TODAY’S AGENDA

Isolation Levels
Modern Multi-Version Concurrency Control
Observation

Serializability is useful because it allows programmers to ignore concurrency issues but enforcing it may allow too little parallelism and limit performance.

We may want to use a weaker level of consistency to improve scalability.
ISOLATION LEVELS

Controls the extent that a txn is exposed to the actions of other concurrent txns.

Provides for greater concurrency at the cost of exposing txns to uncommitted changes:

→ Dirty Read Anomaly
→ Unrepeatable Reads Anomaly
→ Phantom Reads Anomaly
<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Phantom and Read Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SERIALIZABLE</strong></td>
<td>No phantoms, all reads repeatable, no dirty reads.</td>
</tr>
<tr>
<td><strong>REPEATABLE READS</strong></td>
<td>Phantoms may happen.</td>
</tr>
<tr>
<td><strong>READ COMMITTED</strong></td>
<td>Phantoms and unrepeateable reads may happen.</td>
</tr>
<tr>
<td><strong>READ UNCOMMITTED</strong></td>
<td>All of them may happen.</td>
</tr>
</tbody>
</table>
ISOLATION LEVEL HIERARCHY

1. READ UNCOMMITTED
2. READ COMMITTED
3. REPEATABLE READS
4. SERIALIZABLE
## ANSI ISOLATION LEVELS

<table>
<thead>
<tr>
<th></th>
<th>Default</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actian Ingres 10.0/10S</td>
<td><strong>SERIALIZABLE</strong></td>
<td><strong>SERIALIZABLE</strong></td>
</tr>
<tr>
<td>Greenplum 4.1</td>
<td><strong>READ COMMITTED</strong></td>
<td><strong>SERIALIZABLE</strong></td>
</tr>
<tr>
<td>MySQL 5.6</td>
<td><strong>REPEATABLE READS</strong></td>
<td><strong>SERIALIZABLE</strong></td>
</tr>
<tr>
<td>MemSQL 1b</td>
<td><strong>READ COMMITTED</strong></td>
<td><strong>READ COMMITTED</strong></td>
</tr>
<tr>
<td>MS SQL Server 2012</td>
<td><strong>READ COMMITTED</strong></td>
<td><strong>SERIALIZABLE</strong></td>
</tr>
<tr>
<td>Oracle 11g</td>
<td><strong>READ COMMITTED</strong></td>
<td><strong>SNAPSHOT ISOLATION</strong></td>
</tr>
<tr>
<td>Postgres 9.2.2</td>
<td><strong>READ COMMITTED</strong></td>
<td><strong>SERIALIZABLE</strong></td>
</tr>
<tr>
<td>SAP HANA</td>
<td><strong>READ COMMITTED</strong></td>
<td><strong>SERIALIZABLE</strong></td>
</tr>
<tr>
<td>ScaleDB 1.02</td>
<td><strong>READ COMMITTED</strong></td>
<td><strong>READ COMMITTED</strong></td>
</tr>
<tr>
<td>VoltDB</td>
<td><strong>SERIALIZABLE</strong></td>
<td><strong>SERIALIZABLE</strong></td>
</tr>
</tbody>
</table>

Source: Peter Bailis
## ANSI ISOLATION LEVELS

<table>
<thead>
<tr>
<th></th>
<th>Default</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actian Ingres 10.0/10S</td>
<td>TRANSACTIONAL</td>
<td>TRANSACTIONAL</td>
</tr>
<tr>
<td>Greenplum 4.1</td>
<td>READ COMMITTED</td>
<td>TRANSACTIONAL</td>
</tr>
<tr>
<td>MySQL 5.6</td>
<td>REPEATABLE READS</td>
<td>TRANSACTIONAL</td>
</tr>
<tr>
<td>MemSQL 1b</td>
<td>READ COMMITTED</td>
<td>READ COMMITTED</td>
</tr>
<tr>
<td>MS SQL Server 2012</td>
<td>READ COMMITTED</td>
<td>TRANSACTIONAL</td>
</tr>
<tr>
<td>Oracle 11g</td>
<td>READ COMMITTED</td>
<td>TRANSACTIONAL</td>
</tr>
<tr>
<td>Postgres 9.2.2</td>
<td>READ COMMITTED</td>
<td>TRANSACTIONAL</td>
</tr>
<tr>
<td>SAP HANA</td>
<td>READ COMMITTED</td>
<td>TRANSACTIONAL</td>
</tr>
<tr>
<td>ScaleDB 1.02</td>
<td>READ COMMITTED</td>
<td>READ COMMITTED</td>
</tr>
<tr>
<td>VoltDB</td>
<td>TRANSACTIONAL</td>
<td>TRANSACTIONAL</td>
</tr>
</tbody>
</table>

