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

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>END</th>
<th>POINTER</th>
<th>ATTR1</th>
<th>ATTR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td></td>
<td>John</td>
<td>$100</td>
</tr>
<tr>
<td>20</td>
<td>∞</td>
<td></td>
<td>John</td>
<td>$110</td>
</tr>
</tbody>
</table>
HEKATON: OPERATIONS

BEGIN @ 25

INDEX

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>END</th>
<th>POINTER</th>
<th>ATTR1</th>
<th>ATTR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td>∞</td>
<td>John</td>
<td>$100</td>
</tr>
<tr>
<td>20</td>
<td>∞</td>
<td></td>
<td>John</td>
<td>$110</td>
</tr>
</tbody>
</table>
BEGIN @ 25
Read "John"

HEKATON: OPERATIONS

INDEX

BEGIN | END | POINTER | ATTR1 | ATTR2
------|------|---------|-------|-------
10    | 20   |         | John  | $100  
20    | ∞    |         | John  | $110  
BEGIN @ 25
Read "John"

HEKATON: OPERATIONS

INDEX

BEGIN | END | POINTER | ATTR1 | ATTR2
--- | --- | --- | --- | ---
10 | 20 | | John | $100

BEGIN | END | POINTER
--- | --- | ---
20 | ∞ |
BEGIN @ 25
Read "John"
Update "John"
BEGIN @ 25
Read "John"
Update "John"
HEKATON: OPERATIONS

BEGIN @ 25
Read "John"
Update "John"
BEGIN @ 25
Read "John"
Update "John"
COMMIT @ 35
BEGIN @ 25
Read "John"
Update "John"
COMMIT @ 35
BEGIN @ 25
Read "John"
Update "John"
COMMIT @ 35
HEKATON: OPERATIONS

BEGIN @ 25
Read "John"
Update "John"

<table>
<thead>
<tr>
<th>Begin</th>
<th>End</th>
<th>Pointer</th>
<th>Attr1</th>
<th>Attr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td></td>
<td>John</td>
<td>$100</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>Txn25</td>
<td>John</td>
<td>$110</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Txn25</td>
<td>John</td>
<td>$130</td>
</tr>
</tbody>
</table>
HEKATON: OPERATIONS

BEGIN @ 25
Read "John"
Update "John"

BEGIN @ 30
BEGIN @ 25
Read "John"
Update "John"

BEGIN @ 30
Read "John"
BEGIN @ 25
Read "John"
Update "John"

BEGIN @ 30
Read "John"
Update "John"
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 thetxn.

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.
# HYPER: STORAGE ARCHITECTURE

## Main Data Table

<table>
<thead>
<tr>
<th>ATTR1</th>
<th>ATTR2</th>
<th>Version Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tupac</td>
<td>$100</td>
<td></td>
</tr>
<tr>
<td>IceT</td>
<td>$200</td>
<td></td>
</tr>
<tr>
<td>B.I.G</td>
<td>$150</td>
<td>Ø</td>
</tr>
<tr>
<td>DrDre</td>
<td>$99</td>
<td></td>
</tr>
</tbody>
</table>

## Delta Storage (Per Txn)

**Txn #1**

(ATTR2→$199)

**Txn #2**

(ATTR2→$122)

**Txn #3**

(ATTR2→$100)

(ATTR2→$139)
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)
  FALSE

- Txn #1002
  (ATTR2→122)

- Txn #1003
  (ATTR1→'IceCube',
   ATTR2→199)
**HYPER: PRECISION LOCKING**

### Validating Txn

- **Select Query 1**
  ```sql
  SELECT * FROM foo
  WHERE attr2 > 20
  AND attr2 < 30
  ```

- **Select Query 2**
  ```sql
  SELECT COUNT(attr1)
  FROM foo
  WHERE attr2 IN (10, 20, 30)
  ```

