15-721 DATABASE SYSTEMS

[Source]

Lecture #04 – Concurrency Control Part II

Andy Pavlo // Carnegie Mellon University // Spring 2016

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

ANSI ISOLATION LEVELS

Isolation (High→Low)

SERIALIZABLE
→ No phantoms, all reads repeatable, no dirty reads.

REPEATABLE READS
→ Phantoms may happen.

READ COMMITTED
→ Phantoms and unrepeatable reads may happen.

READ UNCOMMITTED
→ All of them may happen.

ISOLATION LEVEL HIERARCHY

SERIALIZABLE

↑

REPEATABLE READS

↑

READ COMMITTED

↑

READ UNCOMMITTED

ANSI ISOLATION LEVELS

	Default	*Maximum*
Actian Ingres 10.0/10S	SERIALIZABLE	SERIALIZABLE
Greenplum 4.1	READ COMMITTED	SERIALIZABLE
MySQL 5.6	REPEATABLE READS	SERIALIZABLE
MemSQL 1b	READ COMMITTED	READ COMMITTED
MS SQL Server 2012	READ COMMITTED	SERIALIZABLE
Oracle 11g	READ COMMITTED	SNAPSHOT ISOLATION
Postgres 9.2.2	READ COMMITTED	SERIALIZABLE
SAP HANA	READ COMMITTED	SERIALIZABLE
ScaleDB 1.02	READ COMMITTED	READ COMMITTED
VoltDB	SERIALIZABLE	SERIALIZABLE

Source: Peter Bailis

ANSI ISOLATION LEVELS

	Default	*Maximum*
Actian Ingres 10.0/10S	SERIALIZABLE	SERIALIZABLE
Greenplum 4.1	READ COMMITTED	SERIALIZABLE
MySQL 5.6	REPEATABLE READS	SERIALIZABLE
MemSQL 1b	READ COMMITTED	READ COMMITTED
MS SQL Server 2012	READ COMMITTED	SERIALIZABLE
Oracle 11g	READ COMMITTED	SNAPSHOT ISOLATION
Postgres 9.2.2	READ COMMITTED	SERIALIZABLE
SAP HANA	READ COMMITTED	SERIALIZABLE
ScaleDB 1.02	READ COMMITTED	READ COMMITTED
VoltDB	SERIALIZABLE	SERIALIZABLE

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**

A CRITIQUE OF ANSI SQL ISOLATION LEVELS
SIGMOD 1995

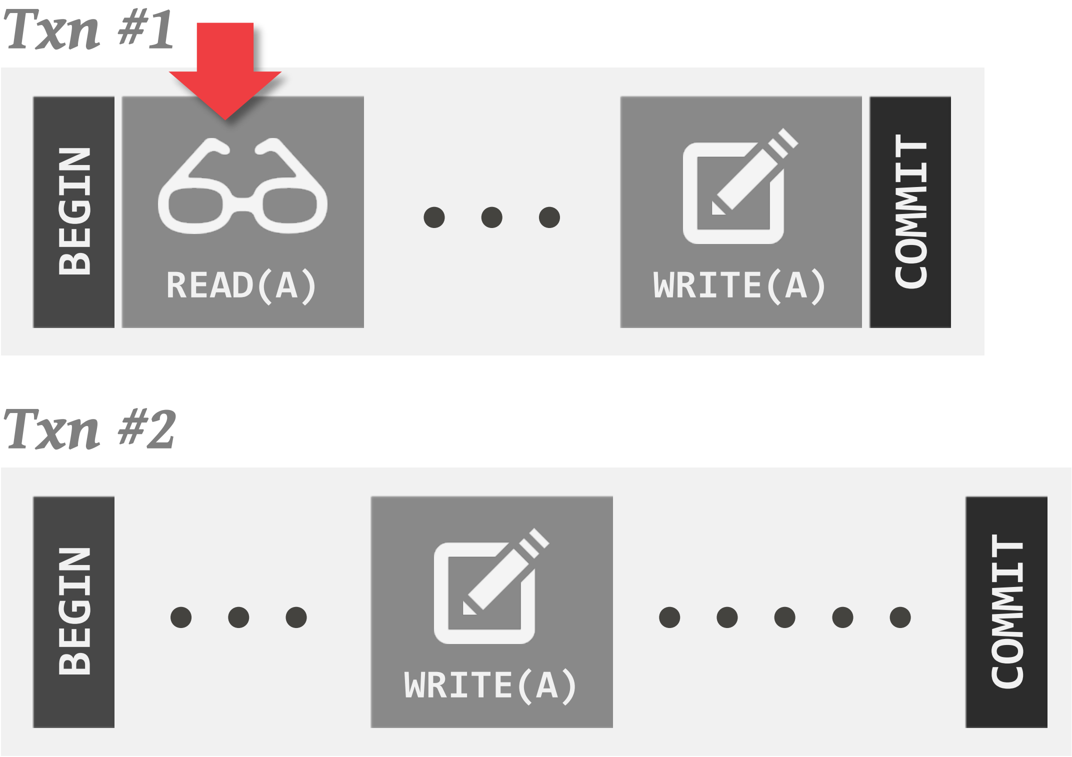
CURSOR STABILITY (CS)

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

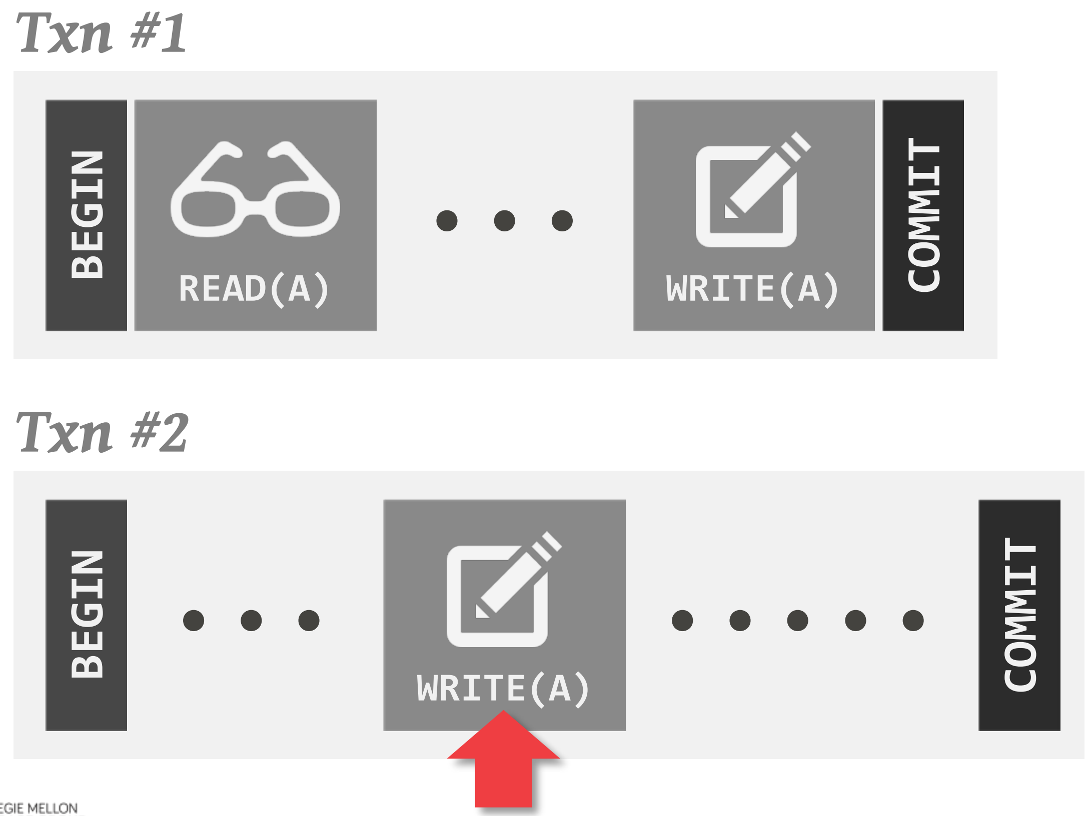
CS is a stronger isolation level in between `REPEATABLE READS` and `READ COMMITTED` that can (sometimes) prevent the **Lost Update Anomaly**.

