Carnegie Mellon University

Lecture #06

# 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).

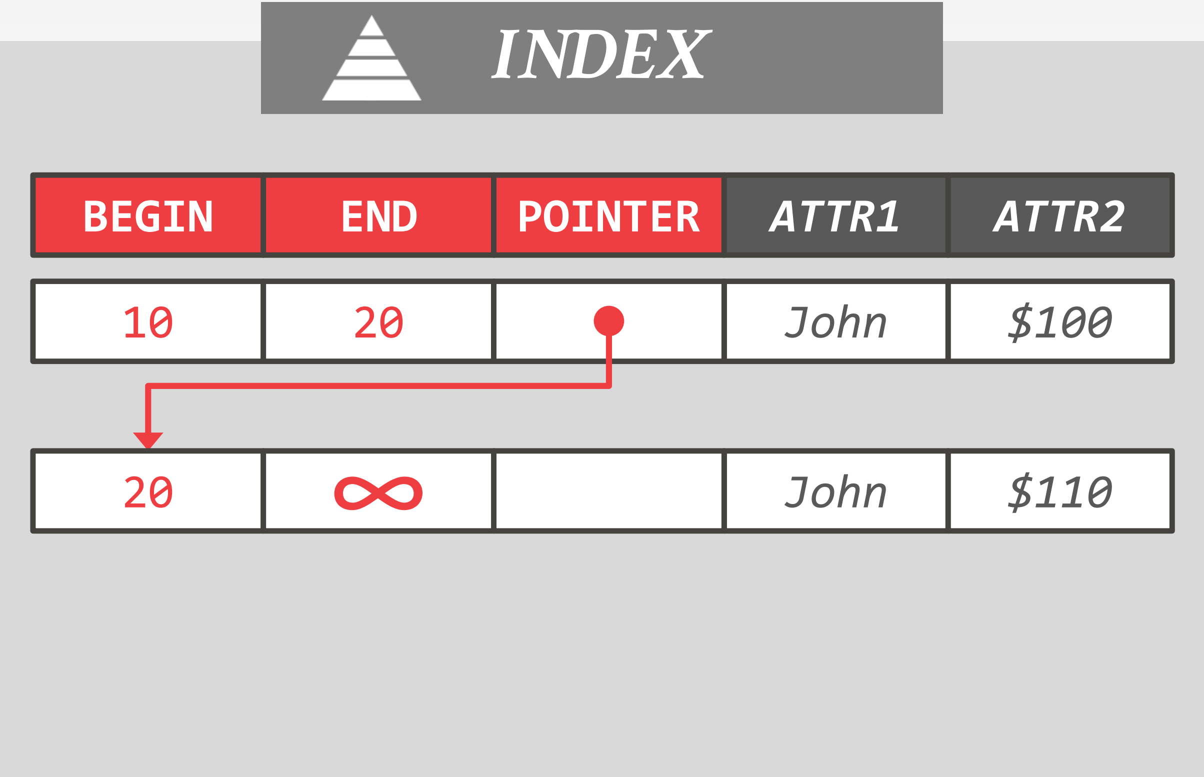
Each tuple contains two timestamps that represents their visibility and current state:

→ **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.

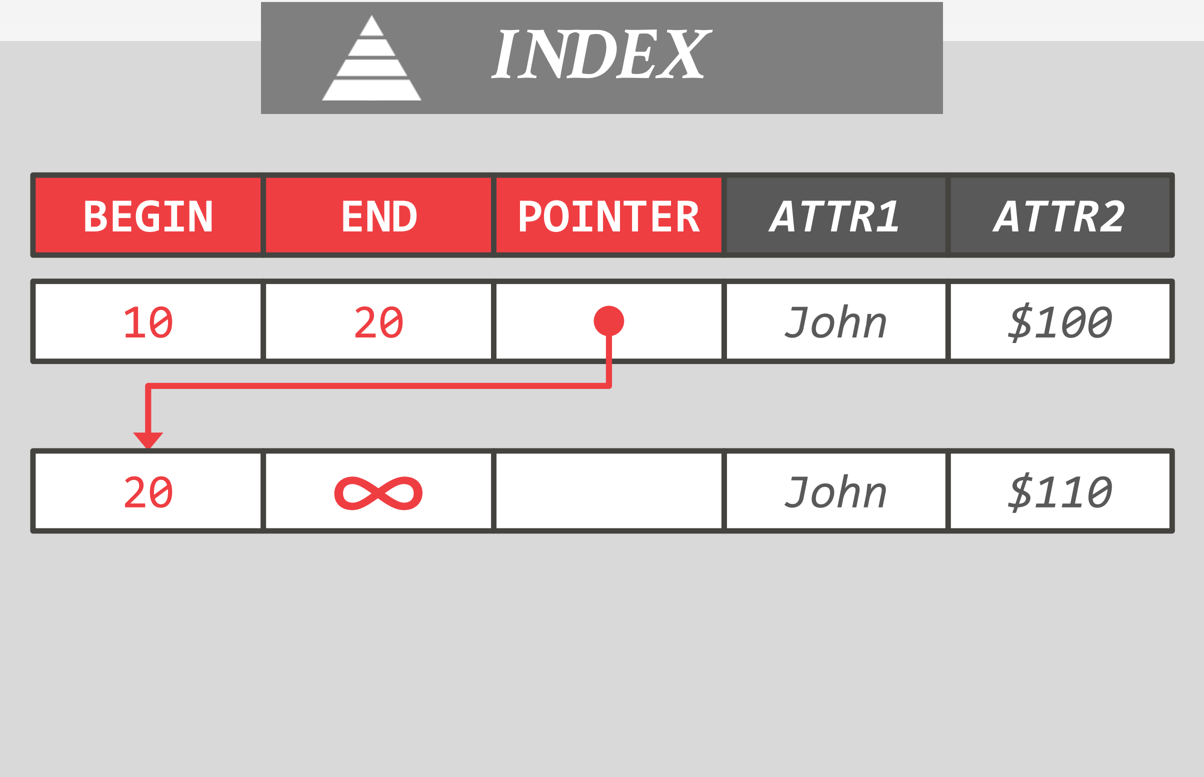
HIGH-PERFORMANCE CONCURRENCY CONTROL MECHANISMS FOR MAIN-MEMORY DATABASES
VLDB 2011

