Project #3 is due Sun Nov 22nd @ 11:59pm.

Q&A Session on Wed Nov 11th @ 8:00pm
→ https://cmu.zoom.us/j/96880648178?pwd=Z0loZUVORVV1eURFc2R0aDR6QU5udz09

No class on Wed Nov 11th
UPCOMING DATABASE TALKS

EraDB "Magical SuperIndexes"
→ Monday Nov 9th @ 5pm ET

FaunaDB Serverless DBMS
→ Monday Nov 16th @ 5pm ET

Confluent ksqlDB (Kafka)
→ Monday Nov 16th @ 5pm ET
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.
Protocol was first proposed in 1978 MIT PhD dissertation.

First implementations was Rdb/VMS and InterBase at DEC in early 1980s.

→ Both were by Jim Starkey, co-founder of NuoDB.
→ DEC Rdb/VMS is now "Oracle Rdb"
→ InterBase was open-sourced as Firebird.
Multi-Version Concurrency Control

Writers do not block readers.
Readers do not block writers.

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

Easily support time-travel queries.
TMVC – EXAMPLE #1

T\textsubscript{S}(T\textsubscript{1}) = 1

\begin{array}{c|c|c}
\hline
Version & Value & Begin & End \\
\hline
A_0 & 123 & 0 & - \\
\hline
\end{array}

\begin{tikzpicture}

\node[coordinate] (begin) at (0,0) {BEGIN};
\node[coordinate] (read) at (1.5,0) {R(A)};
\node[coordinate] (commit1) at (3,0) {COMMIT};
\node[coordinate] (begin2) at (0,1.5) {BEGIN};
\node[coordinate] (write) at (1.5,1.5) {W(A)};
\node[coordinate] (commit2) at (3,1.5) {COMMIT};

\draw[->, thick, red] (begin) -- (read);
\draw[->, thick, red] (read) -- (commit1);
\draw[->, thick, red] (begin2) -- (write);
\draw[->, thick, red] (write) -- (commit2);

\node at (0,2) {T\textsubscript{1}};
\node at (3,2) {T\textsubscript{2}};
\node at (1.5,1) {TS(T\textsubscript{2}) = 2};
\node at (1.5,3) {TS(T\textsubscript{1}) = 1};
\end{tikzpicture}
**MVCC – EXAMPLE #1**

<table>
<thead>
<tr>
<th>Version</th>
<th>Value</th>
<th>Begin</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_0$</td>
<td>123</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>$A_1$</td>
<td>456</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>

$T_1$ creates version $A_0$ and sets $A_0$ End-TS.

$T_2$ creates version $A_1$ and sets $A_0$ End-TS.
**MVCC – EXAMPLE #1**

<table>
<thead>
<tr>
<th>Time</th>
<th>Schedule</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>BEGIN R(A)</td>
<td>$TS(T_1)=1$</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
<td></td>
</tr>
<tr>
<td>$T_2$</td>
<td>BEGIN W(A)</td>
<td>$TS(T_2)=2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$T_2$ creates version $A_1$ and sets $A_0$ End-TS.

<table>
<thead>
<tr>
<th>Version</th>
<th>Value</th>
<th>Begin</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_0$</td>
<td>123</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>$A_1$</td>
<td>456</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
**MVCC – EXAMPLE #1**

**Transaction Status Table**

<table>
<thead>
<tr>
<th>TxnId</th>
<th>Timestamp</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>1</td>
<td>Active</td>
</tr>
<tr>
<td>T₂</td>
<td>2</td>
<td>Active</td>
</tr>
</tbody>
</table>

**Database**

<table>
<thead>
<tr>
<th>Version</th>
<th>Value</th>
<th>Begin</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₀</td>
<td>123</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>A₁</td>
<td>456</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>

**MVCC Example #1**

**Schedule**

- **T₁** begins transaction
  - **R(A)**
  - **TS(T₁) = 1**
- **T₂** begins transaction
  - **R(A)**
  - **W(A)**
  - **TS(T₂) = 2**
- **T₂** creates version A₁ and sets A₀ End-TS.

