Intro to Database Systems (15-445/645)

18 Multi-Version Concurrency Control

Carnegie Mellon University

SPRING 2023

Charlie Garrod
ADMINISTRIVIA

Project 3 ongoing
→ Due Sunday, April 9th at 11:59 p.m.

Homework 4 released today
→ Due Friday, April 7th at 11:59 p.m.

Final exam Monday, May 1st, 8:30 – 11:30 a.m.
LAST TIME: TIMESTAMP ORDERING

- Basic timestamp ordering
- Optimistic concurrency control
- The phantom problem
  - Re-execute scans
  - Predicate locking
  - Index locking schemes
    - Key-value locks
    - Gap locks
    - Key-range locks
    - Hierarchical locking
TODAY’S PLAN

Isolation levels
Multi-version concurrency control
WEAKER LEVELS OF ISOLATION

Serializability is useful because it allows programmers to ignore concurrency issues.

But enforcing it may allow too little concurrency and limit performance.

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

Controls the extent that a txn is exposed to the actions of other concurrent txns.

Provides for greater concurrency at the cost of exposing txns to uncommitted changes:
→ Dirty Reads
→ Unrepeatable Reads
→ Phantom Reads
ISOLATION LEVELS

**SERIALIZABLE**: No phantoms, all reads repeatable, no dirty reads.

**REPEATABLE READS**: Phantoms may happen.

**READ COMMITTED**: Phantoms and unrepeatable reads may happen.

**READ UNCOMMITTED**: All of them may happen.
SERIALIZABLE: No phantoms, all reads repeatable, no dirty reads.

REPEATABLE READS: Phantoms may happen.

READ COMMITTED: Phantoms and unrepeatable reads may happen.

READ UNCOMMITTED: All of them may happen.

Isolation (Low → High)
## ISOLATION LEVELS

<table>
<thead>
<tr>
<th></th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
<th>Phantom</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SERIALIZABLE</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>REPEATABLE READ</strong></td>
<td>No</td>
<td>No</td>
<td>Maybe</td>
</tr>
<tr>
<td><strong>READ COMMITTED</strong></td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td><strong>READ UNCOMMITTED</strong></td>
<td>Maybe</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
</tbody>
</table>
ISOLATION LEVELS

**SERIALIZABLE**: Obtain all locks first; plus index locks, plus strict 2PL.

**REPEATABLE READS**: Same as above, but no index locks.

**READ COMMITTED**: Same as above, but S locks are released immediately.

**READ UNCOMMITTED**: Same as above but allows dirty reads (no S locks).
SQL-92 ISOLATION LEVELS

You set a txn's isolation level before you execute any queries in that txn.

Not all DBMS support all isolation levels in all execution scenarios → Replicated Environments

The default depends on implementation…

```
SET TRANSACTION ISOLATION LEVEL <isolation-level>;
```

```
BEGIN TRANSACTION ISOLATION LEVEL <isolation-level>;
```
## ISOLATION LEVELS

<table>
<thead>
<tr>
<th>Database</th>
<th>Default</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actian Ingres</td>
<td>SERIALIZABLE</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>IBM DB2</td>
<td>CURSOR STABILITY</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>CockroachDB</td>
<td>SERIALIZABLE</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>Google Spanner</td>
<td>STRICT SERIALIZABLE</td>
<td>STRICT SERIALIZABLE</td>
</tr>
<tr>
<td>MSFT SQL Server</td>
<td>READ COMMITTED</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>MySQL</td>
<td>REPEATABLE READS</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>Oracle</td>
<td>READ COMMITTED</td>
<td>SNAPSHOT ISOLATION</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>READ COMMITTED</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>SAP HANA</td>
<td>READ COMMITTED</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>VoltDB</td>
<td>SERIALIZABLE</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>YugaByte</td>
<td>SNAPSHOT ISOLATION</td>
<td>SERIALIZABLE</td>
</tr>
</tbody>
</table>
DATABASE ADMIN SURVEY

What isolation level do transactions execute at on this DBMS?

- Read Uncommitted: 10 None, 12 Few, 8 Most, 4 All
- Read Committed: 2 None, 6 Few, 26 Most, 22 All
- Cursor Stability: 12 None, 10 Few, 1 Most, 1 All
- Repeatable Read: 10 None, 12 Few, 5 Most, 2 All
- Snapshot Isolation: 11 None, 3 Few, 3 Most, 5 All
- Serializable: 8 None, 11 Few, 2 Most, 0 All
SQL-92 ACCESS MODES

You can provide hints to the DBMS about whether a txn will modify the database during its lifetime.

