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

\[
\_\_sync\_bool\_compare\_and\_swap(\&M, 20, 30)
\]
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

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

\[ \_\_\text{sync\_bool\_compare\_and\_swap}(\&M, 25, 35) \]
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.
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

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Version Storage</th>
<th>Garbage Collection</th>
<th>Indexes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle</td>
<td>MV2PL</td>
<td>Delta</td>
<td>Vacuum</td>
</tr>
<tr>
<td>Postgres</td>
<td>MV-2PL/MV-T0</td>
<td>Append-Only</td>
<td>Vacuum</td>
</tr>
<tr>
<td>MySQL-InnoDB</td>
<td>MV-2PL</td>
<td>Delta</td>
<td>Vacuum</td>
</tr>
<tr>
<td>HYRISE</td>
<td>MV-OCC</td>
<td>Append-Only</td>
<td>-</td>
</tr>
<tr>
<td>Hekaton</td>
<td>MV-OCC</td>
<td>Append-Only</td>
<td>Cooperative</td>
</tr>
<tr>
<td>MemSQL</td>
<td>MV-OCC</td>
<td>Append-Only</td>
<td>Vacuum</td>
</tr>
<tr>
<td>SAP HANA</td>
<td>MV-2PL</td>
<td>Time-travel</td>
<td>Hybrid</td>
</tr>
<tr>
<td>NuoDB</td>
<td>MV-2PL</td>
<td>Append-Only</td>
<td>Vacuum</td>
</tr>
<tr>
<td>HyPer</td>
<td>MV-OCC</td>
<td>Delta</td>
<td>Txn-level</td>
</tr>
</tbody>
</table>
TUPLE FORMAT

Unique Txn Identifier
Version Lifetime
Next/Prev Version
Additional Metadata
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)

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

<table>
<thead>
<tr>
<th>TXN-ID</th>
<th>READ-TS</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_x</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B_x</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Use “read-ts” field in the header to keep track of the timestamp of the last txn that read it.

<table>
<thead>
<tr>
<th>TXN-ID</th>
<th>READ-TS</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_x$</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$B_x$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
TIMESTAMP ORDERING (MVTO)

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

<table>
<thead>
<tr>
<th>TXN-ID</th>
<th>READ-TS</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_x$</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$B_x$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
TIMESTAMP ORDERING (MVTO)

$T_{id}=10$

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

$T_{id} = 10$

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

<table>
<thead>
<tr>
<th></th>
<th>TXN-ID</th>
<th>READ-TS</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_x$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$B_x$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>
Use "read-ts" field in the header to keep track of the timestamp of the last txn that read it.

\( T_{id} = 10 \)

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

<table>
<thead>
<tr>
<th>TXN-ID</th>
<th>READ-TS</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A_x )</td>
<td>0</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>( B_x )</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
**TIMESTAMP ORDERING (MVTO)**

**$T_{id}=10$**

- **READ(A)**
- **WRITE(B)**

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”.

<table>
<thead>
<tr>
<th>TXN-ID</th>
<th>READ-TS</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_x$</td>
<td>0</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>$B_x$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
TIMESTAMP ORDERING (MVTO)

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”.

$T_{id} = 10$

<table>
<thead>
<tr>
<th>TXN-ID</th>
<th>READ-TS</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_x$</td>
<td>0</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>$B_x$</td>
<td>10</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
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)

<table>
<thead>
<tr>
<th>TXN-ID</th>
<th>READ-TS</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_x$</td>
<td>0</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>$B_x$</td>
<td>10</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$B_{x+1}$</td>
<td>10</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>
### Timestamp Ordering (MVTO)

**Timestamp Ordering** is a method for ensuring that transactions are processed in a consistent order. It uses the `read-ts` field in the header of each transaction to keep track of the timestamp of the last transaction that read a particular version.

**Example:**

- **Transaction 1 (Txn A):**
  - **Read Operation:** `READ(A)`
  - **Timestamp:** `T_id = 10`

- **Transaction 2 (Txn B):**
  - **Write Operation:** `WRITE(B)`

- **Transactions:**
  - **Txn C:**
    - **Write Operation:** `WRITE(B)`
    - **Timestamp:** `T_id = 10`

**Table: Timestamp Ordering**

<table>
<thead>
<tr>
<th>TXN-ID</th>
<th>READ-TS</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_x</td>
<td>0</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>B_x</td>
<td>10</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>B_{x+1}</td>
<td>10</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

**Rules:**

- **Txn is allowed to read version if the lock is unset and its T_id is between “begin-ts” and “end-ts”:**
  - In the example, `Txn A` can read `A_x` because `B_x` has already set its lock.

- **Txn creates a new version if no other txn holds lock and T_id is greater than “read-ts”:**
  - `Txn C` creates a new version of `B_{x+1}` because `B_x` has not released its lock.