**TS(T₁) = 1**

**TS(T₂) = 2**
**MVCC – EXAMPLE #1**

**Txn Status Table**

<table>
<thead>
<tr>
<th>TxnId</th>
<th>Timestamp</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>1</td>
<td>Active</td>
</tr>
<tr>
<td>T₂</td>
<td>2</td>
<td>Active</td>
</tr>
</tbody>
</table>

**Database**

<table>
<thead>
<tr>
<th>Version</th>
<th>Value</th>
<th>Begin</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₀</td>
<td>123</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>A₁</td>
<td>456</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>

**MVCC Example #1**

- **BEGIN** R(A)
- **R(A)**
- **BEGIN** W(A)
- **W(A)**
- **COMMIT**
- **COMMIT**

**Schedule**

- **TS(T₁)=1**
- **TS(T₂)=2**

**TIME**
**MVCC – EXAMPLE #1**

**Database**

<table>
<thead>
<tr>
<th>Version</th>
<th>Value</th>
<th>Begin</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₀</td>
<td>123</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>A₁</td>
<td>456</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>

**Txn Status Table**

<table>
<thead>
<tr>
<th>TxnId</th>
<th>Timestamp</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>1</td>
<td>Active</td>
</tr>
<tr>
<td>T₂</td>
<td>2</td>
<td>Active</td>
</tr>
</tbody>
</table>

**Schedule**

- **TS(T₁)=1**
- **TS(T₂)=2**

**T₁** reads version **A₀**.
**MVCC – EXAMPLE #1**

**Schedule**

- **TS(T₁)=1**
- **TS(T₂)=2**

**Database**

<table>
<thead>
<tr>
<th>Version</th>
<th>Value</th>
<th>Begin</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₀</td>
<td>123</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>A₁</td>
<td>456</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>

**Txn Status Table**

<table>
<thead>
<tr>
<th>TxnId</th>
<th>Timestamp</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>1</td>
<td>Active</td>
</tr>
<tr>
<td>T₂</td>
<td>2</td>
<td>Active</td>
</tr>
</tbody>
</table>

**Example #1**

- **BEGIN**
  - R(A)
  - W(A)
- **COMMIT**

**Time:**

- T₁
- T₂

**TS(T₁)=1**

**TS(T₂)=2**
**MVCC – EXAMPLE #2**

**Txn Status Table**

<table>
<thead>
<tr>
<th>TxnId</th>
<th>Timestamp</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T_1)</td>
<td>1</td>
<td>Active</td>
</tr>
</tbody>
</table>

**Database**

<table>
<thead>
<tr>
<th>Version</th>
<th>Value</th>
<th>Begin</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A_0)</td>
<td>123</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**TS(\(T_1\))=1**

\(T_1\)

BEGIN

\(R(A)\)

\(W(A)\)

\(R(A)\)

COMMIT

**TS(\(T_2\))=2**

\(T_2\)

BEGIN

\(R(A)\)

\(W(A)\)

\(R(A)\)

COMMIT

TIME

**Schedule**

\(T_1\)

\(T_2\)
**MVCC – EXAMPLE #2**

- **Txn Status Table**
  - **TxnId** | **Timestamp** | **Status**
  - $T_1$ | 1 | Active

- **Transaction Schedule**
  - $T_1$
    - BEGIN
    - R(A)
    - W(A)
    - R(A)
    - COMMIT
  - $T_2$
    - BEGIN
    - R(A)
    - W(A)
    - COMMIT

- **Database**
  - | Version | Value | Begin | End |
  - | A_0   | 123   | 0     |     |
  - | A_1   | 456   | 1     |     |

- **MVCC Example**
  - $TS(T_1)=1$
  - $TS(T_2)=2$
MVCC – EXAMPLE #2

**Schedule**

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>BEGIN R(A) W(A)</td>
</tr>
<tr>
<td></td>
<td>R(A) COMMIT</td>
</tr>
<tr>
<td>$T_2$</td>
<td>BEGIN R(A) W(A)</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

**TS($T_1$) = 1**

**TS($T_2$) = 2**

**Txn Status Table**

<table>
<thead>
<tr>
<th>TxnId</th>
<th>Timestamp</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>1</td>
<td>Active</td>
</tr>
</tbody>
</table>

