TODAY’S AGENDA

Compare-and-Swap (CAS)
Isolation Levels
MVCC Design Decisions
Project #2
Atomic instruction that compares contents of a memory location \( M \) to a given value \( V \)
→ If values are equal, installs new given value \( V' \) in \( M \)
→ Otherwise operation fails

\[
\text{sync_bool_compare_and_swap}(&M, 20, 30)
\]
COMPARE- AND- SWAP

Atomic instruction that compares contents of a memory location \( M \) to a given value \( V \)
→ If values are equal, installs new given value \( V' \) in \( M \)
→ Otherwise operation fails

\[
\text{__sync_bool_compare_and_swap}(&M, 20, 30)
\]
COMPARE-AND-SWAP

Atomic instruction that compares contents of a memory location $M$ to a given value $V$
→ If values are equal, installs new given value $V'$ in $M$
→ Otherwise operation fails

```
__sync_bool_compare_and_swap(&M, 25, 35)
```
OBSERVATION

Serializability is useful because it allows programmers to ignore concurrency issues but enforcing it may allow too little parallelism and limit performance.

We may want to use a weaker level of consistency to improve scalability.
ISOLATION LEVELS

Controls the extent that a txn is exposed to the actions of other concurrent txns.
Provides for greater concurrency at the cost of exposing txns to uncommitted changes:
→ Dirty Read Anomaly
→ Unrepeatable Reads Anomaly
→ Phantom Reads Anomaly
ANSI ISOLATION LEVELS

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

REPEATABLE READS
→ Phantoms may happen.

READ COMMITTED
→ Phantoms and unrepeateable reads may happen.

READ UNCOMMITTED
→ All of them may happen.
ISOLATION LEVEL HIERARCHY

- Read Uncommitted
- Read Committed
- Repeatable Reads
- Serializable
# Real-World Isolation Levels

<table>
<thead>
<tr>
<th>Database</th>
<th>Default</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actian Ingres</td>
<td>Serializable</td>
<td>Serializable</td>
</tr>
<tr>
<td>Greenplum</td>
<td>Read Committed</td>
<td>Serializable</td>
</tr>
<tr>
<td>IBM DB2</td>
<td><strong>Cursor Stability</strong></td>
<td>Serializable</td>
</tr>
<tr>
<td>MySQL</td>
<td>Repeatable Reads</td>
<td>Serializable</td>
</tr>
<tr>
<td>MemSQL</td>
<td>Read Committed</td>
<td>Read Committed</td>
</tr>
<tr>
<td>MS SQL Server</td>
<td>Read Committed</td>
<td>Serializable</td>
</tr>
<tr>
<td>Oracle</td>
<td>Read Committed</td>
<td><strong>Snapshot Isolation</strong></td>
</tr>
<tr>
<td>Postgres</td>
<td>Read Committed</td>
<td>Serializable</td>
</tr>
<tr>
<td>SAP HANA</td>
<td>Read Committed</td>
<td>Serializable</td>
</tr>
<tr>
<td>VoltDB</td>
<td>Serializable</td>
<td>Serializable</td>
</tr>
</tbody>
</table>

Source: Peter Bailis
The isolation levels defined as part of SQL-92 standard only focused on anomalies that can occur in a 2PL-based DBMS.

Two additional isolation levels:
→ CURSOR STABILITY
→ SNAPSHOT ISOLATION
CURSOR STABILITY (CS)

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

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

Txn #1

BEGIN READ(A) ... WRITE(A) COMMIT

Txn #2

BEGIN ... WRITE(A) ... COMMIT
LOST UPDATE ANOMALY

Txn #1

BEGIN
READ(A)

... ...

WRITE(A)

COMMIT

Txn #2

BEGIN

...

WRITE(A)

COMMIT
LOST UPDATE ANOMALY

**Txn #1**

- BEGIN
- READ(A)
- WRITE(A)
- COMMIT

**Txn #2**

- BEGIN
- WRITE(A)
- COMMIT
LOST UPDATE ANOMALY

Txn #1

BEGIN
READ(A)...
WRITE(A)
COMMIT

Txn #2

BEGIN
...WRITE(A)...COMMIT
LOST UPDATE ANOMALY

Txn #1

BEGIN
READ(A)  ...  WRITE(A)

COMMIT

Txn #2

BEGIN  ...  WRITE(A)  ...

