Lecture 18: Case Studies

Recap

Multi-Version Concurrency Control

- 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.

Multi-Version Concurrency Control

- Writers don't block readers. Readers don't block writers.
- Read-only txns can read a **consistent snapshot** without acquiring locks or txn ids.
  - ▶ Use timestamps to determine visibility.
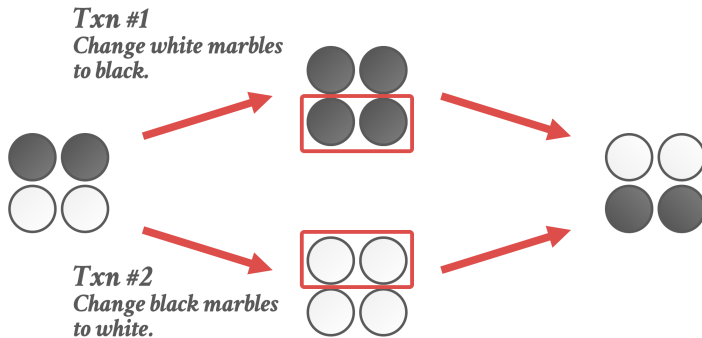- Easily support time-travel queries.

Today's Agenda

- MVCC Protocols
- Microsoft Hekaton (SQL Server)
- TUM HyPer
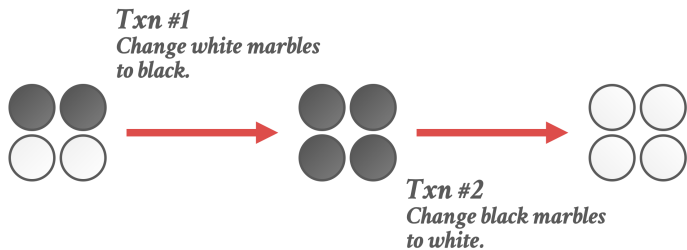- SAP HANA
- CMU Cicada

MVCC Protocols

Snapshot Isolation (SI)

- When a txn starts, it sees a consistent snapshot of the database that existed when that the txn started.
    - ▶ No torn writes from active txns.
    - ▶ If two txns update the same object, then first writer wins.
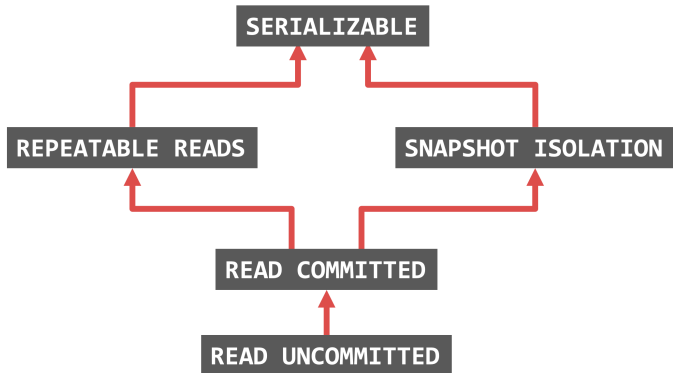- SI is susceptible to the **Write Skew Anomaly**.

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
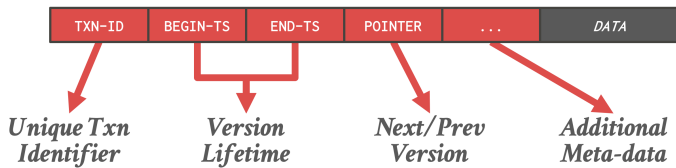*Change black marbles*
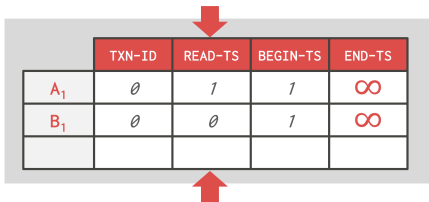*to white.*

Isolation Level Hierarchy

Concurrency Control Protocol

- **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.

