MULTI-VERSION CONCURRENCY CONTROL

The DBMS maintains multiple physical versions of a single logical object in the database:
→ When a txn writes to an object, the DBMS creates a new version of that object.
→ When a txn reads an object, it reads the newest version that existed when the txn started.

First proposed in 1978 MIT PhD dissertation. First implementation was InterBase (Firebird). Used in almost every new DBMS in last 10 years.
MULTI-VERSION CONCURRENCY CONTROL

Writers don't block readers.  
Readers don't block writers.

Read-only txns can read a consistent snapshot without acquiring locks or txn ids.  
→ Use timestamps to determine visibility.

Easily support time-travel queries.
SNAPSHOT ISOLATION (SI)

When a txn starts, it sees a consistent snapshot of the database that existed when that the txn started.
→ No torn writes from active txns.
→ If two txns update the same object, then first writer wins.

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

\( T_{xn} \#1 \)
Change white marbles to black.

\( T_{xn} \#2 \)
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

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

**Txn #2**
Change black marbles to white.
ISOLATION LEVEL HIERARCHY

- **SERIALIZABLE**
- **REPEATABLE READS**
- **SNAPSHOT ISOLATION**
- **READ COMMITTED**
- **READ UNCOMMITTED**
ISOLATION LEVEL HIERARCHY

- REPEATABLE READS
- SNAPSHOT ISOLATION
- READ COMMITTED
- READ UNcommitted
MVCC DESIGN DECISIONS

Concurrency Control Protocol
Version Storage
Garbage Collection
Index Management
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

Unique Txn Identifier

Version Lifetime

Next/Prev Version

Additional Meta-data

TXN-ID | BEGIN-TS | END-TS | POINTER | ... | DATA
Use *read-ts* field in the header to keep track of the timestamp of the last txn that read it.
Use `read-ts` field in the header to keep track of the timestamp of the last txn that read it.

**Thread #1**

\[ T_{id}=10 \]

<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)**

Thread #1

\[ T_{id} = 10 \]

<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>1</td>
<td>1</td>
</tr>
<tr>
<td>B\textsubscript{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 can read version if the latch is unset and its \( T_{id} \) is between **begin-ts** and **end-ts**.

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

 Txn can read version if the latch is unset and its \( T_{id} \) is between **begin-ts** and **end-ts**.
**TIMESTAMP ORDERING (MVTO)**

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>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>

Txn can read version if the latch is unset and its $T_{id}$ is between *begin-ts* and *end-ts*.
**Timestamp Ordering (MVTO)**

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

Thread #1

\[ T_{id}=10 \]

![Diagram](image)

<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>

Txn can read version if the latch is unset and its \(T_{id}\) is between **begin-ts** and **end-ts**.
TIMESTAMP ORDERING (MVTO)

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

Thread #1

\[ T_{id}=10 \]

<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>

Txn can read version if the latch is unset and its \( T_{id} \) is between *begin-ts* and *end-ts*.

Txn creates a new version if no other txn holds latch and \( T_{id} \) is greater than *read-ts*. 
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>10</td>
<td>1</td>
</tr>
<tr>
<td>$B_1$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

$T_{id}=10$

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

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

Txn can read version if the latch is unset and its \( T_{id} \) is between `begin-ts` and `end-ts`.

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

- Txn can read version if the latch is unset and its $T_{id}$ is between **begin-ts** and **end-ts**.
- Txn creates a new version if no other txn holds latch and $T_{id}$ is greater than **read-ts**.
TIMESTAMP ORDERING (MVTO)

Thread #1

Use \textit{read-ts} field in the header to keep track of the timestamp of the last txn that read it.

Txn can read version if the latch is unset and its \( T_{id} \) is between \textit{begin-ts} and \textit{end-ts}.

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

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

Txn can read version if the latch is unset and its $T_{id}$ is between `begin-ts` and `end-ts`.

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

<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>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>B₂</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

Thread #1 $T_{id}=10$

**Read (A)**

**Write (B)**
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)

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.

Thread #1

$T_{id} = 10$

<table>
<thead>
<tr>
<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>0</td>
<td>1</td>
</tr>
<tr>
<td>B₁</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Thread #1

$T_{id} = 10$

READ(A)

WRITE(B)
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.

### Thread #1

\[ T_{id} = 10 \]

<table>
<thead>
<tr>
<th></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>( \infty )</td>
</tr>
<tr>
<td>( B_1 )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( \infty )</td>
</tr>
</tbody>
</table>
TWO-PHASE LOCKING (MV2PL)

Thread #1

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

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

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></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>

Thread #1

$T_{id}=10$

**READ(A)**

**WRITE(B)**
### TWO-PHASE LOCKING (MV2PL)

**Thread #1**

\[ T_{id}=10 \]

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>B_1</td>
<td>10</td>
<td>1</td>
<td>1</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.
## TWO-PHASE LOCKING (MV2PL)

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>B₁</td>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B₂</td>
<td>10</td>
<td>0</td>
<td>10</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)

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.