COMMIT
LOST UPDATE ANOMALY

**Txn #1**

```
BEGIN
READ(A)  . . .  WRITE(A)  COMMIT
```

**Txn #2**

```
BEGIN  . . .  WRITE(A)  . . .  COMMIT
```

Txn #2’s write to A will be lost even though it commits after Txn #1. A cursor lock on A would prevent this problem (but not always).
SNAPSHOT ISOLATION (SI)

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

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

SI is susceptible to the **Write Skew Anomaly**
WRITE SKEW ANOMALY

Txn #1
Change white marbles to black.

Txn #2
Change black marbles to white.
**WRITE SKEW ANOMALY**

**Txn #1**
Change white marbles to black.

**Txn #2**
Change black marbles to white.
WRITE SKEW ANOMALY

**Txn #1**
Change white marbles to black.

**Txn #2**
Change black marbles to white.
WRITE SKEW ANOMALY

**Txn #1**
Change white marbles to black.

**Txn #2**
Change black marbles to white.
WRITE SKEW ANOMALY

\( \text{Txn } #1 \)  
Change white marbles to black.

\( \text{Txn } #2 \)  
Change black marbles to white.
ISOLATION LEVEL HIERARCHY

- Serializable
- Repeatable Reads
- Snapshot Isolation
- Cursor Stability
- Read Committed
- Read Uncommitted
Figure 4-1: A partial order to relate various isolation levels.

Source: Atul Adya
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.
MVCC DESIGN DECISIONS

Concurrency Control Protocol
Version Storage
Garbage Collection
Index Management
Txn Id Wraparound (New)
If You Only Read One Empirical Evaluation Paper on In-Memory Multi-Version Concurrency Control, Make It This One!

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

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

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

DECIIONS

ABSTRACT
Multi-version concurrency control and multi-version concurrency control. Although the is used in almost every DBMS in the last three decades, it is used in almost every major relational DBMS released in the last decade. Maintaining multiple versions of data potentially increases performance with our sacrificing reliability when processing transactions. But scaling MCC in a multicore 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-version. To understand how MCC in modern hardware realizes concurrency in the lock-based scheme’s four key design version store, garbage implementation state of the DBMS and evaluated the impact on the performance.

I. INTRODUCTION
Computer architecture, in-memory DBMS, and management schemes are the same object. They are used in almost every major relational DBMS released in the last decade. Maintaining multiple versions of data potentially increases performance with our sacrificing reliability when processing transactions. But scaling MCC in a multicore 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.

1. INTRODUCTION
Computer architecture, in-memory DBMS, and management schemes are the same object. They are used in almost every major relational DBMS released in the last decade. Maintaining multiple versions of data potentially increases performance with our sacrificing reliability when processing transactions. But scaling MCC in a multicore 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.

1. INTRODUCTION
Computer architecture, in-memory DBMS, and management schemes are the same object. They are used in almost every major relational DBMS released in the last decade. Maintaining multiple versions of data potentially increases performance with our sacrificing reliability when processing transactions. But scaling MCC in a multicore 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.

1. INTRODUCTION
Computer architecture, in-memory DBMS, and management schemes are the same object. They are used in almost every major relational DBMS released in the last decade. Maintaining multiple versions of data potentially increases performance with our sacrificing reliability when processing transactions. But scaling MCC in a multicore 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.

1. INTRODUCTION
Computer architecture, in-memory DBMS, and management schemes are the same object. They are used in almost every major relational DBMS released in the last decade. Maintaining multiple versions of data potentially increases performance with our sacrificing reliability when processing transactions. But scaling MCC in a multicore 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.

1. INTRODUCTION
Computer architecture, in-memory DBMS, and management schemes are the same object. They are used in almost every major relational DBMS released in the last decade. Maintaining multiple versions of data potentially increases performance with our sacrificing reliability when processing transactions. But scaling MCC in a multicore 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.

1. INTRODUCTION
Computer architecture, in-memory DBMS, and management schemes are the same object. They are used in almost every major relational DBMS released in the last decade. Maintaining multiple versions of data potentially increases performance with our sacrificing reliability when processing transactions. But scaling MCC in a multicore 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.

1. INTRODUCTION
Computer architecture, in-memory DBMS, and management schemes are the same object. They are used in almost every major relational DBMS released in the last decade. Maintaining multiple versions of data potentially increases performance with our sacrificing reliability when processing transactions. But scaling MCC in a multicore 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.

1. INTRODUCTION
Computer architecture, in-memory DBMS, and management schemes are the same object. They are used in almost every major relational DBMS released in the last decade. Maintaining multiple versions of data potentially increases performance with our sacrificing reliability when processing transactions. But scaling MCC in a multicore 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.