Source: Peter Bailis
CRITICISM OF ISOLATION LEVELS

The isolation levels defined as part of SQL-92 standard only focused on anomalies that can occur in a 2PL-based DBMS.

Two additional isolation levels:
→ CURSOR STABILITY
→ SNAPSHOT ISOLATION
CURSOR STABILITY (CS)

The DBMS’s internal cursor maintains a lock on a item in the database until it moves on to the next item.

CS is a stronger isolation level in between **REPEATABLE READS** and **READ COMMITTED** that can (sometimes) prevent the **Lost Update Anomaly**.
LOST UPDATE ANOMALY

**Txn #1**

BEGIN

READ(A)

• • •

WRITE(A)

COMMIT

**Txn #2**

BEGIN

• • •

WRITE(A)

• • • • • • • •

COMMIT
LOST UPDATE ANOMALY

**Txn #1**

BEGIN

READ(A)  ...  WRITE(A)

COMMIT

**Txn #2**

BEGIN

...  WRITE(A)  ...

COMMIT
LOST UPDATE ANOMALY

**Txn #1**

BEGIN
READ(A)  ...  WRITE(A)  COMMIT

**Txn #2**

BEGIN  ...  WRITE(A)  ...  COMMIT
LOST UPDATE ANOMALY

Txn #1
BEGIN
READ(A)
... ...
WRITE(A)
COMMIT

Txn #2
BEGIN
... ...
WRITE(A)
... ...
COMMIT
LOST UPDATE ANOMALY

**Txn #1**

BEGIN

READ(A)

\[\cdots\]\n
WRITE(A)

COMMIT

**Txn #2**

BEGIN

\[\cdots\]\n
\[\cdots\]\n
\[\cdots\]\n
\[\cdots\]\n
\[\cdots\]\n
\[\cdots\]\n
WRITE(A)

COMMIT
LOST UPDATE ANOMALY

 Txn #1

 BEGIN  | READ(A)  | • • •  | WRITE(A)  | COMMIT

 Txn #2

 BEGIN  | • • •  | WRITE(A)  | • • • • • • •  | COMMIT
LOST UPDATE ANOMALY

**Txn #1**

BEGIN

READ(A)

... ...

WRITE(A)

COMMIT

**Txn #2**

BEGIN

... ...

WRITE(A)

... ...

COMMIT

Txn #2’s write to A will be lost even though it commits after Txn #1.
LOST UPDATE ANOMALY

Txn #1

BEGIN
READ(A)

... ...

WRITE(A)

COMMIT

Txn #2

BEGIN

... ...

WRITE(A)

COMMIT

Txn #2’s write to A will be lost even though it commits after Txn #1.

A **cursor lock** on A would prevent this problem (but not always).
SNAPSHOT ISOLATION (SI)

Guarantees that all reads made in a txn see a consistent snapshot of the database that existed at the time the txn started.
→ A txn will commit under SI only if its writes do not conflict with any concurrent updates made since that snapshot.

SI is susceptible to the Write Skew Anomaly
WRITE SKEW ANOMALY
WRITE SKEW ANOMALY
WRITE SKEW ANOMALY

Txn #1
Change white marbles to black.

Txn #2
Change black marbles to white.
WRITE SKEW ANOMALY

**Txn #1**
Change white marbles to black.

**Txn #2**
Change black marbles to white.
WRITE SKEW ANOMALY

Txn #1
Change white marbles to black.