Only two possible modes:
→ **READ WRITE** (Default)
→ **READ ONLY**

Not all DBMSs will optimize execution if you set a txn to in **READ ONLY** mode.

```
SET TRANSACTION <access-mode>;
BEGIN TRANSACTION <access-mode>;
```
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 thetxn 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
→ Txn $T$ can see values written by txns that committed before $T$’s timestamp

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

**Schedule**

1. **T1**
   - **BEGIN**
   - **R(A)**
   - **R(A)**
   - **COMMIT**

2. **T2**
   - **BEGIN**
   - **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>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>

**Comments**

- **T₁ reads version A₀.**
- **T₂ creates version A₁ and sets A₀ End-TS.**
- 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>2</td>
</tr>
<tr>
<td>A₂</td>
<td>789</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>

**Txn Status Table**

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

**Notes:**

- **T₁ reads version A₁ because T₁ has not committed yet.**
- **T₂ reads version A₀ because T₁ has not committed yet.**
- **Now T₂ can create the new version.**
SNAPSHOT ISOLATION (SI)

When a txn starts, it sees a consistent snapshot of the database
→ Can see all data committed before that txn started
→ No torn writes from active txns.
→ If two txns update the same object, then first writer wins.

Snapshot isolation is not the same as serializable. It is susceptible to the **Write Skew Anomaly**.
WRITE SKEW ANOMALY

A possible outcome with MVCC:

Both transactions read the same snapshot

Both transactions commit because there is no write conflict
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 write a 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 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.
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></th>
<th>VALUE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A3</td>
<td>$333</td>
<td></td>
</tr>
<tr>
<td>B0</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>A1</td>
<td>(VALUE→$111)</td>
<td>Ø</td>
</tr>
<tr>
<td>A2</td>
<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.

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.
**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 version chain. Only works with O2N.
TRANSACTION-LEVEL GC

Each txn keeps track of its read/write set. On commit/abort, the txn provides this information to a centralized vacuum worker.

The DBMS periodically determines when all versions created by a finished txn are no longer visible.
**TRANSACTION-LEVEL GC**

**Txn #1**

- **BEGIN @ 10**
- **UPDATE(A)**
- **UPDATE(B)**
- **COMMIT @ 15**

**Old Versions**

- A₂
- B₆

**Vacuum**

- TS<10

**Table**

<table>
<thead>
<tr>
<th></th>
<th>BEGIN-TS</th>
<th>END-TS</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₂</td>
<td>1</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>B₆</td>
<td>8</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>A₃</td>
<td>10</td>
<td>∞</td>
<td>-</td>
</tr>
<tr>
<td>B₇</td>
<td>10</td>
<td>∞</td>
<td>-</td>
</tr>
</tbody>
</table>
INDEX MANAGEMENT

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…
INDEX MANAGEMENT

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

GET(A) → PRIMARY INDEX

SECONDARY INDEX

GET(A)

TupleId → Addresses

Record Id

Record Id

A₄ → A₃ → A₂ → A₁

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

**Txn #1**
BEGIN @ 10
READ(A)  READ(A)

**Txn #2**
BEGIN @ 20
COMMIT @ 25
UPDATE(A)
DELETE(A)

**Txn #3**
BEGIN @ 30
INSERT(A)

---

**Index**

**Table:**
- A1: BEGIN-TS 1, END-TS 20, POINTER
- X: BEGIN-TS 20, END-TS 20, POINTER Ø
- A1: BEGIN-TS 30, END-TS ∞, POINTER Ø
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-T0</td>
<td>Append-Only</td>
<td>Vacuum</td>
</tr>
<tr>
<td>MySQL-InnoDB</td>
<td>MV-2PL</td>
<td>Delta</td>
<td>Vacuum</td>
</tr>
<tr>
<td>HYRISE</td>
<td>MV-OCC</td>
<td>Append-Only</td>
<td>–</td>
</tr>
<tr>
<td>Hekaton</td>
<td>MV-OCC</td>
<td>Append-Only</td>
<td>Cooperative</td>
</tr>
<tr>
<td>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>CockroachDB</td>
<td>MV-2PL</td>
<td>Delta (LSM)</td>
<td>Compaction</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 and recovery!