Thread #1 $T_{id}=10$

- **READ(A)**
- **WRITE(B)**

<table>
<thead>
<tr>
<th>Txn</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>10</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)

Thread #1

Thread ID \( T_{id} = 10 \)

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>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>
</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>
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.
Thread #1

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

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.
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.

<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>-</td>
<td>99999</td>
</tr>
<tr>
<td>A₂</td>
<td>1</td>
<td>-</td>
<td>2³¹⁻¹</td>
</tr>
<tr>
<td>A₃</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
</tbody>
</table>
OBSERVATION

<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>-</td>
<td>99999</td>
</tr>
<tr>
<td>A₂</td>
<td>0</td>
<td>-</td>
<td>(2^{31} - 1)</td>
</tr>
<tr>
<td>A₃</td>
<td>0</td>
<td>-</td>
<td>1</td>
</tr>
</tbody>
</table>

Thread #1

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

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.

Thread #2

\(T_{id}=1\)
POSTGRES TXN ID WRAPAROUND

Set a flag in each tuple header that says that it is "frozen" in the past. Any new txn id will always be newer than a frozen version.

Runs the vacuum before the system gets close to this upper limit.
Otherwise it must stop accepting new commands when the system gets close to the max txn id.
The DBMS uses the tuples’ pointer field to create a latch-free **version chain** per logical tuple.

→ This allows the DBMS to find the version that is visible to a particular txn at runtime.

→ Indexes always point to the “head” of the chain.

Different storage schemes determine where/what to store for each version.
VERSION STORAGE

Approach #1: Append-Only Storage
→ New versions are appended to the same table space.

Approach #2: Time-Travel Storage
→ Old versions are copied to separate table space.

Approach #3: Delta Storage
→ The original values of the modified attributes are copied into a separate delta record space.
All the physical versions of a logical tuple are stored in the same table space. The versions are mixed together.

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

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

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

On every update, append a new version of the tuple into an empty space in the table.
VERSION CHAIN ORDERING

Approach #1: Oldest-to-Newest (O2N)
→ Append every new version to end of the chain.
→ Must traverse chain on look-ups.

Approach #2: Newest-to-Oldest (N2O)
→ Must update index pointers for every new version.
→ Don’t have to traverse chain on look ups.

The ordering of the chain has different performance trade-offs.
TIME-TRAVEL STORAGE

### Main Table

<table>
<thead>
<tr>
<th></th>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₂</td>
<td>$222</td>
<td></td>
</tr>
<tr>
<td>B₁</td>
<td>$10</td>
<td></td>
</tr>
</tbody>
</table>

### Time-Travel Table

<table>
<thead>
<tr>
<th></th>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>$111</td>
<td>Ø</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>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₂</td>
<td>$222</td>
</tr>
<tr>
<td>B₁</td>
<td>$10</td>
</tr>
</tbody>
</table>

**Time-Travel Table**

<table>
<thead>
<tr>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>$111</td>
</tr>
<tr>
<td>Ø</td>
<td></td>
</tr>
<tr>
<td>A₂</td>
<td>$222</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>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₂</td>
<td>$222</td>
</tr>
<tr>
<td>B₁</td>
<td>$10</td>
</tr>
</tbody>
</table>

**Time-Travel Table**

<table>
<thead>
<tr>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>$111</td>
</tr>
<tr>
<td>A₂</td>
<td>$222</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 and update pointers.
### TIME-TRAVEL STORAGE

**Main Table**

<table>
<thead>
<tr>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A_3)</td>
<td>$333</td>
</tr>
<tr>
<td>(B_1)</td>
<td>$10</td>
</tr>
</tbody>
</table>

**Time-Travel Table**

<table>
<thead>
<tr>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A_1)</td>
<td>$111</td>
</tr>
<tr>
<td>(A_2)</td>
<td>$222</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 and update pointers.
On every update, copy the current version to the time-travel table. Update pointers.