LOST UPDATE ANOMALY

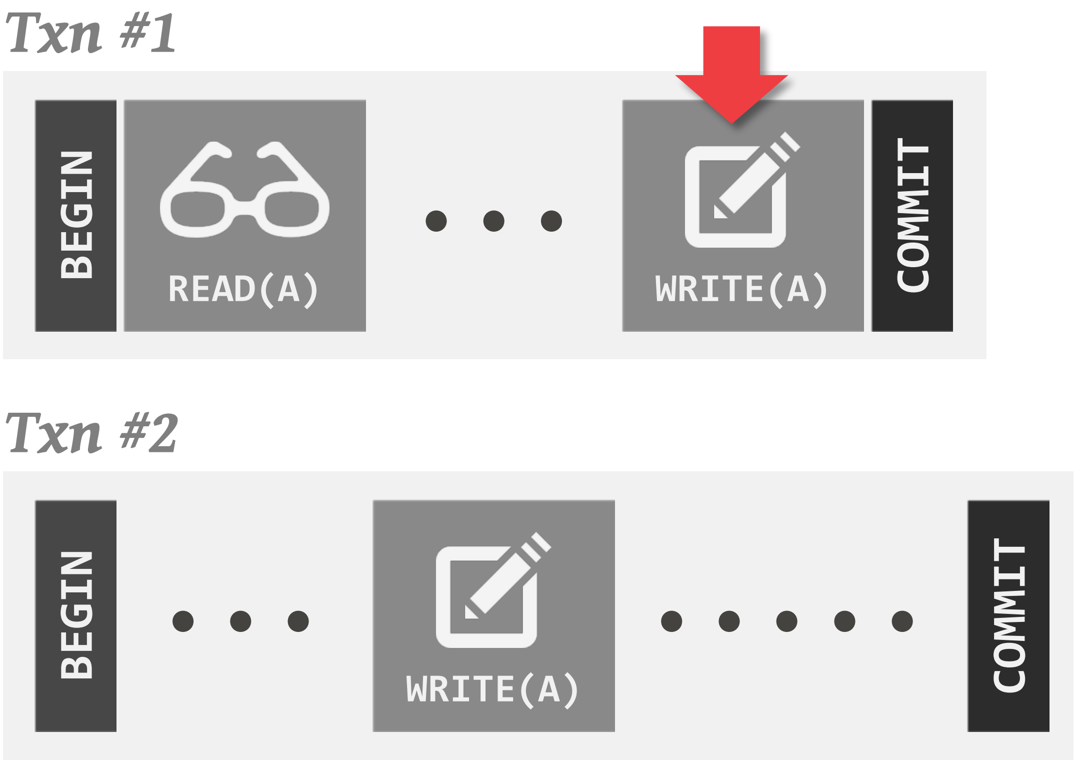
LOST UPDATE ANOMALY



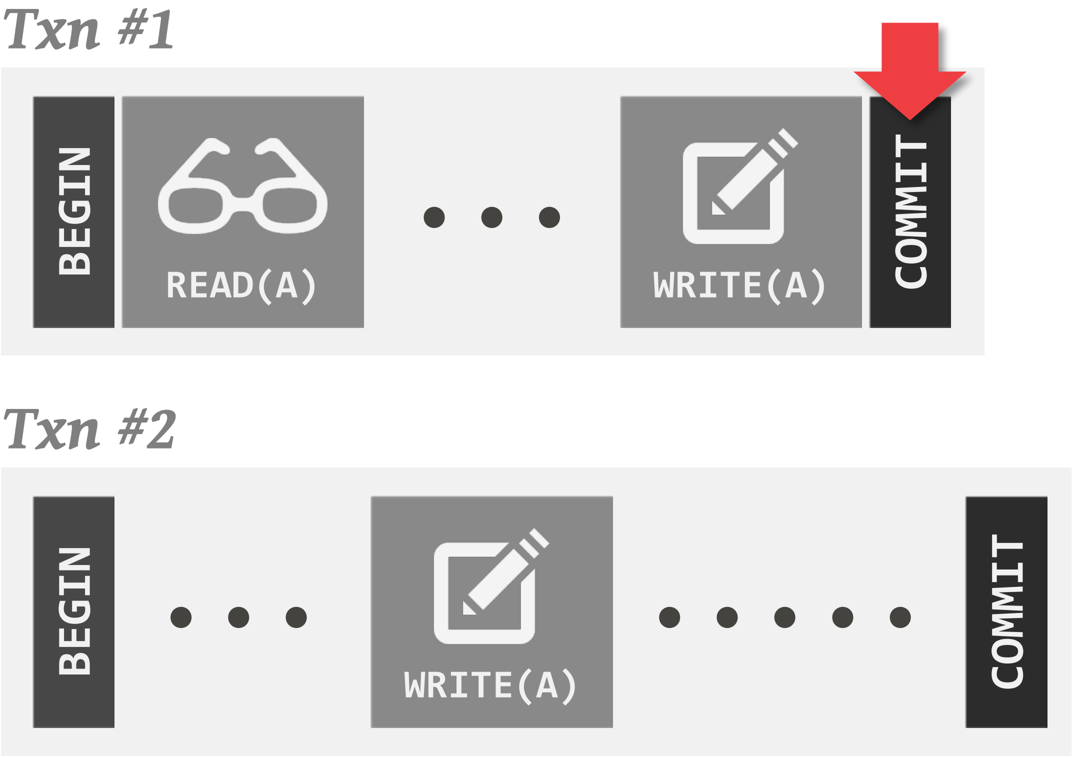
LOST UPDATE ANOMALY



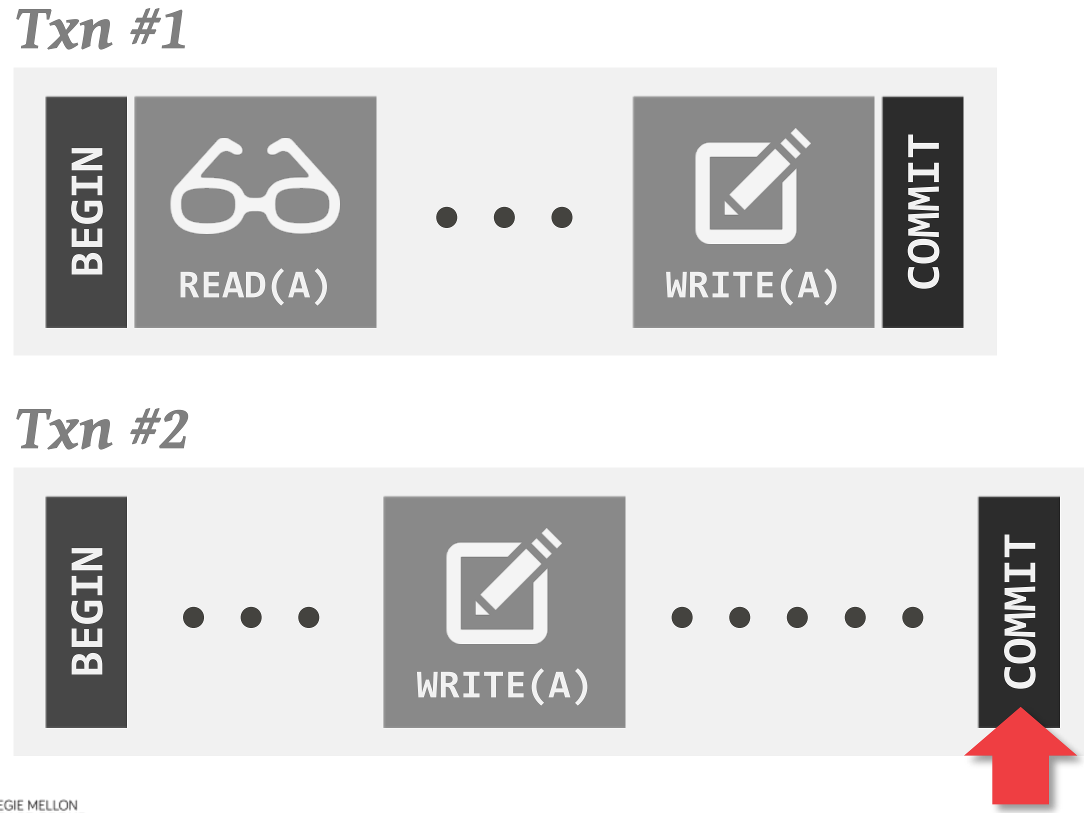
LOST UPDATE ANOMALY



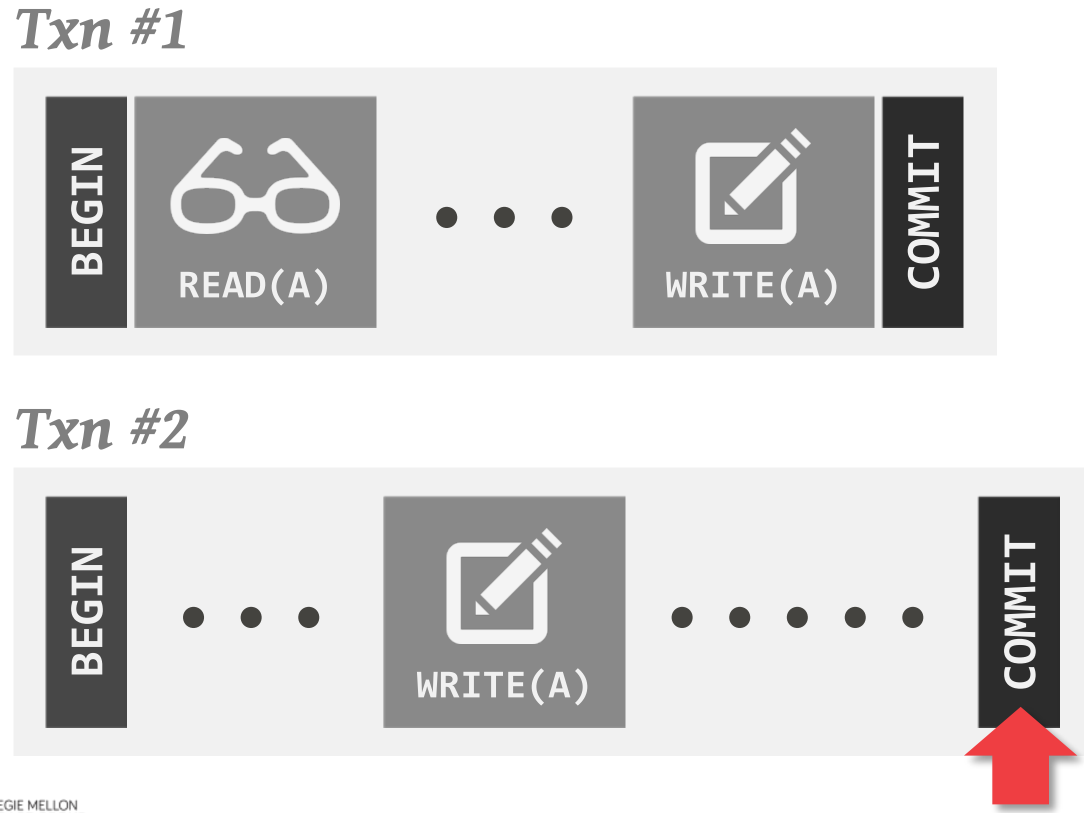
LOST UPDATE ANOMALY



LOST UPDATE ANOMALY

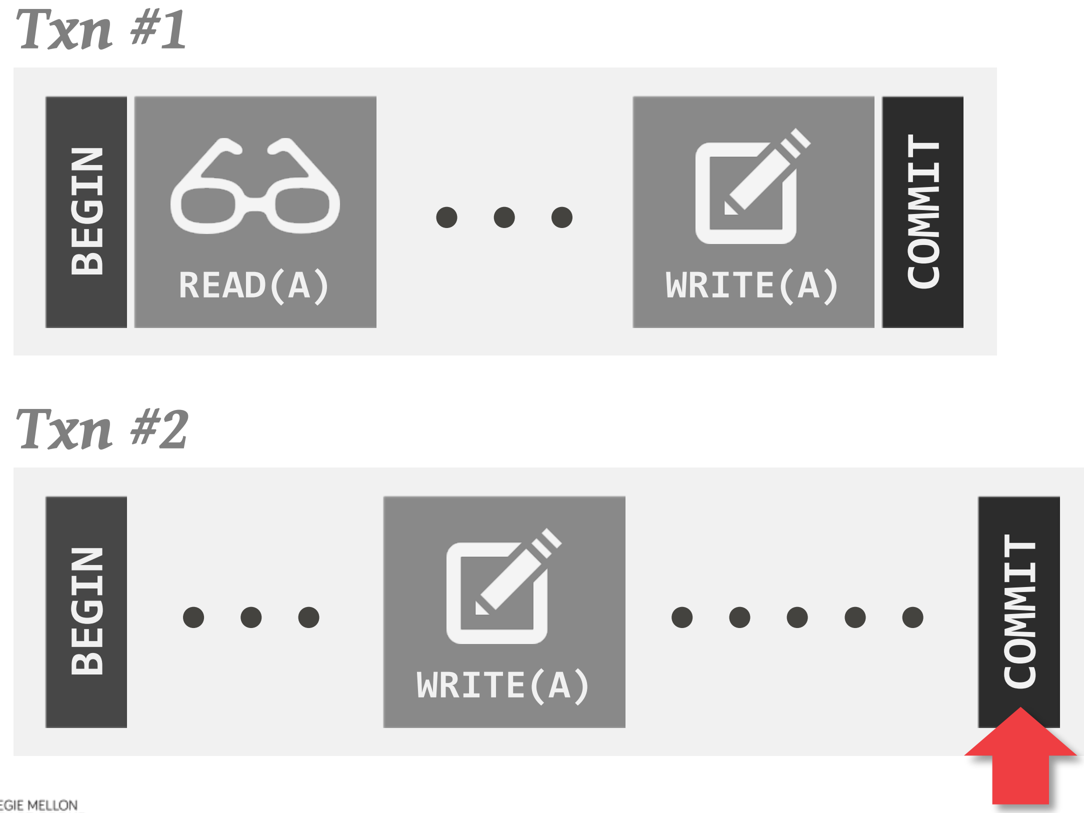


LOST UPDATE ANOMALY



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

LOST UPDATE ANOMALY



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

A **<u>cursor lock</u>** on **A** would prevent this problem (but not always).

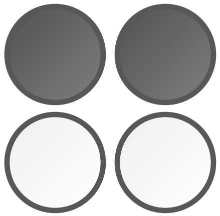
SNAPSHOT ISOLATION (SI)

Guarantees that all reads made in a txn see a consistent snapshot of the database that existed at the time the txn started.

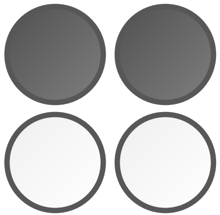
→ A txn will commit under SI only if its writes do not conflict with any concurrent updates made since that snapshot.

SI is susceptible to the **Write Skew Anomaly**

WRITE SKEW ANOMALY

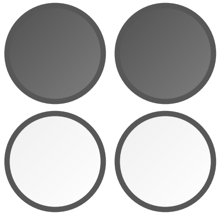


WRITE SKEW ANOMALY



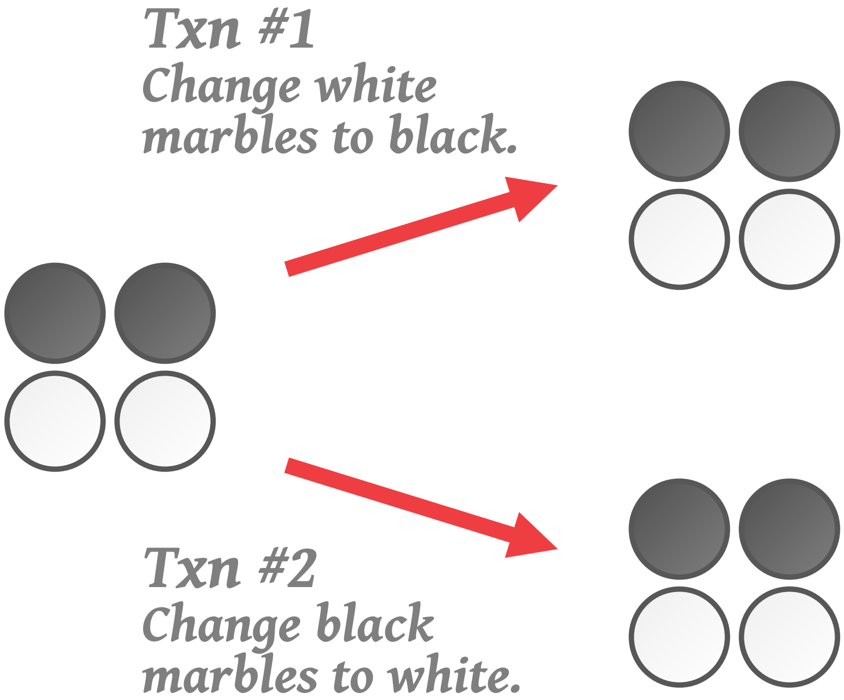
WRITE SKEW ANOMALY

Txn #1
Change white marbles to black.

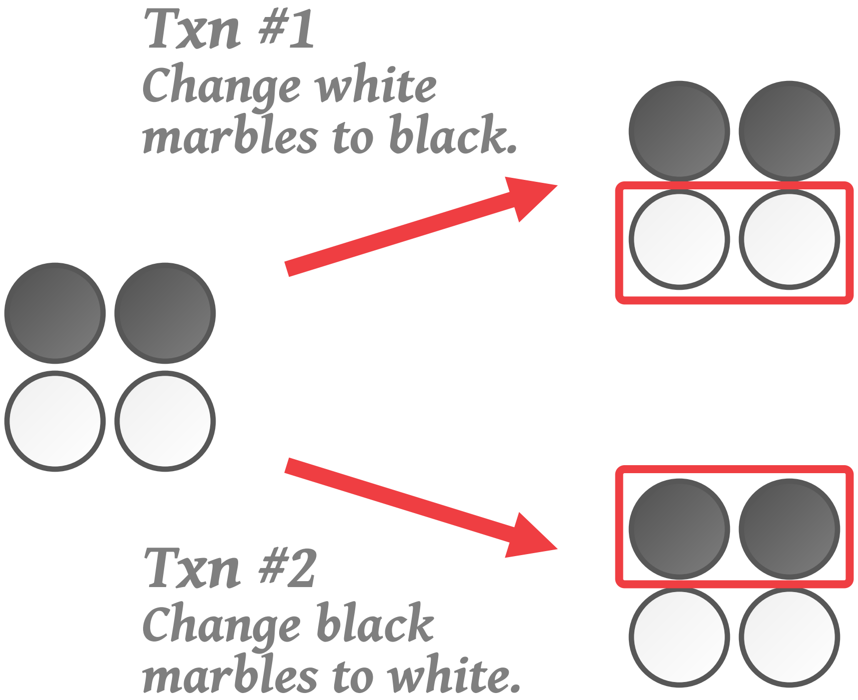


Txn #2
Change black marbles to white.