# HEKATON: OPERATIONS



# HEKATON: OPERATIONS

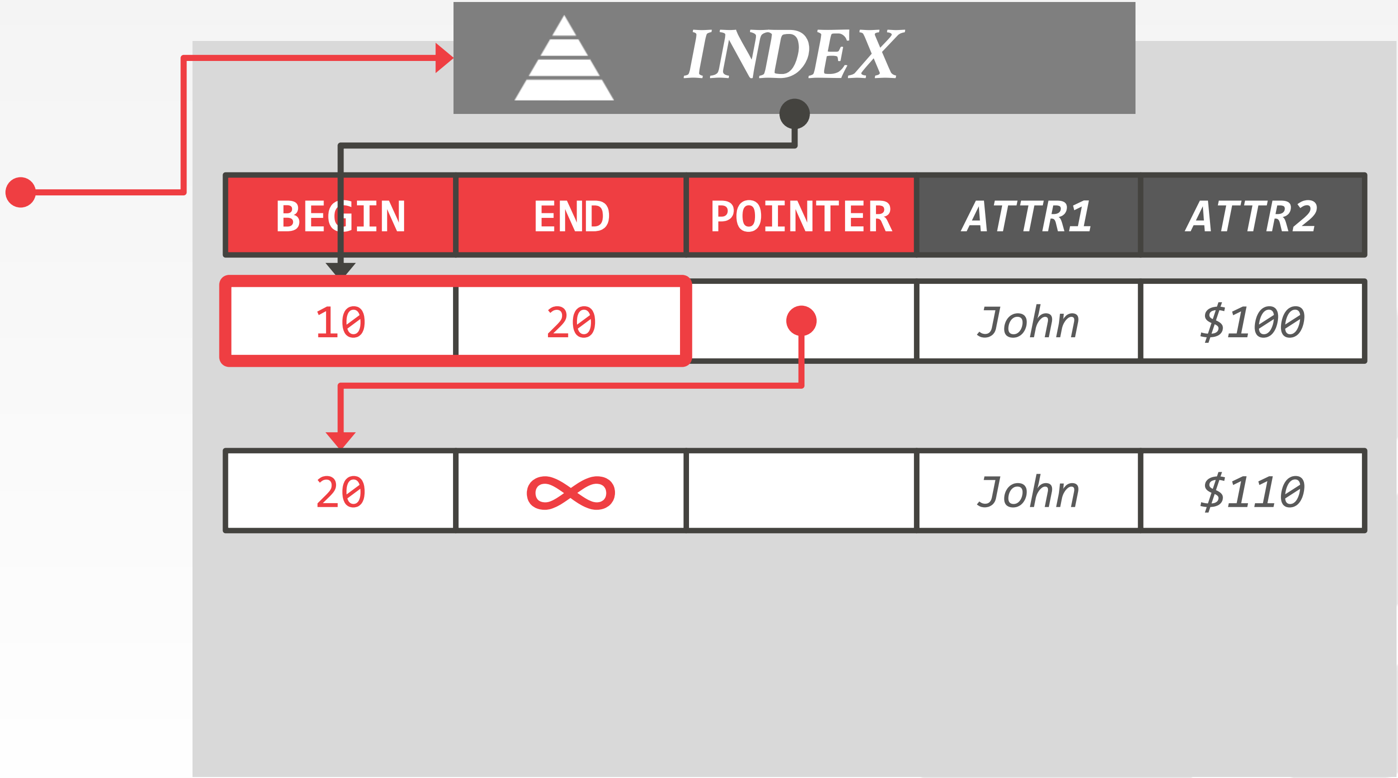
**BEGIN @ 25**



# HEKATON: OPERATIONS

**BEGIN @ 25**

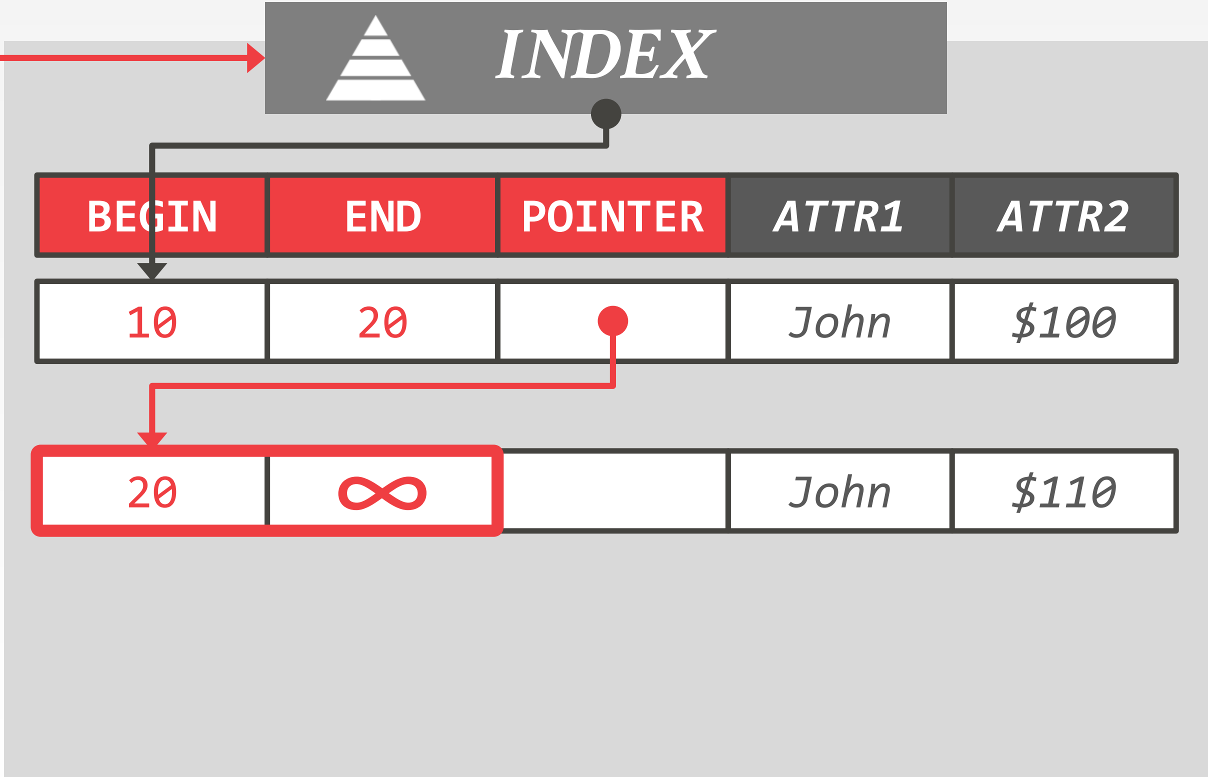
Read "John"



