

CS 6530: Advanced Database Systems Fall 2023

# Lecture 08

## Multi Version Concurrency Control

Prashant Pandey

[prashant.pandey@utah.edu](mailto:prashant.pandey@utah.edu)

Acknowledgement: Slides taken from Prof. Andy Pavlo, CMU

# Some announcements...



- Paper report #1 grades posted
  - Don't copy full sentences from different sources
  - Write your analysis of the paper in your own words!!!
- Project #1 due Tuesday Sep 12<sup>th</sup>
- Start forming Project #2 groups
  - Post a discussion on Canvas to find group members.

# 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.
- First proposed in 1978 MIT PhD [dissertation](#).
- First implementation was InterBase ([Firebird](#)).
- Used in almost every new DBMS in last 10 years.

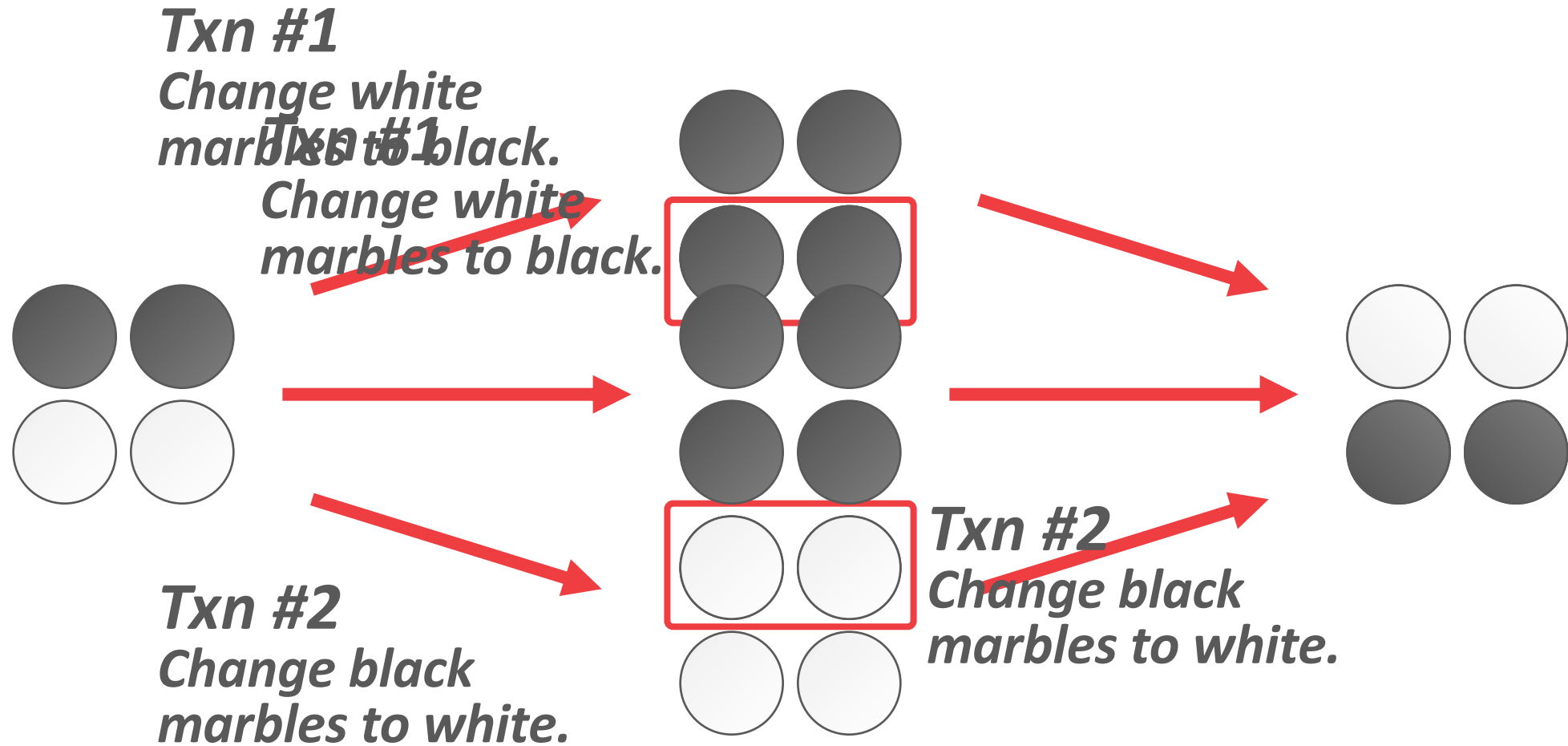
# MULTI-VERSION CONCURRENCY CONTROL

- **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.