Txn #2
Change black marbles to white.
**WRITE SKEW ANOMALY**

*Txn #1*
Change white marbles to black.

*Txn #2*
Change black marbles to white.
WRITE SKEW ANOMALY

Txn #1
Change white marbles to black.

Txn #2
Change black marbles to white.
WRITE SKEW ANOMALY
WRITE SKEW ANOMALY

**Txn #1**
Change white marbles to black.

**Txn #2**
Change black marbles to white.
**ISOLATION LEVEL HIERARCHY**

- **Serializable**
  - **Repeatable Reads**
  - **Snapshot Isolation**
  - **Cursor Stability**
- **Read Committed**
- **Read Uncommitted**
MULTI-VERSION CONCURRENCY CONTROL

Timestamp-ordering scheme that maintains multiple versions of database objects:
→ 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.
MULTI-VERSION CONCURRENCY CONTROL

_txn #1_

BEGIN
READ(A)
WRITE(B)

• • • •  • • •

WRITE(A)

COMMIT
MULTI-VERSION CONCURRENCY CONTROL

**Txn #1**

```
BEGIN
  READ(A)
  WRITE(B)

• • • •  • • •

COMMIT
  WRITE(A)
```

<table>
<thead>
<tr>
<th>Record</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>10000</td>
</tr>
<tr>
<td>B₁</td>
<td>10000</td>
</tr>
</tbody>
</table>
MULTI-VERSION CONCURRENCY CONTROL

#1

BEGIN
READ(A)
WRITE(B)

• • • •  • • •

Record | Write Timestamp
--- | ---
A₁ | 10000
B₁ | 10000

COMMIT
MULTI-VERSION CONCURRENCY CONTROL

10001

BEGIN

READ(A)
WRITE(B)

COMMIT

<table>
<thead>
<tr>
<th>Record</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>10000</td>
</tr>
<tr>
<td>B₁</td>
<td>10000</td>
</tr>
</tbody>
</table>
MULTI-VERSION CONCURRENCY CONTROL

**Transaction #1**

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

**Record Table**

<table>
<thead>
<tr>
<th>Record</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>10000</td>
</tr>
<tr>
<td>B₁</td>
<td>10000</td>
</tr>
</tbody>
</table>

**Database**
MULTI-VERSION CONCURRENCY CONTROL

Txn #1
BEGIN
READ(A)
WRITE(B)

• • • •  • • •

Record

<table>
<thead>
<tr>
<th>Record</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>10000</td>
</tr>
<tr>
<td>B₁</td>
<td>10000</td>
</tr>
<tr>
<td>B₂</td>
<td>10001</td>
</tr>
</tbody>
</table>

COMMIT

10001
MULTI-VERSION CONCURRENCY CONTROL

#1

BEGIN
READ(A)
WRITE(B)

• • • •  • • •

Record | Write Timestamp
-------|------------------
A₁     | 10000
B₁     | 10000
B₂     | 10001
MULTI-VERSION CONCURRENCY CONTROL

### Example Transaction 1

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

**Intermediate State**

<table>
<thead>
<tr>
<th>Record</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>10000</td>
</tr>
<tr>
<td>B₁</td>
<td>10000</td>
</tr>
<tr>
<td>B₂</td>
<td>10001</td>
</tr>
</tbody>
</table>

**COMMIT**
- **WRITE(A)**
MULTI-VERSION CONCURRENCY CONTROL

**Txn #1**

BEGIN
READ(A)
WRITE(B)

• • • •  • • •

Record  | Write Timestamp
---|---
A₁ | 10000
B₁ | 10000
B₂ | 10001
A₂ | 10003

COMMIT
MULTI-VERSION CONCURRENCY CONTROL

#1

BEGIN
• • • •  • • •
COMMIT

READ(A)  WRITE(B)

WRITE(A)

<table>
<thead>
<tr>
<th>Record</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>10000</td>
</tr>
<tr>
<td>B₁</td>
<td>10000</td>
</tr>
<tr>
<td>B₂</td>
<td>10001</td>
</tr>
<tr>
<td>A₂</td>
<td>10003</td>
</tr>
</tbody>
</table>
**MULTI-VERSION CONCURRENCY CONTROL**

<table>
<thead>
<tr>
<th>Record</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>10000</td>
</tr>
<tr>
<td>B₁</td>
<td>10000</td>
</tr>
<tr>
<td>B₂</td>
<td>10001</td>
</tr>
<tr>
<td>A₂</td>
<td>10003</td>
</tr>
</tbody>
</table>
MODERN MVCC

Microsoft Hekaton (SQL Server)
TUM HyPer
HPI HYRISE
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 committedtxn that created it.