Tuple Format

TXN-ID	BEGIN-TS	END-TS	POINTER	...	*DATA*

Unique Txn Identifier

Version Lifetime

Next/Prev Version

Additional Meta-data

Timestamp Ordering (MVTO)

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	1	1	∞
B_1	0	0	1	∞

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

Timestamp Ordering (MVTO)

Thread #1
$T_{id}=10$

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	1	1	∞
B_1	0	0	1	∞

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

Txn can read version if the latch is unset and its T_{id} is between *begin-ts* and *end-ts*.

Timestamp Ordering (MVTO)

Thread #1
$T_{id}=10$

READ(A)

WRITE(B)

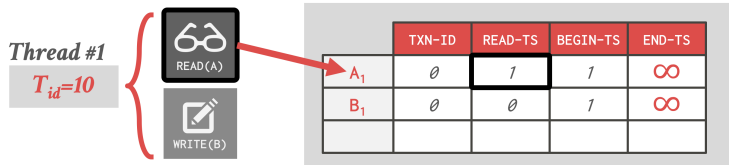|	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	1	1	∞
B_1	0	0	1	∞

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

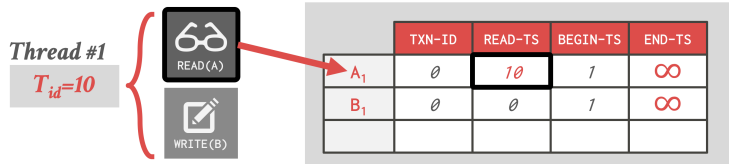Txn can read version if the latch is unset and its T_{id} is between **begin-ts** and **end-ts**.

Timestamp Ordering (MVTO)

Thread #1
T_{id}=10

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	10	1	∞
B_1	0	0	1	∞

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

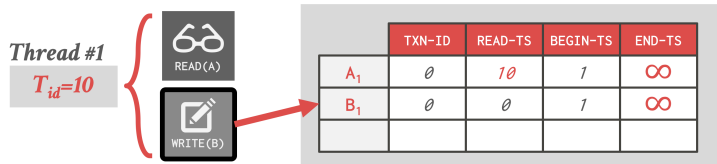Txn can read version if the latch is unset and its T_{id} is between *begin-ts* and *end-ts*.

Timestamp Ordering (MVTO)

Thread #1
$T_{id}=10$

READ(A)

WRITE(B)

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	10	1	∞
B_1	0	0	1	∞

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

Txn can read version if the latch is unset and its T_{id} is between ***begin-ts*** and ***end-ts***.

Txn creates a new version if no other txn holds latch and T_{id} is greater than ***read-ts***.

Timestamp Ordering (MVTO)

Thread #1
$T_{id}=10$

READ(A)

WRITE(B)

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	10	1	∞
B_1	10	0	1	∞

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

Txn can read version if the latch is unset and its T_{id} is between **begin-ts** and **end-ts**.

Txn creates a new version if no other txn holds latch and T_{id} is greater than **read-ts**.

Timestamp Ordering (MVTO)

Thread #1
$T_{id}=10$

READ(A)

WRITE(B)

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	10	1	∞
B_1	10	0	1	10
B_2	10	0	10	∞

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

Txn can read version if the latch is unset and its T_{id} is between **begin-ts** and **end-ts**.

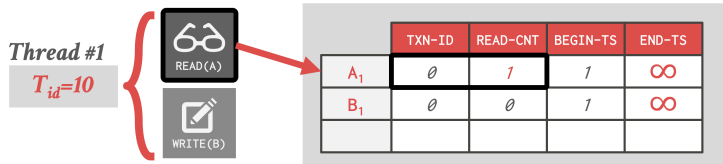Txn creates a new version if no other txn holds latch and T_{id} is greater than **read-ts**.

Two-Phase Locking (MV2PL)

Thread #1
$T_{id}=10$

READ(A)

WRITE(B)

	TXN-ID	READ-CNT	BEGIN-TS	END-TS
A_1	0	0	1	∞
B_1	0	0	1	∞

Txns use the tuple's **read-cnt** field as SHARED lock. Use **txn-id** and **read-cnt** together as EXCLUSIVE lock.