# 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



# MVCC DESIGN DECISIONS

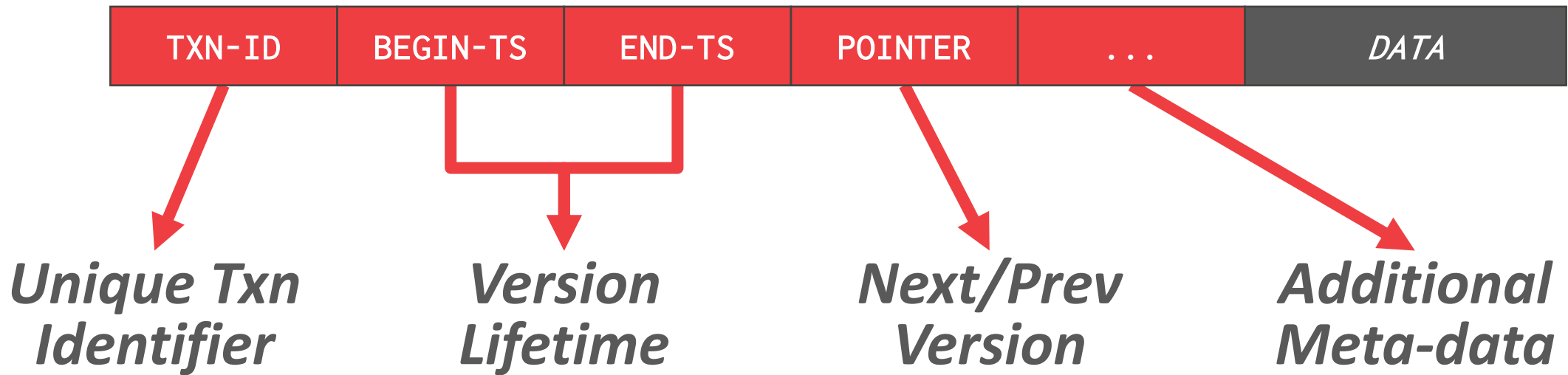
- Concurrency Control Protocol
- Version Storage
- Garbage Collection
- Index Management

# 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



# TIMESTAMP ORDERING (MVTO)

Thread #1

$T_{id}=10$



	TXN-ID	READ-TS	BEGIN-TS	END-TS
A <sub>1</sub>	0	10	1	∞
B <sub>1</sub>	10	0	1	10
B <sub>2</sub>	0	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

$T_{id}=10$



	TXN-ID	READ-CNT	BEGIN-TS	END-TS
A <sub>1</sub>	0	1	1	∞
B <sub>1</sub>	10	1	1	10
B <sub>2</sub>	0	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**

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



**Thread #2**

$T_{id}=1$



	TXN-ID	READ-TS	BEGIN-TS	END-TS
$A_1$	$2^{31}-1$	-	99999	$2^{31}-1$
$A_2$	1	-	$2^{31}-1$	1
$A_3$	1	-	1	$\infty$

- 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.

# POSTGRES 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.

# VERSION STORAGE

- 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.
- 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

- All the physical versions of a logical tuple are stored in the same table space. The versions are mixed together.
- On every update, append a new version of the tuple into an empty space in the table.