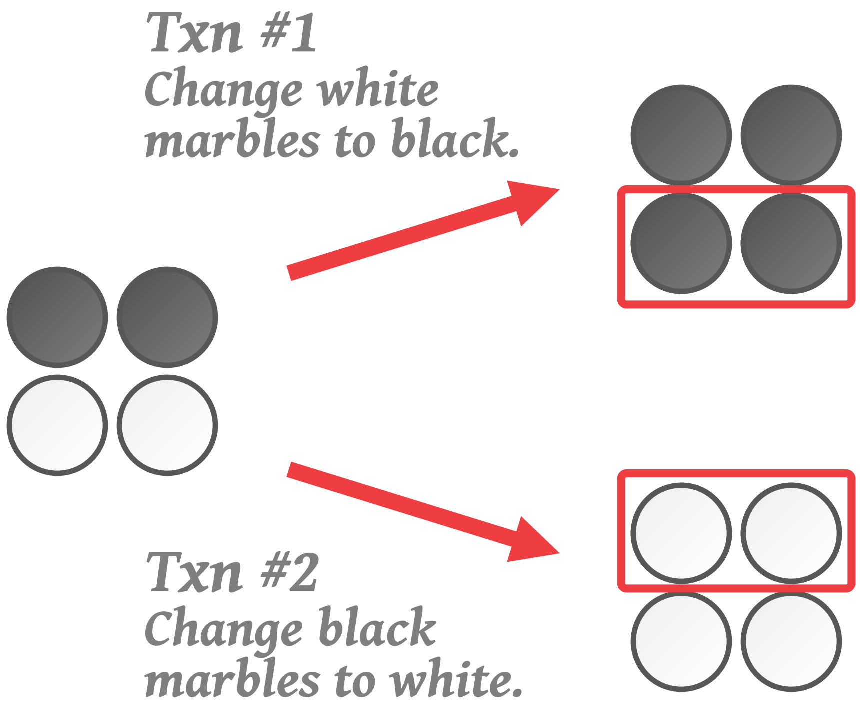
WRITE SKEW ANOMALY



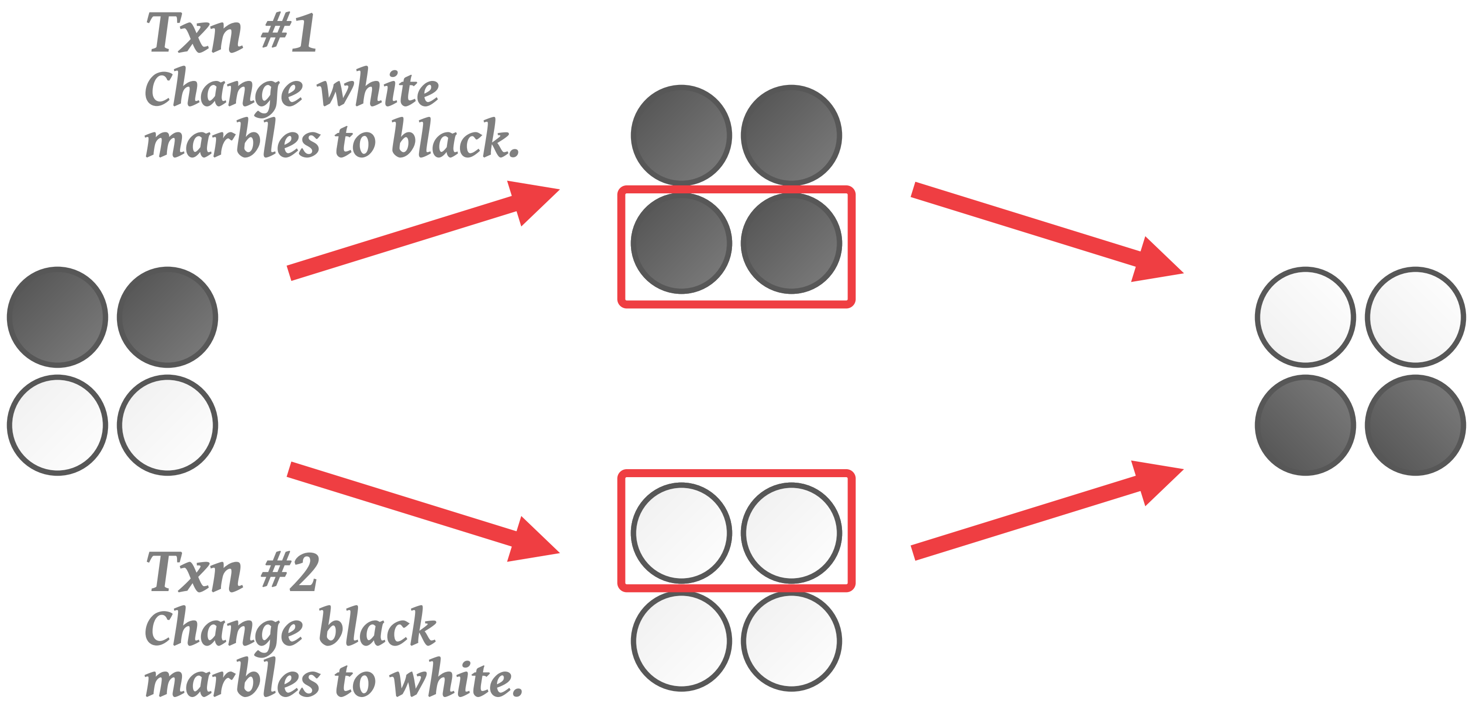
WRITE SKEW ANOMALY



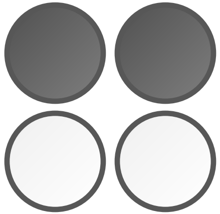
WRITE SKEW ANOMALY



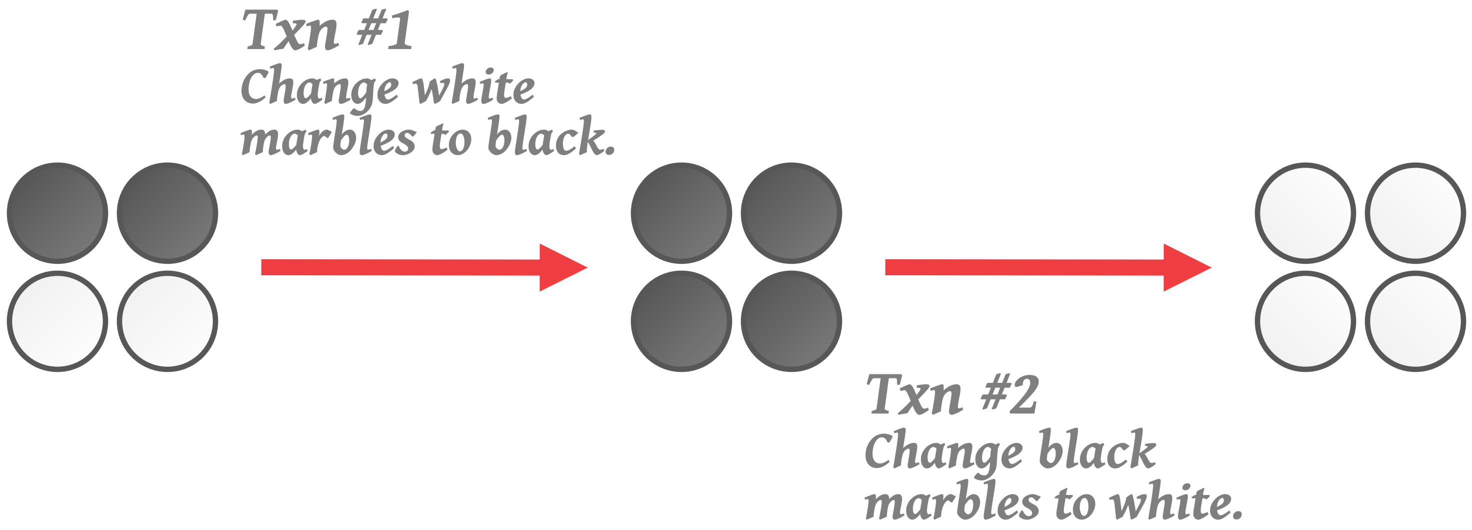
WRITE SKEW ANOMALY



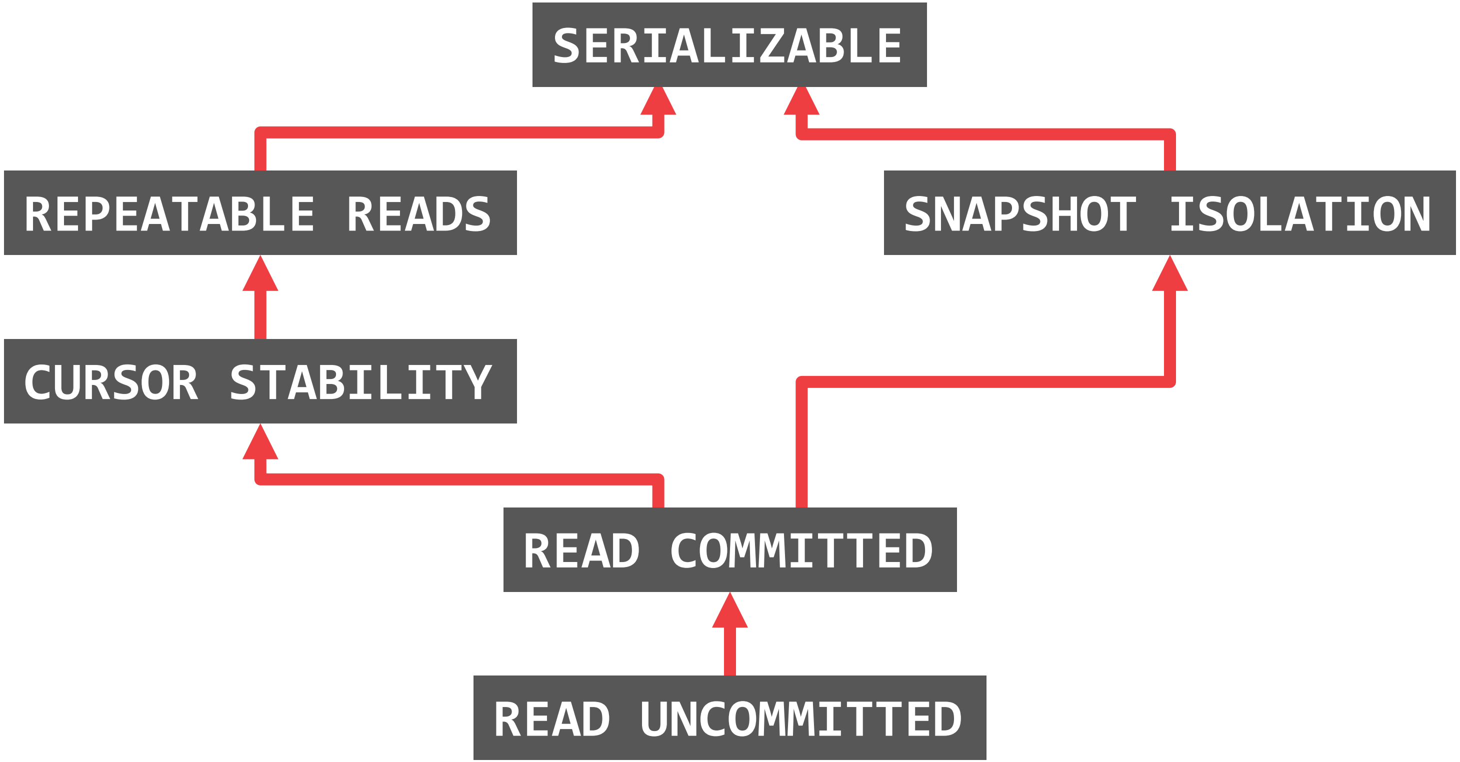
WRITE SKEW ANOMALY



WRITE SKEW ANOMALY



ISOLATION LEVEL HIERARCHY



MULTI-VERSION CONCURRENCY CONTROL

Timestamp-ordering scheme that maintains multiple versions of database objects:

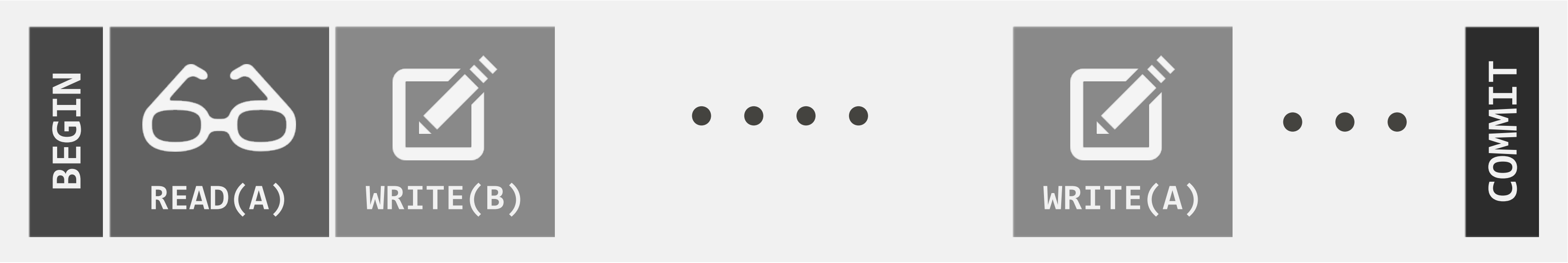
→ When a txn writes to an object, the DBMS creates a new version of that object.

→ When a txn reads an object, it reads the newest version that existed when the txn started.

First proposed in 1978 MIT PhD dissertation.

