Multi-Version Concurrency Control (Design Decisions)
CORRECTION

Original SQL-92 isolation levels were not devised assuming a 2PL-based DBMS.
A Critique of ANSI SQL Isolation Levels

Hal Berenson

Abstract: ANSI SQL-92 (ANSI) defines isolation levels for database transactions that are based on two-phase locking (2PL). This paper argues that the isolation levels defined in ANSI-92 are not sufficient to protect against anomalies in database transactions. Specifically, the isolation levels are not strong enough to prevent update anomalies, and they do not provide a clear way to specify the level of locking required for a transaction. The paper proposes a new set of isolation levels that are stronger than those defined in ANSI-92, and it provides examples to demonstrate how the new levels can be used to specify different levels of locking in a transaction.
A Critique of ANSI SQL Isolation Levels

Abstract: ANSI SQL-92 defines isolation levels in terms of phenomena. Two phenomena, Namely, Loss of Locking and Phantom Locking, are properly defined, but the notion of isolation levels, assuming the traditional locking implementation of the level, cannot stand standard implementation of the level. Proposed isolation level definitions are inconsistent with the traditional locking implementation of the level. Moreover, the phenomenon of lock starvation is not properly defined.

INTRODUCTION

Constraint-violating statements at different isolation levels allow applications to violate constraints. Thus, they allow applications to violate the database's consistency assumptions. Because of this, it is important to define isolation levels that prevent such violations. In this paper, we propose a new isolation level definition that prevents such violations.

The ANSI SQL-92 specifications define four isolation levels: SERIALIZABLE, COMMITTED READ, READ COMMITTED, and READ UNCOMMITTED. These levels are defined in terms of a database's consistency assumptions. The level of isolation is defined in terms of the database's consistency assumptions. The level of isolation is defined by the database's consistency assumptions. The level of isolation is defined by the database's consistency assumptions. The level of isolation is defined by the database's consistency assumptions.

Section 2 introduces the basic concept of isolation levels. Section 3 describes the phenomena of loss of locking and phantom locking. Section 4 introduces isolation levels and proposes a new isolation level. Section 5 explores new techniques for isolating locks. Section 6 presents a conclusion.
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
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,

[Signature]

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

Yuling Wu
National University of Singapore
yuling@comp.nus.edu.sg

Jiaxi Lin
Carnegie Mellon University
jiaxil@cs.cmu.edu

Run Xiong
Carnegie Mellon University
runxiong@cs.cmu.edu

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

ABSTRACT
Multi-version concurrency control (MVCC) is commonly used in modern database systems as a robust and scalable concurrency control protocol. In most cases, MVCC is used in a multi-tenant database system where multiple tenants are sharing a single database instance. In such a system, each tenant is assigned a unique set of transactions, and the database system is responsible for ensuring that all transactions are isolated from each other. However, in recent years, there has been a growing trend towards using MVCC in in-memory databases, which have become popular due to their high performance and scalability. In this paper, we present an empirical evaluation of the performance of MVCC in an in-memory database system.

1. INTRODUCTION
In-memory databases (IMDBs) have become popular due to their high performance and scalability. However, they have also been found to have significant drawbacks, such as limited scalability and limited support for concurrent transactions. In-memory databases are often used in scenarios where data is accessed frequently, and the high performance is critical. In such scenarios, MVCC can be an effective concurrency control mechanism.

What is interesting about this trend is that IMDBs using MVCC are becoming more popular. In a recent study, it was found that MVCC is used in more than 50% of IMDBs developed in the last decade. IMDBs use MVCC to support multi-tenancy and to provide scalability. However, the use of MVCC in IMDBs is still a relatively new area of research, and there is limited research on the performance of MVCC in IMDBs.

In this paper, we present an empirical evaluation of the performance of MVCC in an in-memory database system. We use a state-of-the-art IMDB system, which supports MVCC, and evaluate its performance in a realistic scenario. We compare the performance of MVCC in IMDBs with that of other concurrency control mechanisms, such as single-version concurrency control (SVCC) and nested transactional dependency (NTD).

In this paper, we perform a study of the performance of MVCC in IMDBs using a state-of-the-art IMDB system. We evaluate the performance of MVCC in IMDBs in a realistic scenario, and compare it with the performance of other concurrency control mechanisms, such as SVCC and NTD.