# HEKATON: OPERATIONS

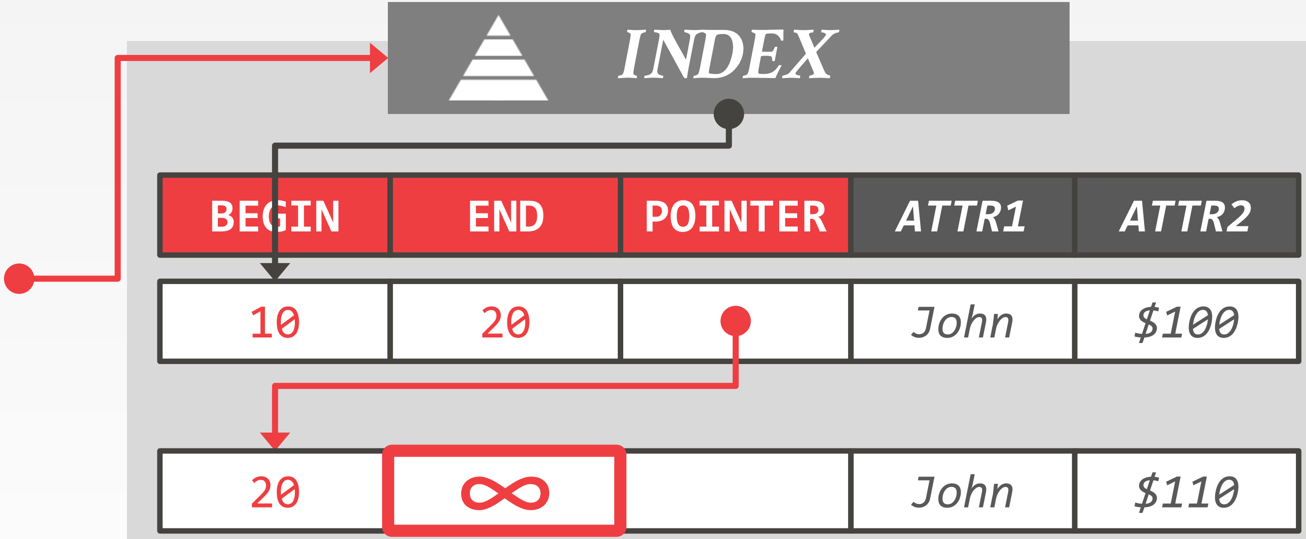
**BEGIN @ 25**

Read "John"



# HEKATON: OPERATIONS

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

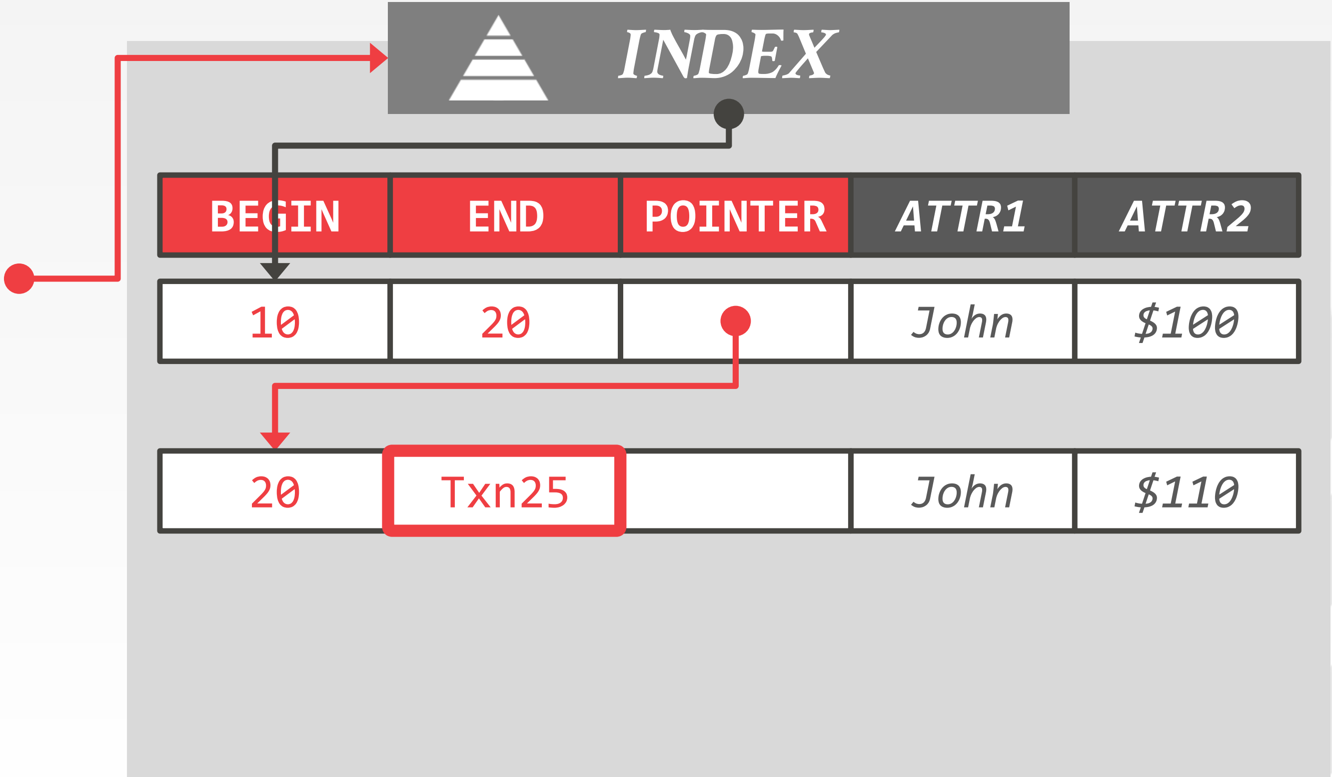


# HEKATON: OPERATIONS

**BEGIN @ 25**

Read "John"

Update "John"

