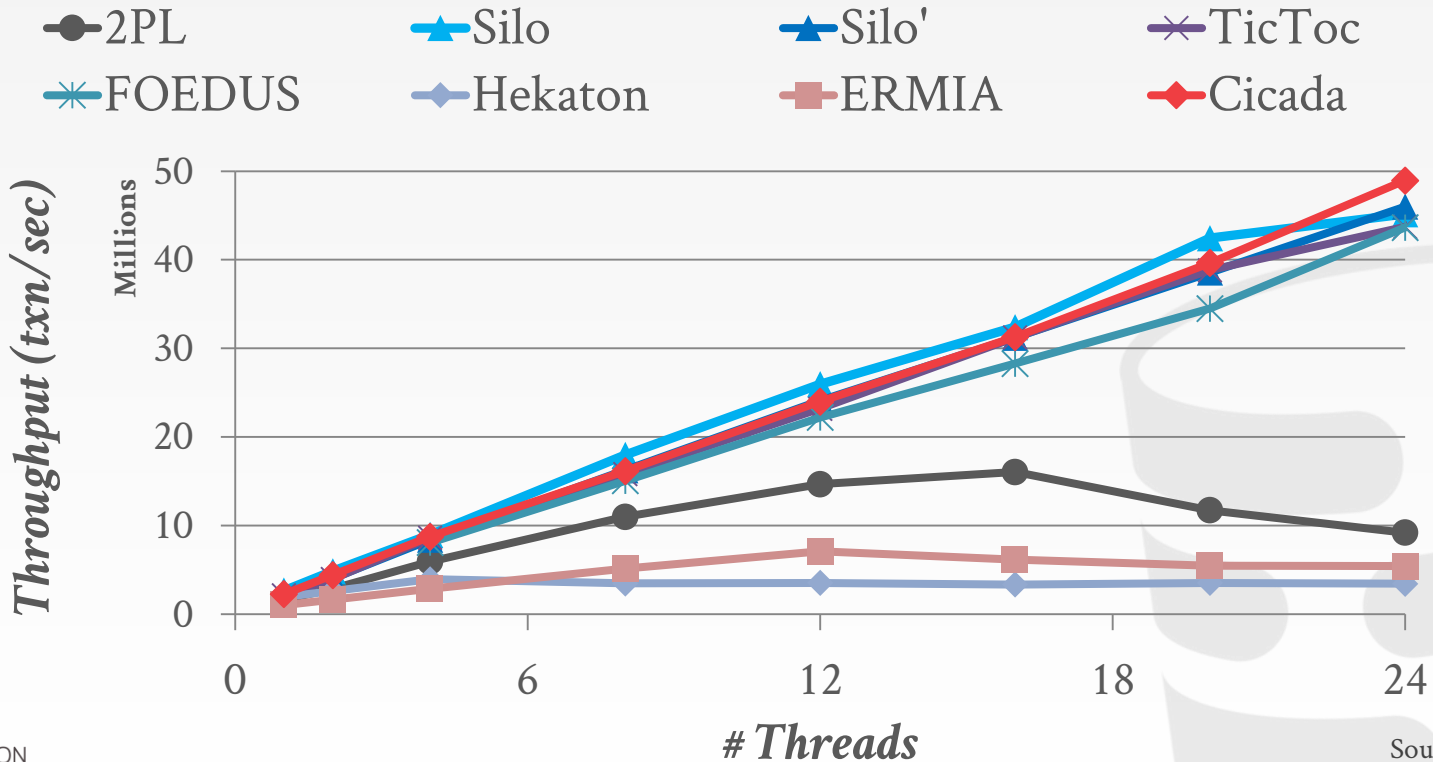
## Index Node Table

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

Diagram illustrating the Index Node Table structure. The table contains entries for nodes  $A_1$ ,  $B_2$ ,  $B_1$ ,  $E_3$ ,  $E_2$ , and  $E_1$ . Each entry lists its keys and pointers. The pointers column shows the pointer values, with red dots and arrows indicating the mapping from the Index structure to the table entries.