## Main Table

	VALUE	POINTER
$A_0$	\$111	●
$A_1$	\$222	●
$B_1$	\$10	∅
$A_2$	\$333	∅

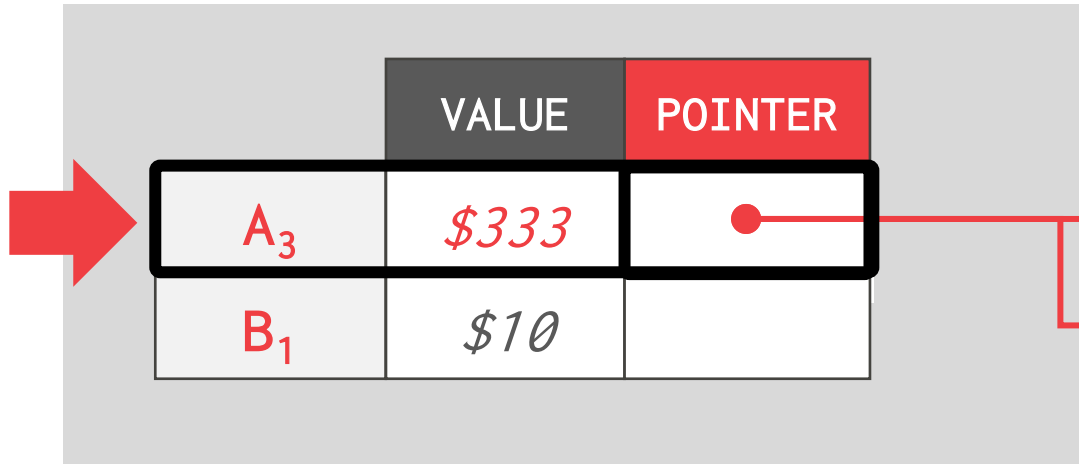


# VERSION CHAIN ORDERING

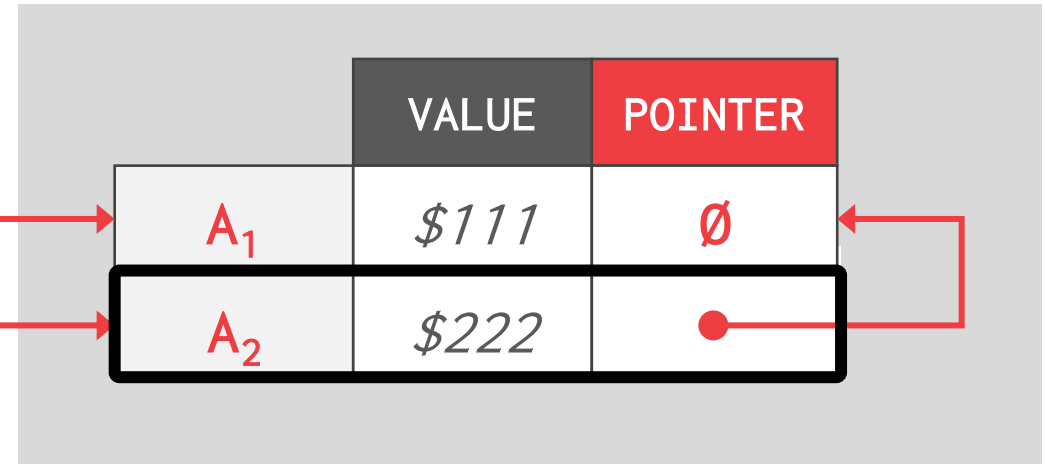
- **Approach #1: Oldest-to-Newest (O2N)**
  - Append every new version to end of the chain.
  - Must traverse chain on look-ups.
- **Approach #2: Newest-to-Oldest (N2O)**
  - Must update index pointers for every new version.
  - Don't have to traverse chain on look ups.
- The ordering of the chain has different performance trade-offs.