Overwrite master version in the main table and update pointers.
TIME-TRAVEL STORAGE

### Main Table

<table>
<thead>
<tr>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₃</td>
<td>$333</td>
</tr>
<tr>
<td>B₁</td>
<td>$10</td>
</tr>
</tbody>
</table>

### Time-Travel Table

<table>
<thead>
<tr>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>$111</td>
</tr>
<tr>
<td>A₂</td>
<td>$222</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 and update pointers.
DELTA STORAGE

Main Table

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

Delta Storage Segment

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.
DELTA STORAGE

Main Table

<table>
<thead>
<tr>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2</td>
<td>$222</td>
</tr>
<tr>
<td>B1</td>
<td>$10</td>
</tr>
</tbody>
</table>

Delta Storage Segment

<table>
<thead>
<tr>
<th>DELTA</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>(VALUE→$111)</td>
<td>Ø</td>
</tr>
<tr>
<td>(VALUE→$222)</td>
<td></td>
</tr>
</tbody>
</table>

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>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_3</td>
<td>$333</td>
</tr>
<tr>
<td>B_1</td>
<td>$10</td>
</tr>
</tbody>
</table>

**Delta Storage Segment**

<table>
<thead>
<tr>
<th>DELTA</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>(VALUE→$111)</td>
<td>Ø</td>
</tr>
<tr>
<td>(VALUE→$222)</td>
<td>Ø</td>
</tr>
</tbody>
</table>

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>INT_VAL</th>
<th>STR_VAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>$100</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></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**

- MY_LONG_STRING
- 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></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

Main Table

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

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 \textit{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:

→ \textbf{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.
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.
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**

<table>
<thead>
<tr>
<th>Tid</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tid</td>
<td>12</td>
</tr>
</tbody>
</table>

**Thread #2**

<table>
<thead>
<tr>
<th>Tid</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tid</td>
<td>25</td>
</tr>
</tbody>
</table>

#### Vacuum

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

<table>
<thead>
<tr>
<th>Block</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_{101}</td>
<td>10</td>
<td>20</td>
</tr>
</tbody>
</table>
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**

**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.
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 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 must 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 a DELETE followed by an INSERT.

Secondary indexes are more complicated...
INDEX MANAGEMENT

PKey indexes always point to version chain head.

→ How often the DBMS must 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 a 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

GET(A) → PRIMARY INDEX

Physical Address

SECONDARY INDEX

Append-Only
Newest-to-Oldest

A₄ → A₃ → A₂ → A₁
INDEX POINTERS

PRIMARY INDEX

SECONDARY INDEX

GET(A)

Physical Address

Append-Only Newest-to-Oldest
INDEX POINTERS

PRIMARY INDEX

SECONDARY INDEX

SECONDARY INDEX

SECONDARY INDEX

SECONDARY INDEX

Append-Only
Newest-to-Oldest

GET(A)
INDEX POINTERS

---

**PRIMARY INDEX**

Physical Address

---

**SECONDARY INDEX**

Primary Key

---

WHERE A = A

NEWEST TO OLDEST

---

GET(A)

---

A4 → A3 → A2 → A1

---

Append-Only

Newest-to-Oldest
INDEX POINTERS

PRIMARY INDEX

SECONDARY INDEX

TupleId

TupleId → Address

Physical Address

A4 → A3 → A2 → A1

GET(A)

Append-Only
Newest-to-Oldest
MVCC INDEXES

MVCC DBMS indexes (usually) do not store version information about tuples with their keys. → Exception: Index-organized tables (e.g., MySQL)

Every index must support duplicate keys from different snapshots:
→ The same key may point to different logical tuples in different snapshots.
MVCC DUPLICATE KEY PROBLEM

Thread #1

Begin @ 10

READ(A)
Thread #1

Begin @ 10

Thread #2

Begin @ 20

MVCC DUPLICATE KEY PROBLEM

Index

Update(A)

Thread #1

Begin @ 10

Read(A)

Thread #2

Begin @ 20

Update(A)

CMU-DB
**MVCC DUPLICATE KEY PROBLEM**

**Thread #1**
*Begin @ 10*

**Thread #2**
*Begin @ 20*

![Index Diagram]

<table>
<thead>
<tr>
<th></th>
<th>BEGIN-TS</th>
<th>END-TS</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>1</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>A₂</td>
<td>20</td>
<td>∞</td>
<td>Ø</td>
</tr>
</tbody>
</table>

**Index**

**Update (A)**

**Read (A)**
MVCC DUPLICATE KEY PROBLEM

Thread #1

Begin @ 10

READ(A)

Thread #2

Begin @ 20

UPDATE(A) DELETE(A)

Index

Table:

<table>
<thead>
<tr>
<th></th>
<th>BEGIN-TS</th>
<th>END-TS</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>1</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>∞</td>
<td>Ø</td>
</tr>
</tbody>
</table>
MVCC DUPLICATE KEY PROBLEM

Thread #1
Begin @ 10
READ(A)

Thread #2
Begin @ 20
Commit @ 25
UPDATE(A)
DELETE(A)

Index

<table>
<thead>
<tr>
<th>BEGIN-TS</th>
<th>END-TS</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_1</td>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>X</td>
<td>25</td>
<td>25</td>
</tr>
</tbody>
</table>
### MVCC DUPLICATE KEY PROBLEM