IMDB Vol 10. This article is an excerpt from your manuscript as an "Experiments and Analyses" section. While the researchers appreciate your research results, they ask you to revise your manuscript in due course. To revise your manuscript, take into consideration the following points:

1. Be more specific about the limitations of your study.
2. Provide more details about the methodology used.
3. Discuss the implications of your findings for future research.

From the title and revise it to be scientific and reflect the content of your manuscript.
If You Only Read One Empirical Evaluation Paper on
In-Memory Multi-Version Concurrency Control

Yingjin Wu
National University of Singapore
yujingj@comp.nus.edu.sg
Jiaxi Li
Carnegie Mellon University
jiaxi@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 technique in modern database management systems (DBMSs). Although MVCC was discovered in the late 1970s, it is used in almost every major relational DBMS in the last decade. Managing multiple versions of data potentially increases parallelism without sacrificing serializability when processing transactions. But scaling MVCC in a multicore and many-core setting is a challenge. When there are a large number of threads running in parallel, the synchronization overhead can overwhelm the benefit of multi-versioning.

To understand how MVCC performs when processing transactions in modern hardware settings, we conduct an extensive study of several key design decisions, including concurrency control protocol, version isolation, garbage collection, and index management. We set up large-scale experiments across all these in an in-memory DBMS and evaluate them using OCP workloads. Our analysis identifies the fundamental bottlenecks of each design choice.

1. INTRODUCTION
Computing architecture and systems have seen a long line of research: an array of memory systems that employ efficient multi-version concurrency management techniques to maximize parallelism without sacrificing serializability. The most popular scheme to isolate concurrent reads and writes is called multi-version concurrency control (MVCC). The basic idea of MVCC is that the DBMS maintains multiple physical representations of the same logical object in the database to allow operations on the same object to proceed in parallel. These objects are called read-only copies, and most of the operations on these objects are independent. This makes it suitable for workloads involving multi-transactions over a tuple with no update.

In-Memory

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

Yingjin Wu
National University of Singapore
yujingj@comp.nus.edu.sg
Jiaxi Li
Carnegie Mellon University
jiaxi@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 technique in modern database management systems (DBMSs). Although MVCC was discovered in the late 1970s, it is used in almost every major relational DBMS in the last decade. Managing multiple versions of data potentially increases parallelism without sacrificing serializability when processing transactions. But scaling MVCC in a multicore and many-core setting is a challenge. When there are a large number of threads running in parallel, the synchronization overhead can overwhelm the benefit of multi-versioning.

To understand how MVCC performs when processing transactions in modern hardware settings, we conduct an extensive study of several key design decisions, including concurrency control protocol, version isolation, garbage collection, and index management. We set up large-scale experiments across all these in an in-memory DBMS and evaluate them using OCP workloads. Our analysis identifies the fundamental bottlenecks of each design choice.

1. INTRODUCTION
Computing architecture and systems have seen a long line of research: an array of memory systems that employ efficient multi-version concurrency management techniques to maximize parallelism without sacrificing serializability. The most popular scheme to isolate concurrent reads and writes is called multi-version concurrency control (MVCC). The basic idea of MVCC is that the DBMS maintains multiple physical representations of the same logical object in the database to allow operations on the same object to proceed in parallel. These objects are called read-only copies, and most of the operations on these objects are independent. This makes it suitable for workloads involving multi-transactions over a tuple with no update.

In-Memory

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

Yingjin Wu
National University of Singapore
yujingj@comp.nus.edu.sg
Jiaxi Li
Carnegie Mellon University
jiaxi@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 technique in modern database management systems (DBMSs). Although MVCC was discovered in the late 1970s, it is used in almost every major relational DBMS in the last decade. Managing multiple versions of data potentially increases parallelism without sacrificing serializability when processing transactions. But scaling MVCC in a multicore and many-core setting is a challenge. When there are a large number of threads running in parallel, the synchronization overhead can overwhelm the benefit of multi-versioning.

To understand how MVCC performs when processing transactions in modern hardware settings, we conduct an extensive study of several key design decisions, including concurrency control protocol, version isolation, garbage collection, and index management. We set up large-scale experiments across all these in an in-memory DBMS and evaluate them using OCP workloads. Our analysis identifies the fundamental bottlenecks of each design choice.