**Database**

<table>
<thead>
<tr>
<th>Version</th>
<th>Value</th>
<th>Begin</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_0$</td>
<td>123</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$A_1$</td>
<td>456</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>
**MVCC – EXAMPLE #2**

**Schedule**

- **TS(T₁) = 1**
  - T₁
    - BEGIN
    - R(A)
    - W(A)
    - COMMIT

- **TS(T₂) = 2**
  - T₂
    - BEGIN
    - R(A)
    - W(A)
    - COMMIT

**Database**

<table>
<thead>
<tr>
<th>Version</th>
<th>Value</th>
<th>Begin</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₀</td>
<td>123</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A₁</td>
<td>456</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

**Timestamp Status Table**

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₂</td>
<td>Active</td>
</tr>
<tr>
<td>T₁</td>
<td>Active</td>
</tr>
</tbody>
</table>

**Explanation:**

- T₂ reads version A₀ because T₁ has not committed yet.
**MVCC – EXAMPLE #2**

<table>
<thead>
<tr>
<th>Transaction Status Table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Version</strong></td>
</tr>
<tr>
<td>:-----------:</td>
</tr>
<tr>
<td>A₀</td>
</tr>
<tr>
<td>A₁</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Txn Status Table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TxnId</strong></td>
</tr>
<tr>
<td>:-------:</td>
</tr>
<tr>
<td>T₁</td>
</tr>
<tr>
<td>T₂</td>
</tr>
</tbody>
</table>

**Database**

**Schedule**

- T₁: BEGIN R(A) W(A) R(A) COMMIT
- T₂: BEGIN R(A) W(A) COMMIT

**MVCC**

- **TS(T₁)** = 1
- **TS(T₂)** = 2
### MVCC – EXAMPLE #2

1. **Schedule**
   - **TS(T₁)=1**
   - **TS(T₂)=2**

2. **Database**
   - **Txn Status Table**
     - | Version | Value | Begin | End |
     - | --- | ------ | --- | --- |
     - | A₀ | 123 | 0 | 1 |
     - | A₁ | 456 | 1 | - |

3. **Txn Status Table**
   - | Stamp | Status |
   - | T₁ | Active |
   - | T₂ | Active |

**T₂ must stall until T₁ commits.**
**MVCC – EXAMPLE #2**

- **Txn Status Table**
  - **TxnId** | **Timestamp** | **Status**
  - T₁ | 1 | Active
  - T₂ | 2 | Active

- **Database**
  - **Version** | **Value** | **Begin** | **End**
  - A₀ | 123 | 0 | 1
  - A₁ | 456 | 1 | -

- **Txn Schedule**
  - BEGIN
  - R(A)
  - W(A)
  - T₁
  - R(A)
  - W(A)
  - COMMIT
  - TS(T₁) = 1

- **Transaction Status Table**
  - **TxnId** | **Timestamp** | **Status**
  - T₁ | 1 | Active
  - T₂ | 2 | Active

- **TS(T₁) = 1**
  - T₁ reads version A₁ that it wrote earlier.

- **TS(T₂) = 2**
**MVCC – EXAMPLE #2**

**Txn Status Table**

<table>
<thead>
<tr>
<th>TxnId</th>
<th>Timestamp</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>1</td>
<td>Committed</td>
</tr>
<tr>
<td>T₂</td>
<td>2</td>
<td>Active</td>
</tr>
</tbody>
</table>

**Database**

<table>
<thead>
<tr>
<th>Version</th>
<th>Value</th>
<th>Begin</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₀</td>
<td>123</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A₁</td>
<td>456</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

**Schedule**

- TS(T₁)=1
- TS(T₂)=2

**TIME**

- T₁: BEGIN
  - R(A)
  - W(A)
  - R(A)
  - COMMIT

- T₂: BEGIN
  - R(A)
  - W(A)
  - COMMIT

EXAM PLE #2

- BEGIN
  - R(A)
  - W(A)

