Lecture #19: Multi-Version Concurrency Control

15-445/645 Database Systems (Fall 2018)
https://15445.courses.cs.cmu.edu/fall2018/
Carnegie Mellon University
Prof. Andy Pavlo

1 Multi-Version Concurrency Control

This is a larger concept than just a concurrency control protocol. It involves all aspect of the DBMS's design and implementation. MVCC is the most widely used scheme in DBMS. It is now used in almost every new DBMS implemented in last 10 years. Even (NoSQL) systems that do not support multi-statement transactions use it.

The DBMS maintains multiple physical versions of a single logical object in the database.

- When a transaction writes to an object, the DBMS creates a new version of that object.
- When a transaction reads an object, it reads the newest version that existed when the transaction started.

History of MVCC

Protocol was first proposed in 1978 MIT PhD dissertation.

First implementation was InterBase (now Firebird). Implemented by Jim Starkey, co-founder of NuoDB.

Key Properties

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

Read-only transactions can read a consistent **snapshot** without acquiring locks. Timestamps are used to determine visibility.

Easily support *time-travel queries* where the DBMS can execute on a point-in-time snapshot.

There are four important MVCC design decisions:

1. Concurrency Control Protocol (T/O, OCC, 2PL, etc).
2. Version Storage
3. Garbage collection
4. Index Management

2 Version Storage

This how the DBMS will store the different physical versions of a logical object.

The DBMS uses the tuple's pointer field to create a **version chain** per logical tuple. This allows the DBMS to find the version that is visible to a particular transaction at runtime. Indexes always point to the head of the chain. A thread traverses chain until you find the version thats visible to you. Different storage schemes determine where/what to store for each version.

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

- **Oldest-To-Newest (O2N):** Append new version to end of chain, look-ups require entire chain traversal.
- **Newest-To-Oldest (N2O):** Head of chain is newest, look-ups are quick, but indexes need to be updated every version.

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.

3 Garbage Collection

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

Approach #1: Tuple Level Garbage Collection – Find old versions by examining tuples directly

- **Background Vacuuming**: Separate threads periodically scan the table and look for reclaimable versions, works with any version storage scheme.
- **Cooperative Cleaning**: Worker threads identify reclaimable versions as they traverse version chain. Only works with O2N.

Approach #2: Transaction Level – Each transaction keeps track of its own read/write set. When a transaction completes, the garbage collector can use that to identify what tuples to reclaim. The DBMS determines when all versions created by a finished transaction are no longer visible.

4 Index Management

All primary key (pkey) indexes always 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 transaction updates a pkey attribute(s), then this is treated as a `DELETE` followed by an `INSERT`.

Managing secondary indexes is more complicated:

- **Approach #1: Logical Pointers** – Use a fixed identifier per tuple that does not change. Requires an extra indirection layer that maps the logical id to the physical location of the tuple (Primary Key vs Tuple ID).
- **Approach #2: Physical Pointers** – Use the physical address to the version chain head