Project #3 is due Sun Nov 17\textsuperscript{th} @ 11:59pm.

Homework #4 was released last week. It is due Wed Nov 13\textsuperscript{th} @ 11:59pm.
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 don't block readers.
Readers don't block writers.

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

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

**Schedule**

<table>
<thead>
<tr>
<th></th>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
<td>BEGIN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COMMIT</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></td>
</tr>
</tbody>
</table>
**MVCC – EXAMPLE #1**

**Schedule**

<table>
<thead>
<tr>
<th>Time</th>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BEGIN R(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
<td>BEGIN W(A)</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
<td>COMMIT</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>-</td>
</tr>
</tbody>
</table>
MVCC – EXAMPLE #1

**Schedule**

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>TS(T₁) = 1</td>
</tr>
<tr>
<td>T₂</td>
<td>TS(T₂) = 2</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>-</td>
</tr>
</tbody>
</table>

**Example**

- BEGIN
- R(A)
- R(A)
- COMMIT
- BEGIN
- W(A)
- COMMIT
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>-</td>
</tr>
</tbody>
</table>

TIME

- BEGIN
- R(A)
- COMMIT
- R(A)
- BEGIN
- W(A)
- COMMIT
- COMMIT
**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>-</td>
</tr>
<tr>
<td>A₁</td>
<td>456</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>

### Schedule

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

**T₂ creates version A₁ and sets A₀ End-TS.**
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

T₁

BEGIN
R(A)
BEGIN
R(A)
COMMIT

TS(T₂) = 2

T₂

BEGIN
W(A)

T₂ creates version A₁ and sets A₀ End-TS.
**MVCC – EXAMPLE #1**

**Transaction 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>
<tr>
<td>T_2</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>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>

**Schedule**

- **T_1** reads version A_0.
- **TS(T_1)=1**
- **TS(T_2)=2**

**Example MVCC**

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

**Database Timeline**

- TS(T_1) = 1
- TS(T_2) = 2

**Time**

- **T_1**
  - Reads version A_0.
- **T_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>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></td>
</tr>
</tbody>
</table>

**Schedule**

- $TS(T₁)=1$
- $TS(T₂)=2$

TIME

- $T₁$
- $T₂$
**MVCC – EXAMPLE #2**

**Schedule**

- **T1**
  - Begin
  - R(A)
  - W(A)
  - R(A)
  - COMMIT

- **T2**
  - Begin
  - R(A)
  - W(A)
  - COMMIT

**Database**

- **Version**: A0
- **Value**: 123
- **Begin**: 0

**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>
</tbody>
</table>

**MVCC**

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

### Schedule

- **TS(T₁)=1**
  - T₁
    - BEGIN
    - R(A)
    - W(A)
    - R(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></td>
</tr>
<tr>
<td>A₁</td>
<td>456</td>
<td>1</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>
</tbody>
</table>
**MVCC – EXAMPLE #2**

*TS(T₁)=1*

```
BEGIN
R(A)
W(A)
COMMIT
```

*TS(T₂)=2*

```
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></td>
</tr>
<tr>
<td>A₁</td>
<td>456</td>
<td>1</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>
</tbody>
</table>

**Schedule**