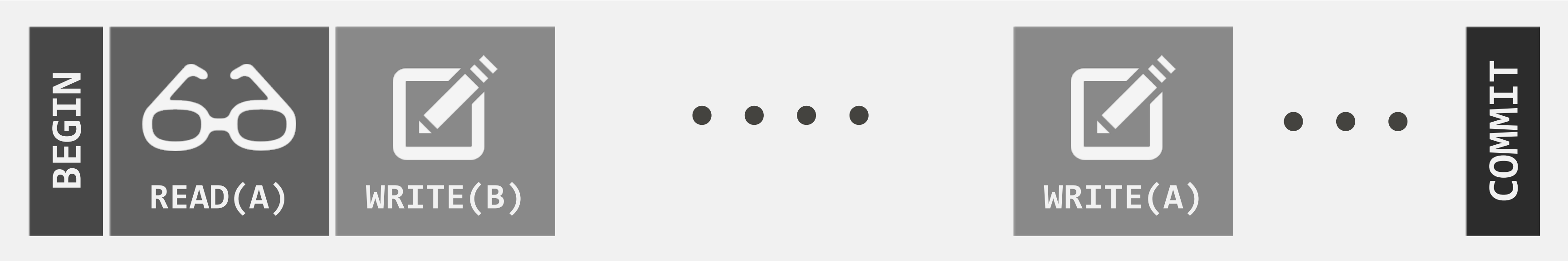
MULTI-VERSION CONCURRENCY CONTROL

Txn #1

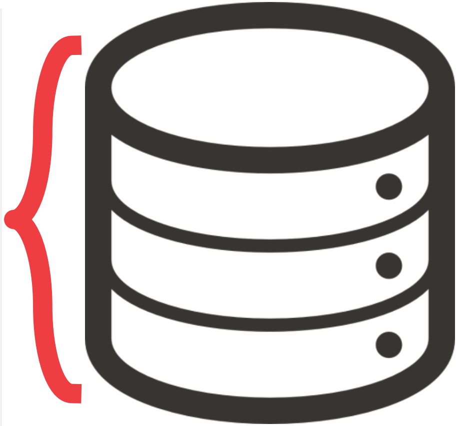


MULTI-VERSION CONCURRENCY CONTROL

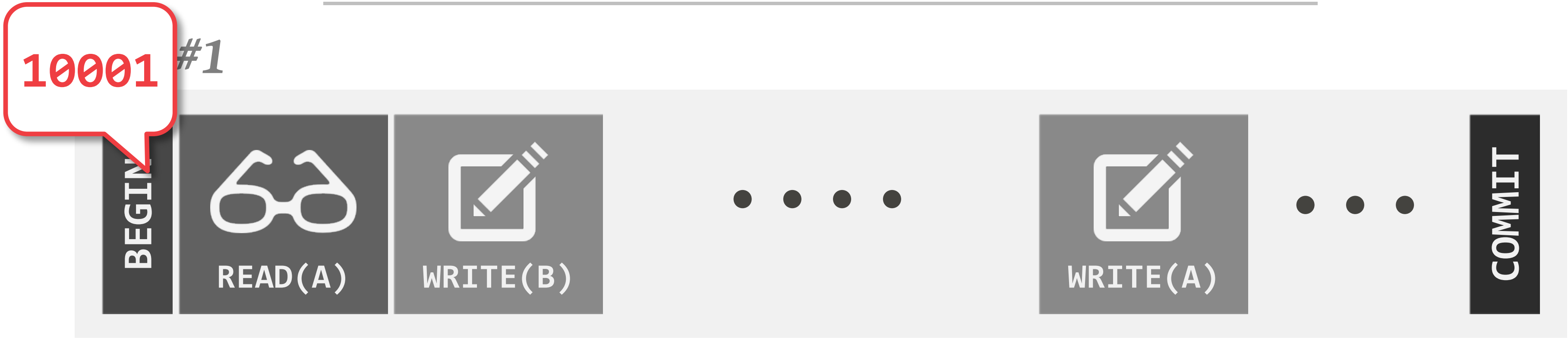
Txn #1



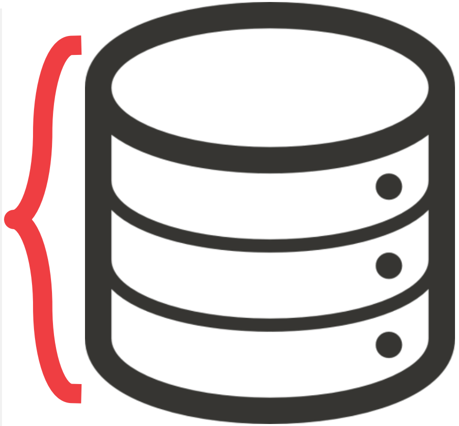
Record	Write Timestamp
A_1	10000
B_1	10000



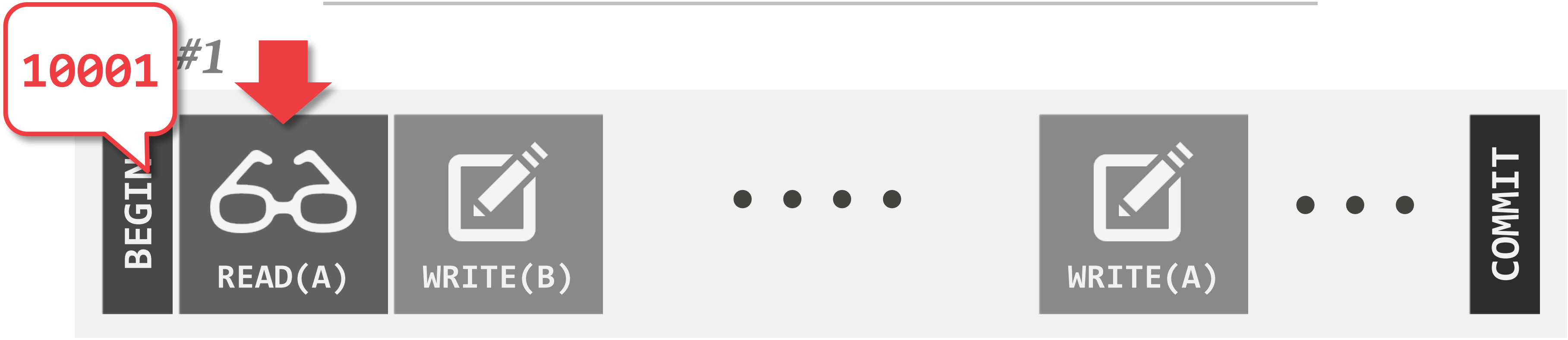
MULTI-VERSION CONCURRENCY CONTROL



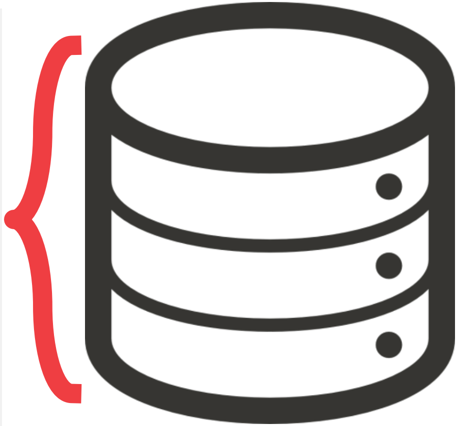
Record	Write Timestamp
A_1	10000
B_1	10000



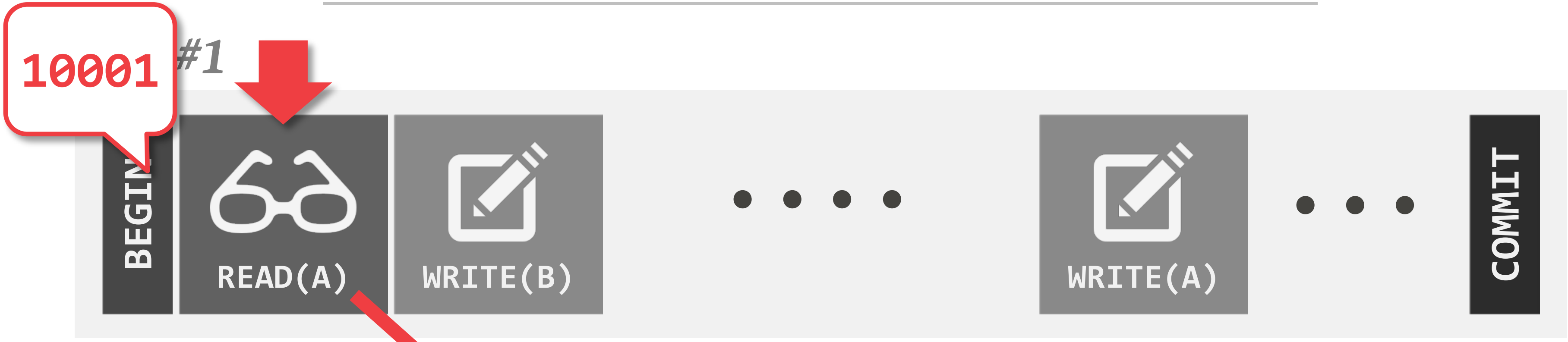
MULTI-VERSION CONCURRENCY CONTROL



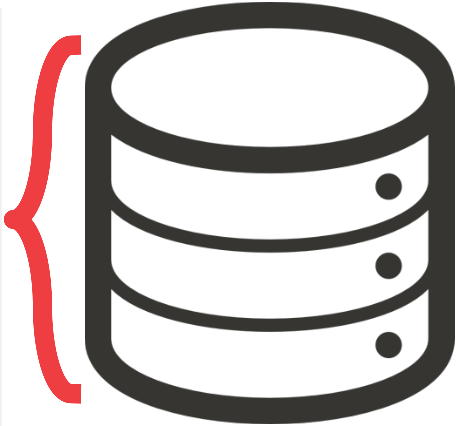
Record	Write Timestamp
A_1	10000
B_1	10000



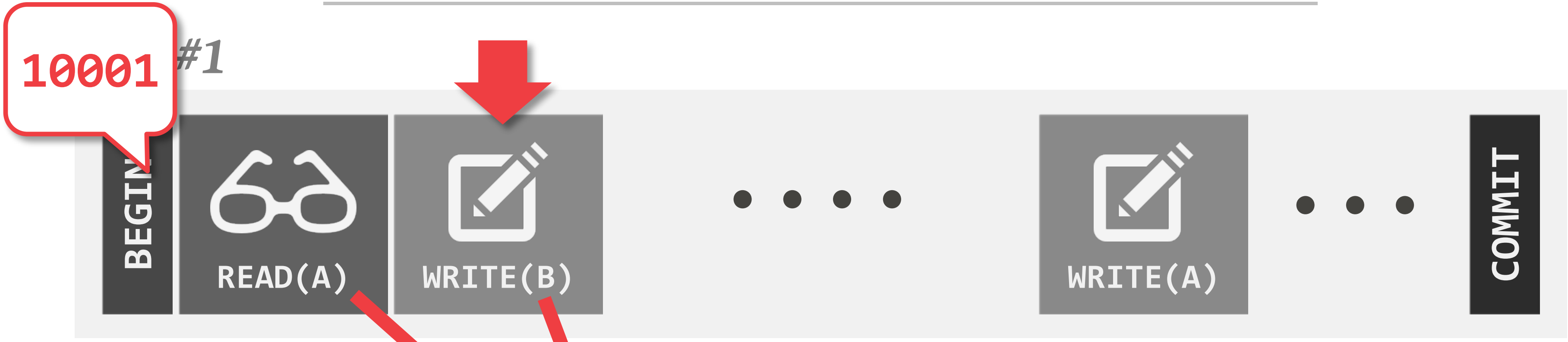
MULTI-VERSION CONCURRENCY CONTROL



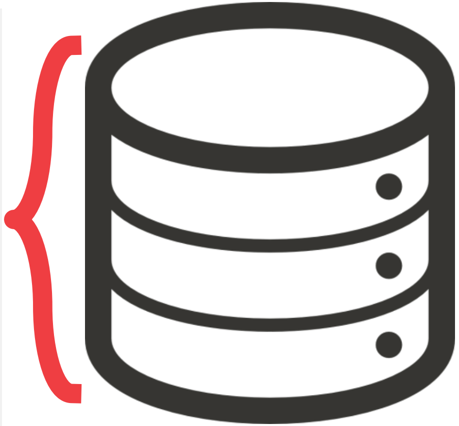
Record	Write Timestamp
A_1	10000
B_1	10000



MULTI-VERSION CONCURRENCY CONTROL

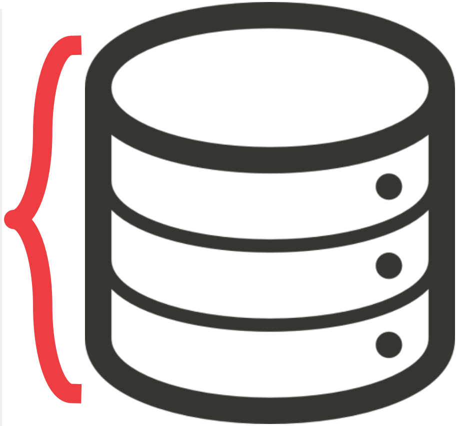


Record	Write Timestamp
A_1	10000
B_1	10000
B_2	10001



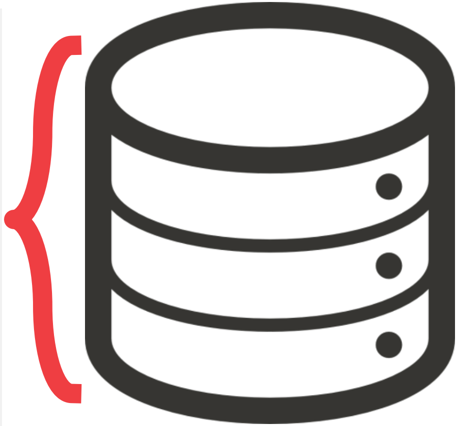
MULTI-VERSION CONCURRENCY CONTROL

Record	Write Timestamp
A_1	10000
B_1	10000
B_2	10001



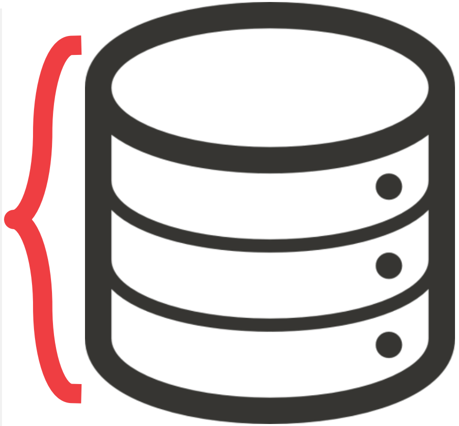
MULTI-VERSION CONCURRENCY CONTROL

Record	Write Timestamp
A_1	10000
B_1	10000
B_2	10001

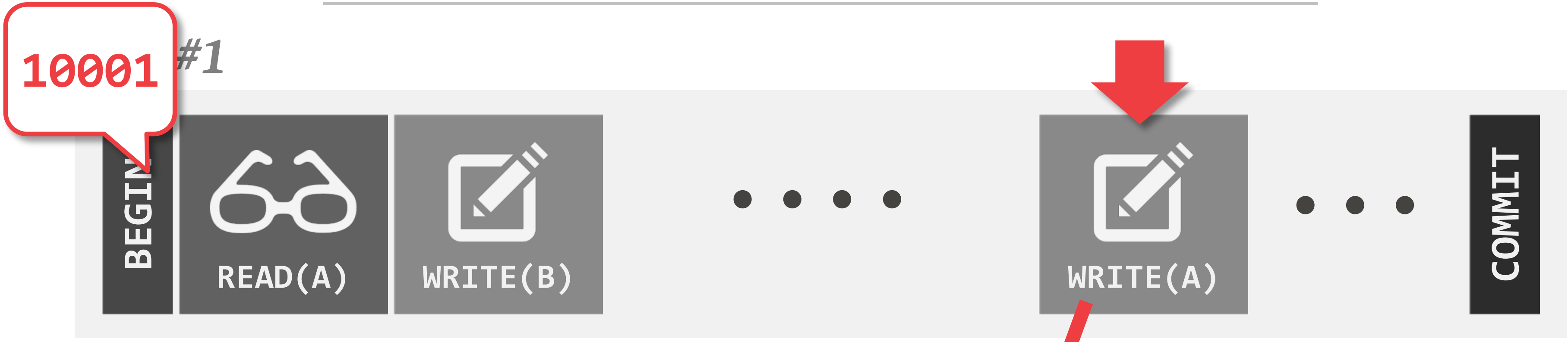


MULTI-VERSION CONCURRENCY CONTROL

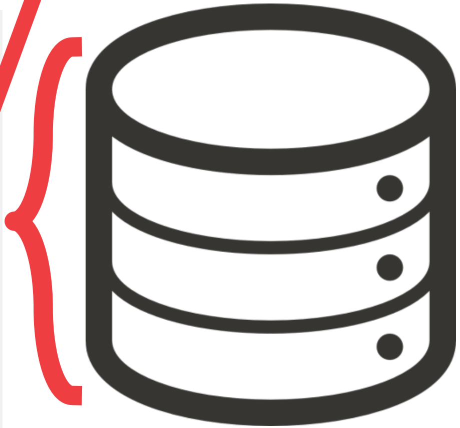
Record	Write Timestamp
A_1	10000
B_1	10000
B_2	10001
A_2	10003