1. INTRODUCTION
Computing architecture and systems have seen a long line of research: an array of memory systems that employ efficient multi-version concurrency management techniques to maximize parallelism without sacrificing serializability. The most popular scheme to isolate concurrent reads and writes is called multi-version concurrency control (MVCC). The basic idea of MVCC is that the DBMS maintains multiple physical representations of the same logical object in the database to allow operations on the same object to proceed in parallel. These objects are called read-only copies, and most of the operations on these objects are independent. This makes it suitable for workloads involving multi-transactions over a tuple with no update.

In-Memory

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

Yingjin Wu
National University of Singapore
yujingj@comp.nus.edu.sg
Jiaxi Li
Carnegie Mellon University
jiaxi@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 technique in modern database management systems (DBMSs). Although MVCC was discovered in the late 1970s, it is used in almost every major relational DBMS in the last decade. Managing multiple versions of data potentially increases parallelism without sacrificing serializability when processing transactions. But scaling MVCC in a multicore and many-core setting is a challenge. When there are a large number of threads running in parallel, the synchronization overhead can overwhelm the benefit of multi-versioning.

To understand how MVCC performs when processing transactions in modern hardware settings, we conduct an extensive study of several key design decisions, including concurrency control protocol, version isolation, garbage collection, and index management. We set up large-scale experiments across all these in an in-memory DBMS and evaluate them using OCP workloads. Our analysis identifies the fundamental bottlenecks of each design choice.

1. INTRODUCTION
Computing architecture and systems have seen a long line of research: an array of memory systems that employ efficient multi-version concurrency management techniques to maximize parallelism without sacrificing serializability. The most popular scheme to isolate concurrent reads and writes is called multi-version concurrency control (MVCC). The basic idea of MVCC is that the DBMS maintains multiple physical representations of the same logical object in the database to allow operations on the same object to proceed in parallel. These objects are called read-only copies, and most of the operations on these objects are independent. This makes it suitable for workloads involving multi-transactions over a tuple with no update.

In-Memory

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

Yingjin Wu
National University of Singapore
yujingj@comp.nus.edu.sg
Jiaxi Li
Carnegie Mellon University
jiaxi@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 technique in modern database management systems (DBMSs). Although MVCC was discovered in the late 1970s, it is used in almost every major relational DBMS in the last decade. Managing multiple versions of data potentially increases parallelism without sacrificing serializability when processing transactions. But scaling MVCC in a multicore and many-core setting is a challenge. When there are a large number of threads running in parallel, the synchronization overhead can overwhelm the benefit of multi-versioning.

To understand how MVCC performs when processing transactions in modern hardware settings, we conduct an extensive study of several key design decisions, including concurrency control protocol, version isolation, garbage collection, and index management. We set up large-scale experiments across all these in an in-memory DBMS and evaluate them using OCP workloads. Our analysis identifies the fundamental bottlenecks of each design choice.

1. INTRODUCTION
Computing architecture and systems have seen a long line of research: an array of memory systems that employ efficient multi-version concurrency management techniques to maximize parallelism without sacrificing serializability. The most popular scheme to isolate concurrent reads and writes is called multi-version concurrency control (MVCC). The basic idea of MVCC is that the DBMS maintains multiple physical representations of the same logical object in the database to allow operations on the same object to proceed in parallel. These objects are called read-only copies, and most of the operations on these objects are independent. This makes it suitable for workloads involving multi-transactions over a tuple with no update.
1. INTRODUCTION

Concurrency control mechanisms have led to the rise of, e.g., popular DBMSs that employ efficient concurrency management in order to facilitate parallel execution. However, the DBMS approaches in the late 1980s and early 1990s were still not mature enough to handle such a large scale of parallelism, but every effort was made to overcome this difficulty. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. For more information, visit http://creativecommons.org/licenses/by-nc-nd/4.0/
CONCURRENCY CONTROL PROTOCOL

Approach #1: Timestamp Ordering
→ Assign txns timestamps that determine serial order.
→ Considered to be original MVCC protocol.

Approach #2: Optimistic Concurrency Control
→ Three-phase protocol from last class.
→ Use private workspace for new versions.

Approach #3: Two-Phase Locking
→ Txns acquire appropriate lock on physical version before they can read/write a logical tuple.
**TUPLE FORMAT**

- **TXN-ID**: Unique Txn Identifier
- **BEGIN-TS**: Version Lifetime
- **END-TS**: Next/Prev Version
- **POINTER**: Additional Meta-data
- **DATA**
Use **read-ts** field in the header to keep track of the timestamp of the last txn that read it.

