Lecture #03

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Multi-Version Concurrency Control (Design Decisions)

@Andy_Pavlo // 15-721 // Spring 2019

CORRECTION

Original SQL-92 isolation levels were <u>not</u> devised assuming a 2PL-based DBMS.

A CRITIQUE OF ANSI SQL ISOLATION LEVELS
SIGMOD 1995

RE

tion

DB

A Critique of ANSI SQL Isolation Levels

Hal Berenson — Microsoft Corp. — haroldb@microsoft.com
Phil Bernstein — Microsoft Corp. — philbe@microsoft.com
Jim Gray — U.C. Berkeley — gray@crl.com
Jim Melton — Sybase Corp. — jim.melton@sybase.com
Elizabeth O'Neil — UMass/Boston — eoneil@cs.umb.edu
Patrick O'Neil — UMass/Boston — poneil@cs.umb.edu

Abstract: ANSI SQL-92 [MS, ANSI] defines Isolation Levels in terms of *phenomena:* Dirty Reads, Non-Repeatable Reads, and Phantoms. This paper shows that these phenomena and the ANSI SQL definitions fail to properly characterize several popular isolation levels, including the standard locking implementations of the levels covered. Ambiguity in the statement of the phenomena is investigated and a more formal statement is arrived at; in addition new phenomena that better characterize isolation types are introduced. Finally, an important multiversion isolation type, called Snapshot Isolation, is defined.

1. Introduction

Running concurrent transactions at different isolation levels allows application designers to trade off concurrency and throughput for correctness. Lower isolation levels increase transaction concurrency at the risk of allowing transactions to observe a fuzzy or incorrect database state. Surprisingly, some transactions can execute at the highest isolation level (perfect serializability) while concurrently executing transactions running at a lower isolation level can access states that are not yet committed or that postdate states the transaction read earlier [GLPT]. Of course, transactions running at lower isolation levels can produce invalid data. Application designers must guard against a later transaction running at a higher isolation level accessing this invalid data and propagating such errors.

The ANSI/ISO SQL-92 specifications [MS, ANSI] define four isolation levels: (1) READ UNCOMMITTED, (2) READ COMMITTED, (3) REPEATABLE READ, (4) SERIALIZABLE. These levels are defined with the classical serializability definition, plus three prohibited operation subsequences, called *phenomena: Dirty Read, Non-repeatable Read,* and *Phantom.* The concept of a *phenomenon* is not explicitly defined in the ANSI specifications, but the specifications suggest that phenomena are operation subsequences that may lead to anomalous (perhaps non-serializable) behavior. We refer to *anomalies* in what follows when making suggested additions to the set of ANSI phenomena. As shown later, there is a technical distinction between anomalies and phenomena, but this distinction is not crucial for a general understanding.

The ANSI isolation levels are related to the behavior of lock schedulers. Some lock schedulers allow transactions to vary the scope and duration of their lock requests, thus departing from pure two-phase locking. This idea was introduced by [GLPT], which defined *Degrees of Consistency* in three ways: locking, data-flow graphs, and anomalies. Defining isolation levels by phenomena (anomalies) was intended to allow non-lock-based implementations of the SQL standard.

This paper shows a number of weaknesses in the anomaly approach to defining isolation levels. The three ANSI phenomena are ambiguous, and even in their loosest interpretations do not exclude some anomalous behavior that may arise in execution histories. This leads to some counter-intuitive results. In particular, lock-based isolation levels have different characteristics than their ANSI equivalents. This is disconcerting because commercial database systems typically use locking implementations. Additionally, the ANSI phenomena do not distinguish between a number of types of isolation level behavior that are popular in commercial systems. Additional phenomena to characterize these isolation levels are suggested here.

Section 2 introduces the basic terminology of isolation levels. It defines the ANSI SQL and locking isolation levels. Section 3 examines some drawbacks of the ANSI isolation levels and proposes a new phenomenon. Other popular isolation levels are also defined. The various definitions map between ANSI SQL isolation levels and the *degrees of consistency* defined in 1977 in [GLPT]. They also encompass Chris Date's definitions of Cursor Stability and Repeatable Read [DAT]. Discussing the isolation levels in a uniform framework reduces misunderstandings arising from independent terminology.

Section 4 introduces a multiversion concurrency control mechanism, called *Snapshot Isolation*, that avoids the ANSI SQL phenomena, but is not serializable. Snapshot Isolation is interesting in its own right, since it provides a reduced-isolation level approach that lies between READ COMMITTED and REPEATABLE READ. A new formalism (available in the longer version of this conference paper [OOBBGM]) connects reduced isolation levels for multiversioned data to the classical single-version locking serializability theory.

Section 5 explores some new anomalies to differentiate the isolation levels introduced in Sections 3 and 4. The extended ANSI SQL phenomena proposed here lack the power to characterize Snapshot isolation and Cursor Stability. Section 6 presents a Summary and Conclusions.

1

Cursor Stability, Snapshot Isolation, Disk Drive performance - cs.cmu.edu/Inbox – Kontact

File Edit View Message Settings Help

Print... Reply Forward Trash Create To-do

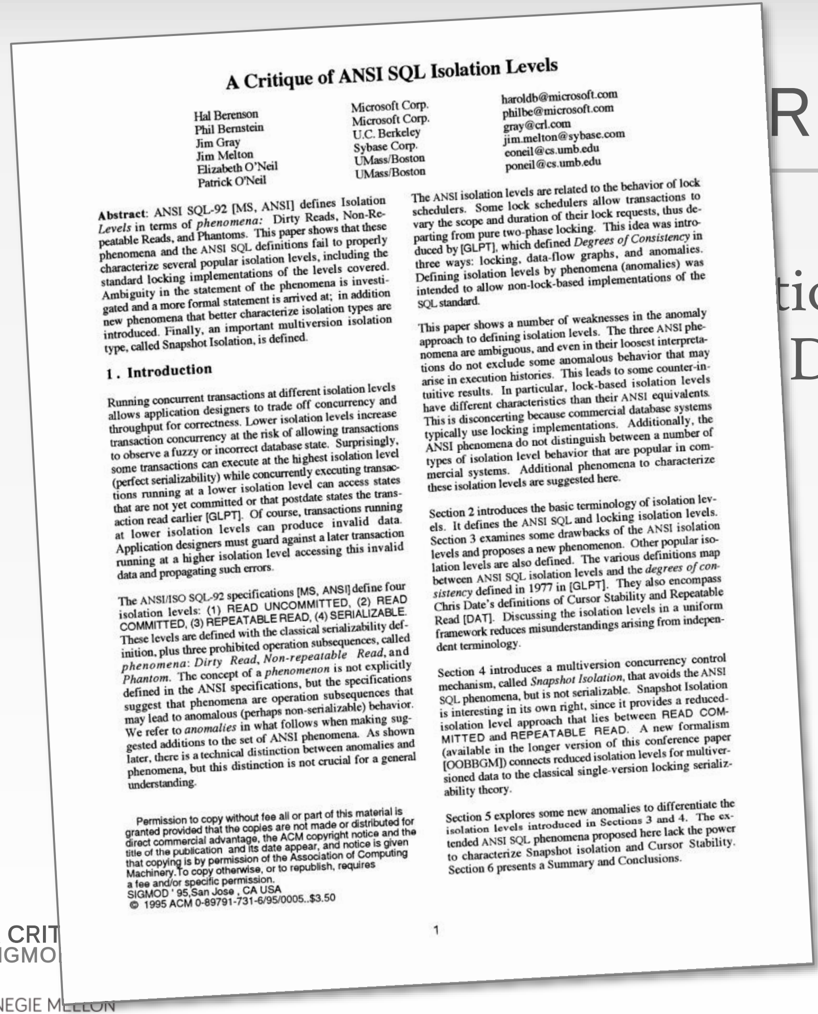
Cursor Stability, Snapshot Isolation, Disk Drive performance

From: Hal Berenson [redacted]
To: Andy Pavlo <pavlo@cs.cmu.edu>
Date: 1/22/19 4:43 PM

DEC Rdb (aka Rdb/VMS aka Oracle Rdb) originally just offered Serializable isolation level as Bill Noyce (the original project leader) was a purist. At some point after I took over Rdb/VMS I let customer feedback outweigh my own "religious" belief in favor of only offering Serializable and added Repeatable Read and Read Committed. Because of the way Rdb worked internally, the Read Committed implementation is actually Cursor Stability.

The reason the SQL-92 standard ended up the way it did is because Jim Melton DIDN'T want to use 2PL language to describe isolation levels. So Jim wrote it in terms on anomalies. He walked into my office and asked me to review his language, and being busy I didn't give it as thorough a review as I should have and missed that he allowed for lost updates within Serializability. In 1994 I was really annoyed that people inside Microsoft kept claiming what they were building was Seitalizable when it allowed lost updates, and was writing an email telling them why what they were doing was wrong. Pat O'Neil happened to walk into my office and asked what I was working on while I was madly pounding on the keyboard. He thought it was a problem in the standard, and when I re-read it I wanted to bang my head against a cement wall. Instead Pat and I decided to write the SIGMOD paper. Asking Jim Gray and Phil Bernstein to work on it was a no-brainer. Asking Jim Melton to join in as well brought it full circle.

A CRIT
SIGMO



A Critique of ANSI SQL Isolation Levels

Hal Berenson	Microsoft Corp.	haroldb@microsoft.com
Phil Bernstein	Microsoft Corp.	philbe@microsoft.com
Jim Gray	U.C. Berkeley	gray@crl.com
Jim Melton	Sybase Corp.	jim.melton@sybase.com
Elizabeth O'Neil	UMass/Boston	eoneil@cs.umb.edu
Patrick O'Neil	UMass/Boston	poneil@cs.umb.edu

Abstract: ANSI SQL-92 [MS, ANSI] defines Isolation Levels in terms of *phenomena*: Dirty Reads, Non-Repeatable Reads, and Phantoms. This paper shows that these phenomena and the ANSI SQL definitions fail to properly characterize several popular isolation levels, including the standard locking implementations of the levels covered. Ambiguity in the statement of the phenomena is investigated and a more formal statement is arrived at; in addition new phenomena that better characterize isolation types are introduced. Finally, an important multiversion isolation type, called Snapshot Isolation, is defined.

1. Introduction

Running concurrent transactions at different isolation levels allows application designers to trade off concurrency and throughput for correctness. Lower isolation levels increase transaction concurrency at the risk of allowing transactions to observe a fuzzy or incorrect database state. Surprisingly, some transactions can execute at the highest isolation level (perfect serializability) while concurrently executing transactions running at a lower isolation level can access states that are not yet committed or that postdate states the transaction read earlier [GLPT]. Of course, transactions running at lower isolation levels can produce invalid data. Application designers must guard against a later transaction running at a higher isolation level accessing this invalid data and propagating such errors.

The ANSI/ISO SQL-92 specifications [MS, ANSI] define four isolation levels: (1) READ UNCOMMITTED, (2) READ COMMITTED, (3) REPEATABLE READ, (4) SERIALIZABLE. These levels are defined with the classical serializability definition, plus three prohibited operation subsequences, called *phenomena: Dirty Read, Non-repeatable Read*, and *Phantom*. The concept of a *phenomenon* is not explicitly defined in the ANSI specifications, but the specifications suggest that phenomena are operation subsequences that may lead to anomalous (perhaps non-serializable) behavior. We refer to *anomalies* in what follows when making suggested additions to the set of ANSI phenomena. As shown later, there is a technical distinction between anomalies and phenomena, but this distinction is not crucial for a general understanding.

The ANSI isolation levels are related to the behavior of lock schedulers. Some lock schedulers allow transactions to vary the scope and duration of their lock requests, thus departing from pure two-phase locking. This idea was introduced by [GLPT], which defined *Degrees of Consistency* in three ways: locking, data-flow graphs, and anomalies. Defining isolation levels by phenomena (anomalies) was intended to allow non-lock-based implementations of the SQL standard.

This paper shows a number of weaknesses in the anomaly approach to defining isolation levels. The three ANSI phenomena are ambiguous, and even in their loosest interpretations do not exclude some anomalous behavior that may arise in execution histories. This leads to some counter-intuitive results. In particular, lock-based isolation levels have different characteristics than their ANSI equivalents. This is disconcerting because commercial database systems typically use locking implementations. Additionally, the ANSI phenomena do not distinguish between a number of types of isolation level behavior that are popular in commercial systems. Additional phenomena to characterize these isolation levels are suggested here.