MULTI-VERSION CONCURRENCY CONTROL



Record	Write Timestamp
A_1	10000
B_1	10000
B_2	10001
A_2	10003



MULTI-VERSION CONCURRENCY CONTROL

Record	Write Timestamp
A_1	10000
B_1	10000
B_2	10001
A_2	10003

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:

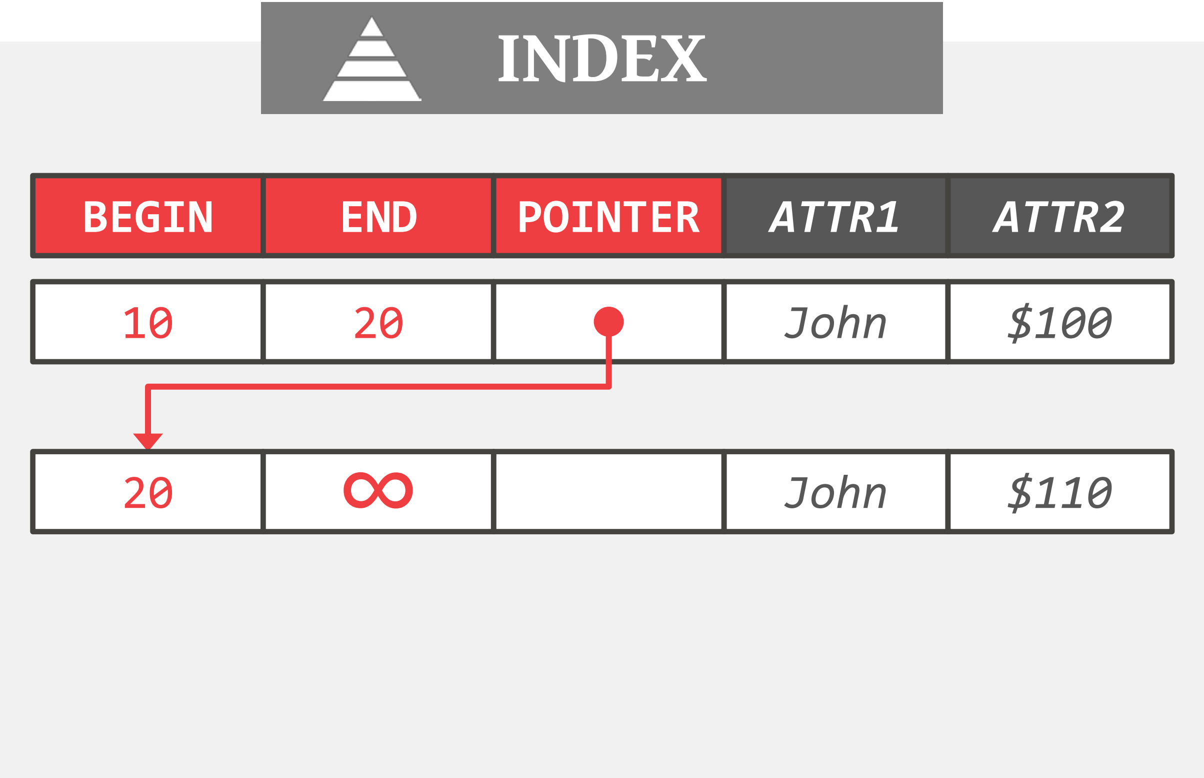
→ **BEGIN**: The BeginTS of the active txn **or** the EndTS of the committed txn that created it.

→ **END**: The BeginTS of the active txn that created the next version **or** infinity **or** the EndTS of the committed txn that created it.

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

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

HEKATON: OPERATIONS



HEKATON: OPERATIONS

BEGIN @ 25

BEGIN	END	POINTER	ATTR1	ATTR2
10	20	●	John	$100
20	∞		John	$110

HEKATON: OPERATIONS

BEGIN @ 25

Read "John"

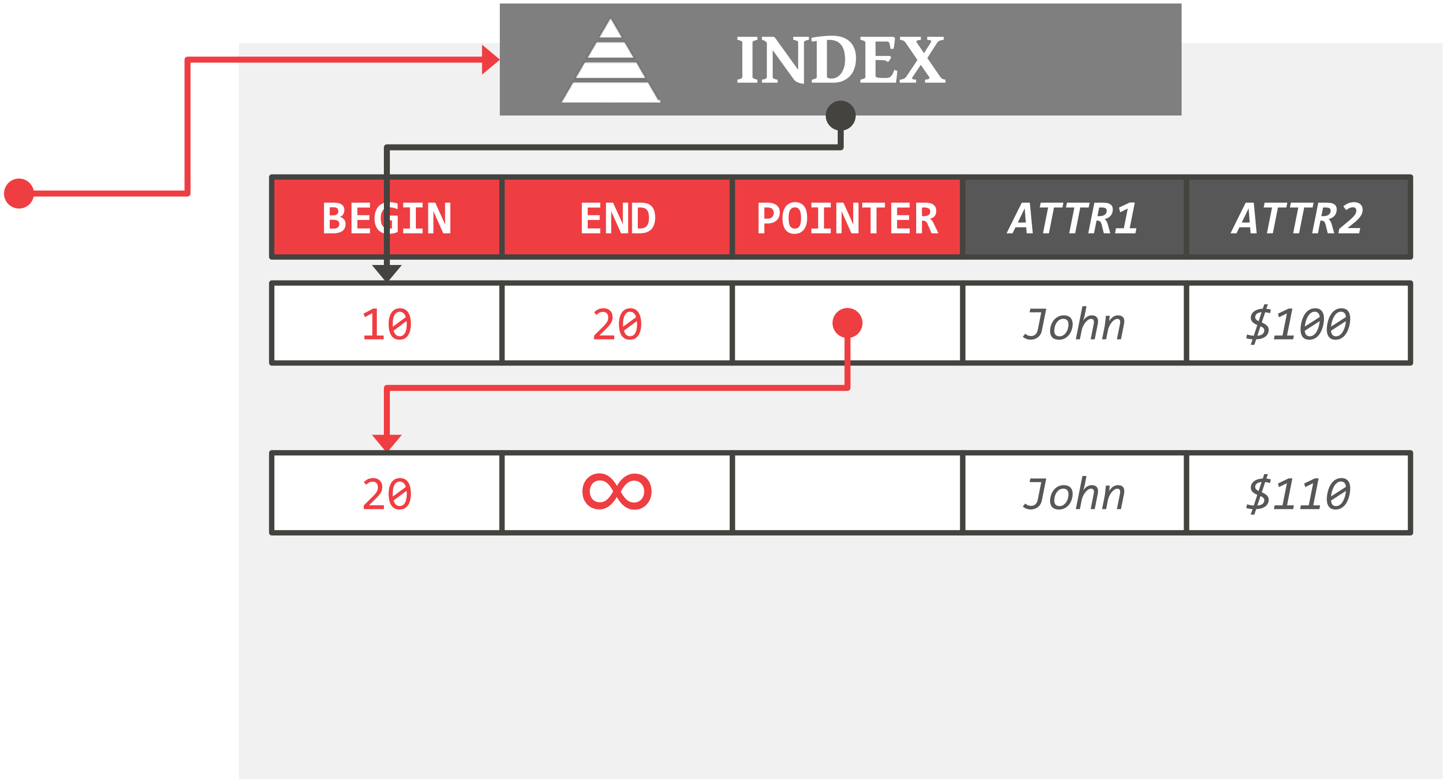
INDEX

BEGIN	END	POINTER	ATTR1	ATTR2
10	20	●	*John*	*$100*
20	∞		*John*	*$110*

HEKATON: OPERATIONS

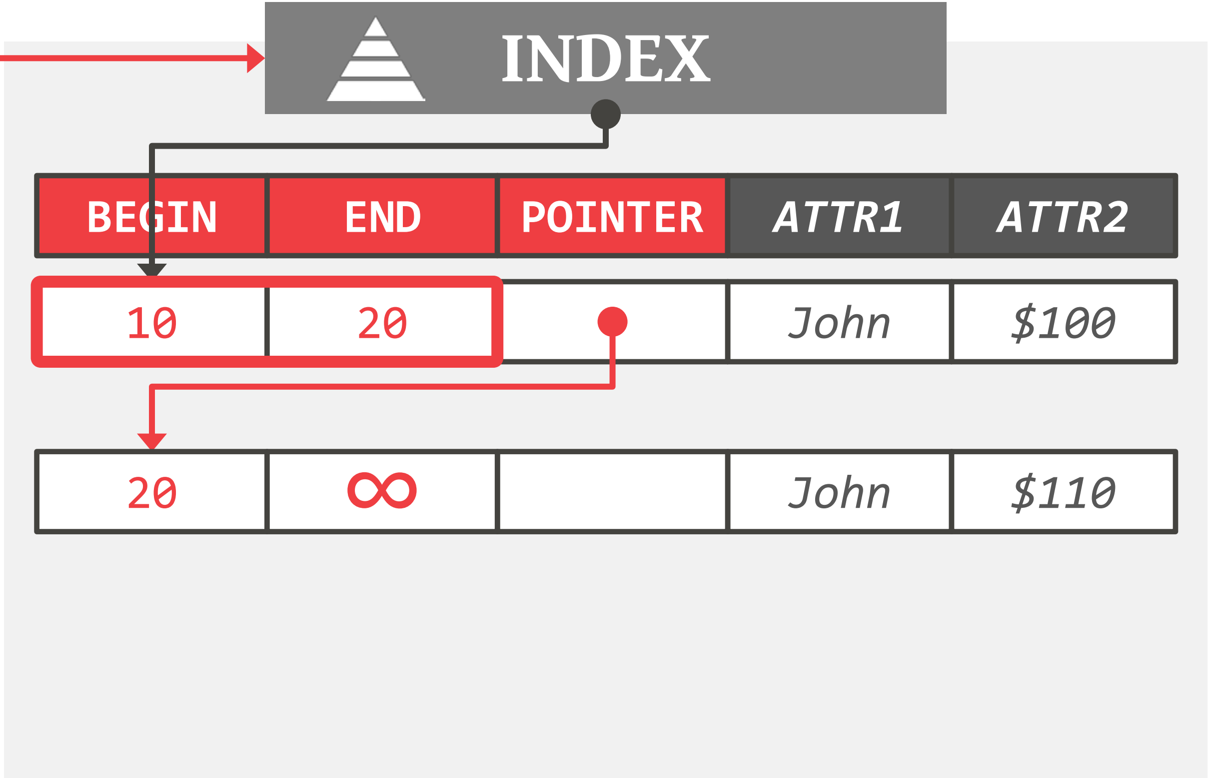
BEGIN @ 25

Read "John"