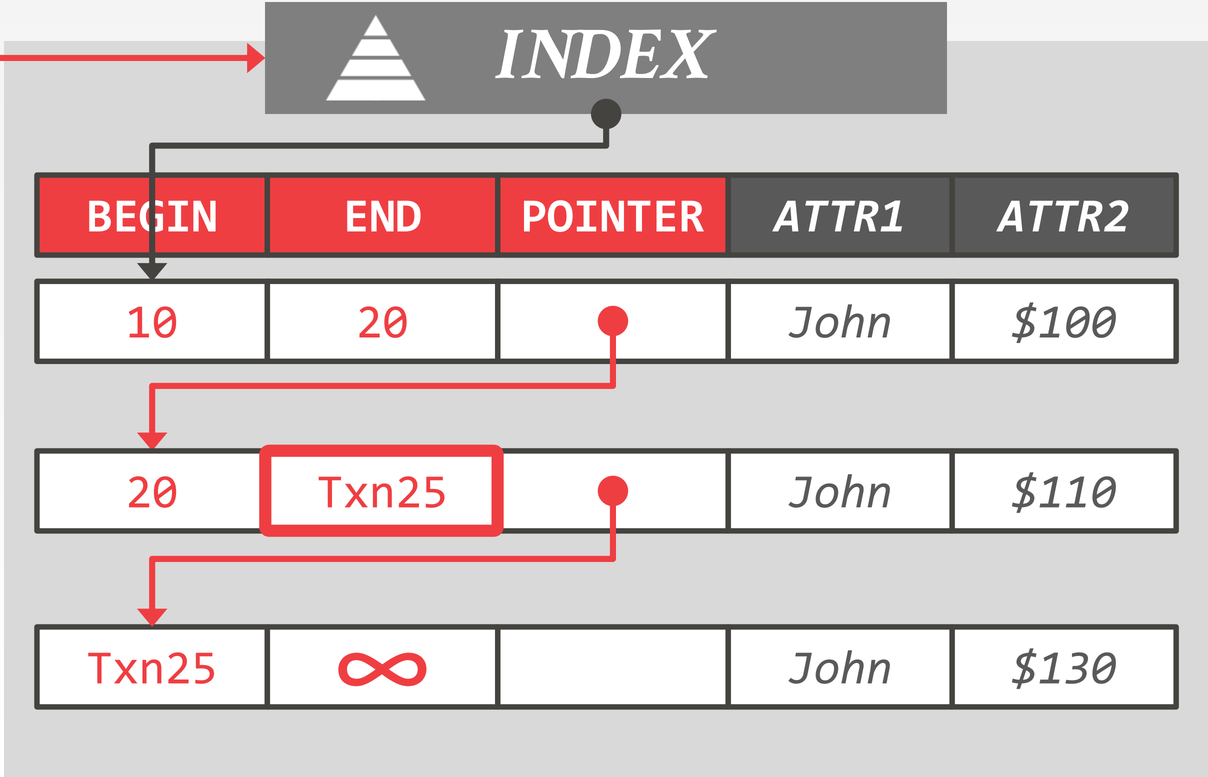


# HEKATON: OPERATIONS

**BEGIN @ 25**

Read "John"

Update "John"



# HEKATON: OPERATIONS

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

INDEX

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

# HEKATON: OPERATIONS

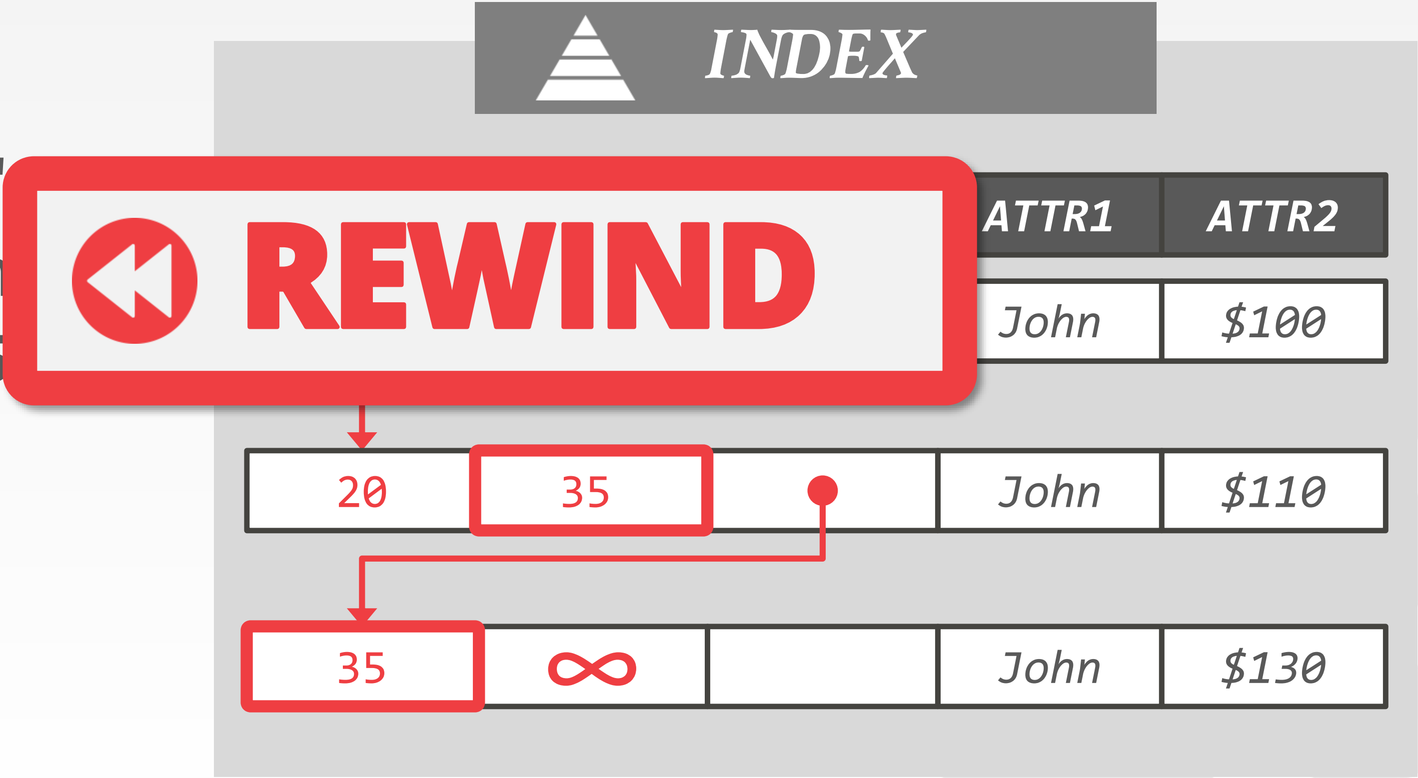
**BEGIN @ 25**
Read "John"
Update "John"
**COMMIT @ 35**