1. INTRODUCTION
Computer architecture, in-memory DBMS, and management schemes are the same object. They are used in almost every major relational DBMS released in the last decade. Maintaining multiple versions of data potentially increases performance with our sacrificing reliability when processing transactions. But scaling MCC in a multicore 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.

1. INTRODUCTION
Computer architecture, in-memory DBMS, and management schemes are the same object. They are used in almost every major relational DBMS released in the last decade. Maintaining multiple versions of data potentially increases performance with our sacrificing reliability when processing transactions. But scaling MCC in a multicore 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.

1. INTRODUCTION
Computer architecture, in-memory DBMS, and management schemes are the same object. They are used in almost every major relational DBMS released in the last decade. Maintaining multiple versions of data potentially increases performance with our sacrificing reliability when processing transactions. But scaling MCC in a multicore 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.

1. INTRODUCTION
Computer architecture, in-memory DBMS, and management schemes are the same object. They are used in almost every major relational DBMS released in the last decade. Maintaining multiple versions of data potentially increases performance with our sacrificing reliability when processing transactions. But scaling MCC in a multicore 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.

1. INTRODUCTION
Computer architecture, in-memory DBMS, and management schemes are the same object. They are used in almost every major relational DBMS released in the last decade. Maintaining multiple versions of data potentially increases performance with our sacrificing reliability when processing transactions. But scaling MCC in a multicore 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.

1. INTRODUCTION
Computer architecture, in-memory DBMS, and management schemes are the same object. They are used in almost every major relational DBMS released in the last decade. Maintaining multiple versions of data potentially increases performance with our sacrificing reliability when processing transactions. But scaling MCC in a multicore 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.
# MVCC Implementations

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>Postgres</td>
<td>MV-2PL/MV-T0</td>
<td>Append-Only</td>
<td>Vacuum</td>
</tr>
<tr>
<td>MySQL-InnoDB</td>
<td>MV-2PL</td>
<td>Delta</td>
<td>Vacuum</td>
</tr>
<tr>
<td>HYRISE</td>
<td>MV-OCC</td>
<td>Append-Only</td>
<td>-</td>
</tr>
<tr>
<td>Hekaton</td>
<td>MV-OCC</td>
<td>Append-Only</td>
<td>Cooperative</td>
</tr>
<tr>
<td>MemSQL</td>
<td>MV-OCC</td>
<td>Append-Only</td>
<td>Vacuum</td>
</tr>
<tr>
<td>SAP HANA</td>
<td>MV-2PL</td>
<td>Time-travel</td>
<td>Hybrid</td>
</tr>
<tr>
<td>NuoDB</td>
<td>MV-2PL</td>
<td>Append-Only</td>
<td>Vacuum</td>
</tr>
<tr>
<td>HyPer</td>
<td>MV-OCC</td>
<td>Delta</td>
<td>Txn-level</td>
</tr>
</tbody>
</table>
TUPLE FORMAT

- TXN-ID: Unique Txn Identifier
- BEGIN-TS: Version Lifetime
- END-TS: Next/Prev Version
- POINTER: Additional Metadata
- DATA:
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.
## TIMESTAMP ORDERING (MVTO)

<table>
<thead>
<tr>
<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>1</td>
<td>1</td>
</tr>
<tr>
<td>B₁</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
## TIMESTAMP ORDERING (MVTO)

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>$B_1$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Use “read-ts” field in the header to keep track of the timestamp of the last txn that read it.
TIMESTAMP ORDERING (MVTO)

$T_{id} = 10$

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

<table>
<thead>
<tr>
<th>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>
</tr>
<tr>
<td>$B_1$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
### TIMESTAMP ORDERING (MVTO)

**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 lock is unset and its $T_{id}$ is between “begin-ts” and “end-ts”.

### Example

<table>
<thead>
<tr>
<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>$\infty$</td>
</tr>
<tr>
<td>$B_1$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

$T_{id} = 10$
**TIMESTAMP ORDERING (MVTO)**

\[ T_{id} = 10 \]

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 lock is unset and its \( T_{id} \) is between “begin-ts” and “end-ts”.

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>B_1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Use “read-ts” field in the header to keep track of the timestamp of the last txn that read it.

$T_{id}=10$

Txn is allowed to read version if the lock is unset and its $T_{id}$ is between “begin-ts” and “end-ts”.

Txn creates a new version if no other txn holds lock and $T_{id}$ is greater than “read-ts”.