HEKATON: OPERATIONS

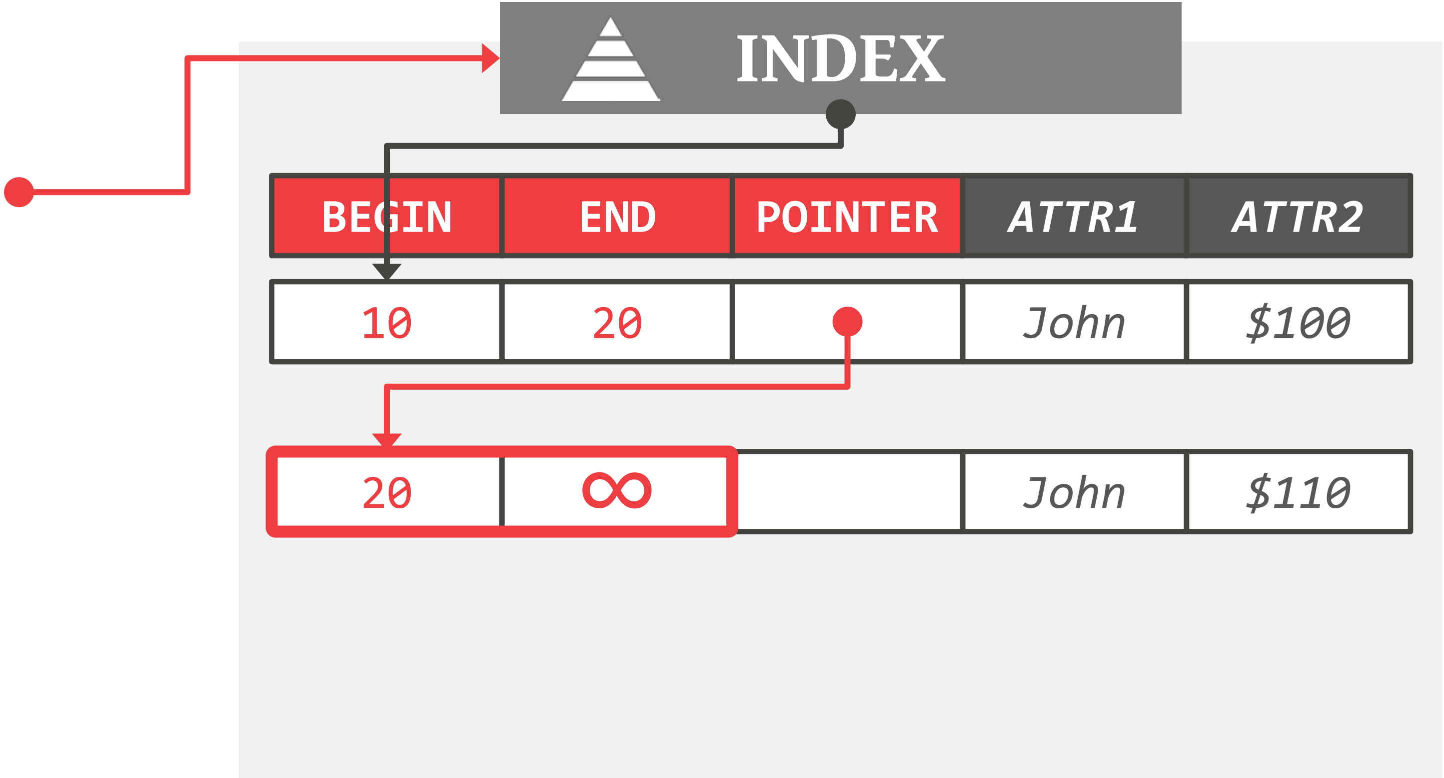
BEGIN @ 25
Read "John"



HEKATON: OPERATIONS

BEGIN @ 25

Read “John”



HEKATON: OPERATIONS

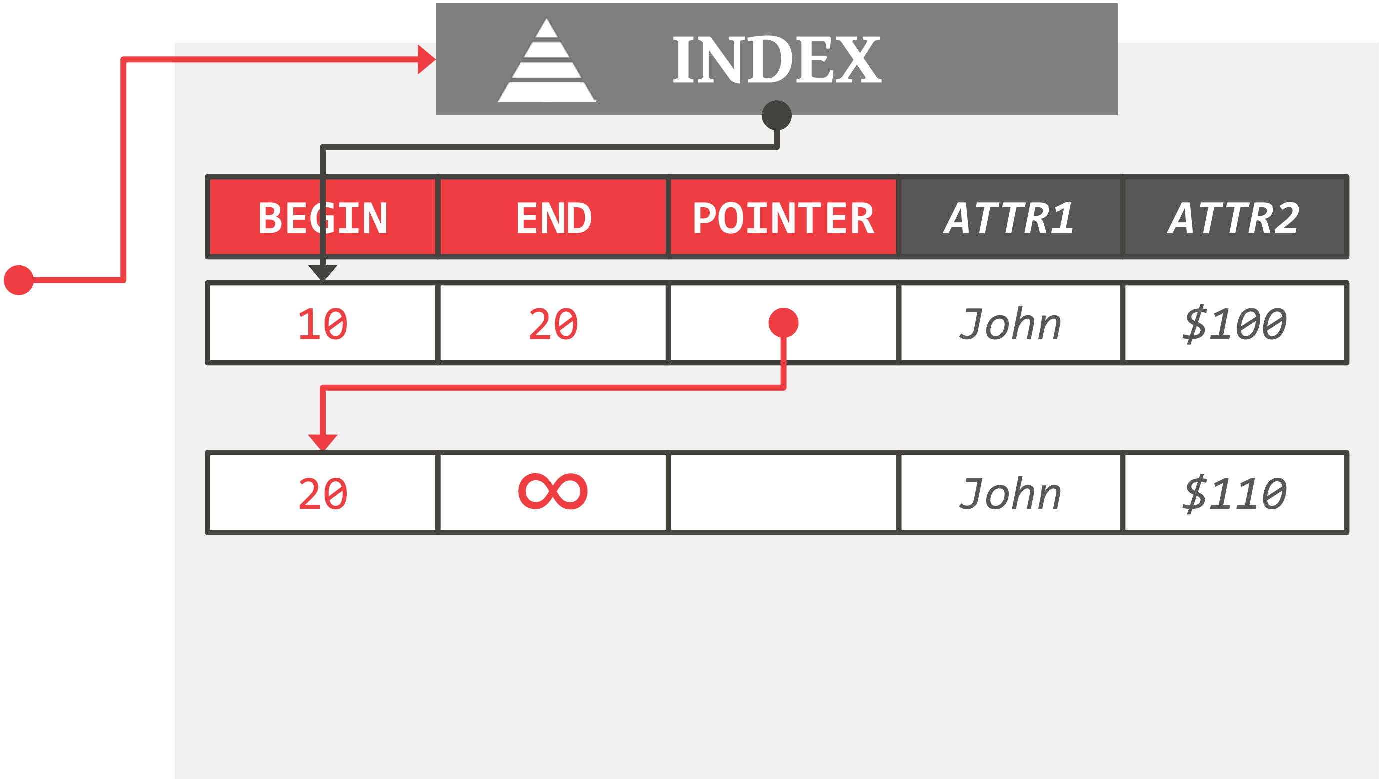
BEGIN @ 25
Read "John"
Update "John"

INDEX

BEGIN	END	POINTER	*ATTR1*	*ATTR2*
10	20	●	*John*	*$100*
20	∞		*John*	*$110*

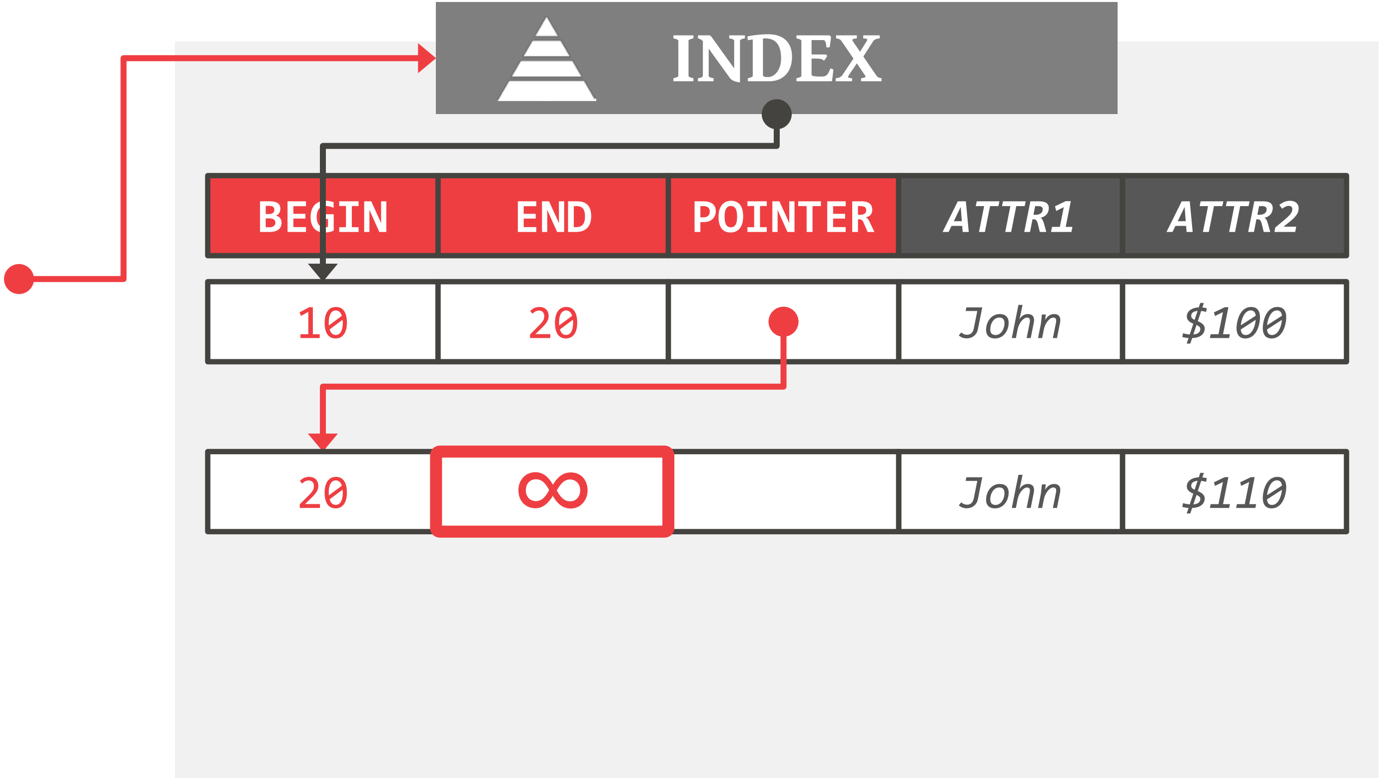
HEKATON: OPERATIONS

BEGIN @ 25
Read "John"
Update "John"



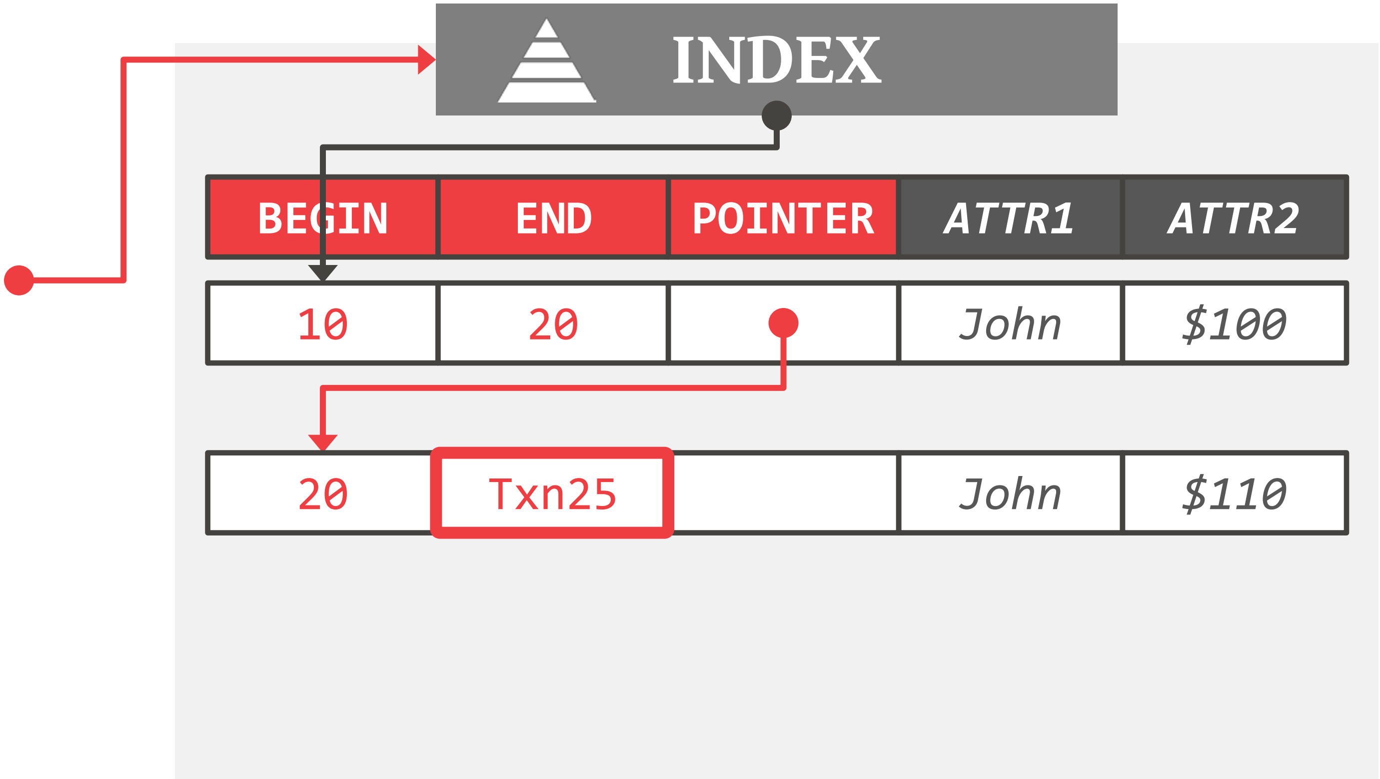
HEKATON: OPERATIONS

BEGIN @ 25
Read "John"
Update "John"