**TIMESTAMP ORDERING (MVTO)**

<table>
<thead>
<tr>
<th>VERSION</th>
<th>TXN-ID</th>
<th>READ-TS</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$B_1$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>
### TIMESTAMP ORDERING (MVTO)

#### Thread #1

<table>
<thead>
<tr>
<th>VERSION</th>
<th>TXN-ID</th>
<th>READ-TS</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$B_1$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

Use **read-ts** field in the header to keep track of the timestamp of the last txn that read it.
Use \textit{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 \textit{begin-ts} and \textit{end-ts}.
TIMESTAMP ORDERING (MVTO)

Thread #1

\[ T_{id} = 10 \]

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**.

### Table

<table>
<thead>
<tr>
<th>VERSION</th>
<th>TXN-ID</th>
<th>READ-TS</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(_1)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>(\infty)</td>
</tr>
<tr>
<td>B(_1)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>(\infty)</td>
</tr>
</tbody>
</table>
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*.
Use \textit{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 \textit{begin-ts} and \textit{end-ts}.

Txn creates a new version if no othertxn holds latch and $T_{id}$ is greater than \textit{read-ts}.
TIMESTAMP ORDERING (MVTO)

Thread #1

\[ T_{id} = 10 \]

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

<table>
<thead>
<tr>
<th>VERSION</th>
<th>TXN-ID</th>
<th>READ-TS</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_1</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>\infty</td>
</tr>
<tr>
<td>B_1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>\infty</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>VERSION</th>
<th>TXN-ID</th>
<th>READ-TS</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>∞</td>
</tr>
<tr>
<td>B₁</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>∞</td>
</tr>
</tbody>
</table>

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**.
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

$T_{id}=10$

<table>
<thead>
<tr>
<th>VERSION</th>
<th>TXN-ID</th>
<th>READ-TS</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$B_1$</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$B_2$</td>
<td>10</td>
<td>0</td>
<td>10</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>
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)**

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

<table>
<thead>
<tr>
<th>VERSION</th>
<th>TXN-ID</th>
<th>READ-TS</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$B_1$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>$B_2$</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

- 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)

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

Thread #1

\[ T_{id} = 10 \]

<table>
<thead>
<tr>
<th>VERSION</th>
<th>TXN-ID</th>
<th>READ-CNT</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_1</td>
<td>θ</td>
<td>θ</td>
<td>1</td>
<td>∞</td>
</tr>
<tr>
<td>B_1</td>
<td>θ</td>
<td>θ</td>
<td>1</td>
<td>∞</td>
</tr>
</tbody>
</table>
TWO-PHASE LOCKING (MV2PL)

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)

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

<table>
<thead>
<tr>
<th>Transaction</th>
<th>TXN-ID</th>
<th>READ-CNT</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>∞</td>
</tr>
<tr>
<td>B₁</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>∞</td>
</tr>
</tbody>
</table>

### 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.
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)

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.

<table>
<thead>
<tr>
<th>VERSION</th>
<th>TXN-ID</th>
<th>READ-CNT</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>∞</td>
</tr>
<tr>
<td>B_1</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>∞</td>
</tr>
<tr>
<td>B_2</td>
<td>10</td>
<td>0</td>
<td>10</td>
<td>∞</td>
</tr>
</tbody>
</table>
### TWO-PHASE LOCKING (MV2PL)

<table>
<thead>
<tr>
<th>VERSION</th>
<th>TXN-ID</th>
<th>READ-CNT</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>∞</td>
</tr>
<tr>
<td>B₁</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>B₂</td>
<td>10</td>
<td>0</td>
<td>10</td>
<td>∞</td>
</tr>
</tbody>
</table>

**Thread #1**

- **$T_{id}=10$**

- **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

\[ T_{id}=10 \]

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.

<table>
<thead>
<tr>
<th>VERSION</th>
<th>TXN-ID</th>
<th>READ-CNT</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>∞</td>
</tr>
<tr>
<td>B_1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>B_2</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>∞</td>
</tr>
</tbody>
</table>
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.
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**

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.