Two-Phase Locking (MV2PL)

Thread #1
$T_{id}=10$

READ(A)

WRITE(B)

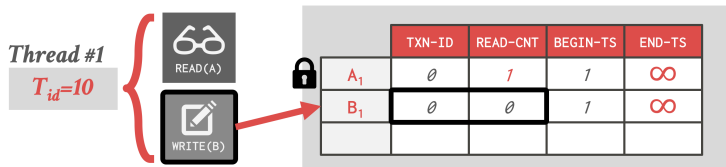|	TXN-ID	READ-CNT	BEGIN-TS	END-TS
A_1	0	1	1	∞
B_1	0	0	1	∞

Txns use the tuple's **read-cnt** field as SHARED lock. Use **txn-id** and **read-cnt** together as EXCLUSIVE lock.

If **txn-id** is zero, then the txn acquires the SHARED lock by incrementing the **read-cnt** field.

Two-Phase Locking (MV2PL)

Thread #1
$T_{id}=10$

	TXN-ID	READ-CNT	BEGIN-TS	END-TS
A_1	0	1	1	∞
B_1	0	0	1	∞

Txns use the tuple's **read-cnt** field as SHARED lock. Use **txn-id** and **read-cnt** together as EXCLUSIVE lock.

If **txn-id** is zero, then the txn acquires the SHARED lock by incrementing the **read-cnt** field.

If both **txn-id** and **read-cnt** are zero, then txn acquires the EXCLUSIVE lock by setting both of them.

Two-Phase Locking (MV2PL)

Thread #1
$T_{id}=10$

READ(A)

WRITE(B)

	TXN-ID	READ-CNT	BEGIN-TS	END-TS
A_1	0	1	1	∞
B_1	10	1	1	∞

Txns use the tuple's *read-cnt* field as SHARED lock. Use *txn-id* and *read-cnt* together as EXCLUSIVE lock.

If *txn-id* is zero, then the txn acquires the SHARED lock by incrementing the *read-cnt* field.

If both *txn-id* and *read-cnt* are zero, then txn acquires the EXCLUSIVE lock by setting both of them.

Two-Phase Locking (MV2PL)

Thread #1
$T_{id}=10$

READ(A)

WRITE(B)

	TXN-ID	READ-CNT	BEGIN-TS	END-TS
A_1	0	1	1	∞
B_1	10	1	1	10
B_2	10	0	10	∞

Txns use the tuple's **read-cnt** field as SHARED lock. Use **txn-id** and **read-cnt** together as EXCLUSIVE lock.

If **txn-id** is zero, then the txn acquires the SHARED lock by incrementing the **read-cnt** field.

If both **txn-id** and **read-cnt** are zero, then txn acquires the EXCLUSIVE lock by setting both of them.

Observation
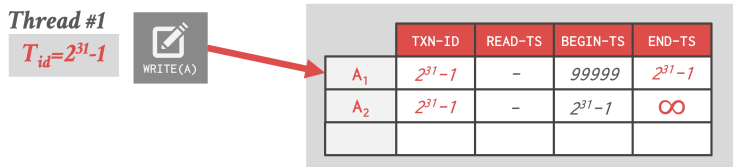
Thread #1
$T_{id}=2^{31}-1$

WRITE(A)

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	$2^{31}-1$	–	99999	$2^{31}-1$
A_2	$2^{31}-1$	–	$2^{31}-1$	∞

If the DBMS reaches the max value for its timestamps, it will have to wrap around and restart at one. This will make all previous versions be in the "future" from new transactions.

Observation

Thread #1
$T_{id}=2^{31}-1$

WRITE(A)

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	–	99999	$2^{31}-1$
A_2	0	–	$2^{31}-1$	∞

If the DBMS reaches the max value for its timestamps, it will have to wrap around and restart at one. This will make all previous versions be in the "future" from new transactions.

Observation

Thread #1
$T_{id}=2^{31}-1$

Thread #2
$T_{id}=1$

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	–	99999	$2^{31}-1$
A_2	1	–	$2^{31}-1$	1
A_3	1	–	1	∞

If the DBMS reaches the max value for its timestamps, it will have to wrap around and restart at one. This will make all previous versions be in the "future" from new transactions.