Section 2 introduces the basic terminology of isolation levels. It defines the ANSI SQL and locking isolation levels. Section 3 examines some drawbacks of the ANSI isolation levels and proposes a new phenomenon. Other popular isolation levels are also defined. The various definitions map between ANSI SQL isolation levels and the *degrees of consistency* defined in 1977 in [GLPT]. They also encompass Chris Date's definitions of Cursor Stability and Repeatable Read [DAT]. Discussing the isolation levels in a uniform framework reduces misunderstandings arising from independent terminology.

Section 4 introduces a multiversion concurrency control mechanism, called *Snapshot Isolation*, that avoids the ANSI SQL phenomena, but is not serializable. Snapshot Isolation is interesting in its own right, since it provides a reduced-isolation level approach that lies between READ COMMITTED and REPEATABLE READ. A new formalism (available in the longer version of this conference paper [OOBBGM]) connects reduced isolation levels for multiversioned data to the classical single-version locking serializability theory.

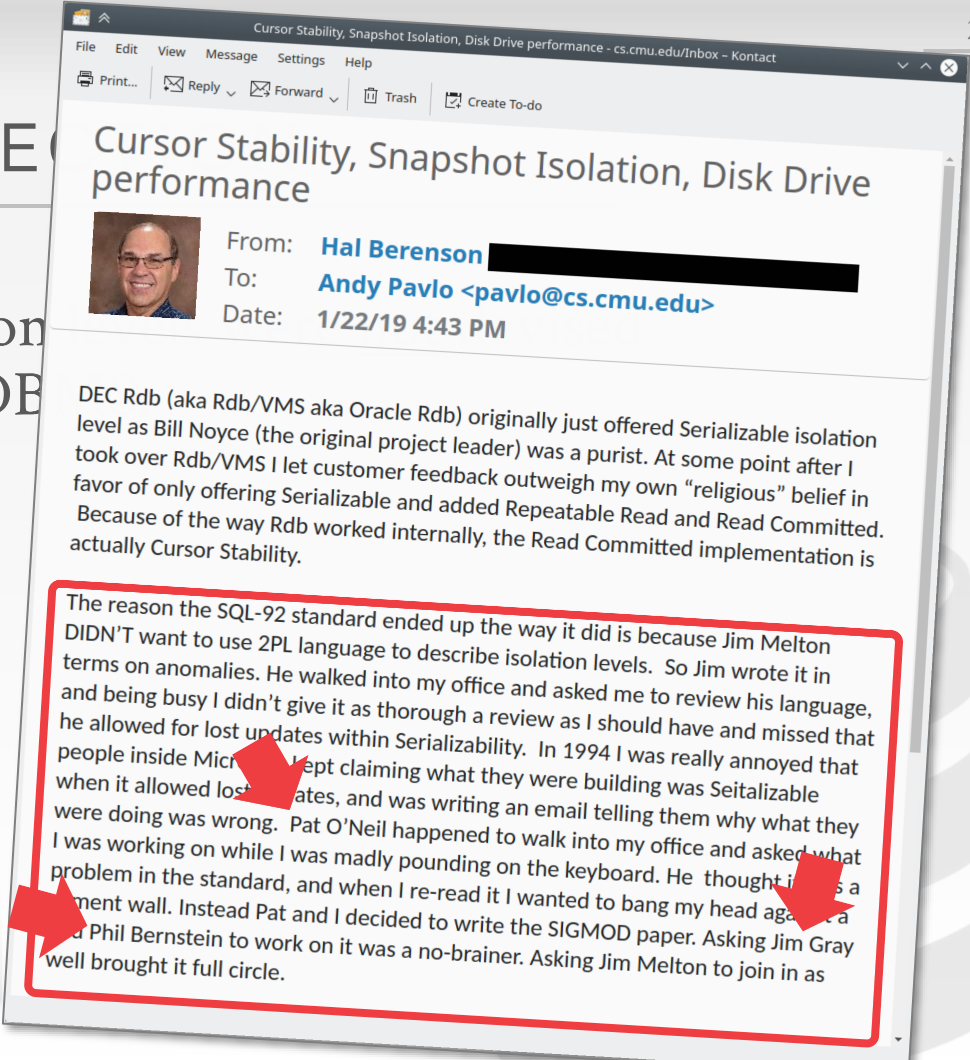
Section 5 explores some new anomalies to differentiate the isolation levels introduced in Sections 3 and 4. The extended ANSI SQL phenomena proposed here lack the power to characterize Snapshot isolation and Cursor Stability. Section 6 presents a Summary and Conclusions.

1

RE

tion

DB



Cursor Stability, Snapshot Isolation, Disk Drive performance - cs.cmu.edu/Inbox – Kontact

File Edit View Message Settings Help

Print... Reply Forward Trash Create To-do

Cursor Stability, Snapshot Isolation, Disk Drive performance

From: **Hal Berenson** [redacted]
To: **Andy Pavlo <pavlo@cs.cmu.edu>**
Date: **1/22/19 4:43 PM**

DEC Rdb (aka Rdb/VMS aka Oracle Rdb) originally just offered Serializable isolation level as Bill Noyce (the original project leader) was a purist. At some point after I took over Rdb/VMS I let customer feedback outweigh my own "religious" belief in favor of only offering Serializable and added Repeatable Read and Read Committed. Because of the way Rdb worked internally, the Read Committed implementation is actually Cursor Stability.

The reason the SQL-92 standard ended up the way it did is because Jim Melton DIDN'T want to use 2PL language to describe isolation levels. So Jim wrote it in terms on anomalies. He walked into my office and asked me to review his language, and being busy I didn't give it as thorough a review as I should have and missed that he allowed for lost updates within Serializability. In 1994 I was really annoyed that people inside Micr... kept claiming what they were building was Seitalizable when it allowed lost ...ates, and was writing an email telling them why what they were doing was wrong. Pat O'Neil happened to walk into my office and asked what I was working on while I was madly pounding on the keyboard. He thought i... a problem in the standard, and when I re-read it I wanted to bang my head aga... ...ment wall. Instead Pat and I decided to write the SIGMOD paper. Asking Jim Gray ...d Phil Bernstein to work on it was a no-brainer. Asking Jim Melton to join in as well brought it full circle.

A CRIT
SIGMO

TODAY'S AGENDA

Overview of In-Memory MVCC

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

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.

SNAPSHOT ISOLATION

When a txn starts, it sees a consistent snapshot of the database that existed at the moment that the txn started.

- → No torn writes from active txns.
- → If two txns update the same object, then first writer wins.

We get SI automatically for "free" with MVCC.

- → If we want serializable isolation, then the DBMS has to do extra stuff…

MVCC DESIGN DECISIONS

Concurrency Control Protocol

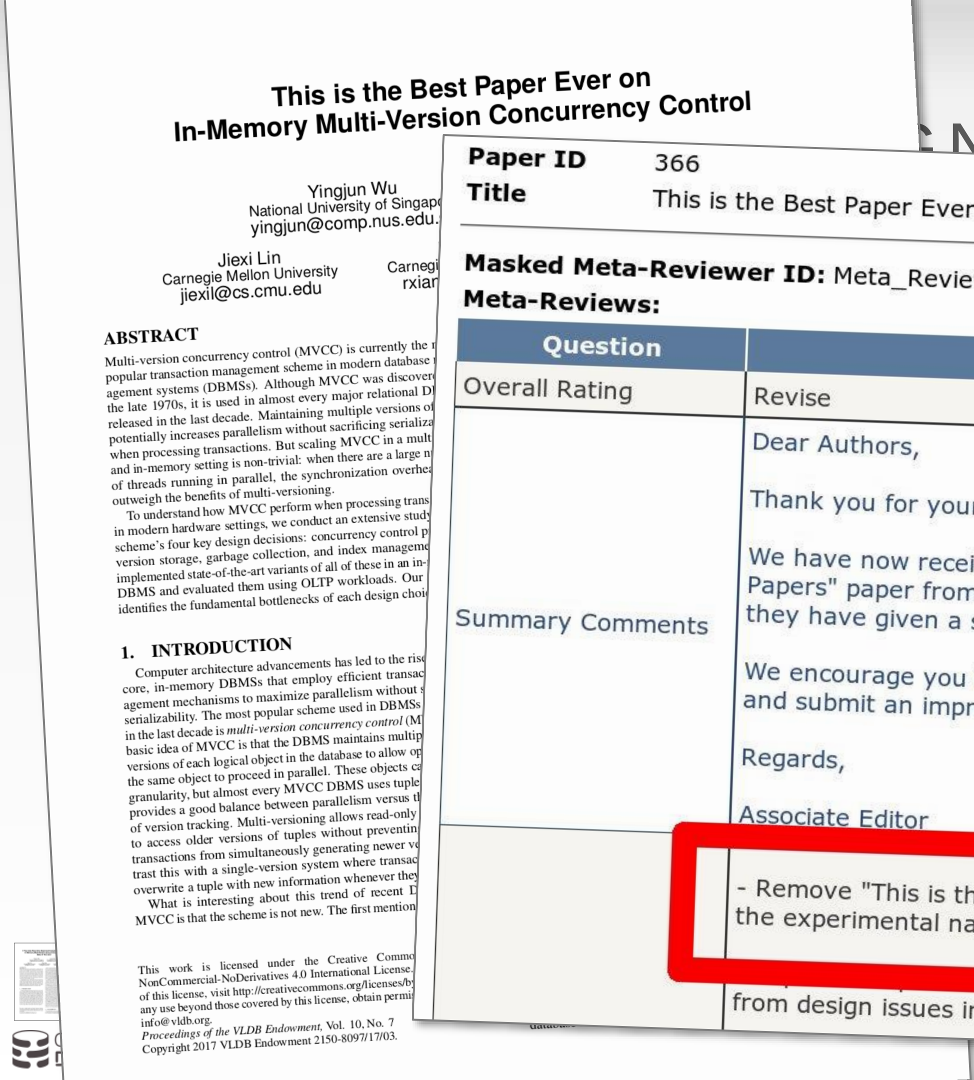
Version Storage

Garbage Collection

Index Management

AN EMPIRICAL EVALUATION OF IN-MEMORY MULTI-VERSION CONCURRENCY CONTROL
VLDB 2017

GN DECISIONS



This is the Best Paper Ever on In-Memory Multi-Version Concurrency Control

Yingjun Wu
National University of Singapo
yingjun@comp.nus.edu.

Jiexi Lin
Carnegie Mellon University
jiexil@cs.cmu.edu

Carnegi
rxian

ABSTRACT

Multi-version concurrency control (MVCC) is currently the m
popular transaction management scheme in modern database
agement systems (DBMSs). Although MVCC was discovere
the late 1970s, it is used in almost every major relational D
released in the last decade. Maintaining multiple versions of
potentially increases parallelism without sacrificing serializa
when processing transactions. But scaling MVCC in a mult
and in-memory setting is non-trivial: when there are a large n
of threads running in parallel, the synchronization overhea
outweigh the benefits of multi-versioning.