# TIME-TRAVEL STORAGE

*Main Table*



*Time-Travel Table*

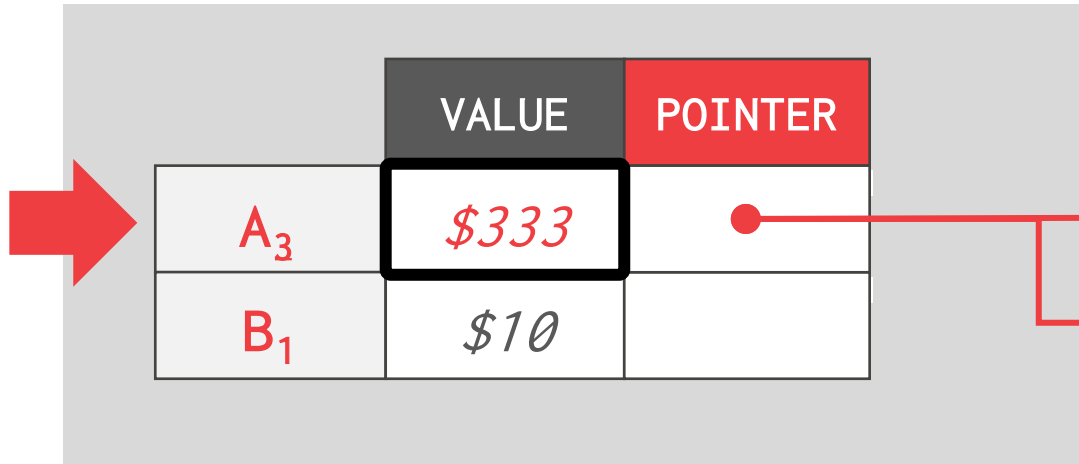


On every update, copy the current version to the time-travel table. Update pointers.

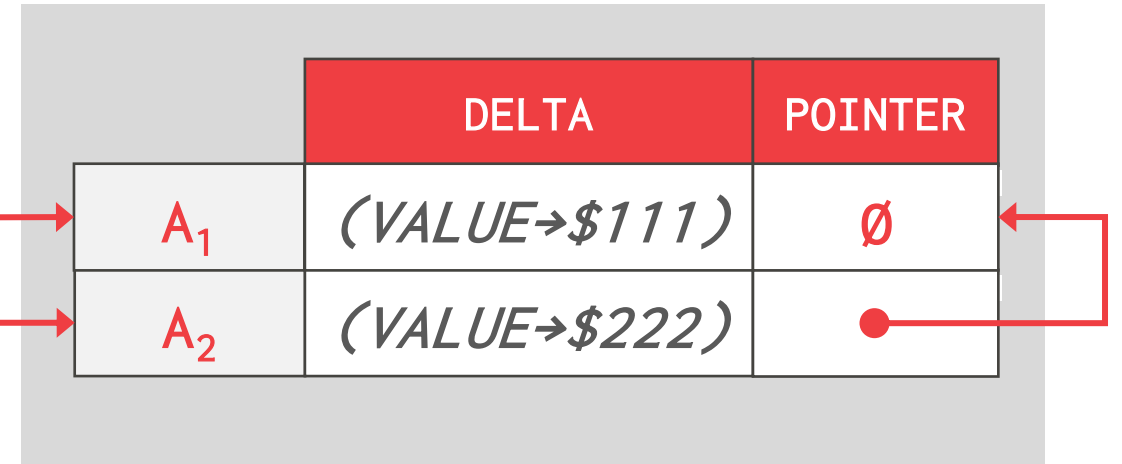
Overwrite master version in the main table and update pointers.

# DELTA STORAGE

*Main Table*



*Delta Storage Segment*



On every update, copy only the values that were modified to the delta storage and overwrite the master version.

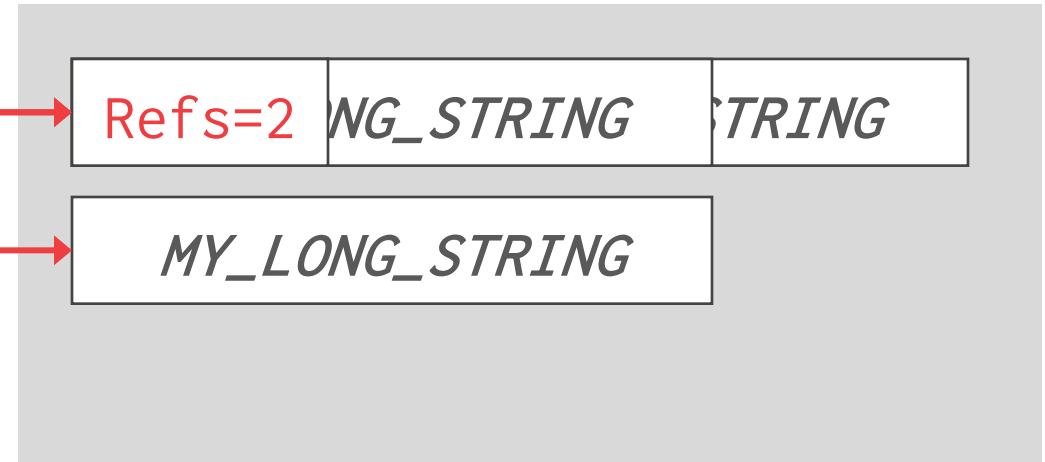
Txns can recreate old versions by applying the delta in reverse order.