Observation

Thread #1
$T_{id} = 2^{31} - 1$

Thread #2
$T_{id} = 1$

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	–	99999	$2^{31} - 1$
A_2	0	–	$2^{31} - 1$	1
A_3	0	–	1	∞

If the DBMS reaches the max value for its timestamps, it will have to wrap around and restart at one. This will make all previous versions be in the "future" from new transactions.

PostgreSQL: Txn Id Wraparound

- Set a flag in each tuple header that says that it is **frozen** in the past. Any new txn id will always be newer than a frozen version.
- Runs the **vacuum** before the system gets close to this upper limit.
- Otherwise it must stop accepting new commands when the system gets close to the max txn id.

Microsoft Hekaton

Microsoft Hekaton

- Incubator project started in 2008 to create new OLTP engine for MSFT SQL Server (MSSQL).
  - ▶ Reference
- Had to integrate with MSSQL ecosystem.
- Had to support all possible OLTP workloads with predictable performance.
  - ▶ Single-threaded partitioning (*e.g.*, H-Store/VoltDB) works well for some applications but terrible for others.

31 / 82

Hekaton MVCC

- Each txn is assigned a timestamp when they **begin** (BeginTS) and when they **commit** (CommitTS).
- Each tuple contains **two timestamps** that represents their visibility and current state:
  - ▶ **BEGIN-TS**: The BeginTS of the active txn **or** the CommitTS of the committed txn that created it.
  - ▶ **END-TS**: The BeginTS of the active txn that created the next version or infinity or the CommitTS of the committed txn that created it.

Hekaton: Operations

Hekaton: Operations

Thread #1
Begin @ 25

READ(A)

Main Data Table

	BEGIN-TS	END-TS	VALUE	POINTER
A₁	10	20	$100	●
A₂	20	∞	$200	Ø

Hekaton: Operations

Hekaton: Operations

Thread #1
Begin @ 25

READ(A)

WRITE(A)

Main Data Table

	BEGIN-TS	END-TS	VALUE	POINTER
A₁	10	20	$100	●
A₂	20	∞	$200	∅
A₃	Txn@25	∞	$300	

Hekaton: Operations

Thread #1
Begin @ 25

READ(A)

WRITE(A)

Main Data Table

	BEGIN-TS	END-TS	VALUE	POINTER
A₁	10	20	$100	●
A₂	20	Txn@25	$200	●
A₃	Txn@25	∞	$300	

Hekaton: Operations

Hekaton: Operations

Thread #1
Begin @ 25
Commit @ 35

READ(A)

WRITE(A)

Main Data Table

	BEGIN-TS	END-TS	VALUE	POINTER
A₁	10	20	$100	●
A₂	20	35	$200	●
A₃	35	∞	$300	

Hekaton: Operations

Thread #1
Begin @ 25
Commit @ 35

READ(A)

WRITE(A)

Main Data Table

	BEGIN-TS	END-TS	VALUE	POINTER
A₁	10	20	$100	●
A₂	20	35	$200	●
A₃	35	∞	$300	

⟲ **REWIND**

Hekaton: Operations

Thread #1
Begin @ 25

READ(A)

WRITE(A)

Thread #2
Begin @ 30

READ(A)

Main Data Table

	BEGIN-TS	END-TS	VALUE	POINTER
A₁	10	20	$100	●
A₂	20	Txn@25	$200	●
A₃	Txn@25	∞	$300	

Hekaton: Operations

Thread #1
Begin @ 25

READ(A)

WRITE(A)

Thread #2
Begin @ 30

READ(A)

WRITE(A)

Main Data Table

	BEGIN-TS	END-TS	VALUE	POINTER
A₁	10	20	$100	●
A₂	20	Txn@25	$200	●
A₃	Txn@25	∞	$300	

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 Lifecycle

Txn Events	Txn Phases	
BEGIN		Get BeginTS, set state to **ACTIVE**
	Normal Processing	Track txn's read set, scan set, and write set.
PRECOMMIT		Get CommitTS, set state to **VALIDATING**
	Validation	Validate reads and scans → If validation OK, write new versions to redo log
COMMIT		Set txn state to **COMMITTED**
	Post-Processing	Update version timestamps → BeginTS in new versions, CommitTS in old versions
TERMINATE		Set txn state to **TERMINATED**
		Remove from txn map