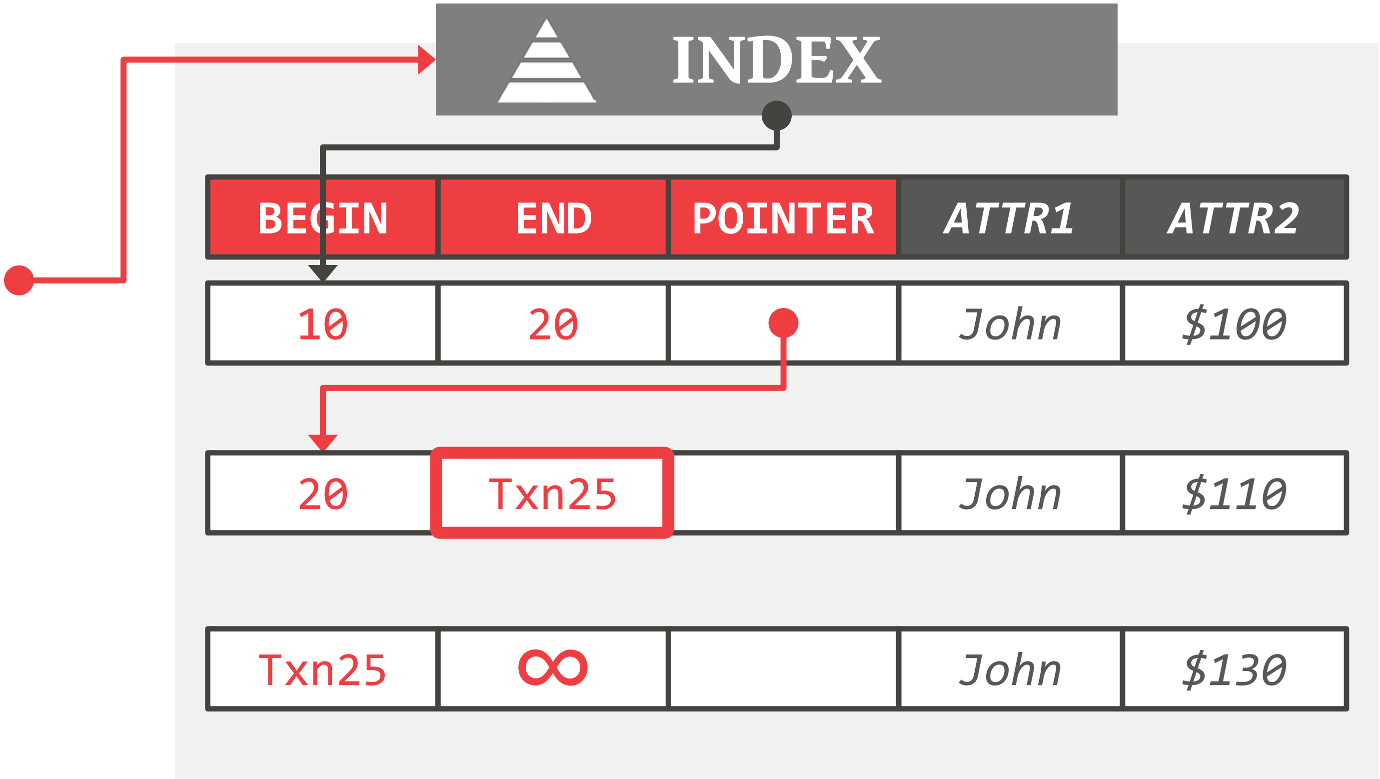
HEKATON: OPERATIONS

BEGIN @ 25
Read "John"
Update "John"



HEKATON: OPERATIONS

BEGIN @ 25
Read "John"
Update "John"

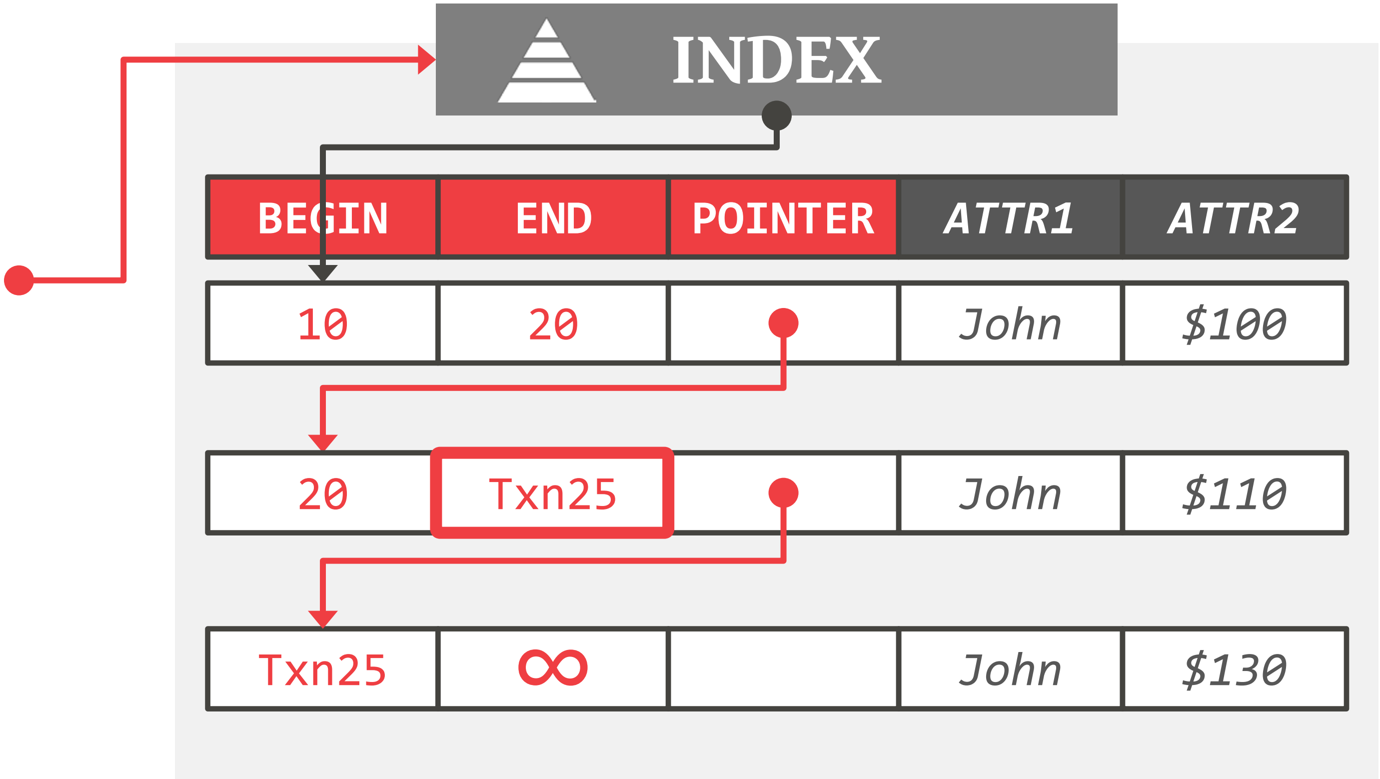


HEKATON: OPERATIONS

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

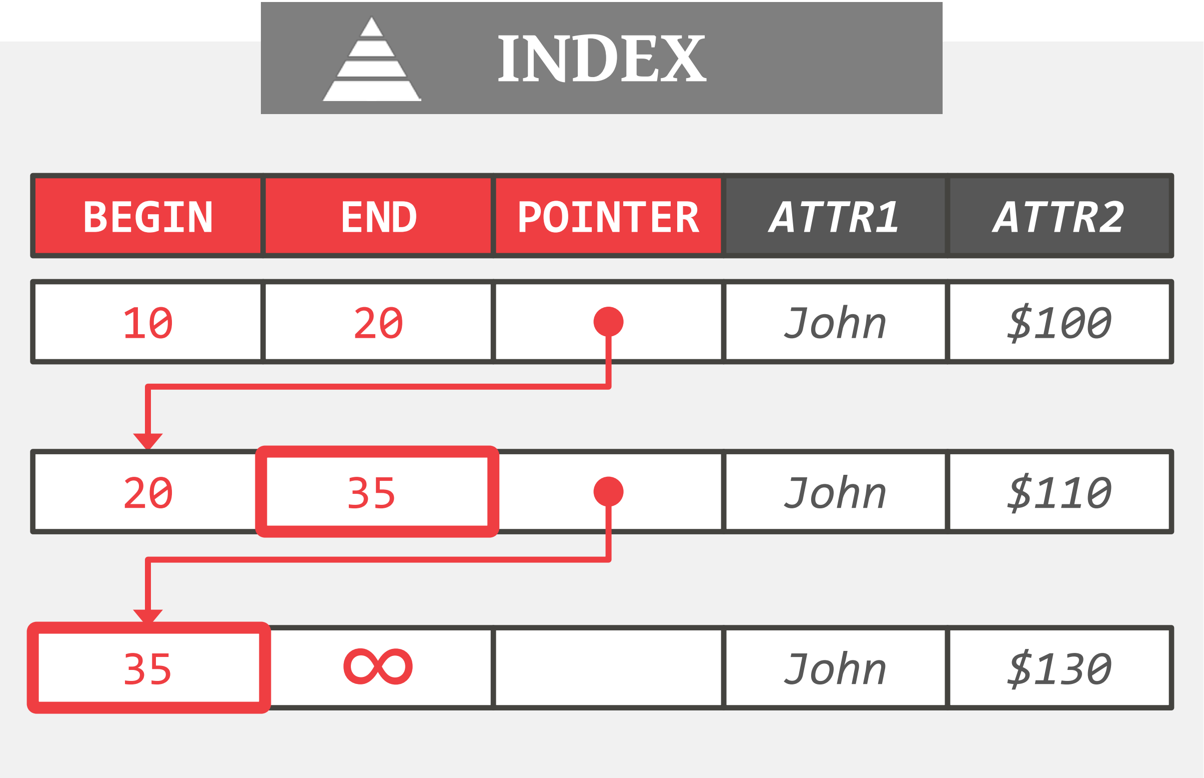
BEGIN @ 25
Read "John"
Update "John"
COMMIT @ 35

INDEX

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

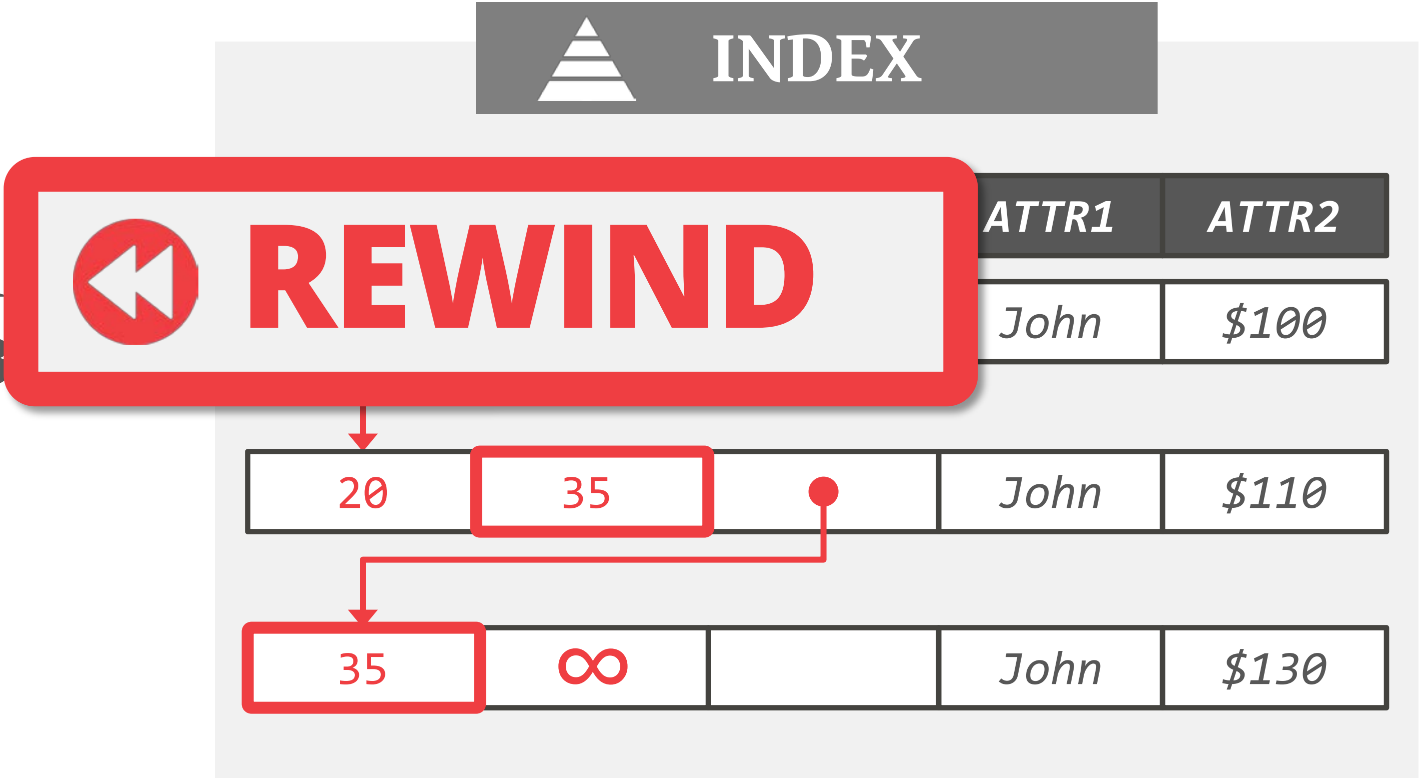
HEKATON: OPERATIONS

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



HEKATON: OPERATIONS

BEGIN @ 25
Read "John"
Update "John
COMMIT @ 3



HEKATON: OPERATIONS

BEGIN @ 25
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

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"

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"

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"

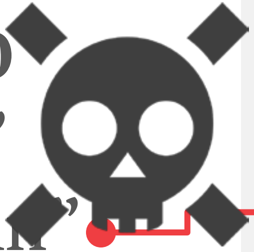
INDEX

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"



INDEX

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

HEKATON: TRANSACTION STATE MAP

Global map of all txns' states in the system:

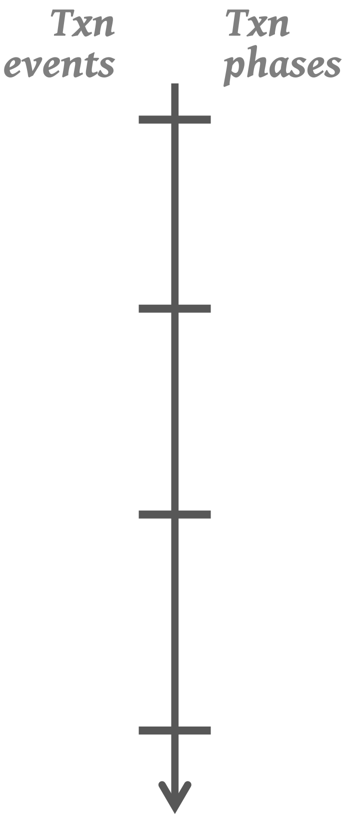
→ **ACTIVE**: The txn is executing read/write operations.

→ **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.