---

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

<table>
<thead>
<tr>
<th>TXN-ID</th>
<th>READ-TS</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_x$</td>
<td>0</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>$B_x$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$B_{x+1}$</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>KEY</th>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_x$</td>
<td>$XXX$</td>
<td>$$111$</td>
</tr>
<tr>
<td>$A_{x+1}$</td>
<td>$XXX$</td>
<td>$$222$</td>
</tr>
<tr>
<td>$B_x$</td>
<td>$YYY$</td>
<td>$$10$</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>KEY</th>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_x$</td>
<td>XXX</td>
<td>$111$</td>
</tr>
<tr>
<td>$A_{x+1}$</td>
<td>XXX</td>
<td>$222$</td>
</tr>
<tr>
<td>$B_x$</td>
<td>YYY</td>
<td>$10$</td>
</tr>
<tr>
<td>$A_{x+2}$</td>
<td>XXX</td>
<td>$333$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>KEY</th>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_x$</td>
<td>$XXX$</td>
<td>$$111$</td>
</tr>
<tr>
<td>$A_{x+1}$</td>
<td>$XXX$</td>
<td>$$222$</td>
</tr>
<tr>
<td>$B_x$</td>
<td>$YYY$</td>
<td>$$10$</td>
</tr>
<tr>
<td>$A_{x+2}$</td>
<td>$XXX$</td>
<td>$$333$</td>
</tr>
</tbody>
</table>

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

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**

<table>
<thead>
<tr>
<th>KEY</th>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₂</td>
<td>$333</td>
<td></td>
</tr>
<tr>
<td>B₁</td>
<td>$10</td>
<td></td>
</tr>
</tbody>
</table>

**Time-Travel Table**

<table>
<thead>
<tr>
<th>KEY</th>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>$111</td>
<td>Ø</td>
</tr>
<tr>
<td>A₂</td>
<td>$222</td>
<td></td>
</tr>
</tbody>
</table>

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

Overwrite master version in the main table. Update pointers.
On every update, copy only the values that were modified to the delta storage and overwrite the master version.
On every update, copy only the values that were modified to the delta storage and overwrite the master version.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2</td>
<td>XXX</td>
<td>$222</td>
</tr>
<tr>
<td>B1</td>
<td>YYY</td>
<td>$10</td>
</tr>
</tbody>
</table>

Delta Storage Segment

- A1: (VALUE→$111)
- Pointer: Ø
On every update, copy only the values that were modified to the delta storage and overwrite the master version.

<table>
<thead>
<tr>
<th>KEY</th>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>XXX</td>
<td>$111</td>
</tr>
<tr>
<td>B₁</td>
<td>YYY</td>
<td>$10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DELTA</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>(VALUE→$222)</td>
<td>Ø</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DELTA</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>(VALUE→$111)</td>
<td>Ø</td>
</tr>
</tbody>
</table>
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.
Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

<table>
<thead>
<tr>
<th>TXN-ID</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A_x)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(B_x)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(B_{x+1})</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>
Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.
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.
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...
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 a DELETE followed by an INSERT.
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

GET(A) 

PRIMARY INDEX

SECONDARY INDEX

Physical Address

\[ A_{x+3} \rightarrow A_{x+2} \rightarrow A_{x+1} \rightarrow A_x \]

Append-Only
Newest-to-Oldest
INDEX POINTERS

PRIMARY INDEX

SECONDARY INDEX

Append-Only
Newest-to-Oldest

Physical Address

GET(A)
INDEX POINTERS

PRIMAR INDEX

SECONDARY INDEX

SECONDARY INDEX

SECONDARY INDEX

SECONDARY INDEX

A_{x+3} \rightarrow A_{x+2} \rightarrow A_{x+1} \rightarrow A_x

GET(A)

Append-Only
Newest-to-Oldest
INDEX POINTERS

GET(A)

PRIMARY INDEX

SECONDARY INDEX

Physical Address

Primary Key

\( \text{Append-Only newest-to-oldest} \)
INDEX POINTERS

<table>
<thead>
<tr>
<th>PRIMARY INDEX</th>
<th>SECONDARY INDEX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TupleId</td>
</tr>
<tr>
<td>TupleId→Address</td>
<td></td>
</tr>
<tr>
<td>A_{x+3}</td>
<td>A_{x+2}</td>
</tr>
<tr>
<td>A_{x+1}</td>
<td>A_{x}</td>
</tr>
</tbody>
</table>

GET(A)

Physical Address

Append-Only
Newest-to-Oldest
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.
HEKATON: OPERATIONS

**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"

INFORMATION

INDEX

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

20 | | | John | $110
HEKATON: OPERATIONS

BEGIN @ 25
Read “John”
BEGIN @ 25
Read "John"
Update "John"
HEKATON: OPERATIONS

BEGIN @ 25
Read “John”
Update “John”
HEKATON: OPERATIONS

BEGIN @ 25
Read “John”
Update “John”
BEGIN @ 25
Read “John”
Update “John”
COMMIT @ 35
# HEKATON: OPERATIONS

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

<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>
<tr>
<td></td>
<td></td>
<td></td>
<td>John</td>
<td>$130</td>
</tr>
</tbody>
</table>

BEGIN
INDEX

Update “John”
Read “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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Txn25</td>
<td></td>
<td>John</td>
<td>$110</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Txn25</td>
<td></td>
<td>$\infty$</td>
<td>John</td>
<td>$130</td>
</tr>
</tbody>
</table>
HEKATON: OPERATIONS

BEGIN @ 25
Read “John”
Update “John”

BEGIN @ 30

<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>Txn25</td>
<td></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
Read “John”
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”
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

Database group
CMU 15-721 (Spring 2017)
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.
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.
### HYPER MVCC

#### 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></td>
</tr>
<tr>
<td>DrDre</td>
<td>$99</td>
<td></td>
</tr>
</tbody>
</table>

#### Delta Storage (Per Txn)

- **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 workloads

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.
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 2\textsuperscript{nd}, 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