Hekaton: Transaction Meta-Data

- **Read Set**
    - ▶ Pointers to physical versions returned to access method.
- **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 again to check result.
- **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
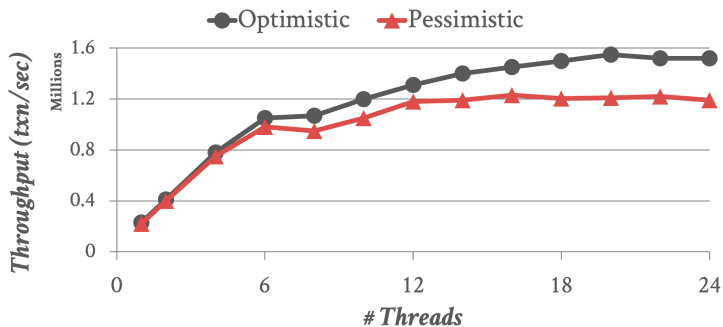
Hekaton: Lessons

- Use only **lock-free data structures**
    - ▶ No latches, spin locks, or critical sections
    - ▶ Indexes, txn map, memory alloc, garbage collector
    - ▶ **Example:** Bw-Trees
- Only one single serialization point in the DBMS to get the txn's begin and commit timestamp
    - ▶ Atomic Addition (CAS)

Observation

- 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

Hyper MVCC

- Column-store with delta record versioning.
- Reference
  - 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.

Hyper: Storage Architecture

Hyper: Storage Architecture

Main Data Table

ATTR1	ATTR2	Version Vector
Peter	$100	●
Qi	$200	●
Gaurav	$150	∅
Alice	$139	∅

Delta Storage (Per Txn)

Txn #1
(ATTR2→$199) ∅

Txn #2
(ATTR2→$122) ∅

Txn #3
(ATTR2→$100) ●
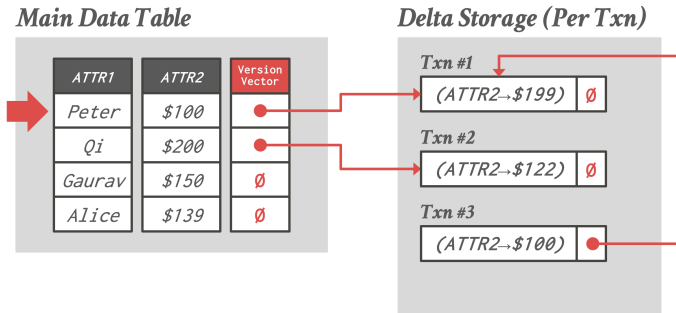
Hyper: Storage Architecture
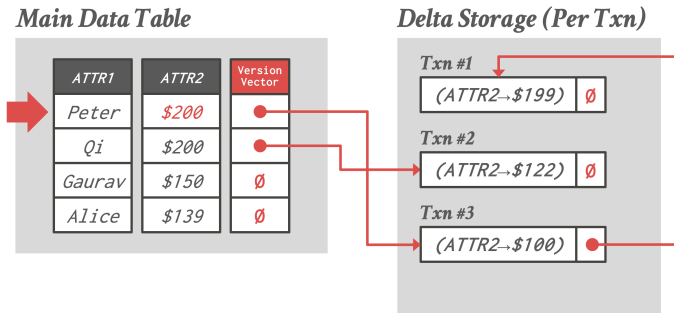
Hyper: Storage Architecture
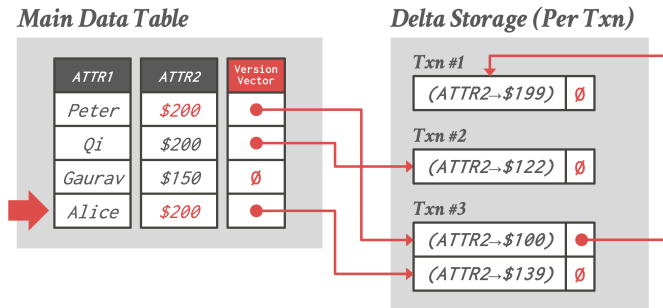
Hyper: Validation