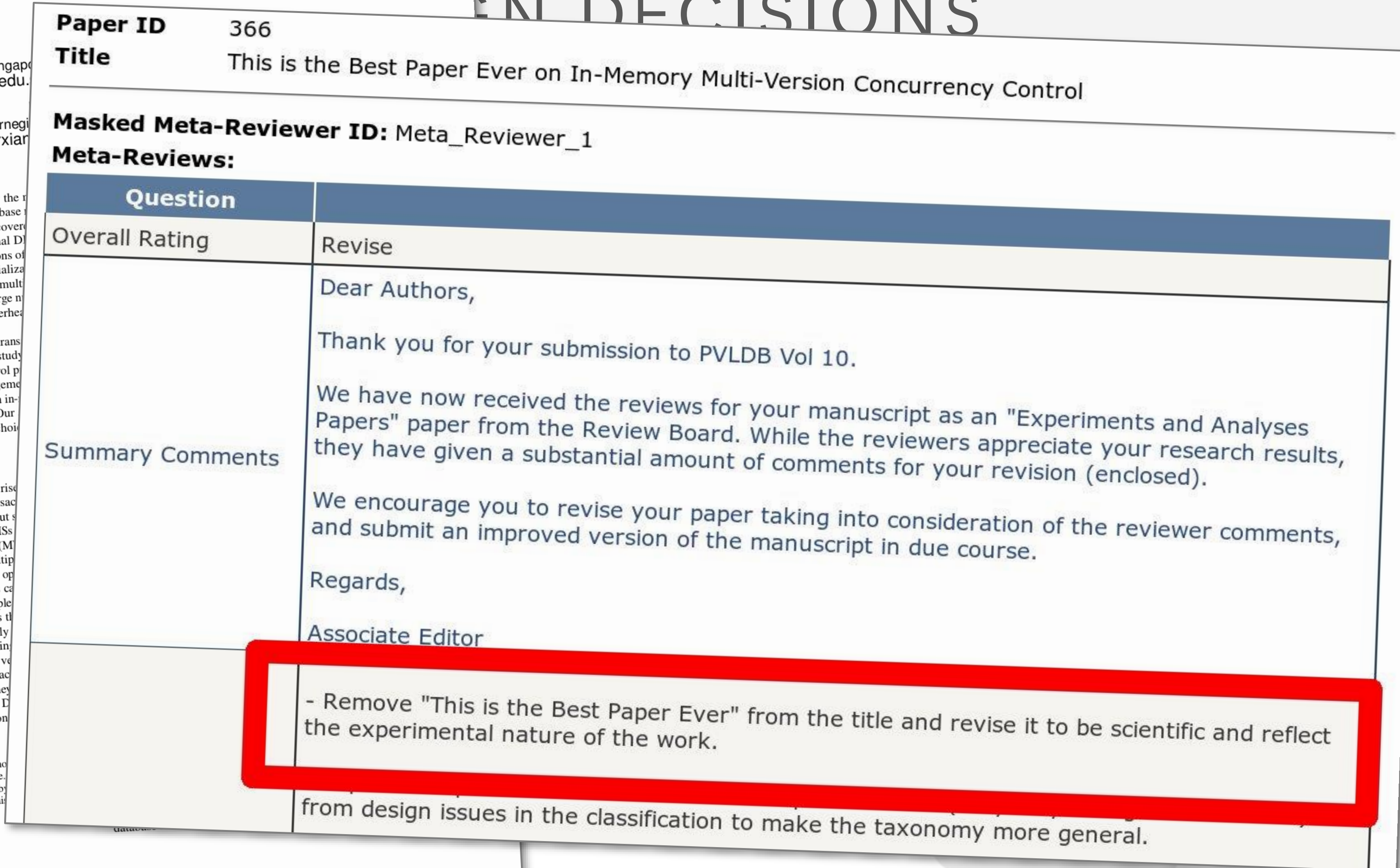
To understand how MVCC perform when processing trans
in modern hardware settings, we conduct an extensive study
scheme's four key design decisions: concurrency control p
version storage, garbage collection, and index manageme
implemented state-of-the-art variants of all of these in an in-
DBMS and evaluated them using OLTP workloads. Our
identifies the fundamental bottlenecks of each design choi

1. INTRODUCTION

Computer architecture advancements has led to the rise
core, in-memory DBMSs that employ efficient transac
agement mechanisms to maximize parallelism without s
serializability. The most popular scheme used in DBMSs
in the last decade is *multi-version concurrency control* (M
basic idea of MVCC is that the DBMS maintains multip
versions of each logical object in the database to allow op
the same object to proceed in parallel. These objects ca
granularity, but almost every MVCC DBMS uses tuple
provides a good balance between parallelism versus t
of version tracking. Multi-versioning allows read-only
to access older versions of tuples without preventin
transactions from simultaneously generating newer ve
trast this with a single-version system where transac
overwrite a tuple with new information whenever they

What is interesting about this trend of recent D
MVCC is that the scheme is not new. The first mention

Proceedings of the VLDB Endowment, Vol. 10, No. 7
Copyright 2017 VLDB Endowment 2150-8097/17/03.



Paper ID 366

Title This is the Best Paper Ever on In-Memory Multi-Version Concurrency Control

Masked Meta-Reviewer ID: Meta_Reviewer_1

Meta-Reviews:

Question	
Overall Rating	Revise
Summary Comments	Dear Authors, Thank you for your submission to PVLDB Vol 10. We have now received the reviews for your manuscript as an "Experiments and Analyses Papers" paper from the Review Board. While the reviewers appreciate your research results, they have given a substantial amount of comments for your revision (enclosed). We encourage you to revise your paper taking into consideration of the reviewer comments, and submit an improved version of the manuscript in due course. Regards, Associate Editor
	- Remove "This is the Best Paper Ever" from the title and revise it to be scientific and reflect the experimental nature of the work. from design issues in the classification to make the taxonomy more general.

...CISIONS

This is the Best Paper Ever on
In-Mem...

If You Only Read One Empirical Evaluation Paper on In-Memory Multi-Version Concurrency Control, Make It This One!

Yingjun Wu
National University of Singapore
yingjun@comp.nus.edu.sg

Joy Arulraj
Carnegie Mellon University
jarulraj@cs.cmu.edu

Jiexi Lin
Carnegie Mellon University
jiexil@cs.cmu.edu

Ran Xian
Carnegie Mellon University
rxian@cs.cmu.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

ABSTRACT

Multi-version concurrency control (MVCC) is currently the most popular transaction management scheme in modern database management systems (DBMSs). Although MVCC was discovered in the late 1970s, it is used in almost every major relational DBMS released in the last decade. Maintaining multiple versions of data potentially increases parallelism without sacrificing serializability when processing transactions. But scaling MVCC in a multi-core and in-memory setting is non-trivial: when there are a large number of threads running in parallel, the synchronization overhead can outweigh the benefits of multi-versioning.

To understand how MVCC perform when processing transactions in modern hardware settings, we conduct an extensive study of the scheme's four key design decisions: concurrency control protocol, version storage, garbage collection, and index management. We implemented state-of-the-art variants of all of these in an in-memory DBMS and evaluated them using OLTP workloads. Our analysis identifies the fundamental bottlenecks of each design choice.

1. INTRODUCTION

Computer architecture advancements has led to the rise of multi-core, in-memory DBMSs that employ efficient transaction management mechanisms to maximize parallelism without sacrificing serializability. The most popular scheme used in DBMSs developed in the last decade is *multi-version concurrency control* (MVCC). The basic idea of MVCC is that the DBMS maintains multiple physical versions of each logical object in the database to allow operations on the same object to proceed in parallel. These objects can be at any granularity, but almost every MVCC DBMS uses tuples because it provides a good balance between parallelism versus the overhead of version tracking. Multi-versioning allows read-only transactions to access older versions of tuples without preventing read-write transactions from simultaneously generating newer versions. Contrast this with a single-version system where transactions always overwrite a tuple with new information whenever they update it.

What is interesting about this trend of recent DBMSs using MVCC is that the scheme is not new. The first mention of it appeared in a 1979 dissertation [38] and the first implementation started in 1981 [22] for the InterBase DBMS (now open-sourced as Firebird). MVCC is also used in some of the most widely deployed disk-oriented DBMSs today, including Oracle (since 1984 [4]), Postgres (since 1985 [41]), and MySQL's InnoDB engine (since 2001). But while there are plenty of contemporaries to these older systems that use a single-version scheme (e.g., IBM DB2, Sybase), almost every new transactional DBMS eschews this approach in favor of MVCC [37]. This includes both commercial (e.g., Microsoft Hekaton [16], SAP HANA [40], MemSQL [1], NuoDB [3]) and academic (e.g., HYRISE [21], HyPer [36]) systems.

Despite all these newer systems using MVCC, there is no one "standard" implementation. There are several design choices that have different trade-offs and performance behaviors. Until now, there has not been a comprehensive evaluation of MVCC in a modern DBMS operating environment. The last extensive study was in the 1980s [13], but it used simulated workloads running in a disk-oriented DBMS with a single CPU core. The design choices of legacy disk-oriented DBMSs are inappropriate for in-memory DBMSs running on a machine with a large number of CPU cores. As such, this previous work does not reflect recent trends in latch-free [27] and serializable [20] concurrency control, as well as in-memory storage [36] and hybrid workloads [40].

In this paper, we perform such a study for key transaction management design decisions in of MVCC DBMSs: (1) concurrency control protocol, (2) version storage, (3) garbage collection, and (4) index management. For each of these topics, we describe the state-of-the-art implementations for in-memory DBMSs and discuss their trade-offs. We also highlight the issues that prevent them from scaling to support larger thread counts and more complex workloads. As part of this investigation, we implemented all of the approaches in the **Peloton** [5] in-memory MVCC DBMS. This provides us with a uniform platform to compare implementations that is not encumbered by other architecture facets. We deployed Peloton on a machine with 40 cores and evaluate it using two OLTP benchmarks. Our analysis identifies the scenarios that stress the implementations and discuss ways to mitigate them (if it all possible).

...ulti-Version Concurrency Control

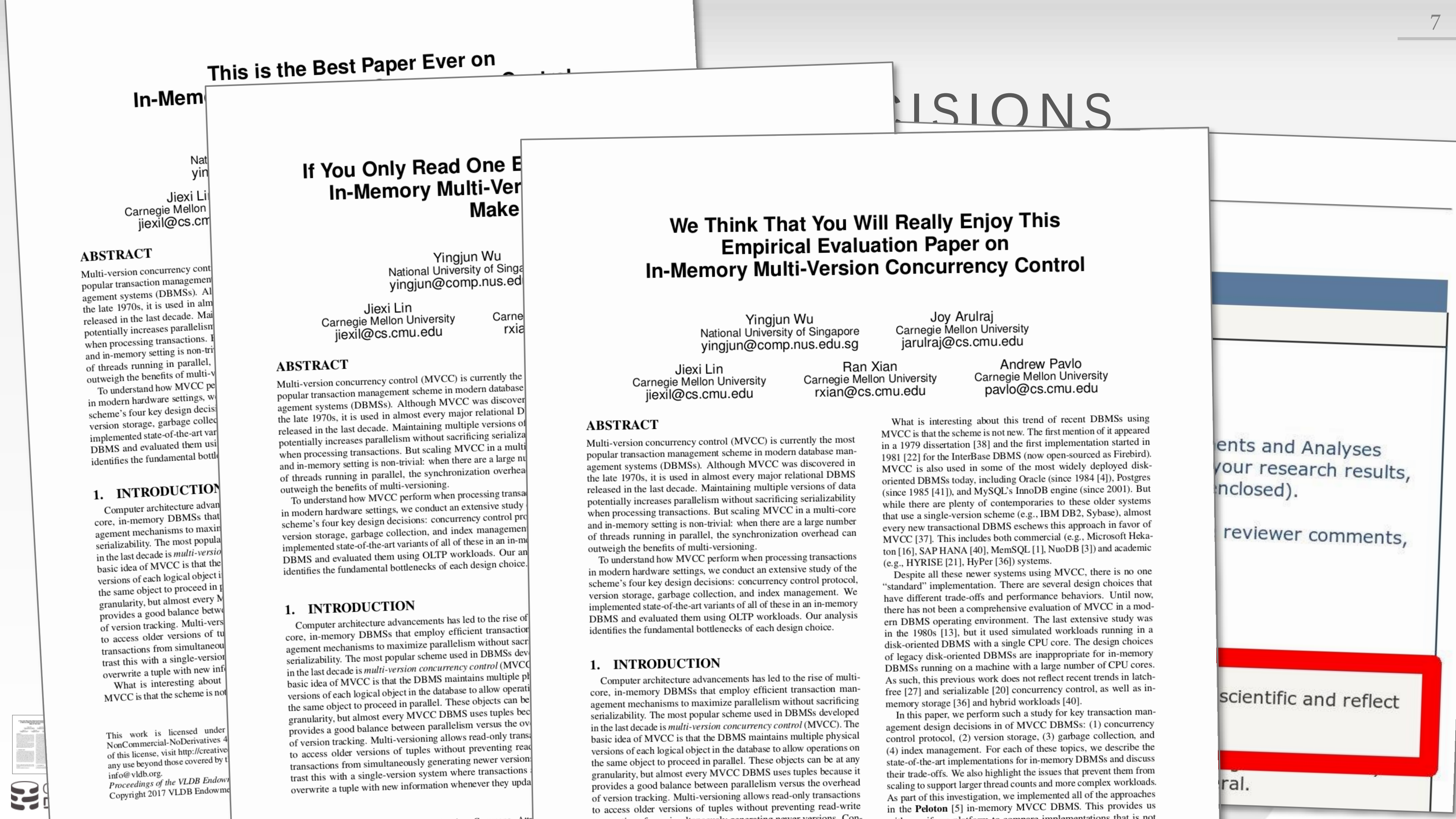
...LDB Vol 10.

...r your manuscript as an "Experiments and Analyses ...l. While the reviewers appreciate your research results, ...t of comments for your revision (enclosed).

...er taking into consideration of the reviewer comments, ...e manuscript in due course.