- COMMIT
**MVCC – EXAMPLE #2**

### Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
</tr>
</tbody>
</table>

### Transaction Status Table

<table>
<thead>
<tr>
<th>TxnId</th>
<th>Timestamp</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>1</td>
<td>Committed</td>
</tr>
<tr>
<td>T₂</td>
<td>-</td>
<td>Active</td>
</tr>
</tbody>
</table>

### MVCC Example

- **TS(T₁)=1**
- **TS(T₂)=2**

<table>
<thead>
<tr>
<th>Version</th>
<th>Value</th>
<th>Begin</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₀</td>
<td>123</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A₁</td>
<td>456</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>A₂</td>
<td>789</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>

Now T₂ can create the new version.
MULTI-VERSION CONCURRENCY CONTROL

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
Deletes
CONCURRENCY CONTROL PROTOCOL

Approach #1: Timestamp Ordering
→ Assign txns timestamps that determine serial order.

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.
The DBMS uses the tuples' pointer field to create a version chain per logical tuple.

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

→ Indexes always point to the "head" of the chain.

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

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

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

Approach #3: Delta Storage
→ The original values of the modified attributes are copied into a separate delta record space.
APPEND-ONLY STORAGE

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

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

### Main Table

<table>
<thead>
<tr>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₀</td>
<td>$111</td>
</tr>
<tr>
<td>A₁</td>
<td>$222</td>
</tr>
<tr>
<td>B₁</td>
<td>$10</td>
</tr>
</tbody>
</table>
All the physical versions of a logical tuple are stored in the same table space. The versions are inter-mixed.

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 inter-mixed.

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 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.
→ Do not have to traverse chain on look-ups.
**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>
</tbody>
</table>

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.
**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.
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>

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

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.

**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>Ø</td>
</tr>
</tbody>
</table>
**DELTA STORAGE**

### Main Table

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

### Delta Storage Segment

<table>
<thead>
<tr>
<th></th>
<th>DELTA</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_1</td>
<td>(VALUE→$111)</td>
<td>ø</td>
</tr>
<tr>
<td>A_2</td>
<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.
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.

**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.
TUPLE-LEVEL GC

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

<table>
<thead>
<tr>
<th>VERSION</th>
<th>BEGIN</th>
<th>END</th>
</tr>
</thead>
<tbody>
<tr>
<td>A$_{100}$</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>B$_{100}$</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>B$_{101}$</td>
<td>10</td>
<td>20</td>
</tr>
</tbody>
</table>

Thread #1

TS(T$_1$) = 12

Thread #2

TS(T$_2$) = 25
**TUPLE-LEVEL GC**

**Thread #1**

\[ TS(T_1) = 12 \]

**Thread #2**

\[ TS(T_2) = 25 \]

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

<table>
<thead>
<tr>
<th>VERSION</th>
<th>BEGIN</th>
<th>END</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_{100}</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>B_{100}</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>B_{101}</td>
<td>10</td>
<td>20</td>
</tr>
</tbody>
</table>
### TUPLE-LEVEL GC

**Thread #1**
- $TS(T_1) = 12$

**Thread #2**
- $TS(T_2) = 25$

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

<table>
<thead>
<tr>
<th>VERSION</th>
<th>BEGIN</th>
<th>END</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_{100}$</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>$B_{100}$</td>
<td>1</td>
<td>9</td>
</tr>
<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.
**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.

**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 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 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 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 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 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.
Primary key indexes 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...
Primary key indexes 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

PRIMARY INDEX

SECONDARY INDEX

A_{100} \rightarrow A_{99} \rightarrow A_{98} \rightarrow A_{97}

Append-Only
Newest-to-Oldest
INDEX POINTERS

GET(A) → PRIMARY INDEX

SECONDARY INDEX

Physical Address

A_{100} → A_{99} → A_{98} → A_{97}

Append-Only
Newest-to-Oldest
INDEX POINTERS

PRIMARY INDEX

SECONDARY INDEX

GET(A)

Physical Address

Append-Only Newest-to-Oldest

$A_{100} \rightarrow A_{99} \rightarrow A_{98} \rightarrow A_{97}$
INDEX POINTERS