- First-Writer Wins
  - ▶ If version vector is not null, then it always points to the last committed version.
  - ▶ Do not need to check whether write-sets overlap.
- Check the redo buffers 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

33>20 AND 33<30

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
```

99 IN (10,20,30)

33 IN (10,20,30)

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
```

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
```

TRUE
'IceCube' LIKE '%Ice%'

Delta Storage (Per Txn)

Txn #1001

$(attr2 \rightarrow 99)$

$(attr2 \rightarrow 33)$

Txn #1002

$(attr2 \rightarrow 122)$

Txn #1003

$(attr1 \rightarrow 'IceCube', attr2 \rightarrow 199)$

Hyper: Version Synopses

Main Data 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 Synopses

Main Data Table

Version Synopsis	ATTR1	ATTR2	Version Vector
[2, 5)	Peter	$100	Ø
	Qi	$200	Ø
	Gaurav	$150	● →
	Alice	$99	Ø
	Mark	$300	● →
	Rahul	$300	Ø
	Alex	$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.

SAP HANA

SAP HANA

- In-memory HTAP DBMS with time-travel version storage (N2O).
- Reference
    - Supports both optimistic and pessimistic MVCC.
    - Latest versions are stored in time-travel space.
    - Hybrid storage layout (row + columnar).
- Based on P*TIME, TREX, and MaxDB.
- First released in 2012.

SAP HANA: Version Storage

- Store the oldest version in the main data table.
- Each tuple maintains a flag to denote whether there exists newer versions in the version space.
- Maintain a separate hash table that maps record identifiers to the head of version chain.

SAP HANA: Version Storage

Main Data Table

RID	VERS?	VERSION	DATA
A	*True*	A_1	–
B	*False*	B_3	–
C	*True*	C_2	–
D	*True*	D_6	–

Version Storage
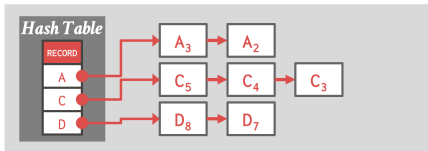
SAP HANA: Transactions

- Instead of embedding meta-data about the txn that created a version with the data, store a pointer to a context object.
  - Reads are slower because you must follow pointers.
  - Large updates are faster because it's a single write to update the status of all tuples.
- Store meta-data about whether a txn has committed in a separate object as well.

SAP HANA: Version Storage

Main Data Table

RID	VERS?	VERSION	DATA