... from the title and revise it to be scientific and reflect

...o make the taxonomy more general.



We Think That You Will Really Enjoy This Empirical Evaluation Paper on In-Memory Multi-Version Concurrency Control

Yingjun Wu
National University of Singapore
yingjun@comp.nus.edu.sg

Joy Arulraj
Carnegie Mellon University
jarulraj@cs.cmu.edu

Jiexi Lin
Carnegie Mellon University
jiexil@cs.cmu.edu

Ran Xian
Carnegie Mellon University
rxian@cs.cmu.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

ABSTRACT

Multi-version concurrency control (MVCC) is currently the most popular transaction management scheme in modern database management systems (DBMSs). Although MVCC was discovered in the late 1970s, it is used in almost every major relational DBMS released in the last decade. Maintaining multiple versions of data potentially increases parallelism without sacrificing serializability when processing transactions. But scaling MVCC in a multi-core and in-memory setting is non-trivial: when there are a large number of threads running in parallel, the synchronization overhead can outweigh the benefits of multi-versioning.

To understand how MVCC perform when processing transactions in modern hardware settings, we conduct an extensive study of the scheme's four key design decisions: concurrency control protocol, version storage, garbage collection, and index management. We implemented state-of-the-art variants of all of these in an in-memory DBMS and evaluated them using OLTP workloads. Our analysis identifies the fundamental bottlenecks of each design choice.

1. INTRODUCTION

Computer architecture advancements has led to the rise of multi-core, in-memory DBMSs that employ efficient transaction management mechanisms to maximize parallelism without sacrificing serializability. The most popular scheme used in DBMSs developed in the last decade is *multi-version concurrency control* (MVCC). The basic idea of MVCC is that the DBMS maintains multiple physical versions of each logical object in the database to allow operations on the same object to proceed in parallel. These objects can be at any granularity, but almost every MVCC DBMS uses tuples because it provides a good balance between parallelism versus the overhead of version tracking. Multi-versioning allows read-only transactions to access older versions of tuples without preventing read-write

What is interesting about this trend of recent DBMSs using MVCC is that the scheme is not new. The first mention of it appeared in a 1979 dissertation [38] and the first implementation started in 1981 [22] for the InterBase DBMS (now open-sourced as Firebird). MVCC is also used in some of the most widely deployed disk-oriented DBMSs today, including Oracle (since 1984 [4]), Postgres (since 1985 [41]), and MySQL's InnoDB engine (since 2001). But while there are plenty of contemporaries to these older systems that use a single-version scheme (e.g., IBM DB2, Sybase), almost every new transactional DBMS eschews this approach in favor of MVCC [37]. This includes both commercial (e.g., Microsoft Hekaton [16], SAP HANA [40], MemSQL [1], NuoDB [3]) and academic (e.g., HYRISE [21], HyPer [36]) systems.

Despite all these newer systems using MVCC, there is no one "standard" implementation. There are several design choices that have different trade-offs and performance behaviors. Until now, there has not been a comprehensive evaluation of MVCC in a modern DBMS operating environment. The last extensive study was in the 1980s [13], but it used simulated workloads running in a disk-oriented DBMS with a single CPU core. The design choices of legacy disk-oriented DBMSs are inappropriate for in-memory DBMSs running on a machine with a large number of CPU cores. As such, this previous work does not reflect recent trends in latch-free [27] and serializable [20] concurrency control, as well as in-memory storage [36] and hybrid workloads [40].

In this paper, we perform such a study for key transaction management design decisions in of MVCC DBMSs: (1) concurrency control protocol, (2) version storage, (3) garbage collection, and (4) index management. For each of these topics, we describe the state-of-the-art implementations for in-memory DBMSs and discuss their trade-offs. We also highlight the issues that prevent them from scaling to support larger thread counts and more complex workloads. As part of this investigation, we implemented all of the approaches in the **Peloton** [5] in-memory MVCC DBMS. This provides us

An Empirical Evaluation of In-Memory Multi-Version Concurrency Control

Yingjun Wu
National University of Singapore
yingjun@comp.nus.edu.sg

Joy Arulraj
Carnegie Mellon University
jarulraj@cs.cmu.edu

Jiexi Lin
Carnegie Mellon University
jiexil@cs.cmu.edu

Ran Xian
Carnegie Mellon University
rxian@cs.cmu.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

ABSTRACT

Multi-version concurrency control (MVCC) is currently the most popular transaction management scheme in modern database management systems (DBMSs). Although MVCC was discovered in the late 1970s, it is used in almost every major relational DBMS released in the last decade. Maintaining multiple versions of data potentially increases parallelism without sacrificing serializability when processing transactions. But scaling MVCC in a multi-core and in-memory setting is non-trivial: when there are a large number of threads running in parallel, the synchronization overhead can outweigh the benefits of multi-versioning.

To understand how MVCC perform when processing transactions in modern hardware settings, we conduct an extensive study of the scheme's four key design decisions: concurrency control protocol, version storage, garbage collection, and index management. We implemented state-of-the-art variants of all of these in an in-memory DBMS and evaluated them using OLTP workloads. Our analysis identifies the fundamental bottlenecks of each design choice.

1. INTRODUCTION

Computer architecture advancements has led to the rise of multi-core, in-memory DBMSs that employ efficient transaction management mechanisms to maximize parallelism without sacrificing serializability. The most popular scheme used in DBMSs developed in the last decade is *multi-version concurrency control* (MVCC). The basic idea of MVCC is that the DBMS maintains multiple physical versions of each logical object in the database to allow operations on the same object to proceed in parallel. These objects can be at any granularity, but almost every MVCC DBMS uses tuples because it provides a good balance between parallelism versus the overhead of version tracking. Multi-versioning allows read-only transactions to access older versions of tuples without preventing read-write transactions from simultaneously generating newer versions. Contrast this with a single-version system where transactions always overwrite a tuple with new information whenever they update it.

What is interesting about this trend of recent DBMSs using MVCC is that the scheme is not new. The first mention of it appeared in a 1979 dissertation [38] and the first implementation started in 1981 [22] for the InterBase DBMS (now open-sourced as Firebird). MVCC is also used in some of the most widely deployed disk-oriented DBMSs today, including Oracle (since 1984 [4]), Postgres (since 1985 [41]), and MySQL's InnoDB engine (since 2001). But while there are plenty of contemporaries to these older systems that use a single-version scheme (e.g., IBM DB2, Sybase), almost every new transactional DBMS eschews this approach in favor of MVCC [37]. This includes both commercial (e.g., Microsoft Hekaton [16], SAP HANA [40], MemSQL [1], NuoDB [3]) and academic (e.g., HYRISE [21], HyPer [36]) systems.

Despite all these newer systems using MVCC, there is no one "standard" implementation. There are several design choices that have different trade-offs and performance behaviors. Until now, there has not been a comprehensive evaluation of MVCC in a modern DBMS operating environment. The last extensive study was in the 1980s [13], but it used simulated workloads running in a disk-oriented DBMS with a single CPU core. The design choices of legacy disk-oriented DBMSs are inappropriate for in-memory DBMSs running on a machine with a large number of CPU cores. As such, this previous work does not reflect recent trends in latch-free [27] and serializable [20] concurrency control, as well as in-memory storage [36] and hybrid workloads [40].

In this paper, we perform such a study for key transaction management design decisions in of MVCC DBMSs: (1) concurrency control protocol, (2) version storage, (3) garbage collection, and (4) index management. For each of these topics, we describe the state-of-the-art implementations for in-memory DBMSs and discuss their trade-offs. We also highlight the issues that prevent them from scaling to support larger thread counts and more complex workloads. As part of this investigation, we implemented all of the approaches in the **Peloton** [5] in-memory MVCC DBMS. This provides us with a uniform platform to compare implementations that is not encumbered by other architecture facets. We deployed Peloton on a machine with 40 cores and evaluate it using two OLTP benchmarks. Our analysis identifies the scenarios that stress the implementations and discuss ways to mitigate them (if it all possible).

2. BACKGROUND

We first provide an overview of the high-level concepts of MVCC. We then discuss the meta-data that the DBMS uses to track transactions and maintain versioning information.

2.1 MVCC Overview

Proceedings of the VLDB Endowment, Vol. 10, No. 7

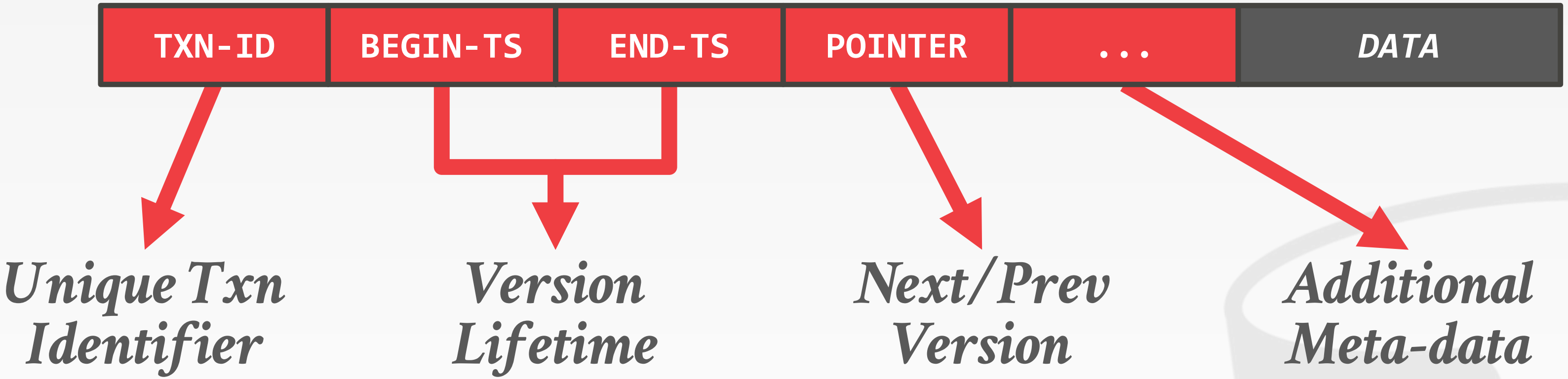
CONCURRENCY CONTROL PROTOCOL

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)

VERSION	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

- READ(A)
- WRITE(B)

VERSION	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)

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

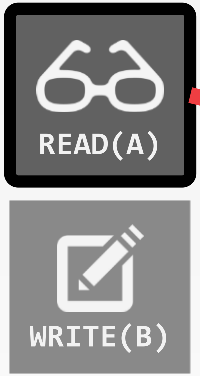
Use ***read-ts*** field in the header to keep track of the timestamp of the last txn that read it.

Txn is allowed to 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



VERSION	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	$\boxed{1}$	1	∞
B_1	0	0	1	∞

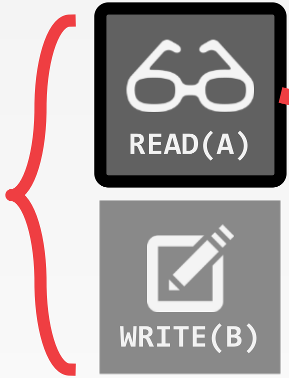
Use ***read-ts*** field in the header to keep track of the timestamp of the last txn that read it.

Txn is allowed to 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



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

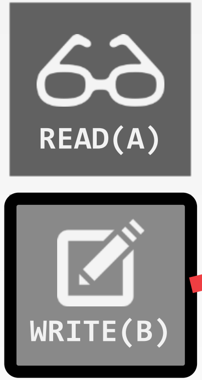
Use ***read-ts*** field in the header to keep track of the timestamp of the last txn that read it.

Txn is allowed to 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



VERSION	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 last txn that read it.

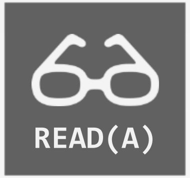
Txn is allowed to read version if the latch is unset and its T_{id} is between ***begin-ts*** and ***end-ts***.

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



VERSION	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 is allowed to 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)

VERSION	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 is allowed to 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)

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

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

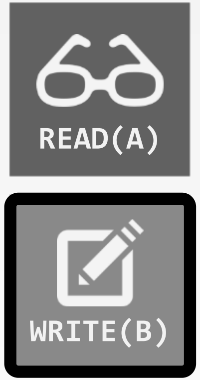
Txn is allowed to read version if the latch is unset and its T_{id} is between ***begin-ts*** and ***end-ts***.

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

TIMESTAMP ORDERING (MVTO)

Thread #1

$T_{id}=10$



VERSION	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 last txn that read it.