<table>
<thead>
<tr>
<th>VERSION</th>
<th>TXN-ID</th>
<th>READ-TS</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>θ</td>
<td>-</td>
<td>99999</td>
<td>(2^{31} - 1)</td>
</tr>
<tr>
<td>A₂</td>
<td>θ</td>
<td>-</td>
<td>(2^{31} - 1)</td>
<td>∞</td>
</tr>
</tbody>
</table>

**Thread #1**

\(T_{id}=2^{31} - 1\)

**WRITE(A)**
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.

**Thread #1**

\[ T_{id} = 2^{31} - 1 \]

**Thread #2**

\[ T_{id} = 1 \]
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

\[ T_{id} = 1 \]

<table>
<thead>
<tr>
<th>VERSION</th>
<th>TXN-ID</th>
<th>READ-TS</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_1</td>
<td>0</td>
<td>-</td>
<td>99999</td>
<td>2^{31} - 1</td>
</tr>
<tr>
<td>A_2</td>
<td>0</td>
<td>-</td>
<td>2^{31} - 1</td>
<td>1</td>
</tr>
<tr>
<td>A_3</td>
<td>0</td>
<td>-</td>
<td>1</td>
<td>\infty</td>
</tr>
</tbody>
</table>

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.
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.

<table>
<thead>
<tr>
<th>VERSION</th>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_0$</td>
<td>$111</td>
<td>Ø</td>
</tr>
<tr>
<td>$A_1$</td>
<td>$222</td>
<td>Ø</td>
</tr>
<tr>
<td>$B_1$</td>
<td>$10</td>
<td>Ø</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>VERSION</th>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₀</td>
<td>$111</td>
<td></td>
</tr>
<tr>
<td>A₁</td>
<td>$222</td>
<td>Ø</td>
</tr>
<tr>
<td>B₁</td>
<td>$10</td>
<td>Ø</td>
</tr>
<tr>
<td>A₂</td>
<td>$333</td>
<td>Ø</td>
</tr>
</tbody>
</table>
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.
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.
On every update, copy the current version to the time-travel table. Update pointers.
TIME-TRAVEL STORAGE

Main Table

<table>
<thead>
<tr>
<th>VERSION</th>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_2</td>
<td>$222</td>
<td></td>
</tr>
<tr>
<td>B_1</td>
<td>$10</td>
<td></td>
</tr>
</tbody>
</table>

Time-Travel Table

<table>
<thead>
<tr>
<th>VERSION</th>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_1</td>
<td>$111</td>
<td>Ø</td>
</tr>
<tr>
<td>A_2</td>
<td>$222</td>
<td></td>
</tr>
</tbody>
</table>

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

#### Main Table

<table>
<thead>
<tr>
<th>VERSION</th>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A&lt;sub&gt;3&lt;/sub&gt;</td>
<td>$333</td>
<td></td>
</tr>
<tr>
<td>B&lt;sub&gt;1&lt;/sub&gt;</td>
<td>$10</td>
<td></td>
</tr>
</tbody>
</table>

#### Time-Travel Table

<table>
<thead>
<tr>
<th>VERSION</th>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A&lt;sub&gt;1&lt;/sub&gt;</td>
<td>$111</td>
<td>Ø</td>
</tr>
<tr>
<td>A&lt;sub&gt;2&lt;/sub&gt;</td>
<td>$222</td>
<td></td>
</tr>
</tbody>
</table>

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

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

Overwrite master version in the main table. Update pointers.
On every update, copy only the values that were modified to the delta storage and overwrite the master version.
On every update, copy only the values that were modified to the delta storage and overwrite the master version.
On every update, copy only the values that were modified to the delta storage and overwrite the master version.
On every update, copy only the values that were modified to the delta storage and overwrite the master version.
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**

<table>
<thead>
<tr>
<th>VERSION</th>
<th>INT_VAL</th>
<th>STR_VAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>$100$</td>
<td></td>
</tr>
</tbody>
</table>

**Variable-Length Data**

`MY_LONG_STRING`
Reuse pointers to variable-length pool for values that do not change between versions.
# NON-INLINE ATTRIBUTES

## Main Table

<table>
<thead>
<tr>
<th>VERSION</th>
<th>INT_VAL</th>
<th>STR_VAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>$100</td>
<td></td>
</tr>
<tr>
<td>A₂</td>
<td>$90</td>
<td></td>
</tr>
</tbody>
</table>

## 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

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

<table>
<thead>
<tr>
<th>VERSION</th>
<th>INT_VAL</th>
<th>STR_VAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>$100</td>
<td></td>
</tr>
<tr>
<td>A₂</td>
<td>$90</td>
<td></td>
</tr>
</tbody>
</table>