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>$B_1$</td>
<td>0</td>
<td>0</td>
<td>1</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 lock is unset and its $T_{id}$ is between “begin-ts” and “end-ts”.

Txn creates a new version if no other txn holds lock and $T_{id}$ is greater than “read-ts”.

$T_{id} = 10$
Use "read-ts" field in the header to keep track of the timestamp of the last txn that read it.

\[ T_{id} = 10 \]

Txn is allowed to read version if the lock is unset and its \( T_{id} \) is between "begin-ts" and "end-ts".

Txn creates a new version if no other txn holds lock and \( T_{id} \) is greater than "read-ts".

TIMESTAMP ORDERING (MVTO)

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>B_1</td>
<td>10</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>B_2</td>
<td>10</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>
Use “read-ts” field in the header to keep track of the timestamp of the last txn that read it.

$T_{id} = 10$

Txn is allowed to read version if the lock is unset and its $T_{id}$ is between “begin-ts” and “end-ts”.

Txn creates a new version if no other txn holds lock and $T_{id}$ is greater than “read-ts”.

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>B₁</td>
<td>10</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>B₂</td>
<td>10</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>
Use “read-ts” field in the header to keep track of the timestamp of the last txn that read it.

\[ T_{id} = 10 \]

Txn is allowed to read version if the lock is unset and its \( T_{id} \) is between “begin-ts” and “end-ts”.

Txn creates a new version if no other txn holds lock and \( T_{id} \) is greater than “read-ts”.

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>B(_1)</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>B(_2)</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>
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.
All of the physical versions of a logical tuple are stored in the same table space.

On every update, append a new version of the tuple into an empty space in the table.
APPEND-ONLY STORAGE

All of the physical versions of a logical tuple are stored in the same table space.

On every update, append a new version of the tuple into an empty space in the table.
### APPEND-ONLY STORAGE

**Main Table**

<table>
<thead>
<tr>
<th>KEY</th>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>XXX</td>
<td>$111</td>
</tr>
<tr>
<td>A₂</td>
<td>XXX</td>
<td>$222</td>
</tr>
<tr>
<td>B₁</td>
<td>YYY</td>
<td>$10</td>
</tr>
<tr>
<td>A₃</td>
<td>XXX</td>
<td>$333</td>
</tr>
</tbody>
</table>

All of the physical versions of a logical tuple are stored in the same table space.

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.
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.
DELTA STORAGE

**Main Table**

<table>
<thead>
<tr>
<th>KEY</th>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>XXX</td>
<td>$111</td>
</tr>
<tr>
<td>B₁</td>
<td>YYY</td>
<td>$10</td>
</tr>
</tbody>
</table>

**Delta Storage Segment**

<table>
<thead>
<tr>
<th>DELTA</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>(VALUE→$111)</td>
</tr>
</tbody>
</table>

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>KEY</th>
<th>INT_VAL</th>
<th>STR_VAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>XXX</td>
<td>$100</td>
</tr>
</tbody>
</table>

### Variable-Length Data

- `MY_LONG_STRING`
NON INLINE ATTRIBUTES

Main Table

<table>
<thead>
<tr>
<th>KEY</th>
<th>INT_VAL</th>
<th>STR_VAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>XXX</td>
<td>$100</td>
</tr>
</tbody>
</table>

Variable-Length Data

MY_LONG_STRING
**NON-INLINE ATTRIBUTES**

**Main Table**

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

**Variable-Length Data**

- `MY_LONG_STRING`
- `MY_LONG_STRING`
NON INLINE ATTRIBUTES

Main Table

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

Variable-Length Data

- MY_LONG_STRING
- MY_LONG_STRING

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

Main Table

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

Variable-Length Data

<table>
<thead>
<tr>
<th>Refs=1</th>
<th>MY_LONG_STRING</th>
</tr>
</thead>
</table>

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

Requires reference counters to know when it safe to free memory. Unable to relocate memory easily.
Reusing pointers to variable-length pool for values that do not change between versions.

Requires reference counters to know when it's 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 activetxn in the DBMS can “see” that version (SI).
→ The version was created by an abortedtxn.

Two additional design decisions:

→ How to look for expired versions?
→ How to decide when it is safe to reclaim memory?
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.
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.