INDEX

| BEGIN | END | POINTER | ATTR1 | ATTR2 |
|---|---|---|---|---|
| 10 | 20 | • | John | $100 |
| 20 | 35 | • | John | $110 |
| 35 | ∞ | | John | $130 |

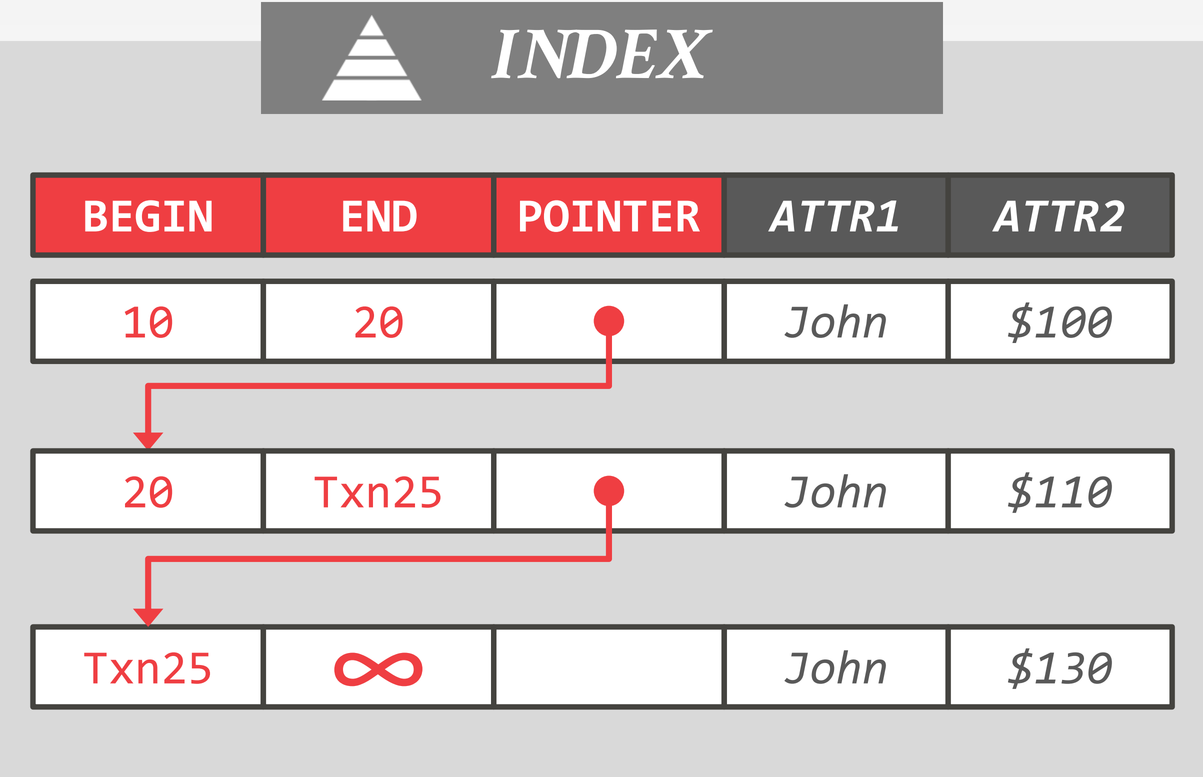
# HEKATON: OPERATIONS

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



# HEKATON: OPERATIONS

```
BEGIN @ 25
Read "John"
Update "John"
```



# 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"

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"

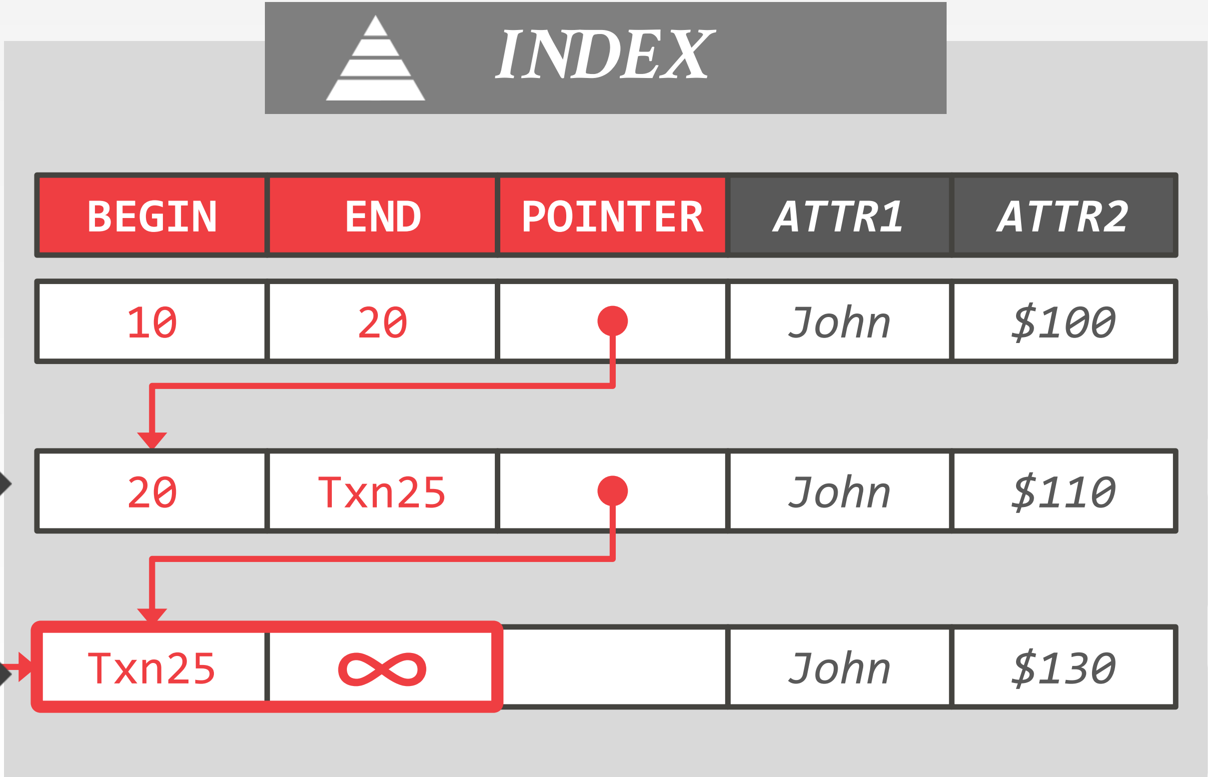
**INDEX**

| 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:

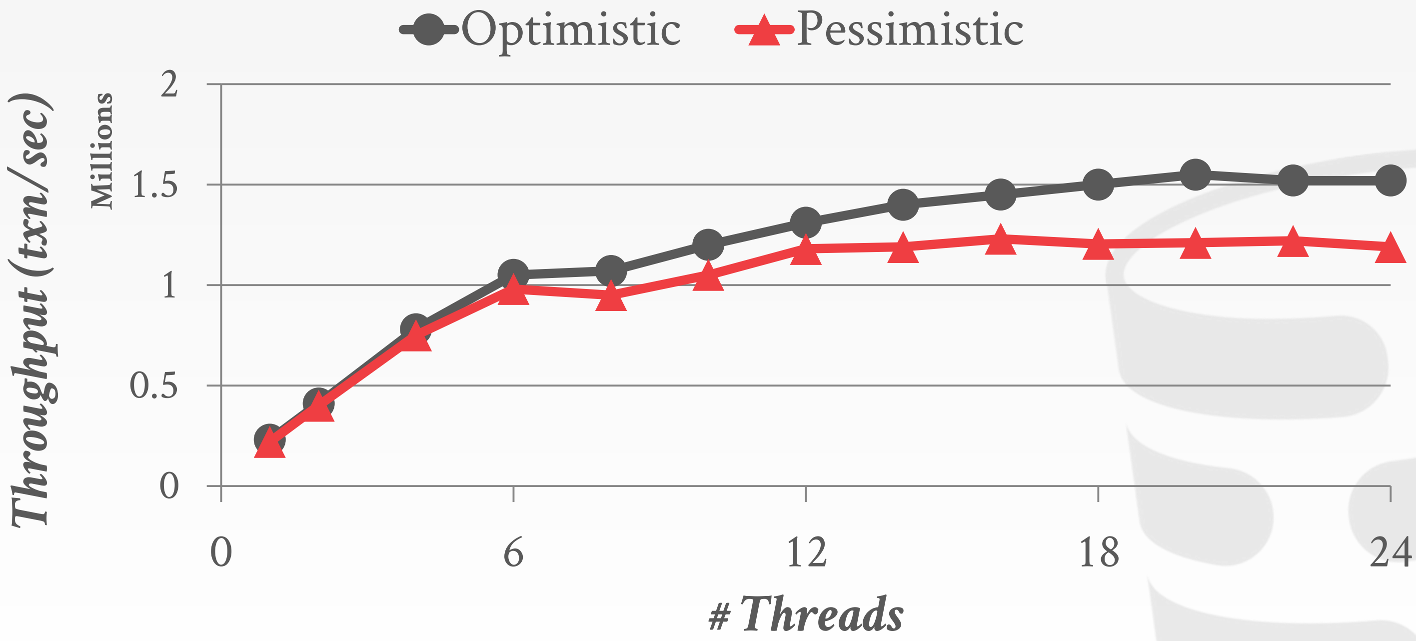
→ Check whether a version read is still visible at the end of the txn.
→ Repeat all index scans to check for phantoms.

## 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: 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.

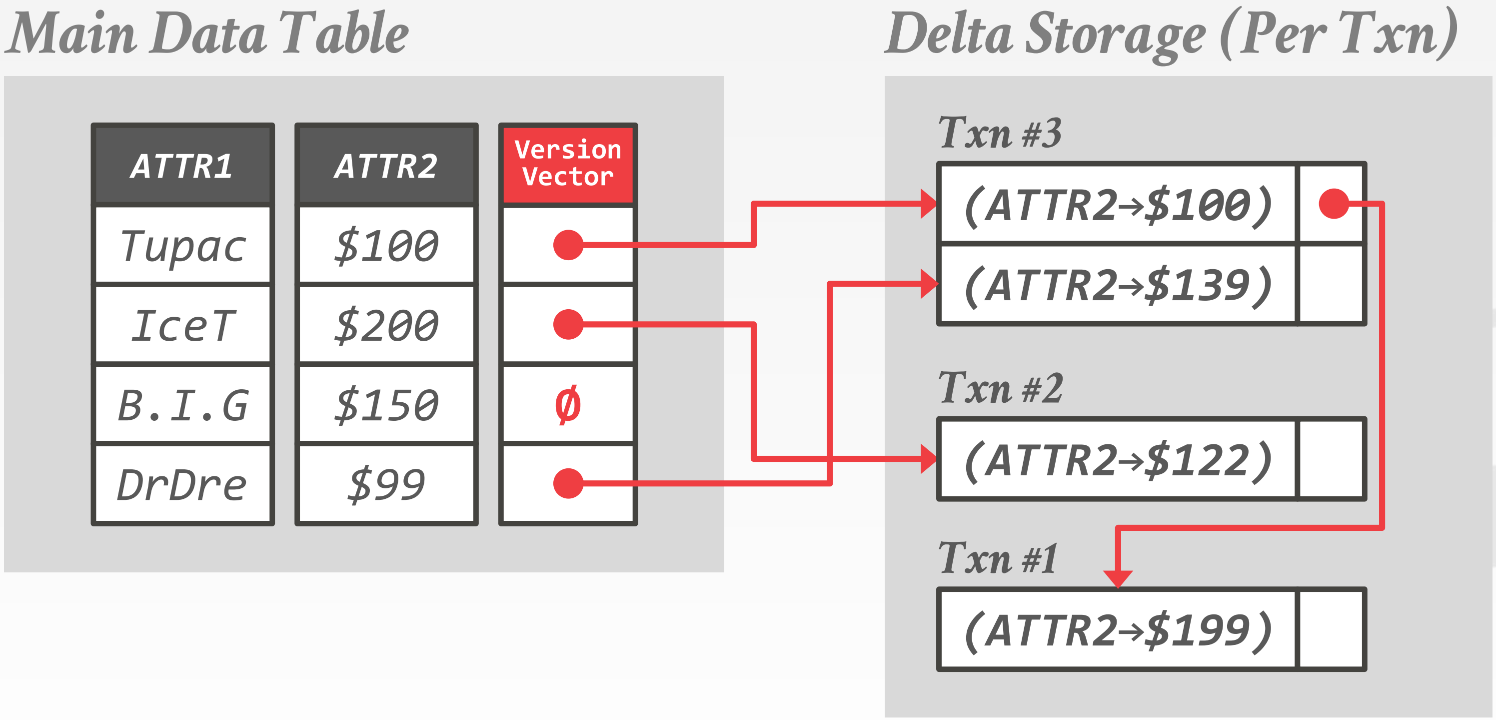
→ In-Place updates for non-indexed attributes
→ Delete/Insert updates for indexed attributes.
→ Newest-to-Oldest Version Chains
→ No Predicate Locks / No Scan Checks