# NON-INLINE ATTRIBUTES

## Main Table

	<i>INT_VAL</i>	<i>STR_VAL</i>
<i>A<sub>1</sub></i>	<i>\$100</i>	●
<i>A<sub>2</sub></i>	<i>\$90</i>	●

## Variable-Length Data



Reuse pointers to variable-length pool for values that do not change between versions.

Requires reference counters to know when it is safe to free memory. Unable to relocate memory easily.

# 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.
- Three additional design decisions:
  - How to look for expired versions?
  - How to decide when it is safe to reclaim memory?
  - Where to look for expired versions?

# 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.

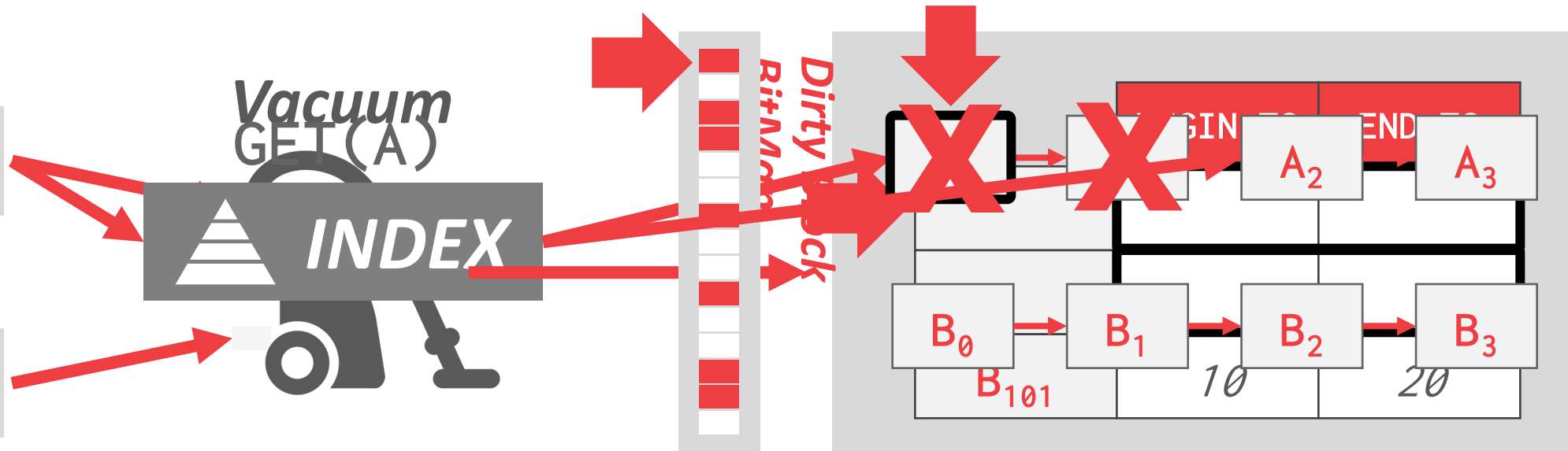
# TUPLE-LEVEL GC

Thread #1

$T_{id}=12$

Thread #2

$T_{id}=25$



**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.

# TRANSACTION-LEVEL GC

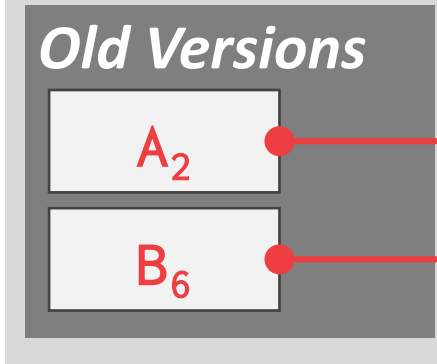
- Each txn keeps track of its read/write set.
- The DBMS determines when all versions created by a finished txn are no longer visible.
- May still require multiple threads to reclaim the memory fast enough for the workload.