Txn is allowed to 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)

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

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 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)

VERSION	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)

VERSION	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.

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

READ(A)

WRITE(B)

VERSION	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)

VERSION	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.

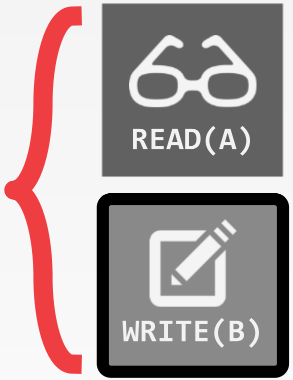
If ***txn-id*** is zero, then the txn acquires the SHARED lock by incrementing the ***read-cnt*** field.

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



VERSION	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)

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

TWO-PHASE LOCKING (MV2PL)

Thread #1

T_{id}=10

READ(A)

WRITE(B)

VERSION	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.

TWO-PHASE LOCKING (MV2PL)

Thread #1

T_{id}=10

READ(A)

WRITE(B)

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

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

WRITE(A)

VERSION	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	-	99999	∞

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)

VERSION	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)

VERSION	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

$T_{id}=1$

WRITE(A)

VERSION	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

$T_{id}=1$

WRITE(A)

VERSION	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

$T_{id}=1$

VERSION	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.

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 has to 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.

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

All of 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

VERSION	VALUE	POINTER
A_0	$111	●
A_1	$222	Ø
B_1	$10	Ø

APPEND-ONLY STORAGE

All of 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

VERSION	VALUE	POINTER
A_0	$111	
A_1	$222	Ø
B_1	$10	Ø
A_2	$333	Ø

APPEND-ONLY STORAGE

All of 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

VERSION	VALUE	POINTER
A_0	$111	•
A_1	$222	•
B_1	$10	Ø
A_2	$333	Ø

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.

TIME-TRAVEL STORAGE

Main Table

VERSION	VALUE	POINTER
A_2	$222	•
B_1	$10	

Time-Travel Table

VERSION	VALUE	POINTER
A_1	$111	Ø

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

TIME-TRAVEL STORAGE

Main Table

VERSION	VALUE	POINTER
A_2	$222	●
B_1	$10	

Time-Travel Table

VERSION	VALUE	POINTER
A_1	$111	Ø
A_2	$222	●

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

TIME-TRAVEL STORAGE

Main Table

VERSION	VALUE	POINTER
A_3	*$333*	
B_1	*$10*	

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

Time-Travel Table

VERSION	VALUE	POINTER
A_1	*$111*	Ø
A_2	*$222*	

Overwrite master version in the main table.
Update pointers.

TIME-TRAVEL STORAGE

Main Table

VERSION	VALUE	POINTER
A_3	$333	●
B_1	$10	

Time-Travel Table

VERSION	VALUE	POINTER
A_1	$111	Ø
A_2	$222	●

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

Overwrite master version in the main table. Update pointers.

DELTA STORAGE

Main Table

VERSION	VALUE	POINTER
A_1	$111	
B_1	$10	

Delta Storage Segment

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

DELTA STORAGE

Main Table

VERSION	VALUE	POINTER
A_1	$111	
B_1	$10	

Delta Storage Segment

	DELTA	POINTER
A_1	(VALUE→$111)	Ø

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

DELTA STORAGE

Main Table

VERSION	VALUE	POINTER
A_2	*$222*	•
B_1	*$10*	

Delta Storage Segment

	DELTA	POINTER
A_1	*(VALUE→$111)*	Ø

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

DELTA STORAGE

Main Table

VERSION	VALUE	POINTER
A_2	$222	
B_1	$10	

Delta Storage Segment

	DELTA	POINTER
A_1	(VALUE→$111)	Ø
A_2	(VALUE→$222)	

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

DELTA STORAGE

Main Table

VERSION	VALUE	POINTER
A_3	$\boxed{\$333}$	●
B_1	\$10	

Delta Storage Segment

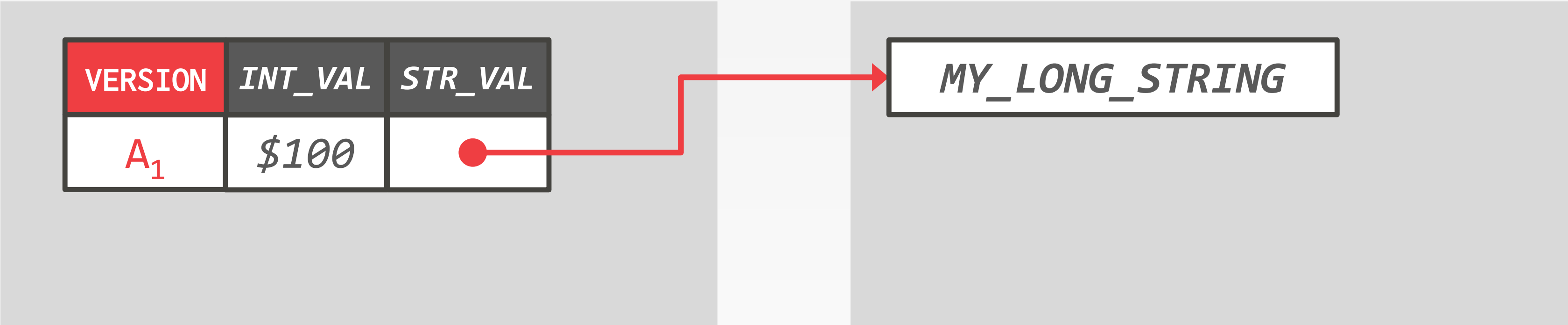
	DELTA	POINTER
A_1	(VALUE→\$111)	Ø
A_2	(VALUE→\$222)	●

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.

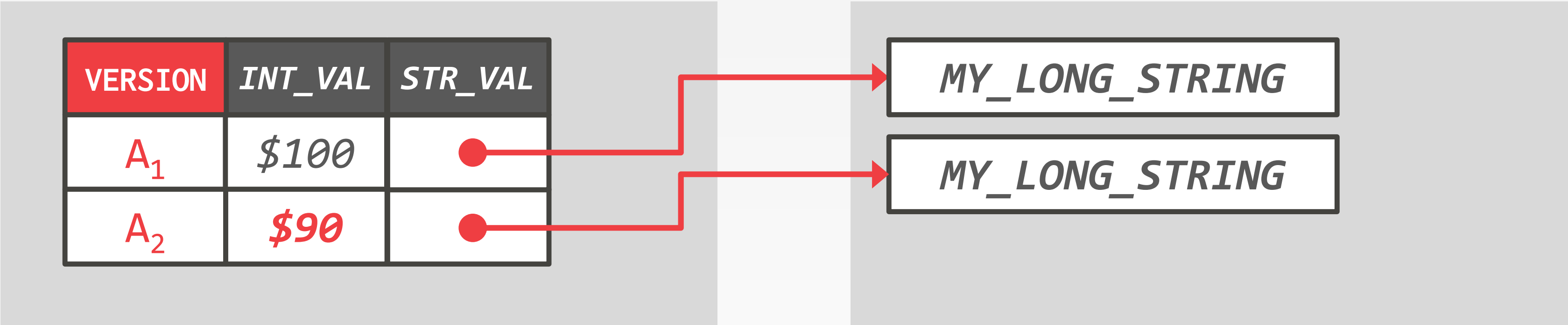
NON-INLINE ATTRIBUTES

Main Table *Variable-Length Data*



NON-INLINE ATTRIBUTES

Main Table *Variable-Length Data*



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

NON-INLINE ATTRIBUTES

Main Table

VERSION	*INT_VAL*	*STR_VAL*
A_1	*$100*	•
A_2	***$90***	

Variable-Length Data

Refs=1	*MY_LONG_STRING*

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.