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

Designed for HTAP workloads.

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

# HYPER: STORAGE ARCHITECTURE



# 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*

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

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

```sql
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 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) |

*Offsets*

| Offset | 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 | 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.

# CMU CICADA

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

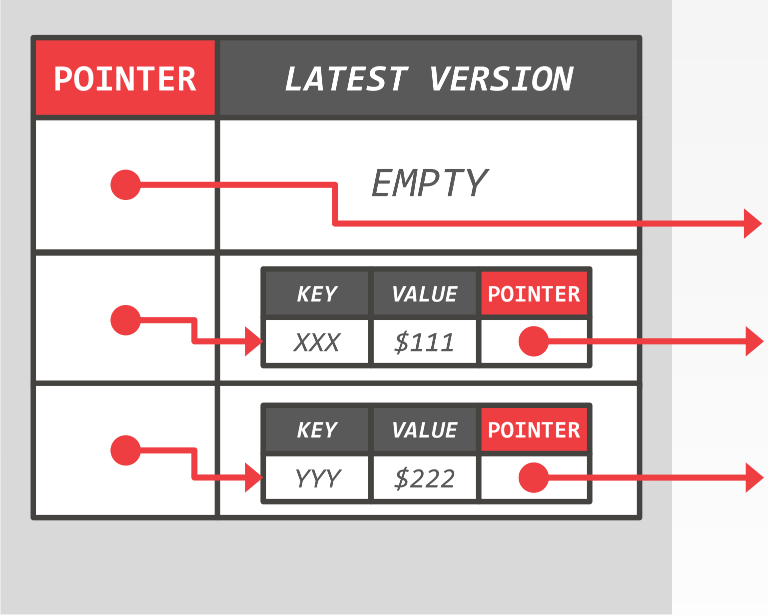
→ Best-effort Inlining
→ Loosely Synchronized Clocks
→ Contention-Aware Validation
→ Index Nodes Stored in Tables

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.

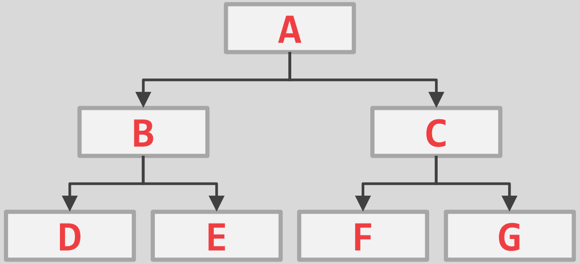