A	*True*	A_1	–
B	*False*	B_3	–
C	*True*	C_2	–
D	*True*	D_6	–

Version Storage

Hash Table

RECORD

A → A_3 → A_2

C → C_5 → C_4 → C_3

D → D_8 → D_7

Thread #1

$T_{id} = 3$

WRITE(C)   WRITE(D)
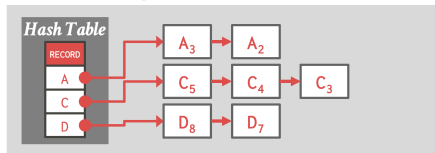
Txn Meta-Data

Txn Contexts

$T_{id}=1$ $T_{id}=2$ $T_{id}=3$

Group Commit Context

Group 1

SAP HANA: Version Storage
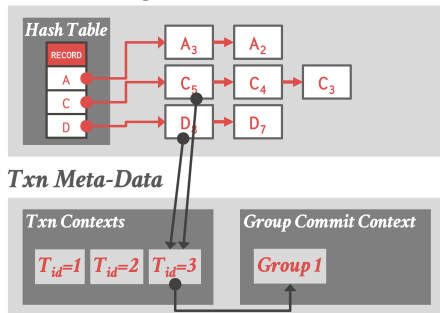
Cicada

MVCC Limitations

- **Computation & Storage Overhead**
  - ▶ Most MVCC schemes use indirection to search a tuple's version chain. This increases CPU cache misses.
  - ▶ Also requires frequent garbage collection to minimize the number versions that a thread must evaluate.
- **Shared Memory Writes**
  - ▶ Most MVCC schemes store versions in "global" memory in the heap without considering locality.
- **Timestamp Allocation**
  - ▶ All threads access single shared counter.

OCC Limitations

- **Frequent Aborts**
  - ▶ Txns will abort too quickly under high contention, causing high churn.
- **Extra Reads & Writes**
  - ▶ Each txn must copy tuples into their private workspace to ensure repeatable reads. It then has to check whether it read consistent data when it commits.
- **Index Contention**
  - ▶ Txns install "virtual" index entries to ensure unique-key invariants.

CMU Cicada

- In-memory OLTP engine based on optimistic MVCC with append-only storage (N2O).
- Reference
    - ▶ 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
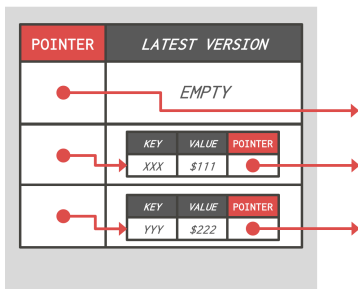
- Record Meta-data

- 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**
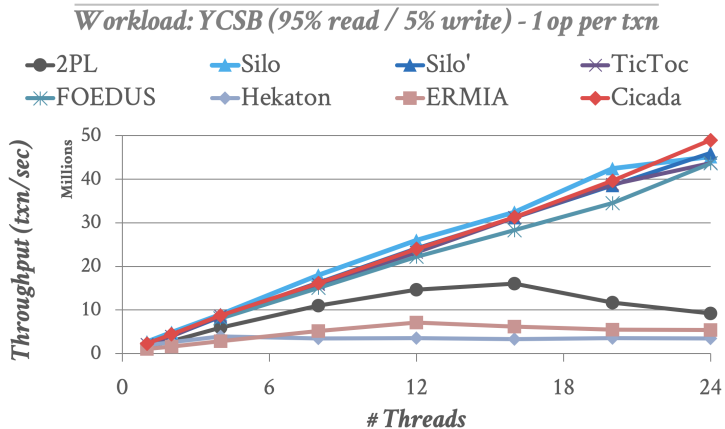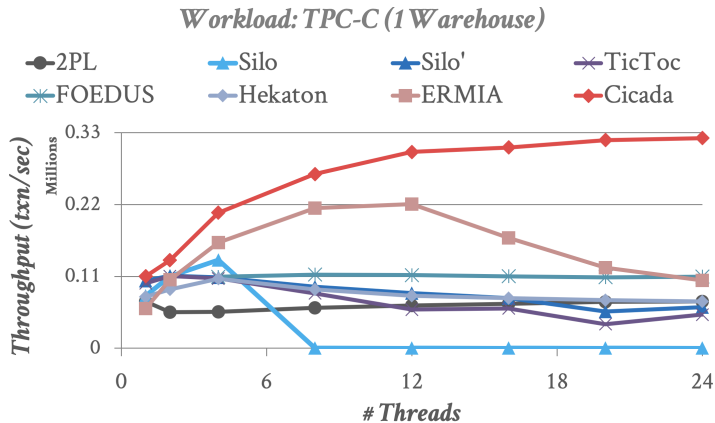 - ▶ Resume from last search location in version list.

Cicada: Index Storage
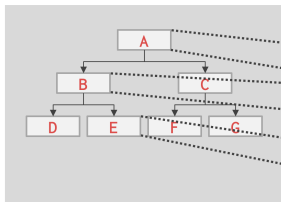
Cicada: Low Contention

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

Cicada: High Contention

Workload: TPC-C (1 Warehouse)

Conclusion

Parting Thoughts

- There are several other implementation factors for an MVCC DBMS beyond the four main design decisions that we discussed last class.
- Need to balance the trade-offs between indirection and performance.

Next Class

- Wellness Day
- Midterm Exam
- Query Optimization