```
T₁ → T₂
```

**TIME**
# 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>1</td>
</tr>
<tr>
<td>$A_1$</td>
<td>456</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

**MVCC**

- $TS(T_1)=1$
- $TS(T_2)=2$

**Schedule**

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

**Example Transactions**

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

- BEGIN
  - $R(A)$
  - $W(A)$
  - COMMIT
**MVCC – EXAMPLE #2**

**Schedule**

- $TS(T_1) = 1$
- $TS(T_2) = 2$

**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>

**Status Table**

<table>
<thead>
<tr>
<th>Stamp</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_2$</td>
<td>Active</td>
</tr>
<tr>
<td>2</td>
<td>Active</td>
</tr>
</tbody>
</table>

$T_2$ reads version $A_0$ because $T_1$ has not committed yet.
**MVCC – EXAMPLE #2**

- **Schedule**
  - $TS(T_1)=1$
  - $TS(T_2)=2$

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

- **Txn Status Table**
  - | Stamp | Status |
  - |------|--------|
  - | $T_1$ | Active |
  - | $T_2$ | Active |

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

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

$T_2$ has to stall until $T_1$ commits.
**MVCC – EXAMPLE #2**

<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>

$TS(T_1) = 1$

$TS(T_2) = 2$

$T_1$ reads version $A_1$ that it wrote earlier.

**Transaction 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>
<tr>
<td>$T_2$</td>
<td>2</td>
<td>Active</td>
</tr>
</tbody>
</table>
**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>

**TS(T₁)=1**

**TS(T₂)=2**

**Schedule**

- `T₁`: BEGIN, R(A), W(A), R(A), COMMIT
- `T₂`: BEGIN, R(A), W(A), COMMIT
MVCC – EXAMPLE #2

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
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.
VERSION STORAGE

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

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

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

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

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

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

Approach #1: Oldest-to-Newest (O2N)
→ Just append new version to end of the chain.
→ Have to traverse chain on look-ups.

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

Main Table

<table>
<thead>
<tr>
<th>VERSION</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>VERSION</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.
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.
### TIME-TRAVEL STORAGE

#### Main Table

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

#### Time-Travel Table

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

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

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

**Main Table**

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

**Time-Travel Table**

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

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

Overwrite master version in the main table. Update pointers.
DELTA STORAGE

Main Table

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

### Main Table

<table>
<thead>
<tr>
<th>VERSION</th>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>$111</td>
<td></td>
</tr>
<tr>
<td>B₁</td>
<td>$10</td>
<td></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.
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.
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.
**TUPLE-LEVEL GC**

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

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

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

![Diagram of Vacuum and Dirty Page BitMap table]

<table>
<thead>
<tr>
<th>VERSION</th>
<th>BEGIN</th>
<th>END</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_{101}$</td>
<td>10</td>
<td>20</td>
</tr>
</tbody>
</table>
TUPLE-LEVEL GC

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

**Cooperative Cleaning:**
Worker threads identify reclaimable versions as they traverse version chain. Only works with O2N.
TUPLE-LEVEL GC

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

**Cooperative Cleaning:**
Worker threads identify reclaimable versions as they traverse version chain. Only works with O2N.
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 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.
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.
INDEX MANAGEMENT

Primary key indexes point to version chain head.

→ How often the DBMS has to update the pkey index depends on whether the system creates new versions when a tuple is updated.

→ If a txn updates a tuple’s pkey attribute(s), then this is treated as an DELETE followed by an INSERT.

Secondary indexes are more complicated...
SECONDARY INDEXES

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

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

PRIMARY INDEX

SECONDARY INDEX

A_{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

Append-Only
Newest-to-Oldest

A_{100} ➔ A_{99} ➔ A_{98} ➔ A_{97}
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)

A100  A99  A98  A97
INDEX POINTERS

PRIMARY INDEX

SECONDARY INDEX

Physical Address

Primary Key

GET(A)

Append-Only
Newest-to-Oldest
INDEX POINTERS

PRIMARY INDEX

SECONDARY INDEX

GET(A)

TupleId

TupleId → Address

Physical Address

A_{100} → A_{99} → A_{98} → A_{97}

Append-Only
Newest-to-Oldest
# MVCC IMPLEMENTATIONS

<table>
<thead>
<tr>
<th></th>
<th>Protocol</th>
<th>Version Storage</th>
<th>Garbage Collection</th>
<th>Indexes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle</td>
<td>MV2PL</td>
<td>Delta</td>
<td>Vacuum</td>
<td>Logical</td>
</tr>
<tr>
<td>Postgres</td>
<td><strong>MV-2PL/MV-TO</strong></td>
<td><strong>Append-Only</strong></td>
<td>Vacuum</td>
<td>Physical</td>
</tr>
<tr>
<td>MySQL-InnoDB</td>
<td><strong>MV-2PL</strong></td>
<td>Delta</td>
<td>Vacuum</td>
<td>Logical</td>
</tr>
<tr>
<td>HYRISE</td>
<td><strong>MV-OCC</strong></td>
<td>Append-Only</td>
<td>–</td>
<td>Physical</td>
</tr>
<tr>
<td>Hekaton</td>
<td><strong>MV-OCC</strong></td>
<td>Append-Only</td>
<td>Cooperative</td>
<td>Physical</td>
</tr>
<tr>
<td>MemSQL</td>
<td><strong>MV-OCC</strong></td>
<td>Append-Only</td>
<td>Vacuum</td>
<td>Physical</td>
</tr>
<tr>
<td>SAP HANA</td>
<td><strong>MV-2PL</strong></td>
<td>Time-travel</td>
<td>Hybrid</td>
<td>Logical</td>
</tr>
<tr>
<td>NuoDB</td>
<td><strong>MV-2PL</strong></td>
<td>Append-Only</td>
<td>Vacuum</td>
<td>Logical</td>
</tr>
<tr>
<td>HyPer</td>
<td><strong>MV-OCC</strong></td>
<td>Delta</td>
<td>Txn-level</td>
<td>Logical</td>
</tr>
<tr>
<td><strong>CMU's TBD</strong></td>
<td><strong>MV-OCC</strong></td>
<td>Delta</td>
<td>Txn-level</td>
<td>Logical</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 6th
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 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.
Thread #1

Begin @ 10

READ(A)

Index

MVCC DUPLICATE KEY PROBLEM

<table>
<thead>
<tr>
<th>VERSION</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
<th>POINTERS</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)*
MVCC DUPLICATE KEY PROBLEM

Thread #1
Begin @ 10
READ(A)

Thread #2
Begin @ 20
UPDATE(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>A₂</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
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>Ø</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*

**UPDATE(A)**

*Commit @ 25*

**DELETE(A)**

---

**Index**

---

**Transaction Log**

<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>Ø</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>∅</td>
</tr>
</tbody>
</table>
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>X</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)**

#### 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>A₂</td>
<td>25</td>
<td>25</td>
<td>∅</td>
</tr>
<tr>
<td>A₁</td>
<td>30</td>
<td>∞</td>
<td>∅</td>
</tr>
</tbody>
</table>
MVCC DUPLICATE KEY PROBLEM

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

Thread #2
- Begin @ 20
- 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 has to 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 have to follow the pointers to find the proper physical version.