# CICADA: LOW CONTENTION

Workload: YCSB (95% read / 5% write) - 1 op per txn

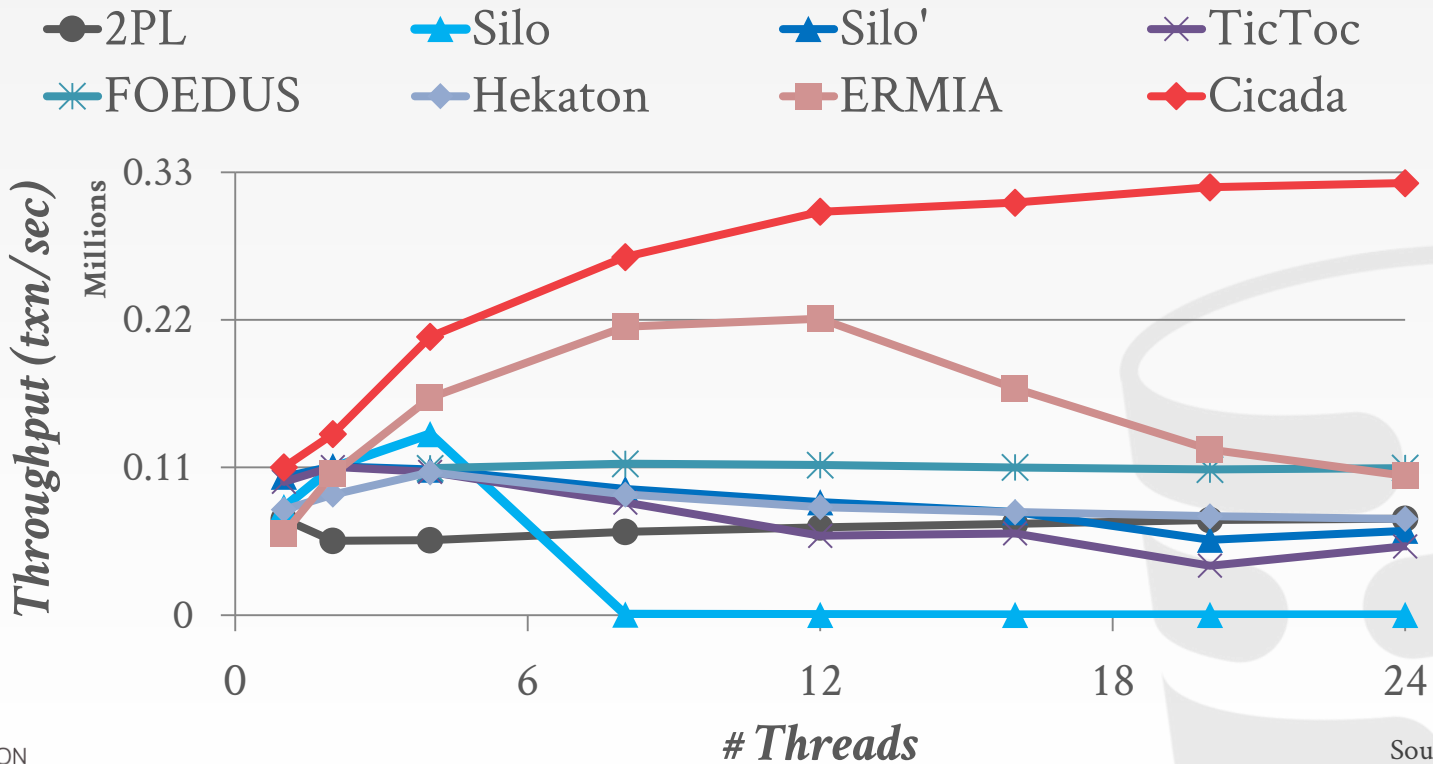


Source: [Hyeontaek Lim](#)

CMU 15-721 (Spring 2018)

# CICADA: HIGH CONTENTION

Workload: TPC-C (1 Warehouse)

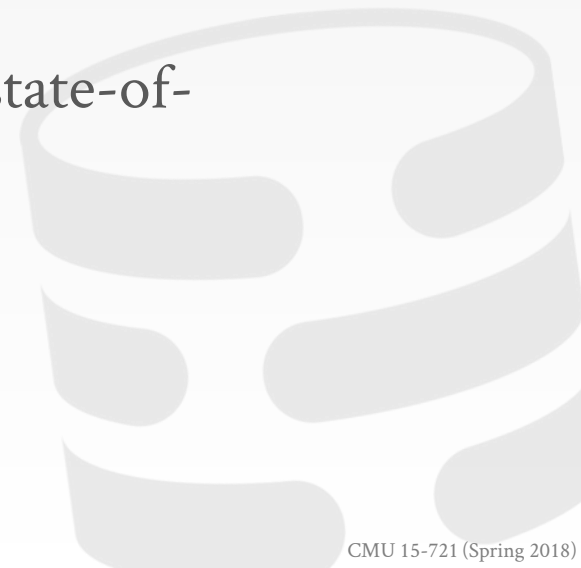


## PARTING THOUGHTS

---

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

Andy considers HyPer and Cicada to be state-of-the-art as of January 2018.



# NEXT CLASS

---

## Index Locking + Latching