PRIMARY INDEX

SECONDARY INDEX

SECONDARY INDEX

SECONDARY INDEX

SECONDARY INDEX

Append-Only
Newest-to-Oldest

GET(A)
INDEX POINTERS

PRIMARY INDEX

SECONDARY INDEX

GET(A)

Physical Address

Primary Key

A_{100} \rightarrow A_{99} \rightarrow A_{98} \rightarrow A_{97}

Append-Only
Newest-to-Oldest
INDEX POINTERS

PRIMARY INDEX

SECONDARY INDEX

GET(A)

TupleId

Physical Address

TupleId → Address

A_{100} → A_{99} → A_{98} → A_{97}

Append-Only
Newest-to-Oldest
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

Index

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

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

*Read (A)*

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

*Update (A)*

**Index**

**Table:**

<table>
<thead>
<tr>
<th>VERSION</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_1</td>
<td>1</td>
<td>2θ</td>
<td></td>
</tr>
<tr>
<td>A_2</td>
<td>2θ</td>
<td>∞</td>
<td>Ø</td>
</tr>
</tbody>
</table>
MVCC DUPLICATE KEY PROBLEM

Thread #1
Begin @ 10
READ(A)

Thread #2
Begin @ 20
UPDATE(A), DELETE(A)

Index

<table>
<thead>
<tr>
<th>VERSION</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>X</td>
<td>2₀</td>
<td>∞</td>
<td>φ</td>
</tr>
</tbody>
</table>

CMU-DB
**MVCC DUPLICATE KEY PROBLEM**

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

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

**Index**

**Version Table**

<table>
<thead>
<tr>
<th>VERSION</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>X</td>
<td>2₀</td>
<td>∞</td>
<td>Ø</td>
</tr>
</tbody>
</table>
MVCC DUPLICATE KEY PROBLEM

Thread #1
Begin @ 10

Thread #2
Begin @ 20
Commit @ 25

Index

<table>
<thead>
<tr>
<th>VERSION</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>1</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>25</td>
<td>Ø</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

### Version Table

<table>
<thead>
<tr>
<th>VERSION</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>1</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>25</td>
<td>Ø</td>
</tr>
</tbody>
</table>

- The page contains a demonstration of the MVCC (Multi-Version Concurrency Control) protocol, showing how transactions can conflict with each other, leading to a duplicate key problem.
- Thread #1 reads A at time 10.
- Thread #2 updates A at time 20 and deletes A at time 25.
- Thread #3 inserts A at time 30.
- The version table shows conflicting versions of A, highlighting the issue of duplicate keys.
- The index structure is also depicted, illustrating how transactions access and update versions of the same key.
MVCC DUPLICATE KEY PROBLEM

Thread #1
Begin @ 10
READ(A)

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

Thread #3
Begin @ 30
INSERT(A)

Index

<table>
<thead>
<tr>
<th>VERSION</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>1</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>✘</td>
<td>25</td>
<td>✔</td>
<td>Ø</td>
</tr>
<tr>
<td>A₁</td>
<td>30</td>
<td>∞</td>
<td>Ø</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:**

<table>
<thead>
<tr>
<th>VERSION</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>1</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>25</td>
<td>25</td>
<td>Ø</td>
</tr>
<tr>
<td>A₁</td>
<td>30</td>
<td>∞</td>
<td>Ø</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.
MVCC DELETES

The DBMS physically deletes a tuple from the database only when all versions of a logically deleted tuple are not visible.

→ If a tuple is deleted, then there cannot be a new version of that tuple after the newest version.

→ No write-write conflicts / first-writer wins

We need a way to denote that tuple has been logically delete at some point in time.
MVCC DELETES

Approach #1: Deleted Flag
→ Maintain a flag to indicate that the logical tuple has been deleted after the newest physical version.
→ Can either be in tuple header or a separate column.

Approach #2: Tombstone Tuple
→ Create an empty physical version to indicate that a logical tuple is deleted.
→ Use a separate pool for tombstone tuples with only a special bit pattern in version chain pointer to reduce the storage overhead.
# 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-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>
CONCLUSION

MVCC is the widely used scheme in DBMSs. Even systems that do not support multi-statement txns (e.g., NoSQL) use it.
NEXT CLASS

No class on Wed November 11th