NON-INLINE ATTRIBUTES

Main Table

VERSION	*INT_VAL*	*STR_VAL*
A_1	*$100*	•
A_2	***$90***	•

Variable-Length Data

Refs=2	*MY_LONG_STRING*

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

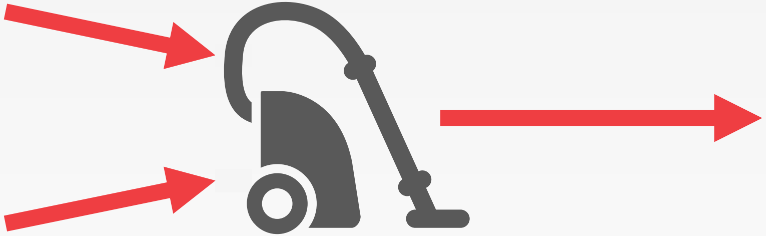
Thread #1

T_{id}=12

Thread #2

T_{id}=25

Vacuum



VERSION	BEGIN-TS	END-TS
A_{100}	1	9
B_{100}	1	9
B_{101}	10	20

Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

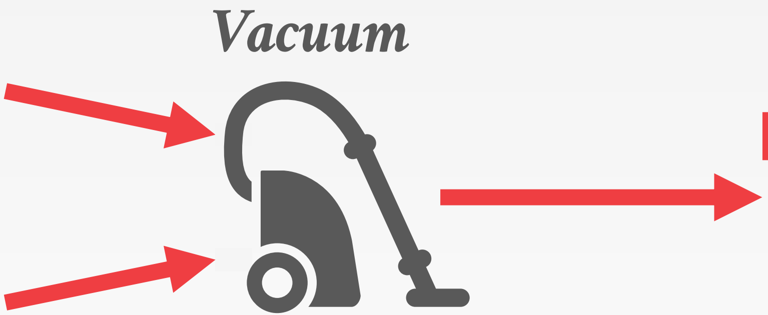
TUPLE-LEVEL GC

Thread #1

T_{id}=12

Thread #2

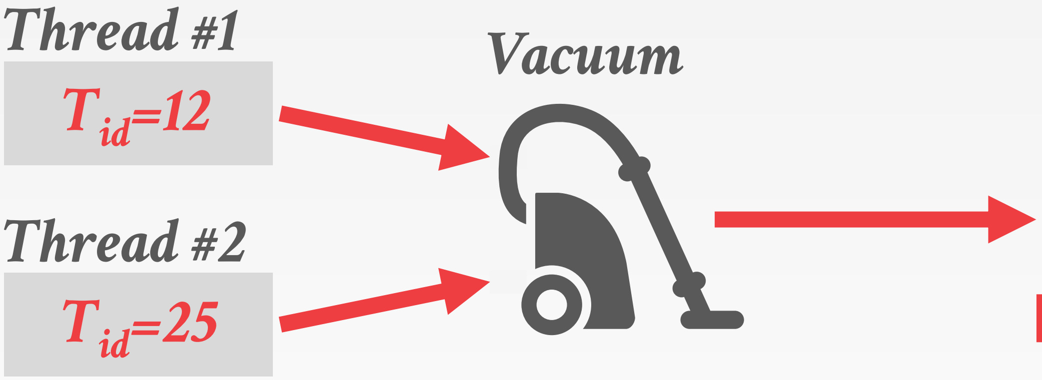
T_{id}=25



VERSION	BEGIN-TS	END-TS
A_{100}	1	9
B_{100}	1	9
B_{101}	10	20

Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

TUPLE-LEVEL GC



VERSION	BEGIN-TS	END-TS
A_{100}	1	9
B_{100}	1	9
B_{101}	10	20

Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

TUPLE-LEVEL GC

Thread #1

T_{id}=12

Thread #2

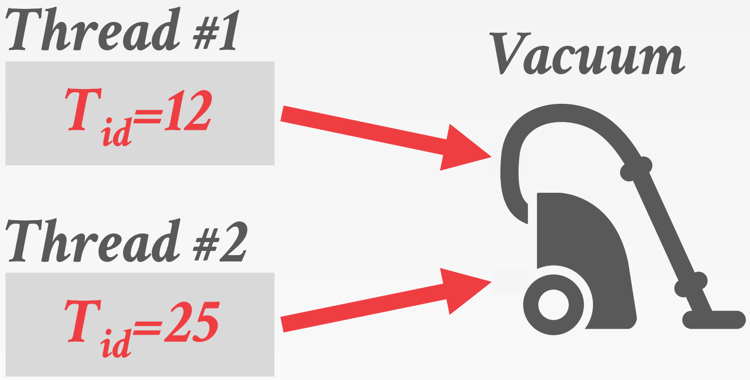
T_{id}=25

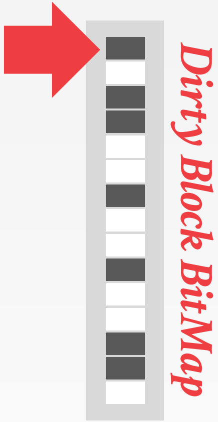
Vacuum

VERSION	BEGIN-TS	END-TS
B_{101}	10	20

Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

TUPLE-LEVEL GC





VERSION	BEGIN-TS	END-TS
B_{101}	10	20

Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

TUPLE-LEVEL GC

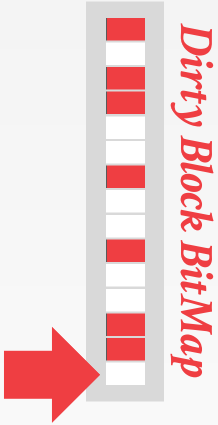
Thread #1

T_{id}=12

Thread #2

T_{id}=25

Vacuum



VERSION	BEGIN-TS	END-TS
B_{101}	10	20

Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

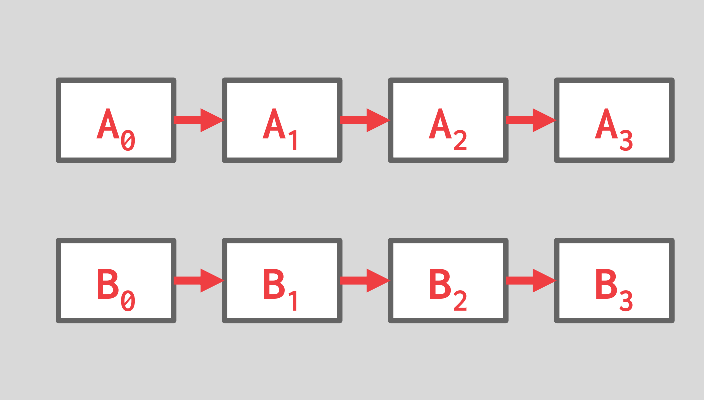
TUPLE-LEVEL GC

Thread #1

T_{id}=12

Thread #2

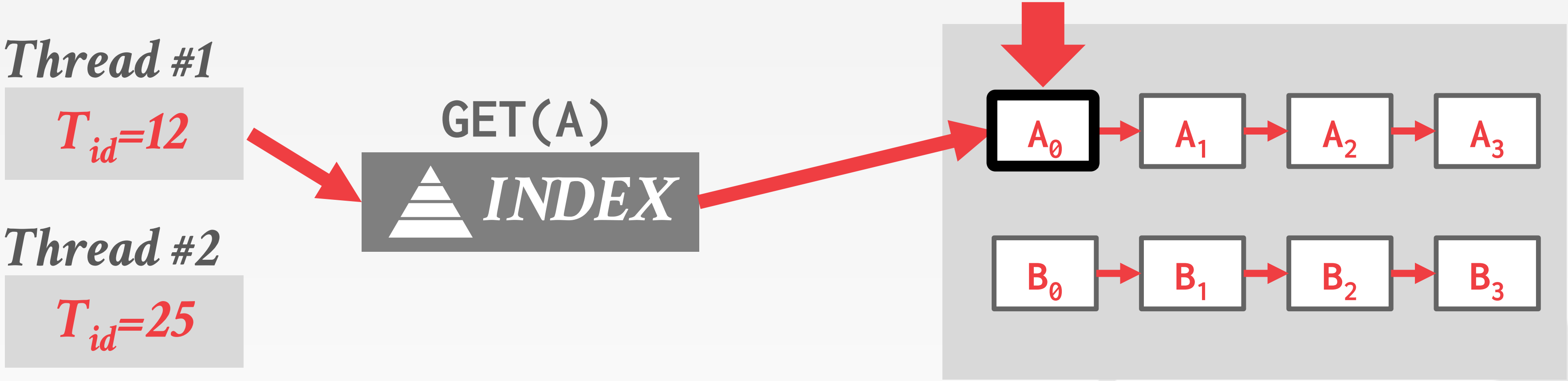
T_{id}=25



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.

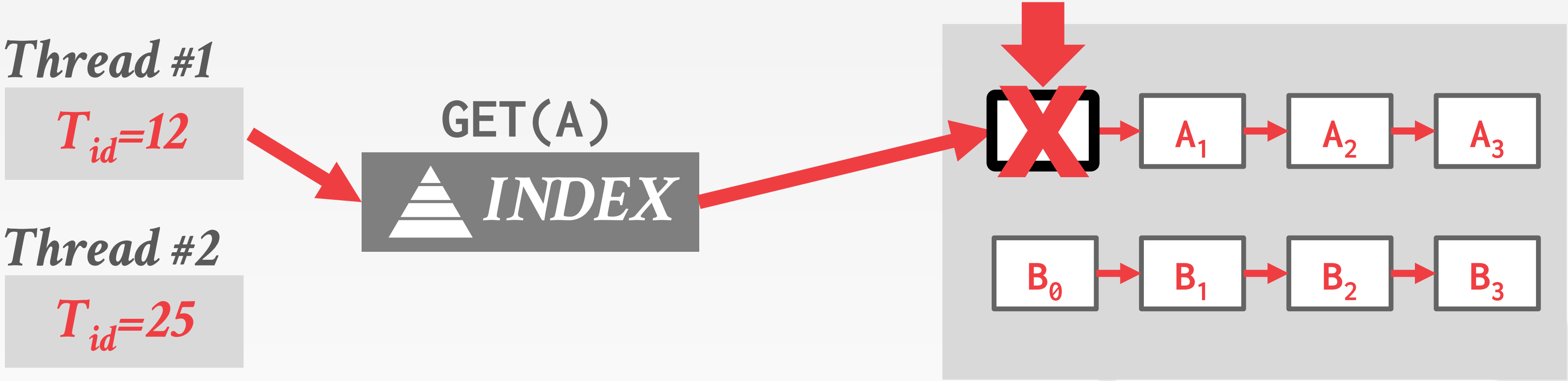
TUPLE-LEVEL GC



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.

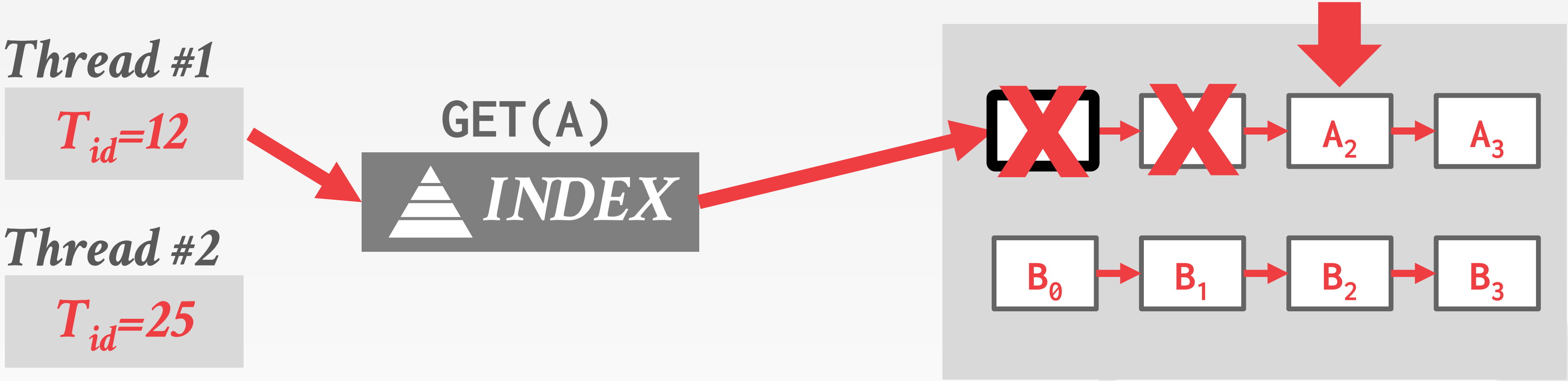
TUPLE-LEVEL GC



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.

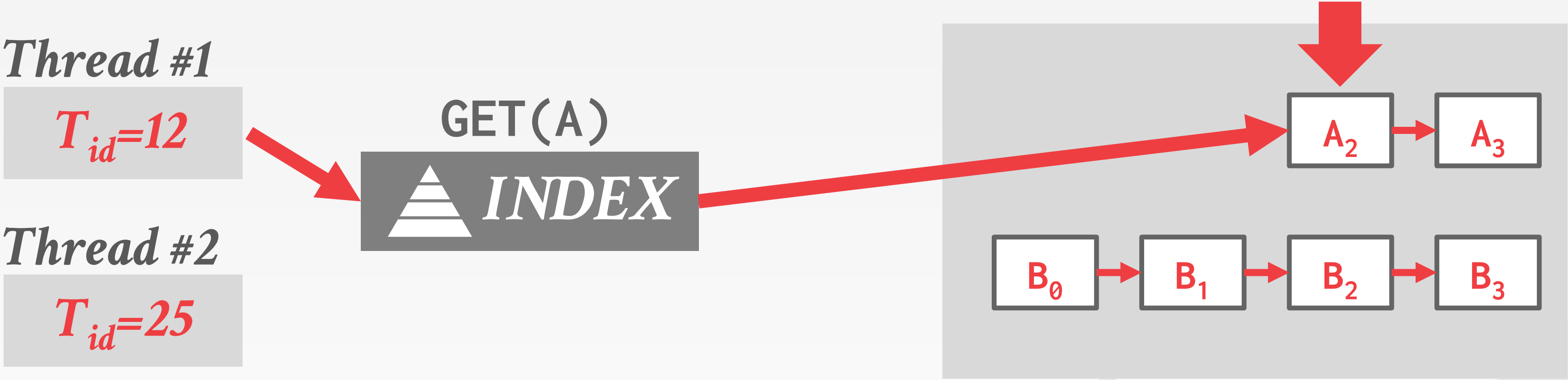
TUPLE-LEVEL GC



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.

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.

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.

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…

UBER Engineering

JOIN THE TEAM

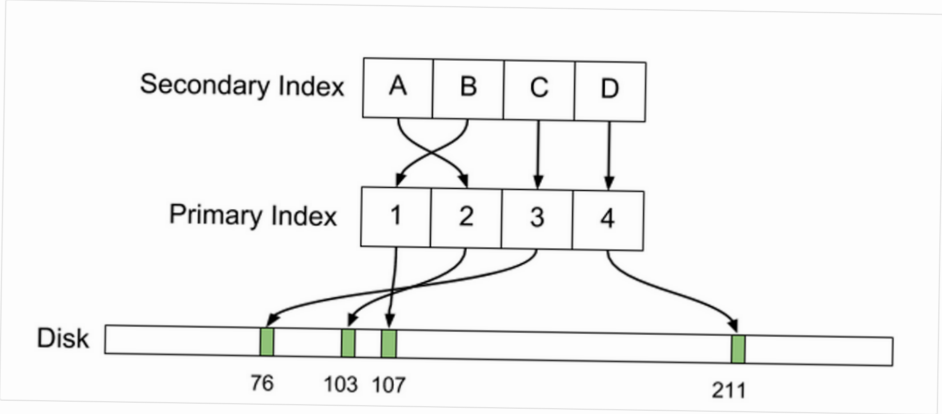
MEET THE PEOPLE

ARCHITECTURE

WHY UBER ENGINEERING SWITCHED FROM POSTGRES TO MYSQL

JULY 26, 2016

BY EVAN KLITZKE



SECONDARY INDEXES

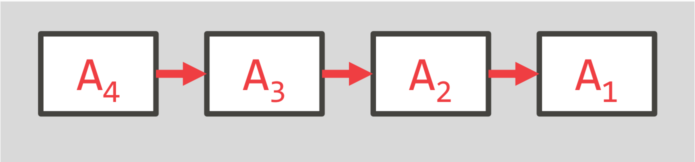
Approach #1: Logical Pointers
→ Use a fixed identifier per tuple that does not change.
→ Requires an extra indirection layer.
→ Primary Key vs. Tuple Id

Approach #2: Physical Pointers
→ Use the physical address to the version chain head.