<table>
<thead>
<tr>
<th>Dirty?</th>
<th>TXN-ID</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_2$</td>
<td>0</td>
<td>10</td>
<td>20</td>
</tr>
</tbody>
</table>
**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.
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
$T_{id}=12$

Tid = 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.
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

Eachtxnkeeps 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.
## OBSERVATION

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

### Thread #1

\[ T_{id}=1 \]

<table>
<thead>
<tr>
<th>TXN-ID</th>
<th>READ-TS</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A\textsubscript{1}</td>
<td>0</td>
<td>(2^{31} - 1)</td>
<td>(2^{31} - 2)</td>
</tr>
<tr>
<td>B\textsubscript{1}</td>
<td>0</td>
<td>(2^{31} - 1)</td>
<td>(2^{31} - 2)</td>
</tr>
</tbody>
</table>
**OBSERVATION**

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

<table>
<thead>
<tr>
<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>$2^{31} - 1$</td>
<td>$2^{31} - 2$</td>
</tr>
<tr>
<td>B₁</td>
<td>0</td>
<td>$2^{31} - 1$</td>
<td>$2^{31} - 2$</td>
</tr>
</tbody>
</table>

Thread #1

$T_{id}=1$
If the DBMS reaches the max value for its timestamps, it will have to wrap around and start at zero. This will make all previous versions be in the "future" from new transactions.
POSTGRES TXN ID WRAPAROUND

Stop accepting new commands when the system gets close to the max txn id.

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.
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**

A4 → A3 → A2 → A1
INDEX POINTERS

PRIMARY INDEX

SECONDARY INDEX

Append-Only
Newest-to-Oldest
INDEX POINTERS

GET(A) →

PRIMARY INDEX

SECONDARY INDEX

Physical Address

A4 → A3 → A2 → A1

Append-Only
Newest-to-Oldest
INDEX POINTERS

PRIMARY INDEX

SECONDARY INDEX

GET(A)

Physical Address

Append-Only
Newest-to-Oldest
INDEX POINTERS

GET(A)

PRIMARY INDEX

SECONDARY INDEX

SECONDARY INDEX

SECONDARY INDEX

SECONDARY INDEX

Append-Only
Newest-to-Oldest
INDEX POINTERS

PRIMAR INDEX

SECONDARY INDEX

GET(A)

A4 → A3 → A2 → A1

Append-Only
Newest-to-Oldest
INDEX POINTERS

GET(A)

PRIMARY INDEX

SECONDARY INDEX

Physical Address

Primary Key

A₁ → A₂ → A₃ → A₄

Append-Only
Newest-to-Oldest
INDEX POINTERS

PRIMARY INDEX

SECONDARY INDEX

GET(A)

TupleId

TupleId → Address

Physical Address

A_4 → A_3 → A_2 → A_1

Append-Only
Newest-to-Oldest
**MVCC CONFIGURATION EVALUATION**

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

Throughput (txn/sec)

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

# Threads

0 8 16 24 32 40
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 old 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_repair, but either way you end up rewriting the whole table. Proposals have been made to try to relocate rows in the fly, but it’s hard to do correctly and risks bloating the...
PARTING THOUGHTS

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

We only discussed MVCC for OLTP.
→ Design decisions may be different for HTAP

Interesting MVCC research/project Topics:
→ Block compaction
→ Version compression
→ On-line schema changes
PROJECT #2

Implement a latch-free Skip List in Peloton.
→ Forward / Reverse Iteration
→ Garbage Collection

Must be able to support both unique and non-unique keys.
We will provide you with a header file with the index API that you have to implement.
→ Data serialization and predicate evaluation will be taken care of for you.

There are several design decisions that you are going to have to make.
→ There is no right answer.
→ Do not expect us to guide you at every step of the development process.

PROJECT #2 – DESIGN
PROJECT #2 – TESTING

We are providing you with C++ unit tests for you to check your implementation. We also have a BwTree implementation to compare against.

We strongly encourage you to do your own additional testing.
You must write sufficient documentation and comments in your code to explain what you are doing in all different parts.

We will inspect the submissions manually.
**PROJECT #2 – GRADING**

We will run additional tests beyond what we provided you for grading.
→ Bonus points will be given to the groups with the fastest implementation.
→ We will use Valgrind when testing your code.

All source code must pass ClangFormat syntax formatting checker.
→ See Peloton [documentation](#) for formatting guidelines.
This is a group project.
→ Everyone should contribute equally.
→ I will review commit history.

Email me if you do not have a group.
PROJECT #2

Due Date: March 12\textsuperscript{th} @ 11:59pm
Projects will be turned in using Autolab.

Full description and instructions:
http://15721.courses.cs.cmu.edu/spring2018/project2.html
Modern MVCC Implementations
→ CMU Cicada
→ Microsoft Hekaton
→ TUM HyPer
→ Serializable Snapshot Isolation