- **Select Query 3**
  ```sql
  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)**
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)

---

FALSE
## 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)

- **Txn #1002**
  - (ATTR2→33)

- **Txn #1003**
  - (ATTR1→'IceCube', ATTR2→199)
**HYPER: PRECISION LOCKING**

**Validating Txn**

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

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

3. `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)

'Ice Cube' LIKE '%Ice%'

TRUE
HYPER: VERSION SYNOPSES

Main Data Table

<table>
<thead>
<tr>
<th>Version Synopsis</th>
<th>ATTR1</th>
<th>ATTR2</th>
<th>Version Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>[2,5)</td>
<td>Tupac</td>
<td>$100</td>
<td>Ø</td>
</tr>
<tr>
<td></td>
<td>IceT</td>
<td>$200</td>
<td>Ø</td>
</tr>
<tr>
<td></td>
<td>B.I.G</td>
<td>$150</td>
<td>Ø</td>
</tr>
<tr>
<td></td>
<td>DrDre</td>
<td>$99</td>
<td>Ø</td>
</tr>
<tr>
<td></td>
<td>RZA</td>
<td>$300</td>
<td>Ø</td>
</tr>
<tr>
<td></td>
<td>GZA</td>
<td>$300</td>
<td>Ø</td>
</tr>
<tr>
<td></td>
<td>ODB</td>
<td>$0</td>
<td>Ø</td>
</tr>
</tbody>
</table>

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 SYNOPSIS

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.

<table>
<thead>
<tr>
<th>Version Synopsis</th>
<th>ATTR1</th>
<th>ATTR2</th>
<th>Version Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>[2,5)</td>
<td>0</td>
<td>$100</td>
<td>Ø</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>$200</td>
<td>Ø</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>$150</td>
<td>Ø</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>$99</td>
<td>Ø</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>$300</td>
<td>Ø</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>$300</td>
<td>Ø</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>$0</td>
<td>Ø</td>
</tr>
</tbody>
</table>
**HYPER: VERSION SYNOPSES**

**Main Data Table**

<table>
<thead>
<tr>
<th>Version Synopsis</th>
<th>ATTR1</th>
<th>ATTR2</th>
<th>Version Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>[2,5)</td>
<td>Tupac</td>
<td>$100</td>
<td>ø</td>
</tr>
<tr>
<td></td>
<td>IceT</td>
<td>$200</td>
<td>ø</td>
</tr>
<tr>
<td></td>
<td>B.I.G</td>
<td>$150</td>
<td>ø</td>
</tr>
<tr>
<td></td>
<td>DrDre</td>
<td>$99</td>
<td>ø</td>
</tr>
<tr>
<td></td>
<td>RZA</td>
<td>$300</td>
<td>ø</td>
</tr>
<tr>
<td></td>
<td>GZA</td>
<td>$300</td>
<td>ø</td>
</tr>
<tr>
<td></td>
<td>ODB</td>
<td>$0</td>
<td>ø</td>
</tr>
</tbody>
</table>

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 SYNOPSIS

Main Data Table

<table>
<thead>
<tr>
<th>Version Synopsis</th>
<th>ATTR1</th>
<th>ATTR2</th>
<th>Version Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>[2,5)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tupac</td>
<td></td>
<td>$100</td>
<td>Ø</td>
</tr>
<tr>
<td>IceT</td>
<td></td>
<td>$200</td>
<td>Ø</td>
</tr>
<tr>
<td>B.I.G</td>
<td></td>
<td>$150</td>
<td>Ø</td>
</tr>
<tr>
<td>DrDre</td>
<td></td>
<td>$99</td>
<td>Ø</td>
</tr>
<tr>
<td>RZA</td>
<td></td>
<td>$300</td>
<td>Ø</td>
</tr>
<tr>
<td>GZA</td>
<td></td>
<td>$300</td>
<td>Ø</td>
</tr>
<tr>
<td>ODB</td>
<td></td>
<td>$0</td>
<td>Ø</td>
</tr>
</tbody>
</table>

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: 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.

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

Skip if all recent txns committed successfully.

Source: Hyeontaek Lim
CMU 15-721 (Spring 2018)
CICADA: INDEX STORAGE

Index

Index Node Table

<table>
<thead>
<tr>
<th>NODE DATA</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_1</td>
<td>Keys→[100,200] Pointer→[B,C]</td>
</tr>
<tr>
<td>B_2</td>
<td>Keys→[50,70] Pointer→[D,E]</td>
</tr>
<tr>
<td>B_1</td>
<td>Keys→[52,70] Pointer→[D,E]</td>
</tr>
<tr>
<td>E_3</td>
<td>Keys→[10,30] Pointer→[RID,RID]</td>
</tr>
<tr>
<td>E_2</td>
<td>Keys→[11,30] Pointer→[RID,RID]</td>
</tr>
<tr>
<td>E_1</td>
<td>Keys→[12,30] Pointer→[RID,RID]</td>
</tr>
</tbody>
</table>
# CICADA: INDEX STORAGE

<table>
<thead>
<tr>
<th>Index Node Table</th>
<th>NODE DATA</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Keys→[100, 200] &lt;br&gt; Pointers→[B, C]</td>
<td>Ø</td>
</tr>
<tr>
<td>B</td>
<td>Keys→[50, 70] &lt;br&gt; Pointers→[D, E]</td>
<td>Ø</td>
</tr>
<tr>
<td>C</td>
<td>Keys→[52, 70] &lt;br&gt; Pointers→[D, E]</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>Keys→[10, 30] &lt;br&gt; Pointers→[RID, RID]</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>Keys→[11, 30] &lt;br&gt; Pointers→[RID, RID]</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>Keys→[12, 30] &lt;br&gt; Pointers→[RID, RID]</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Diagram**

- A connects to B, C, D, E, F, and G.
- B connects to C, D, and E.
- C connects to D and E.
- D connects to E.
- E connects to F and G.

**Notes**

- Keys and Pointers are data structures used in indexing.
- RID stands for Record Identifier.
CICADA: LOW CONTENTION

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

Throughput (txn/sec) vs. # Threads

- 2PL
- Silo
- Silo'
- TicToc
- FOEDUS
- Hekaton
- ERMIA
- Cicada

Source: Hyeontaek Lim
CICADA: HIGH CONTENTION

Workload: TPC-C (1 Warehouse)

- 2PL
- Silo
- Silo'
- TicToc
- FOEDUS
- Hekaton
- ERMIA
- Cicada

Throughput (txn/sec) Millions

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