→ **POINTER**: Location of the next version in the chain.
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>

INDEX
HEKATON: OPERATIONS

BEGIN @ 25

```
<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

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

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

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

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

BEGIN @ 30
Read “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”
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**: Thetxn is executing read/write operations.
→ **VALIDATING**: Thetxn 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

Source: Paul Larson
Get txn start timestamp, set state to **ACTIVE**
HEKATON: TRANSACTION LIFECYCLE

Get txn start timestamp, set state to **ACTIVE**

Perform normal processing
→ Tracktxn’s read set, scan set, and write set.

Source: Paul Larson
**HEKATON: TRANSACTION LIFECYCLE**

**Txn events**
- Begin
- Precommit

**Txn phases**
- Get txn start timestamp, set state to **ACTIVE**
- Perform normal processing
  - → Track txn’s read set, scan set, and write set.
- Get txn end timestamp, set state to **VALIDATING**

Source: Paul Larson
Get txn start timestamp, set state to \textbf{ACTIVE}

Perform normal processing
→ Track txn’s read set, scan set, and write set.

Get txn end timestamp, set state to \textbf{VALIDATING}

Validate reads and scans
→ If validation OK, write new versions to redo log
HEKATON: TRANSACTION LIFECYCLE

Get txn start timestamp, set state to **ACTIVE**

Begin

Precommit

Commit

Perform normal processing
→ Track txn’s read set, scan set, and write set.

Get txn end timestamp, set state to **VALIDATING**

Normal processing

Validation

→ If validation OK, write new versions to redo log

Set txn state to **COMMITTED**

Source: Paul Larson

CMU 15-721 (Spring 2016)
HEKATON: TRANSACTION LIFECYCLE

**Txn events**  
Begin  
Precommit  
Commit

**Txn phases**  
Normal processing  
Validation  
Post-processing

- **Begin**  
  Get txn start timestamp, set state to **ACTIVE**
- **Precommit**  
  Perform normal processing  
  → Track txn’s read set, scan set, and write set.
  
  Get txn end timestamp, set state to **VALIDATING**
- **Commit**  
  Validate reads and scans  
  → If validation OK, write new versions to redo log
  
  Set txn state to **COMMITTED**
  
  Fix up version timestamps  
  → Begin TS in new versions, end TS in old versions

Source: [Paul Larson](http://www.cs.cmu.edu/~plarson/)

CMU 15-721 (Spring 2016)
HEKATON: TRANSACTION LIFECYCLE

<table>
<thead>
<tr>
<th>Txn events</th>
<th>Txn phases</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin</td>
<td>Normal processing</td>
<td>Get txn start timestamp, set state to <strong>ACTIVE</strong></td>
</tr>
<tr>
<td></td>
<td>Validation</td>
<td>Perform normal processing</td>
</tr>
<tr>
<td></td>
<td>Post-processing</td>
<td>→ Track txn’s read set, scan set, and write set.</td>
</tr>
<tr>
<td>Precommit</td>
<td></td>
<td>Get txn end timestamp, set state to <strong>VALIDATING</strong></td>
</tr>
<tr>
<td>Commit</td>
<td></td>
<td>Validate reads and scans</td>
</tr>
<tr>
<td></td>
<td></td>
<td>→ If validation OK, write new versions to redo log</td>
</tr>
<tr>
<td>Terminate</td>
<td></td>
<td>Set txn state to <strong>COMMITTED</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fix up version timestamps</td>
</tr>
<tr>
<td></td>
<td></td>
<td>→ Begin TS in new versions, end TS in old versions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set txn state to <strong>TERMINATED</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Remove from txn map</td>
</tr>
</tbody>
</table>

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: 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

Throughput (txn/sec)

# Threads
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)
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
MVCC DESIGN CHOICES

Version Chains
Version Storage
Garbage Collection
VERSION CHAINS

Approach #1: Oldest-to-Newest
→ Just append new version to end of the chain.