Variable-Length Data

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

- `INT_VAL`
  - A1: $100
  - A2: $90

- `STR_VAL`
  - Refs=2
  - `MY_LONG_STRING`
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.
Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.
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

Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.
Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.
Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.
Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.
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.
**Thread #1**

$T_{id}=12$

**Thread #2**

$T_{id}=25$

---

**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.
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.
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

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 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...
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…
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

A_4 \rightarrow A_3 \rightarrow A_2 \rightarrow A_1

Append-Only
Newest-to-Oldest
INDEX POINTERS

GET(A) → PRIMARY INDEX

Physical Address

SECONDARY INDEX

Append-Only
Newest-to-Oldest
INDEX POINTERS

PRIMARY INDEX

SECONDARY INDEX

GET(A)

Physical Address

Append-Only
Newest-to-Oldest

A_4 \rightarrow A_3 \rightarrow A_2 \rightarrow A_1
INDEX POINTERS

GET(A)

PRIMARY INDEX

SECONDARY INDEX

SECONDARY INDEX

SECONDARY INDEX

SECONDARY INDEX

A_4 → A_3 → A_2 → A_1

Append-Only
Newest-to-Oldest
INDEX POINTERS

PRIMARY INDEX

SECONDARY INDEX

SECONDARY INDEX

SECONDARY INDEX

SECONDARY INDEX

A_4 \rightarrow A_3 \rightarrow A_2 \rightarrow A_1

GET(A)

Append-Only
Newest-to-Oldest
INDEX POINTERS

PRIMARY INDEX

SECONDARY INDEX

Physical Address

Primary Key

Append-Only
Newest-to-Oldest

GET(A)
INDEX POINTERS

PRIMARY INDEX

SECONDARY INDEX

GET(A)

TupleId

TupleId \rightarrow \text{Address}

Physical Address

A_4 \rightarrow A_3 \rightarrow A_2 \rightarrow A_1

Append-Only
Newest-to-Oldest
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.
MVCC DESIGN DECISIONS

CC Protocol: Inconclusive results…

Version Storage: Deltas

Garbage Collection: Tuple-Level Vacuuming

Indexes: Logical Pointers
## MVCC Configuration Evaluation

<table>
<thead>
<tr>
<th>Database</th>
<th>Protocol</th>
<th>Version Storage</th>
<th>Garbage Collection</th>
<th>Indexes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle</td>
<td>MV2PL</td>
<td>Delta</td>
<td>Vacuum</td>
<td>Logical</td>
</tr>
<tr>
<td>Postgres</td>
<td>MV-2PL/MV-T0</td>
<td>Append-Only</td>
<td>Vacuum</td>
<td>Physical</td>
</tr>
<tr>
<td>MySQL-InnoDB</td>
<td>MV-2PL</td>
<td>Delta</td>
<td>Vacuum</td>
<td>Logical</td>
</tr>
<tr>
<td>HYRISE</td>
<td>MV-OCC</td>
<td>Append-Only</td>
<td>–</td>
<td>Physical</td>
</tr>
<tr>
<td>Hekaton</td>
<td>MV-OCC</td>
<td>Append-Only</td>
<td>Cooperative</td>
<td>Physical</td>
</tr>
<tr>
<td>MemSQL</td>
<td>MV-OCC</td>
<td>Append-Only</td>
<td>Vacuum</td>
<td>Physical</td>
</tr>
<tr>
<td>SAP HANA</td>
<td>MV-2PL</td>
<td>Time-travel</td>
<td>Hybrid</td>
<td>Logical</td>
</tr>
<tr>
<td>NuoDB</td>
<td>MV-2PL</td>
<td>Append-Only</td>
<td>Vacuum</td>
<td>Logical</td>
</tr>
<tr>
<td>HyPer</td>
<td>MV-OCC</td>
<td>Delta</td>
<td>Txn-level</td>
<td>Logical</td>
</tr>
<tr>
<td>CMU's TBD</td>
<td>MV-OCC</td>
<td>Delta</td>
<td>Txn-level</td>
<td>Logical</td>
</tr>
</tbody>
</table>
MVCC CONFIGURATION EVALUATION

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

Throughput (txn/sec) vs. # Threads

- Oracle/MySQL
- NuoDB
- HyPer
- HYRISE
- MemSQL
- HANA
- HEKATON
- Postgres
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...
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