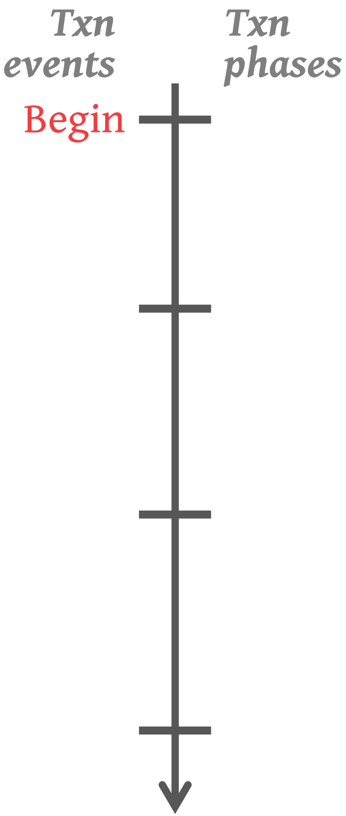
→ **TERMINATED**: The txn has updated the TS for all of the versions that it created.

HEKATON: TRANSACTION LIFECYCLE



Source: Paul Larson

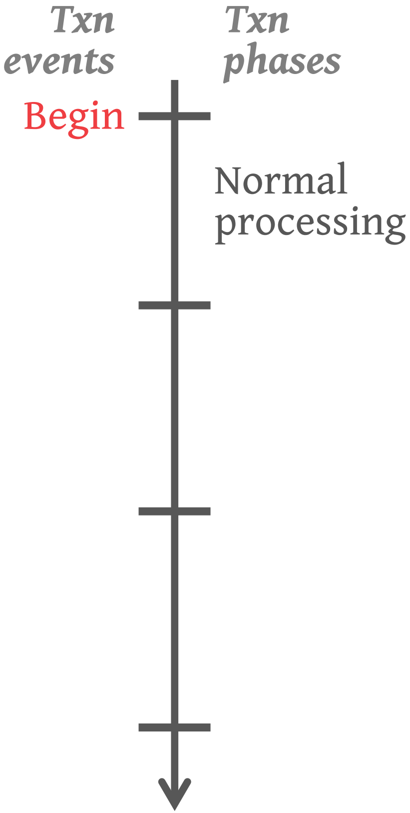
HEKATON: TRANSACTION LIFECYCLE



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

Source: Paul Larson

HEKATON: TRANSACTION LIFECYCLE



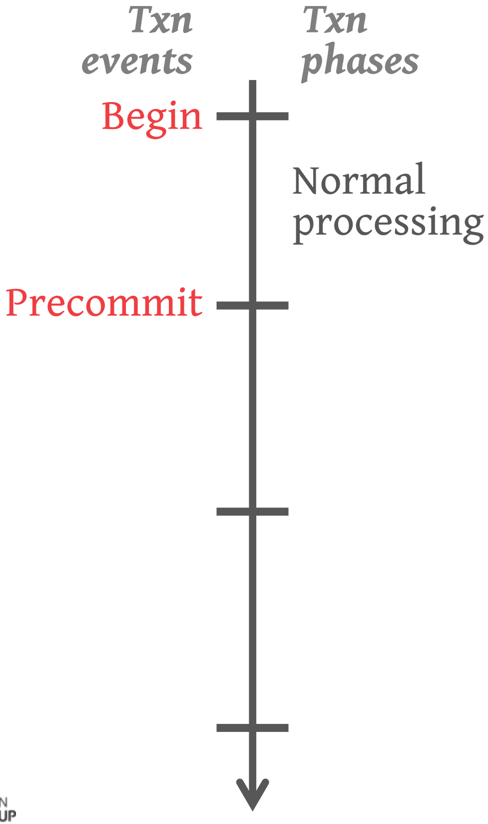
Get txn start timestamp, set state to **ACTIVE**

Perform normal processing

→ Track txn's read set, scan set, and write set.

Source: Paul Larson

HEKATON: TRANSACTION LIFECYCLE



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

HEKATON: TRANSACTION LIFECYCLE

Txn events	*Txn phases*	
Begin		Get txn start timestamp, set state to **ACTIVE**
	Normal processing	Perform normal processing → Track txn's read set, scan set, and write set.
Precommit		Get txn end timestamp, set state to **VALIDATING**
	Validation	Validate reads and scans → If validation OK, write new versions to redo log

Source: Paul Larson

HEKATON: TRANSACTION LIFECYCLE

Txn events	*Txn phases*	
Begin		Get txn start timestamp, set state to **<u>ACTIVE</u>**
	Normal processing	Perform normal processing → Track txn's read set, scan set, and write set.
Precommit		Get txn end timestamp, set state to **<u>VALIDATING</u>**
	Validation	Validate reads and scans → If validation OK, write new versions to redo log
Commit		Set txn state to **<u>COMMITTED</u>**

Source: Paul Larson

HEKATON: TRANSACTION LIFECYCLE

Txn events	*Txn phases*	
Begin		Get txn start timestamp, set state to **ACTIVE**
	Normal processing	Perform normal processing → Track txn's read set, scan set, and write set.
Precommit		Get txn end timestamp, 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	Fix up version timestamps → Begin TS in new versions, end TS in old versions

Source: Paul Larson

HEKATON: TRANSACTION LIFECYCLE

Txn events	*Txn phases*	
Begin		Get txn start timestamp, set state to **ACTIVE**
	Normal processing	Perform normal processing → Track txn's read set, scan set, and write set.
Precommit		Get txn end timestamp, 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	Fix up version timestamps → Begin TS in new versions, end TS in old versions
Terminate		Set txn state to **TERMINATED**
		Remove from txn map

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:

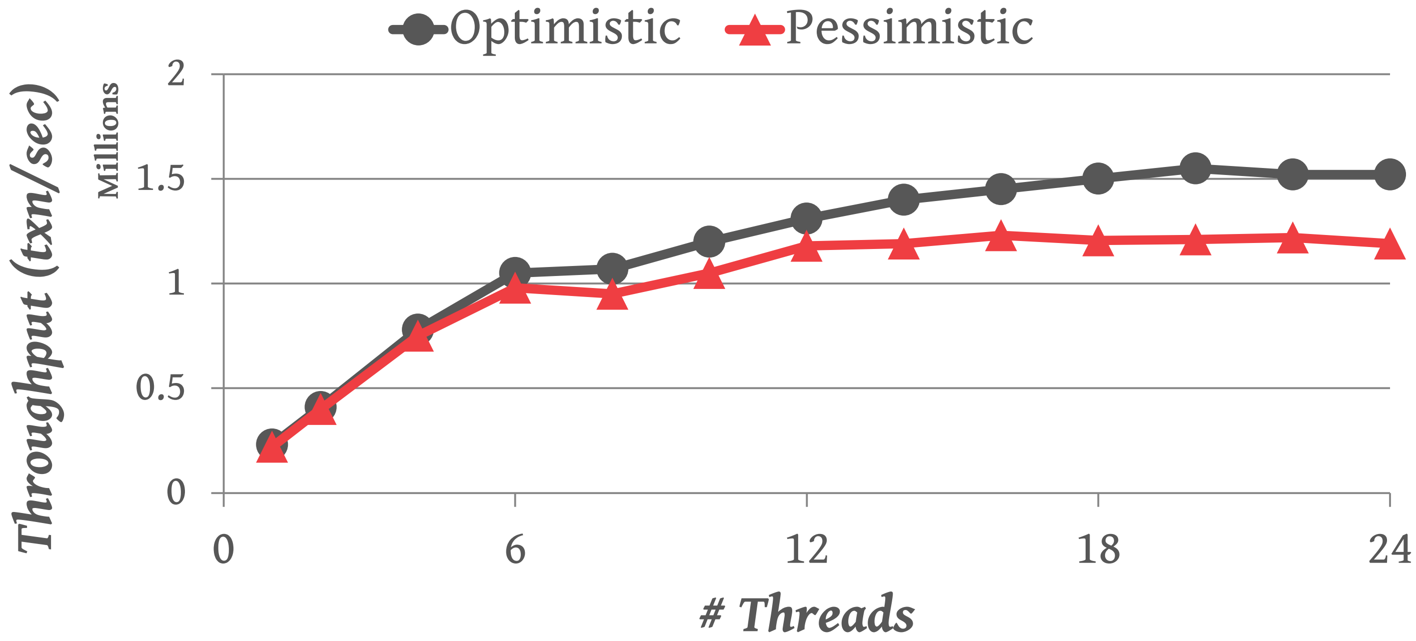
→ Check whether a version read is still visible at the end of the txn.
→ Repeat all index scans to check for phantoms.

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

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

BEGIN	END	ATTR1	ATTR2
10	20	John	$100

ROLLBACK SEGMENTS

Main Data Table

BEGIN	END	ATTR1	ATTR2
10	20	John	$\boxed{\$100}$

ROLLBACK SEGMENTS

Main Data Table

BEGIN	END	*ATTR1*	*ATTR2*
10	20	*John*	*$100*

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

ROLLBACK SEGMENTS

Main Data Table

BEGIN	END	*ATTR1*	*ATTR2*
10	20	*John*	*$100*

Rollback Segment (Per Tuple)

BEGIN	END	DELTA

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

ROLLBACK SEGMENTS

Main Data Table

BEGIN	END	*ATTR1*	*ATTR2*
10	20	*John*	*$100*

Rollback Segment (Per Tuple)

BEGIN	END	DELTA
10	20	*(ATTR2→$100)*

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

ROLLBACK SEGMENTS

Main Data Table

BEGIN	END	ATTR1	ATTR2
20	25	John	$110

Rollback Segment (Per Tuple)

BEGIN	END	DELTA
10	20	(ATTR2→$100)

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

ROLLBACK SEGMENTS

Main Data Table

BEGIN	END	*ATTR1*	*ATTR2*
30	35	*John*	*$130*

Rollback Segment (Per Tuple)

BEGIN	END	DELTA
10	20	*(ATTR2→$100)*
20	25	*(ATTR2→$110)*

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

ROLLBACK SEGMENTS

Main Data Table

BEGIN	END	*ATTR1*	*ATTR2*
30	35	*John*	*$130*

Rollback Segment (Per Tuple)

BEGIN	END	DELTA
10	20	*(ATTR2→$100)*
20	25	*(ATTR2→$110)*

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.

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

Rollback Segment with Deltas
→ In-Place updates for non-indexed attributes
→ Delete/Insert updates for indexed attributes.

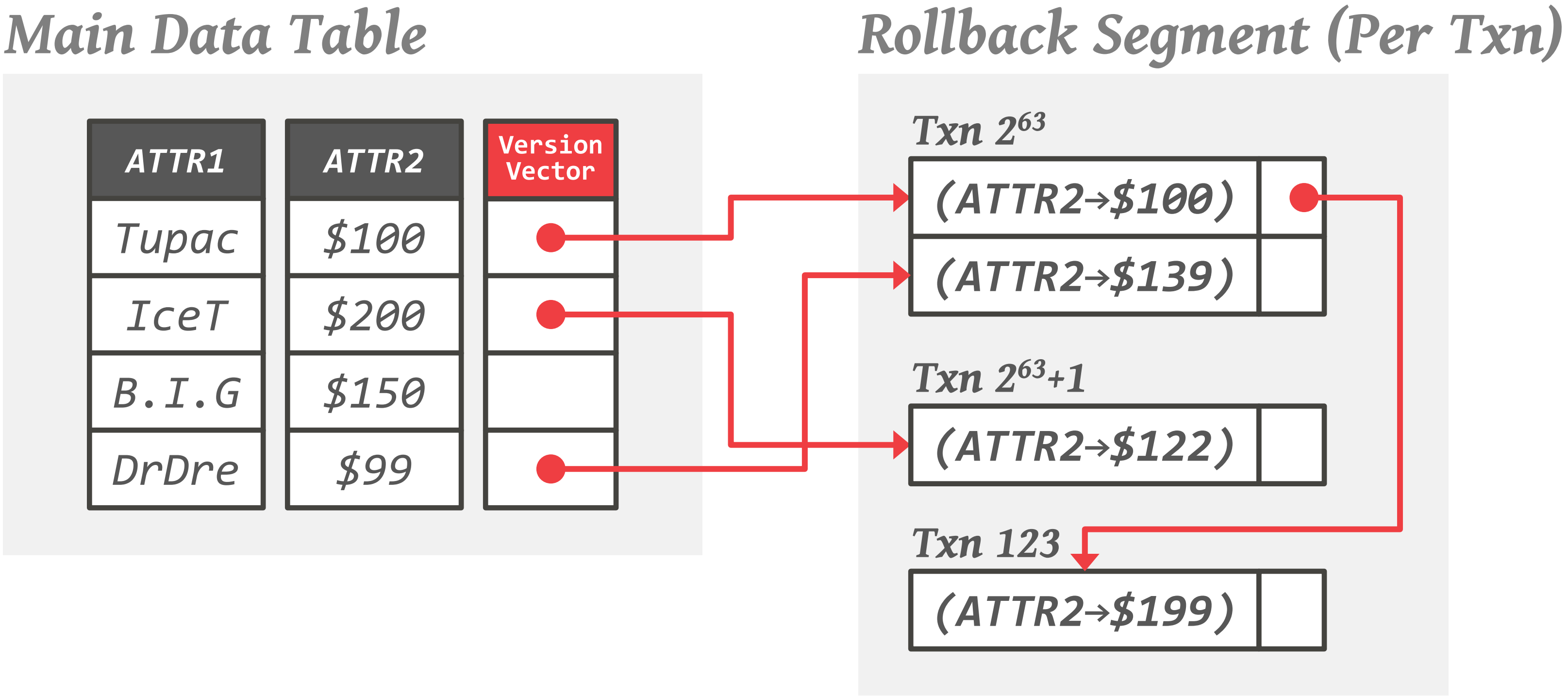
Newest-to-Oldest Version Chains

No Predicate Locks

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

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

HYPER MVCC



Main Data Table
Rollback Segment (Per Txn)
ATTR1
ATTR2
Version Vector
Tupac
$100
IceT
$200
B.I.G
$150
DrDre
$99
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.

EFFICIENT TRANSACTION PROCESSING FOR HYRISE IN MIXED WORKLOAD ENVIRONMENTS
IMDM 2014

SAP HANA MVCC

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

It's not clear what else they are doing...

HIGH-PERFORMANCE TRANSACTION PROCESSING IN SAP HANA
IEEE Data Engineering Bulletin 2013

PARTING THOUGHTS

MVCC is currently the best approach for supporting txns in mixed workoads

→ 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