INDEX POINTERS

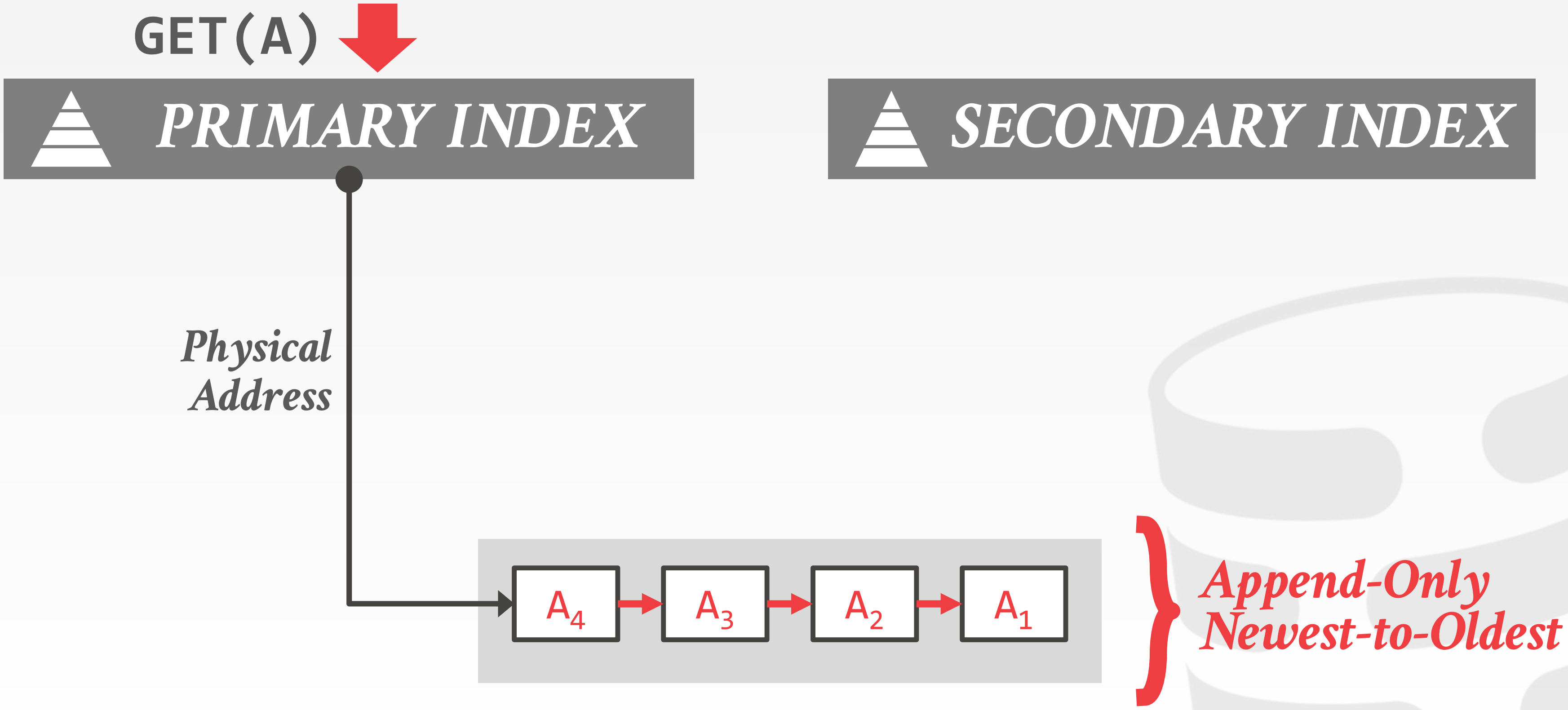
PRIMARY INDEX

SECONDARY INDEX

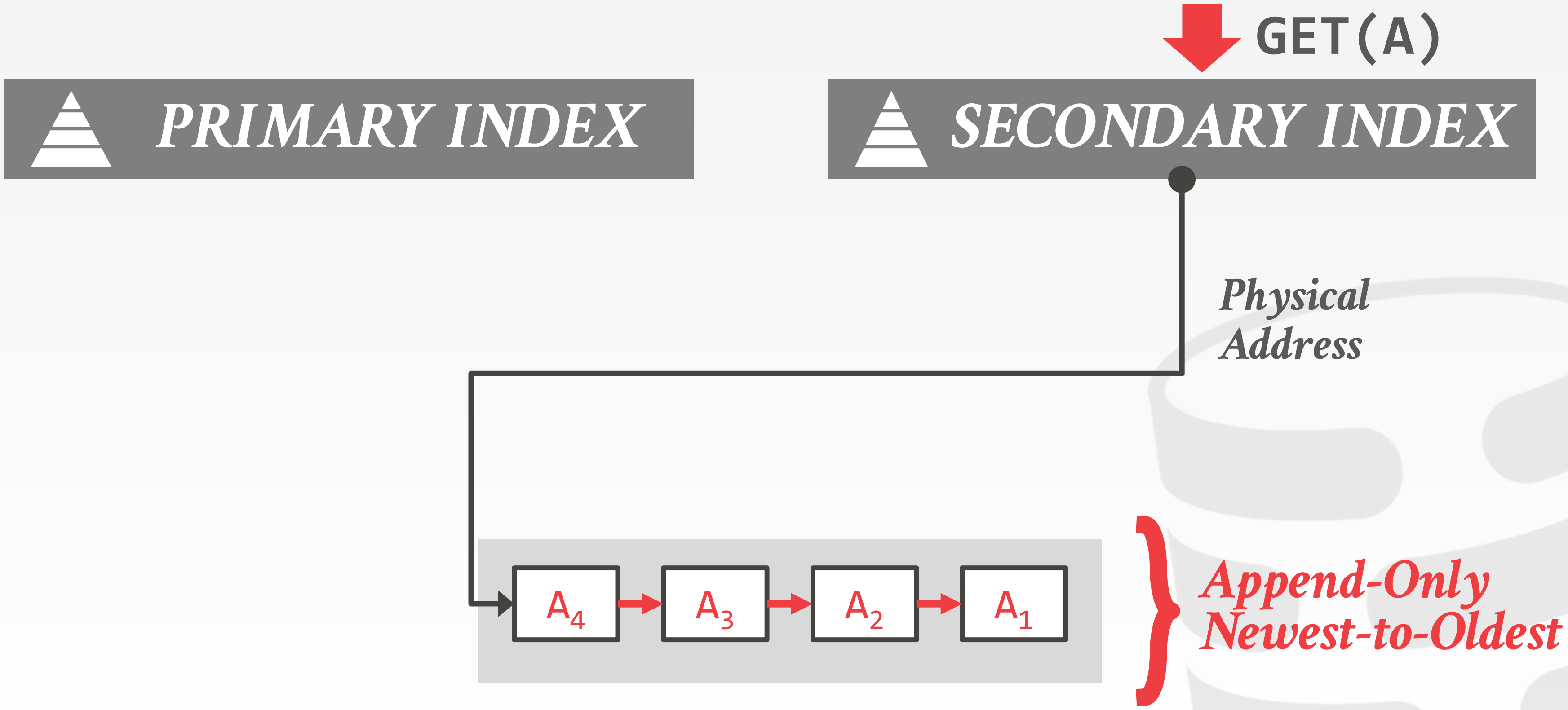


Append-Only
Newest-to-Oldest

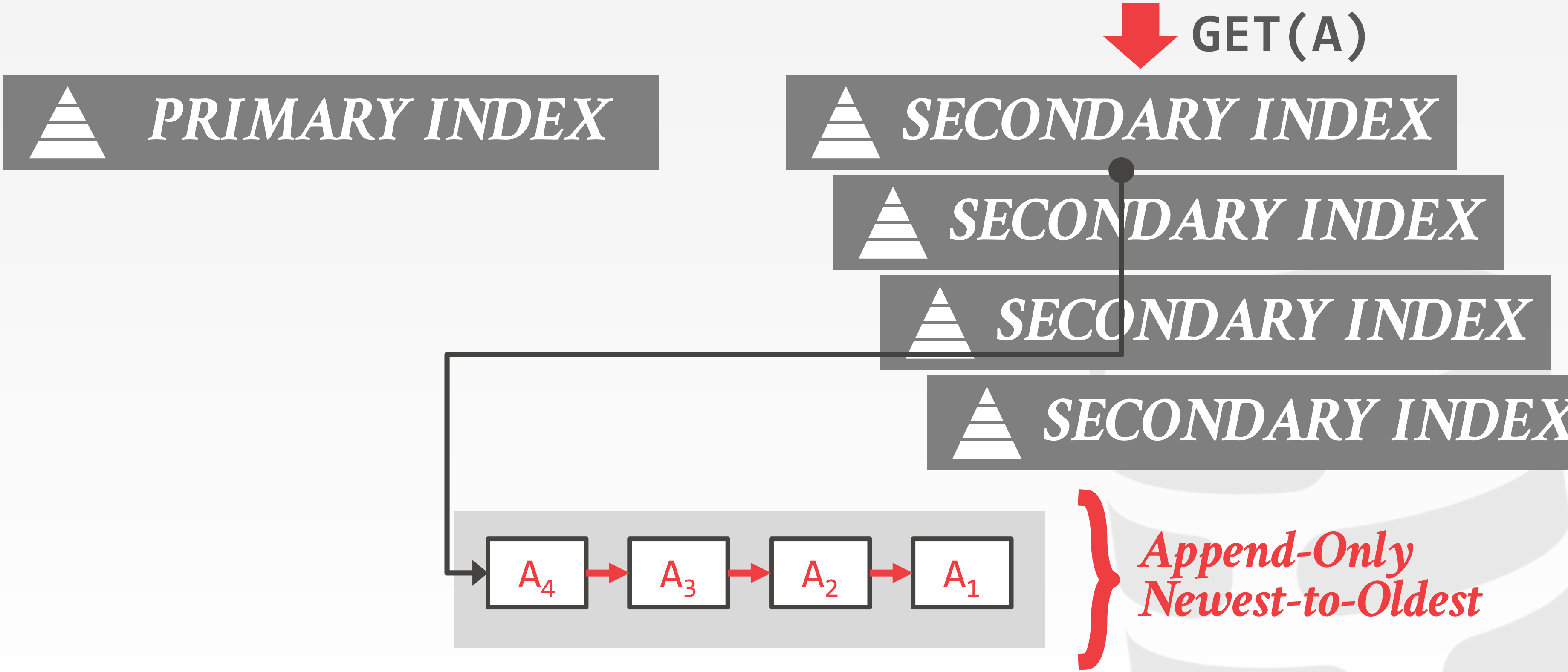
INDEX POINTERS



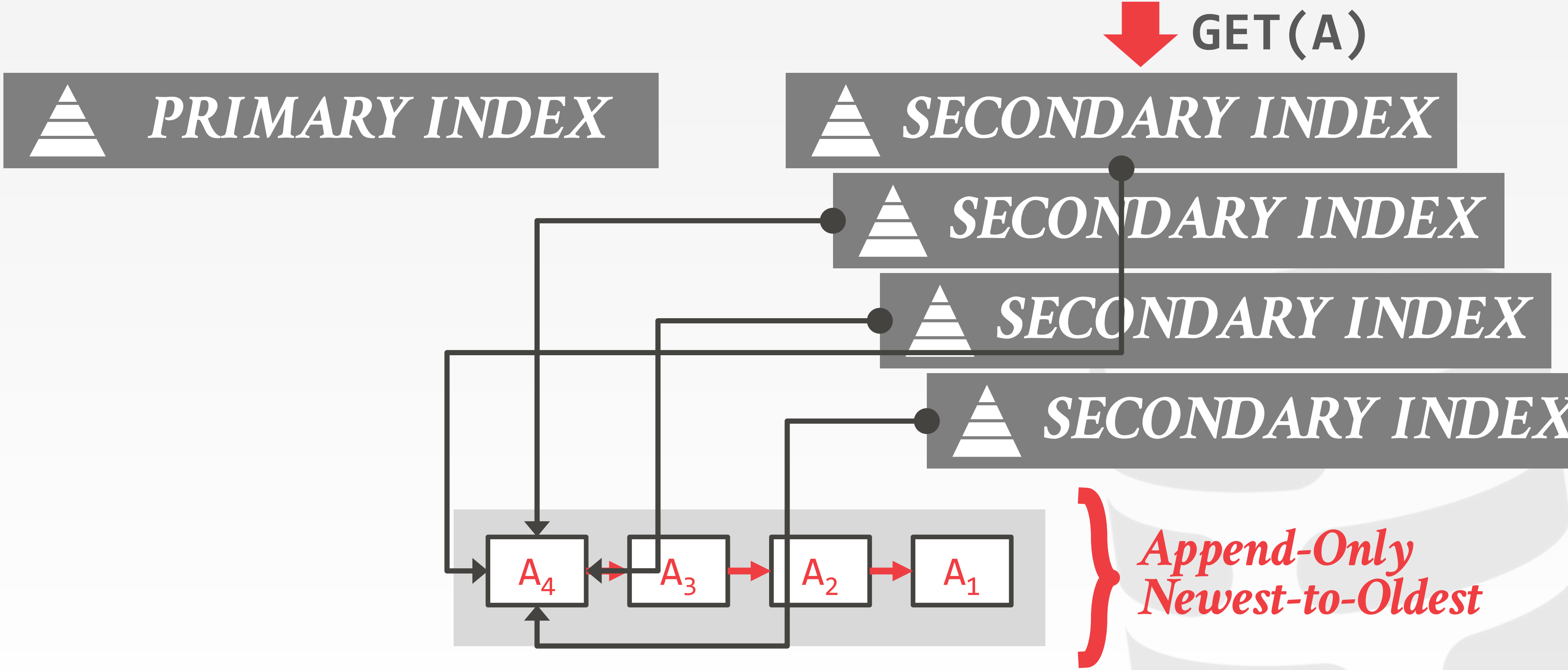
INDEX POINTERS



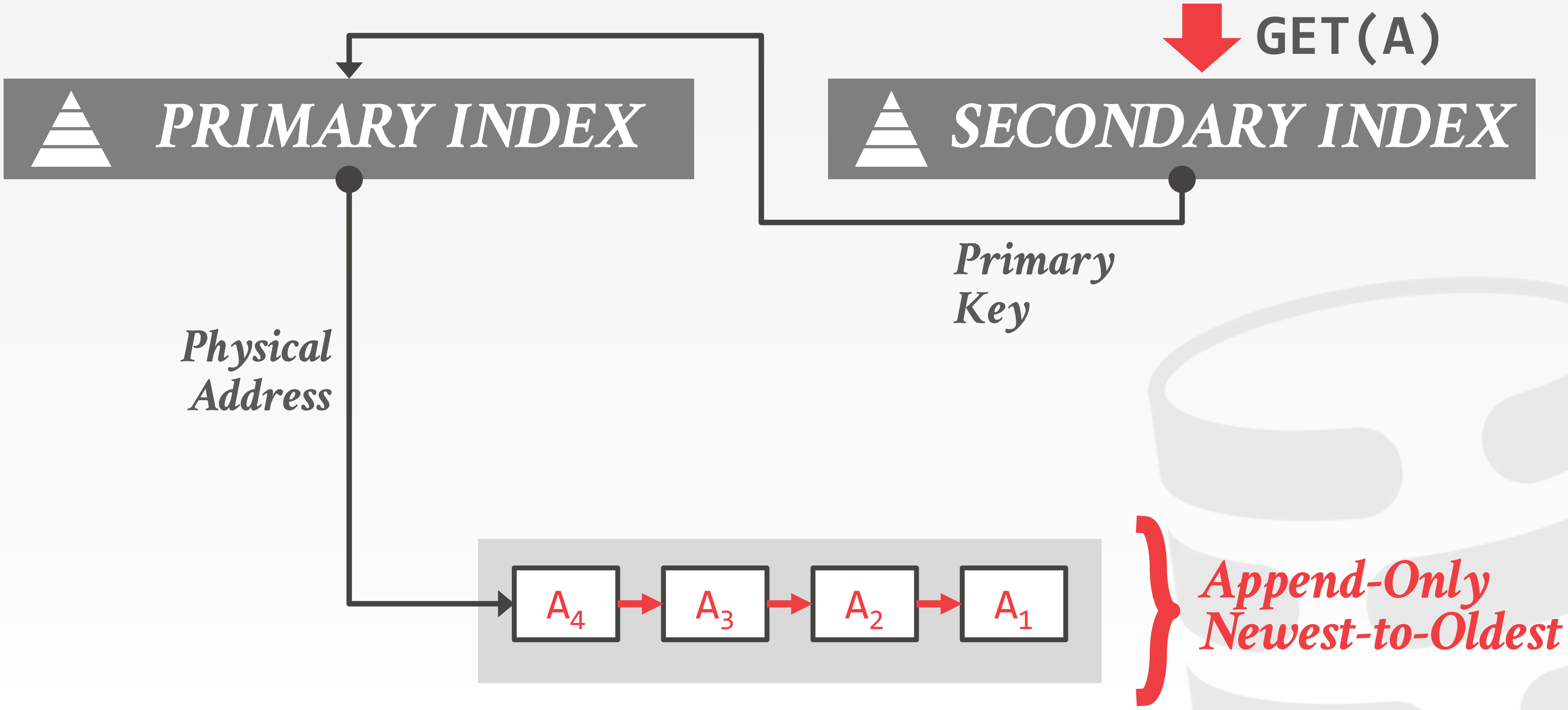
INDEX POINTERS



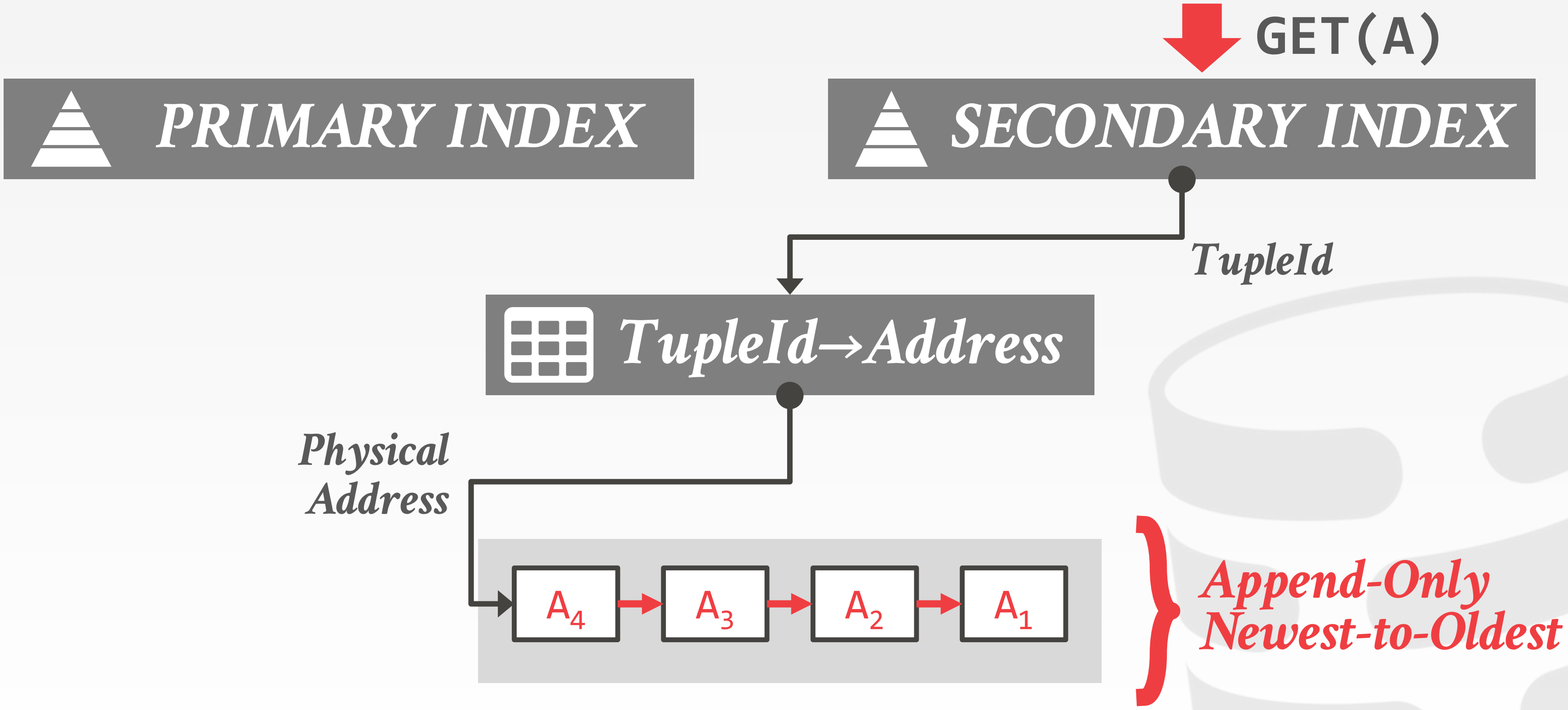
INDEX POINTERS



INDEX POINTERS



INDEX POINTERS



MVCC EVALUATION PAPER

We implemented all of the design decisions in the Peloton DBMS as part of 15-721 in Spring 2016.

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

AN EMPIRICAL EVALUATION OF IN-MEMORY MULTI-VERSION CONCURRENCY CONTROL
VLDB 2017

MVCC DESIGN DECISIONS

