Multi-Version Concurrency Control
ADMINISTRIVIA

Homework #4: Monday Nov 12th @ 11:59pm

Project #3: Monday Nov 19th @ 11:59am
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

\[
\begin{array}{c|c}
T_1 & T_2 \\
\hline
\text{BEGIN} & \text{BEGIN} \\
\text{R(A)} & \text{W(A)} \\
\text{COMMIT} & \text{COMMIT} \\
\end{array}
\]

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>
MVCC – EXAMPLE #1

Schedule

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<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>

TIME

BEGIN R(A)
BEGIN W(A)
COMMIT
MVCC – EXAMPLE #1

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS(T₁) = 1</td>
<td>BEGIN R(A)</td>
</tr>
<tr>
<td>T₁</td>
<td>BEGIN W(A)</td>
</tr>
<tr>
<td>T₂</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

<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
MVCC – EXAMPLE #1

**Schedule**

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

**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>-</td>
</tr>
</tbody>
</table>
MVCC – EXAMPLE #1

**Schedule**

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

- **T2**
  - BEGIN
  - R(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>2</td>
<td>-</td>
</tr>
</tbody>
</table>

- **TS(T₁) = 1**
- **TS(T₂) = 2**

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>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>

**Schedule**

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

**Notes**

- **TS(T₁) = 1**
- **TS(T₂) = 2**
- **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>$A_0$</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>

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

**Schedule**

- $T_1$ reads version $A_0$.
- $T_1$ reads $A_0$.
- $T_2$ writes $A_1$.
- $T_1$ commits.
- $T_2$ commits.

**MVCC Timeline**

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

**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**

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

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

**Time**

- T₁
- T₂
### MVCC – EXAMPLE #2

**Schedule**

- **TS(T₁)=1**
  - Begin: T₁
  - R(A)
  - W(A)
  - R(A)
  - Commit

- **TS(T₂)=2**
  - Begin: T₂
  - 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>
</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**

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

**Schedule**

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

**TS(T₁)=1**

**TS(T₂)=2**
MVCC – EXAMPLE #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>

**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>

**Schedule**

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

- $T_1$
  - BEGIN
  - R(A)
  - W(A)
  - COMMIT

- $T_2$
  - BEGIN
  - R(A)
  - W(A)
  - COMMIT

**TS(T1)=1**

**TS(T2)=2**
**MVCC – EXAMPLE #2**

**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>1</td>
</tr>
<tr>
<td>A₁</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>T₂</td>
<td>Active</td>
</tr>
</tbody>
</table>

**Type Status Table**

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

**Txn Status Table**

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

**Txn Schedule**

<table>
<thead>
<tr>
<th>Time</th>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T₁</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
<tr>
<td></td>
<td>T₂</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

**MVCC Example**

- **TS(T₁) = 1**
- **TS(T₂) = 2**

**T₂ has to stall until T₁ commits.**

**MVCC – EXAMPLE #2**

<table>
<thead>
<tr>
<th>Transaction Status Table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TxnId</strong></td>
</tr>
<tr>
<td>$T_1$</td>
</tr>
<tr>
<td>$T_2$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Version</strong></td>
</tr>
<tr>
<td>$A_0$</td>
</tr>
<tr>
<td>$A_1$</td>
</tr>
</tbody>
</table>

**Schedule**

$T_1$ starts at time 0 and commits at time 1.

$T_2$ starts at time 1 and commits at time 2.

$TS(T_1) = T_1$ and $TS(T_2) = T_2$.

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

$TS(T_1) = 1$ and $TS(T_2) = 2$. 

**MVCC**
**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>Committed</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>1</td>
</tr>
<tr>
<td>$A_1$</td>
<td>456</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

**Schedule**

- $T_1$
  - BEGIN
  - R(A)
  - W(A)
  - R(A)
  - COMMIT
- $T_2$
  - BEGIN
  - R(A)
  - W(A)
  - COMMIT

$TS(T_1)=1$

$TS(T_2)=2$
**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>Committed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Active</td>
</tr>
</tbody>
</table>

### Schedule

- **$T_1$**
  - BEGIN
  - R(A)
  - W(A)
  - COMMIT

- **$T_2$**
  - 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>$A_0$</td>
<td>123</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$A_1$</td>
<td>456</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$A_2$</td>
<td>789</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>

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

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

Now $T_2$ can create the new version.
MVCC is more than just a concurrency control protocol. It completely affects how the DBMS manages transactions and the database.
MVCC DESIGN DECISIONS

Concurrent 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.
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 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.

### Main Table

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

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

### Main Table

<table>
<thead>
<tr>
<th>VERSION</th>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>
<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.
On every update, copy the current version to the time-travel table. Update pointers.

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

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

Txns can recreate old versions by applying the delta in reverse order.
GARBAGE COLLECTION

The DBMS needs to remove **reclaimable** physical versions from the database over time.

→ No active txn in the DBMS can "see" that version (SI).
→ The version was created by an aborted txn.

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.
**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.

**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.

**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.
**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.

---

**TUPLE-LEVEL GC**

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

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

INDEX

GET(A)
**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.

**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...
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

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
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 → ADDRESS

Physical Address

A_{100} → A_{99} → A_{98} → A_{97}

Append-Only
Newest-to-Oldest
## MVCC IMPLEMENTATIONS

<table>
<thead>
<tr>
<th>Database</th>
<th>Protocol</th>
<th>Version Storage</th>
<th>Garbage Collection</th>
<th>Indexes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle</td>
<td>MV2PL</td>
<td>Delta</td>
<td>Vacuum</td>
<td>Logical</td>
</tr>
<tr>
<td>Postgres</td>
<td>MV-2PL/MV-T0</td>
<td>Append-Only</td>
<td>Vacuum</td>
<td>Physical</td>
</tr>
<tr>
<td>MySQL-InnoDB</td>
<td>MV-2PL</td>
<td>Delta</td>
<td>Vacuum</td>
<td>Logical</td>
</tr>
<tr>
<td>HYRISE</td>
<td>MV-0CC</td>
<td>Append-Only</td>
<td>–</td>
<td>Physical</td>
</tr>
<tr>
<td>Hekaton</td>
<td>MV-0CC</td>
<td>Append-Only</td>
<td>Cooperative</td>
<td>Physical</td>
</tr>
<tr>
<td>MemSQL</td>
<td>MV-0CC</td>
<td>Append-Only</td>
<td>Vacuum</td>
<td>Physical</td>
</tr>
<tr>
<td>SAP HANA</td>
<td>MV-2PL</td>
<td>Time-travel</td>
<td>Hybrid</td>
<td>Logical</td>
</tr>
<tr>
<td>NuoDB</td>
<td>MV-2PL</td>
<td>Append-Only</td>
<td>Vacuum</td>
<td>Logical</td>
</tr>
<tr>
<td>HyPer</td>
<td>MV-0CC</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

Logging & Recovery