→ Have to traverse chain on look-ups.

Approach #2: Newest-to-Oldest
→ 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.
VERSION STORAGE

Approach #1: Insert Method
→ New versions are added as new tuples to the table.

Approach #2: Delta Method
→ Copy the current version to a separate storage location and then overwrite it with the new data.
→ Rollback segment with deltas, Time-travel table
ROLLBACK SEGMENTS

**Main Data Table**

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>END</th>
<th>ATTR1</th>
<th>ATTR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td>John</td>
<td>$100</td>
</tr>
</tbody>
</table>
## ROLLBACK SEGMENTS

### Main Data Table

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>END</th>
<th>ATTR1</th>
<th>ATTR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td>John</td>
<td>$100</td>
</tr>
</tbody>
</table>
ROLLBACK SEGMENTS

Main Data Table

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>END</th>
<th>ATTR1</th>
<th>ATTR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td>John</td>
<td>$100</td>
</tr>
</tbody>
</table>

On every update, copy the old version to the rollback segment and overwrite the tuple in the main data table.
On every update, copy the old version to the rollback segment and overwrite the tuple in the main data table.
On every update, copy the old version to the rollback segment and overwrite the tuple in the main data table.

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>END</th>
<th>ATTR1</th>
<th>ATTR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td>John</td>
<td>$100</td>
</tr>
</tbody>
</table>

Rollback Segment (Per Tuple)

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>END</th>
<th>DELTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td>(ATTR2→$100)</td>
</tr>
</tbody>
</table>
On every update, copy the old version to the rollback segment and overwrite the tuple in the main data table.

### Main Data Table

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>END</th>
<th>ATTR1</th>
<th>ATTR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>25</td>
<td>John</td>
<td>$110</td>
</tr>
</tbody>
</table>

### Rollback Segment (Per Tuple)

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>END</th>
<th>DELTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td>((\text{ATTR2} \rightarrow $100))</td>
</tr>
</tbody>
</table>
On every update, copy the old version to the rollback segment and overwrite the tuple in the main data table.
On every update, copy the old version to the rollback segment and overwrite the tuple in the main data table.

Txns can recreate old versions by applying the delta in reverse order.
GARBAGE COLLECTION

Approach #1: Vacuum Thread
→ Use a separate background thread to find old versions and delete them.

Approach #2: Cooperative Threads
→ Worker threads remove old versions that they encounter during scans.

GC overhead depends on read/write ratio
→ Hekaton authors report about a 15% overhead on a write-heavy workload. Typically much less.
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

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.
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>(ATTR2→$100)</td>
</tr>
<tr>
<td>IceT</td>
<td>$200</td>
<td>(ATTR2→$139)</td>
</tr>
<tr>
<td>B.I.G</td>
<td>$150</td>
<td>(ATTR2→$122)</td>
</tr>
<tr>
<td>DrDre</td>
<td>$99</td>
<td>(ATTR2→$199)</td>
</tr>
</tbody>
</table>

Rollback Segment (Per Txn)

- **Txn 2^{63}**
  - (ATTR2→$100)
  - (ATTR2→$139)

- **Txn 2^{63}+1**
  - (ATTR2→$122)

- **Txn 123**
  - (ATTR2→$199)
HYRISE MVCC

Insert Method (no rollback segment)
Oldest-to-Newest
No garbage collection.
All updates are executed as DELETE/INSERT.
SAP HANA MVCC

Insert Method (no rollback segment)
Background GC thread (optional)

It’s not clear what else they are doing...
PARTING THOUGHTS

MVCC is currently the best approach for supporting txns in mixed workloads
→ Readers are not blocked by writers.

HyPer’s MVCC makes a lot of good decisions for HTAP workloads.
NEXT CLASS

Stored Procedures
Optimistic Concurrency Control