CC Protocol: Inconclusive results…

Version Storage: Deltas

Garbage Collection: Tuple-Level Vacuuming

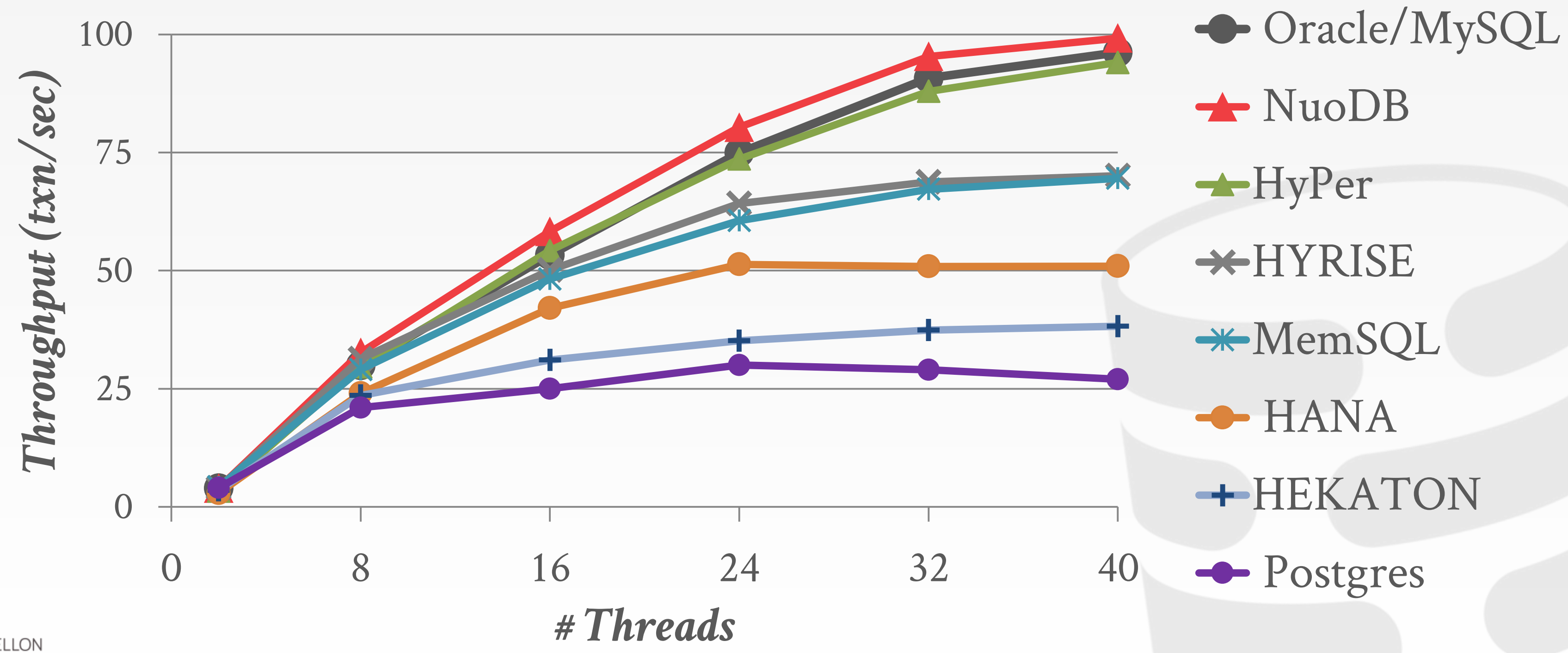
Indexes: Logical Pointers

MVCC CONFIGURATION EVALUATION

	Protocol	*Version Storage*	*Garbage Collection*	*Indexes*
Oracle	MV2PL	Delta	Vacuum	Logical
Postgres	MV-2PL/MV-TO	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
CMU's TBD	MV-OCC	Delta	Txn-level	Logical

MVCC CONFIGURATION EVALUATION

Database: TPC-C Benchmark (40 Warehouses)
Processor: 4 sockets, 10 cores per socket



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)

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

NEXT CLASS

Modern MVCC Implementations
→ TUM HyPer
→ CMU Cicada
→ Microsoft Hekaton