# TRANSACTION-LEVEL GC

**Thread #1**

**Begin @ 10**  
**Commit @ 15**



	BEGIN-TS	END-TS	DATA
A <sub>2</sub>	1	15	-
B <sub>6</sub>	8	15	-
A <sub>3</sub>	15	∞	-
B <sub>7</sub>	15	∞	-

**Vacuum**

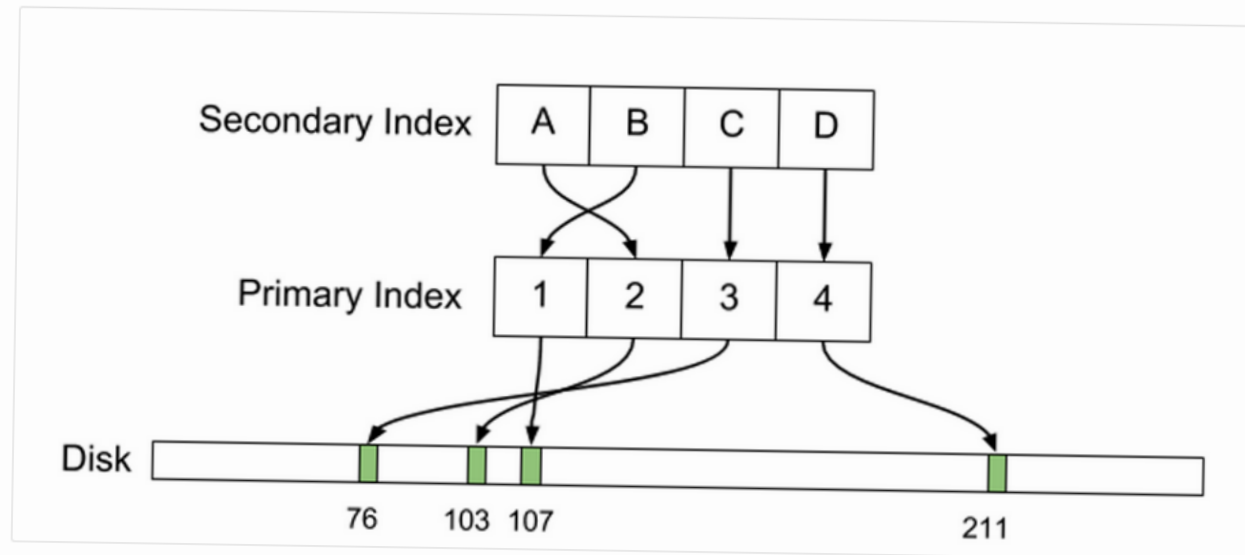


**TS < 15**

ARCHITECTURE

# WHY UBER ENGINEERING SWITCHED FROM POSTGRES TO MYSQL

JULY 26, 2016  
BY EVAN KLITZKE



INDEX

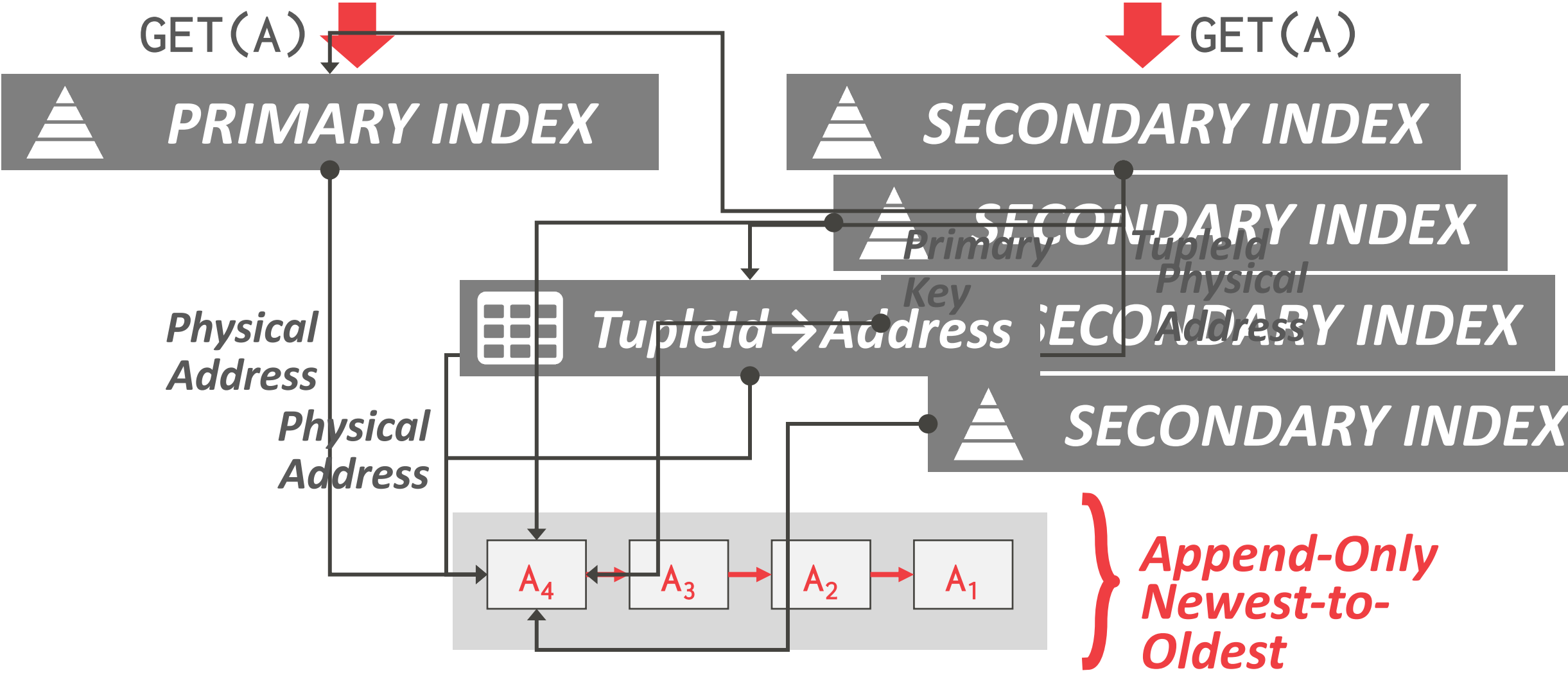
- PKey
- How system
- If a table follows
- Security

the  
TE

# 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



# MVCC INDEXES

- MVCC DBMS indexes (usually) do not store version information about tuples with their keys.
  - Exception: Index-organized tables (e.g., MySQL)
- Every index must support duplicate keys from different snapshots:
  - The same key may point to different logical tuples in different snapshots.