*Skip if all recent txns committed successfully.*

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

Source: Hyeontaek Lim

# CICADA: INDEX STORAGE

## *Index*

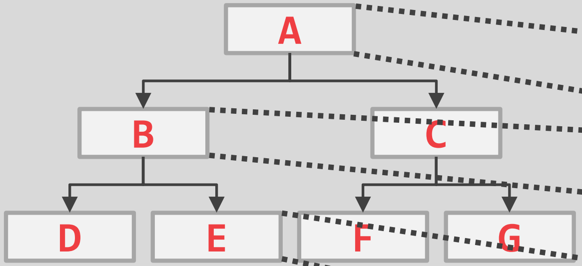



## *Index Node Table*

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

# CICADA: INDEX STORAGE

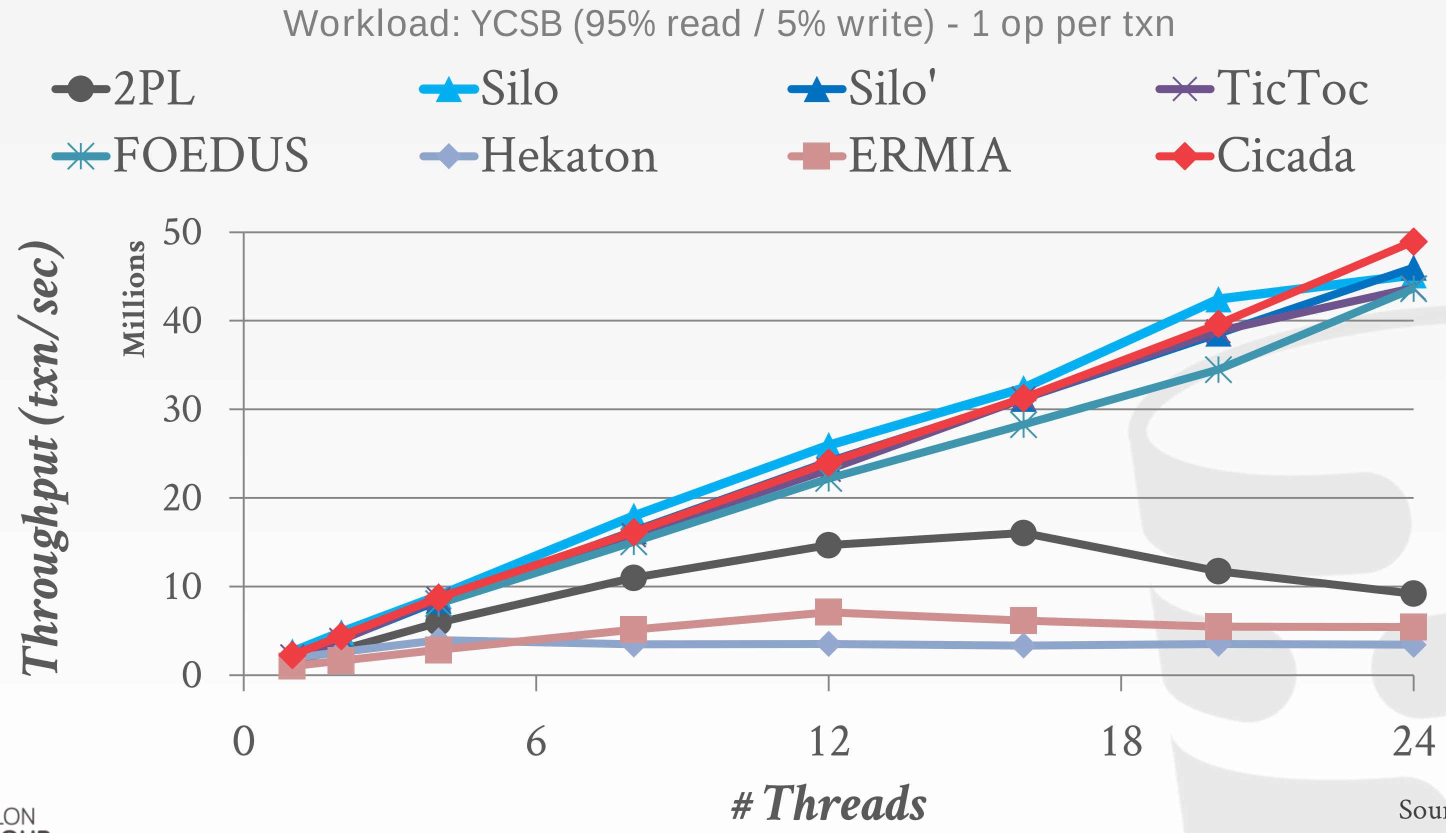
*Index*




*Index Node Table*

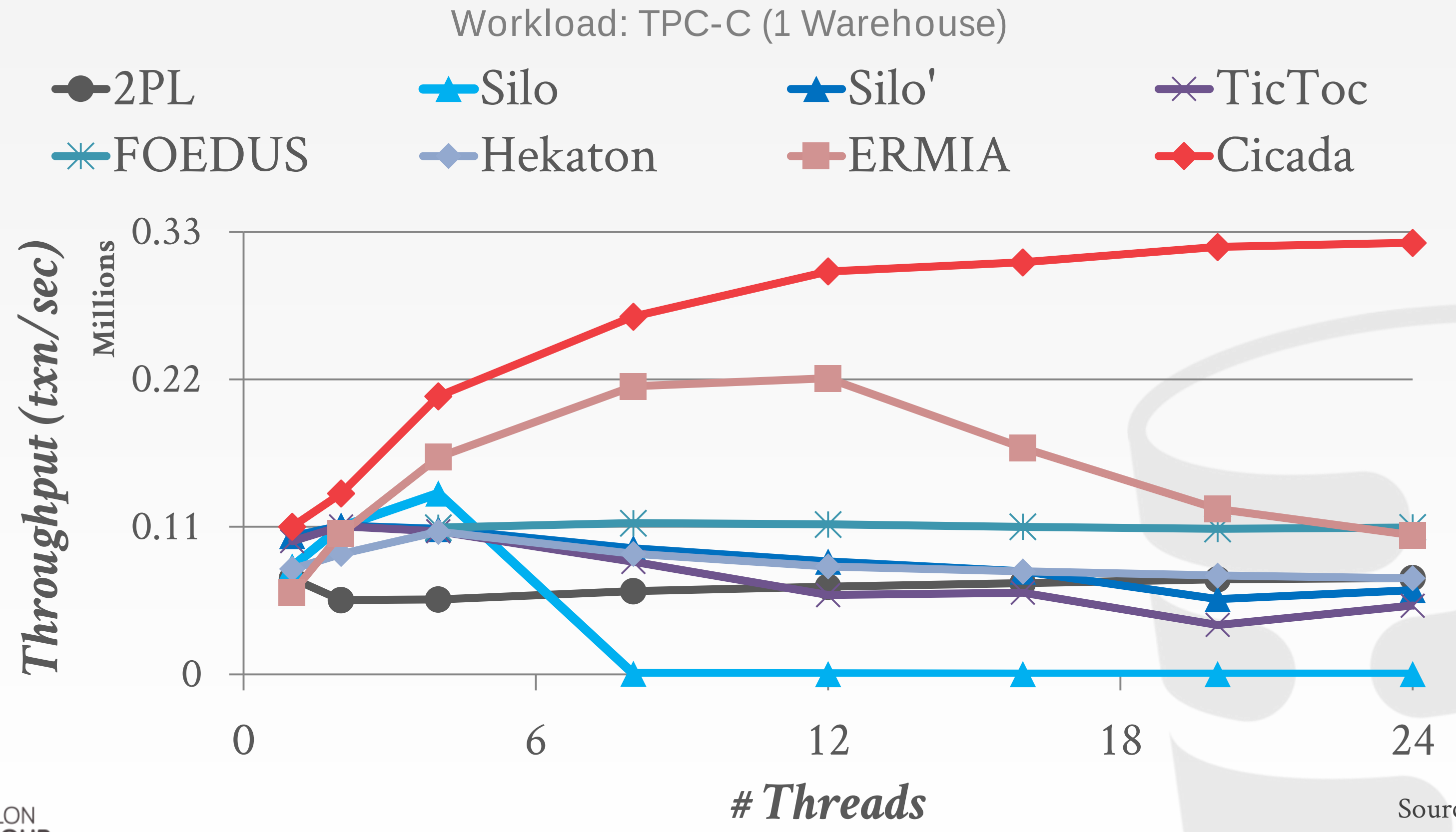
| | NODE DATA | POINTER |
|---|---|---|
| $A_1$ | Keys→[100,200]<br>Pointers→[B,C] | Ø |
| $B_2$ | Keys→[50,70]<br>Pointers→[D,E] | ● |
| $B_1$ | Keys→[52,70]<br>Pointers→[D,E] | Ø |
| $E_3$ | Keys→[10,30]<br>Pointers→[RID,RID] | ● |
| $E_2$ | Keys→[11,30]<br>Pointers→[RID,RID] | ● |
| $E_1$ | Keys→[12,30]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

# CICADA: LOW CONTENTION




Source: Hyeontaek Lim

# CICADA: HIGH CONTENTION




Source: Hyeontaek Lim

# 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