#### Thread #1
*Begin @ 10*
- **READ(A)**

#### Thread #2
*Begin @ 20*
- **UPDATE(A)**
- **DELETE(A)**
*Commit @ 25*

#### Thread #3
*Begin @ 30*
- **INSERT(A)**

---

**Index**

**Table**

<table>
<thead>
<tr>
<th>BEGIN-TS</th>
<th>END-TS</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>X</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>A₁</td>
<td>30</td>
<td>∞</td>
</tr>
</tbody>
</table>
MVCC DUPLICATE KEY PROBLEM

Thread #1
Begin @ 10
READ(A)  READ(A)

Thread #2
Begin @ 20
Commit @ 25
UPDATE(A)  DELETE(A)

Thread #3
Begin @ 30
INSERT(A)

Index

<table>
<thead>
<tr>
<th>BEGIN-TS</th>
<th>END-TS</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>x</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>A₁</td>
<td>30</td>
<td>∞</td>
</tr>
</tbody>
</table>
MVCC INDEXES

Each index's underlying data structure must support the storage of non-unique keys.

Use additional execution logic to perform conditional inserts for pkey / unique indexes.
→ Atomically check whether the key exists and then insert.

Workers may get back multiple entries for a single fetch. They then must follow the pointers to find the proper physical version.
We implemented all 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>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-TO</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>
<tr>
<td>CMU's TBD</td>
<td>MV-OCC</td>
<td>Delta</td>
<td>Txn-level</td>
</tr>
</tbody>
</table>
MVCC CONFIGURATION EVALUATION

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

Throughput (txn/sec)

# Threads

Oracle/MySQL
NuoDB
HyPer
HYRISE
MemSQL
HANA
HEKATON
Postgres

15-721 (Spring 2020)
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 also to blog the

<table>
<thead>
<tr>
<th>Database</th>
<th>Throughput (txn/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle/MySQL</td>
<td>8</td>
</tr>
<tr>
<td>NuoDB</td>
<td>16</td>
</tr>
<tr>
<td>HyPer</td>
<td>24</td>
</tr>
<tr>
<td>HYRISE</td>
<td>32</td>
</tr>
<tr>
<td>MemSQL</td>
<td>40</td>
</tr>
<tr>
<td>HANA</td>
<td>50</td>
</tr>
<tr>
<td>HEKATON</td>
<td>70</td>
</tr>
<tr>
<td>Postgres</td>
<td>75</td>
</tr>
</tbody>
</table>

Database: TPC-C Benchmark (40 Warehouses)

Processor: 4 sockets, 10 cores per socket
PROJECT #1

Identify bottlenecks in the DBMS's sequential scan implementation using profiling tools and refactor the system to remove it.

This project is meant to teach you how to work in a highly concurrent system.
CMU’s new in-memory hybrid relational DBMS
→ HyPer-style MVCC column store
→ Multi-threaded architecture
→ Latch-free Bw-Tree Index
→ Native support for Apache Arrow format
→ Vectorized Execution Engine
→ MemSQL-style LLVM-based Query Compilation
→ Cascades-style Query Optimizer
→ Postgres Wire Protocol / Catalog Compatible

Long term vision is to build a "self-driving" system
PROJECT #1 – TESTING

We are providing you with a suite of C++ benchmarks for you check your implementation.
→ Focus on the ConcurrentSlotIterators microbenchmark but you will want to run all of them to make sure your code works.

We strongly encourage you to do your own additional testing.
→ Different workloads
→ Different # of threads
→ Different access patterns
PROJECT #1 – GRADING

We will run additional tests beyond what we provided you for grading.
We will also use Google's Sanitizers when testing your code.

All source code must pass ClangFormat + ClangTidy syntax formatting checker.
→ See documentation for formatting guidelines
DEVELOPMENT ENVIRONMENT

The DBMS builds on Ubuntu 18.04+ and OSX. → You can also do development on docker or VM.

This is CMU so I’m going to assume that each of you can get access to a machine.

Important: You will not be able to identify the bottleneck on a machine with less than 8 cores.
TESTING ENVIRONMENT

Every student will receive $50 of Amazon AWS credits to run experiments on EC2.
→ Setup monitoring + alerts to prevent yourself from burning through your credits.
→ Use spot instances whenever possible.

Target EC2 Instance: **c5.9xlarge**
→ On Demand: $1.53/hr
→ Spot Instance: $0.34/hr (as of Jan 2020)
PROJECT #1

Due Date: February 16th @ 11:59pm
Source code + final report will be turned in using Gradescope but graded using a different machine.

Full description and instructions:
https://15721.courses.cs.cmu.edu/spring2020/project1.html
PARTING THOUGHTS

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

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

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