# MVCC DUPLICATE KEY PROBLEM

**Thread #1**

*Begin @ 10*



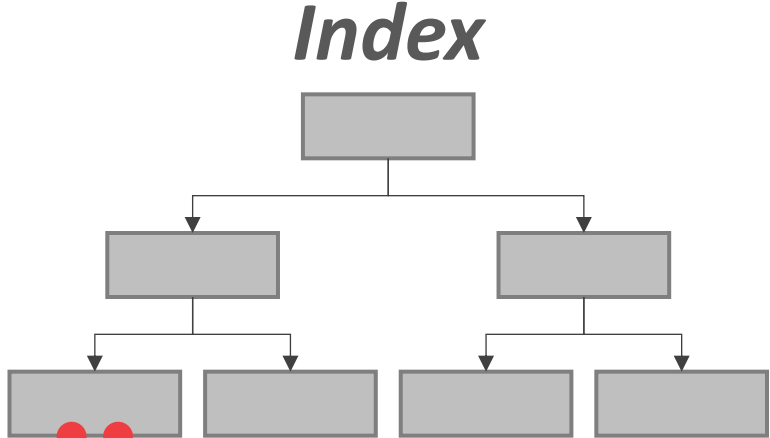
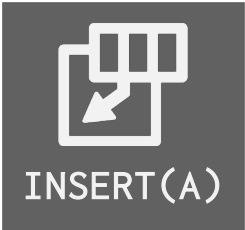
**Thread #2**

*Begin @ 20*  
*Commit @ 25*



**Thread #3**

*Begin @ 30*



	BEGIN-TS	END-TS	POINTER
<i>A<sub>1</sub></i>	<i>1</i>	<i>25</i>	<i>●</i>
<del><i>A<sub>1</sub></i></del>	<i>25</i>	<i>25</i>	<i>∅</i>
<i>A<sub>1</sub></i>	<i>30</i>	<i>∞</i>	<i>∅</i>

# MVCC INDEXES

- Each index's underlying data structure must support the storage of non-unique keys.
- Use additional execution logic to perform conditional inserts for pkey / unique indexes.
  - Atomically check whether the key exists and then insert.
- Workers may get back multiple entries for a single fetch. They then must follow the pointers to find the proper physical version.

# MVCC EVALUATION PAPER

- Two categories of experiments:
  - Evaluate each of the design decisions in isolation to determine their trade-offs.
  - Compare configurations of real-world MVCC systems.



# MVCC DESIGN DECISIONS

- **CC Protocol:** Inconclusive results...
- **Version Storage:** Deltas
- **Garbage Collection:** Tuple-Level Vacuuming
- **Indexes:** Logical Pointers

# MVCC CONFIGURATION EVALUATION

	<i>Protocol</i>	<i>Version Storage</i>	<i>Garbage Collection</i>	<i>Indexes</i>
Oracle	MV2PL	Delta	Vacuum	Logical
Postgres	MV-2PL/MV-T0	Append-Only	Vacuum	Physical
MySQL-InnoDB	MV-2PL	Delta	Vacuum	Logical
HYRISE	MV-OCC	Append-Only	-	Physical
Hekaton	MV-OCC	Append-Only	Cooperative	Physical
MemSQL	MV-OCC	Append-Only	Vacuum	Physical
SAP HANA	MV-2PL	Time-travel	Hybrid	Logical
NuoDB	MV-2PL	Append-Only	Vacuum	Logical
HyPer	MV-OCC	Delta	Txn-level	Logical
<a href="#">CMU's TBD</a>	MV-OCC	Delta	Txn-level	Logical

# Robert Haas

VP, Chief Architect, Database Server @ EnterpriseDB, PostgreSQL Major Contributor and Committer

Tuesday, January 30, 2018

## DO or UNDO - there is no VACUUM

What if PostgreSQL didn't need VACUUM at all? This seems hard to imagine. After all, PostgreSQL uses multi-version concurrency control (MVCC), and if you create multiple versions of rows, you have to eventually get rid of the row versions somehow. In PostgreSQL, VACUUM is in charge of making sure that happens, and the autovacuum process is in charge of making sure that happens soon enough. Yet, other schemes are possible, as shown by the fact that not all relational databases handle MVCC in the same way, and there are reasons to believe that PostgreSQL could benefit significantly from adopting a new approach. In fact, many of my colleagues at EnterpriseDB are busy implementing a new approach, and today I'd like to tell you a little bit about what we're doing and why we're doing it.

While it's certainly true that VACUUM has significantly improved over the years, there are some problems that are very difficult to solve in the current system structure. Because old row versions and new row versions are stored in the same place - the table, also known as the heap - updating a large number of rows must, at least temporarily, make the heap bigger. Depending on the pattern of updates, it may be impossible to easily shrink the heap again afterwards. For example, imagine loading a large number of rows into a table and then updating half of the rows in each block. The table size must grow by 50% to accommodate the new row versions. When VACUUM removes the old versions of those rows, the original table blocks are now all 50% full. That space is available for new row versions, but there is no easy way to move the rows from the new newly-added blocks back to the old half-full blocks: you can use VACUUM FULL or you can use third-party tools like `pg_repack`, but either way you end up rewriting the whole table. Proposals have been made to try to relocate rows on the fly, but it's hard to do correctly and risks bloating the

### About Me



Robert Haas

Follow 0

[View my complete profile](#)

### Blog Archive

- ▼ [2018](#) (2)
  - ▼ [January](#) (2)
    - [DO or UNDO - there is no VACUUM](#)
    - [The State of VACUUM](#)
- ▶ [2017](#) (6)
- ▶ [2016](#) (6)
- ▶ [2015](#) (4)
- ▶ [2014](#) (11)
- ▶ [2013](#) (5)
- ▶ [2012](#) (14)
- ▶ [2011](#) (41)
- ▶ [2010](#) (46)

SQL

# PARTING THOUGHTS

- MVCC is the best approach for supporting txns in mixed workloads.
- We only discussed MVCC for OLTP.
  - Design decisions may be different for HTAP

