Lecture #05: Multi-Version Concurrency Control (Garbage Collection)

15-721 Advanced Database Systems (Spring 2020)
https://15721.courses.cs.cmu.edu/spring2020/
Carnegie Mellon University
Prof. Andy Pavlo

1 Introduction

MVCC maintains multiple physical versions of a single logical object in the database. Over time, old versions will become invisible to active transactions under snapshot isolation. These are *reclaimable* versions that the DBMS must remove in order to reclaim memory.

Up until now, we have assumed that transactions (OLTP) will complete in a short amount of time. This means that the lifetime of an obsolete version is short as well. But HTAP workloads may have long running queries that access old snapshots. Such queries block the previous garbage collection methods (e.g., tuple-level, transaction-level) that we have discussed.

The DBMS can incur several problems if it is unable to clean up old versions:

- Increased Memory Usage
- Memory Allocation Contention
- Longer Version Chains
- Garbage Collection CPU Spikes
- Poor Time-based Version Locality

MVCC Deletes

When the DBMS performs a delete operation, the system will remove the tuple logically removed but it is still physically available. The DBMS *physically* deletes a tuple from the database only when all versions of a logically deleted tuple are not visible. We need a way to denote that a tuple has been logically deleted at some point in time.

1. **Deleted Flag**: Maintain a flag to indicate that the logical tuple has been deleted after the newest physical version. It can either be in tuple header or a separate column. This is the most common method.
2. **Tombstone Flag**: Create an empty physical version to indicate that a logical tuple is deleted. To reduce the overhead of creating a full tuple (i.e., with all attributes) in the append-only and time-travel version storage approaches, the DBMS can use a separate pool for tombstone tuples with only a special bit pattern in version chain pointer.

Indexes with MVCC Tables

Most MVCC DBMS do not store version information about tuples in indexes with their keys. An exception is index-organized tables like MySQL's InnoDB.

Every index must support duplicate keys from different snapshots because the same key may point to different logical tuples in different snapshots. Therefore, the DBMS need to use additional execution logic to

perform conditional inserts for primary key / unique indexes. These conditional inserts need to be atomic as well. The DBMS's execution engine also may get back multiple entries for a single fetch; it then has to follow the pointers to find the proper physical version.

2 Garbage Collection: Design Decisions

There are four design decisions to make when designing the garbage collection mechanism for an MVCC DBMS [1, 2].

Index Clean-up

The DBMS must remove a tuples' keys from indexes when their corresponding versions are no longer visible to active transactions. To achieve this, the DBMS maintains an internal log entry that can track the transaction's modifications to individual indexes to support GC of older versions on commit and removal modifications on abort.

Version Tracking

The version tracking protocol determines how the DBMS discovers reclaimable versions.

- **Tuple-level**: The DBMS's GC component is responsible for finding old versions by examining tuples directly. The system can use either *Background Vacuuming* or *Cooperative Cleaning* to locate old tuples.
- **Transaction-level**: Each transaction keeps track their old versions in thread-local storage. This means that the DBMS does not have to scan tuples to determine visibility.

Granularity

Granularity determines how the DBMS should internally organize the expired versions that it needs to check to determine whether they are reclaimable. Different granularity levels balance trade-off between the ability to reclaim versions sooner versus computational overhead.

- **Single Version:** The DBMS tracks the visibility of individual versions and reclaims them separately. This approach provides more fine-grained control, but has higher overhead.
- **Group Version:** The DBMS organizes versions into groups and reclaim all of them together when the newest tuple in that group is no longer visible to any active transaction. This grouping reduces overhead, but may delay reclamation.
- **Tables:** Reclaim all versions from a table if the DBMS determines that active transactions will never access it. This is a special case scenario that requires transactions to execute using either stored procedures or prepared statements since it requires the DBMS knowing what tables a transaction will access in advance.

Comparison Unit

DBMS needs a way to determine whether version(s) are reclaimable. Examining the list of active transactions and reclaimable versions should be latch-free; we want this process to be as efficient as possible to prevent new transactions from committing. As a result, the reclaimable checks might generate false negatives, but this is allowable as long as the versions are eventually reclaimed.

- **Timestamp:** Use a global minimum timestamp to determine whether versions are safe to reclaim. This approach is the easiest to implement and execute.
- **Interval:** The DBMS identifies ranges of timestamps that are not visible to any active transaction. The lower-bound of this range may not be the lowest timestamp of any active transaction, but range is

not visible under snapshot isolation. This approach was first introduced by SAP HANA [2], but it is also used in HyPer [1].

3 Block Compaction

If the application deletes a tuple, then the slots occupied by that tuple's versions are available to store new data once the versions' storage is reclaimed. Ideally the DBMS should try to reuse those slots to conserve memory. The DBMS also need to deal with the case where the application deletes a bunch of tuples in a short amount of time, which in turn generates a large amount of potentially reusable space.

MVCC Deleted Tuples

The DBMS needs to determine what to do with empty slots after it reclaims tuple versions.

- **Reuse Slot:** The DBMS allow workers to insert new tuples in the empty slots. This approach is an obvious choice for append-only storage since there is no distinction between versions. The problem, however, is that it destroys temporal locality of tuples in delta storage.
- **Leave Slot Unoccupied:** With this approach, workers can only insert new tuples in slots that were not previously occupied. This ensures that tuples in the same block are inserted into the database at around the same time. Overtime the DBMS will need to perform a background compaction step to combine together less-than-full blocks of data

Block Compaction

A mechanism to reuse empty holes in our database is to consolidate less-than-full blocks into fewer blocks and then returning memory to the OS. The DBMS should move data using `DELETE+ INSERT` to ensure transactional guarantees during consolidation.

Ideally the DBMS will want to store tuples that are likely to be accessed together within a window of time together in the same block. This will make operations on blocks (e.g., compression) easier to execute because tuples that are unlikely to be updated will be within the same block.

There are different approaches for identifying what blocks to compact:

1. **Time Since Last Update:** Leverage the `BEGIN-TS` field in each tuple to determine when the version was created.
2. **Time Since Last Access:** Track the timestamp of every read access to a tuple (e.g., the `READ-TS` in the basic T/O concurrency control protocol). This expensive to maintain because now every read operation has to perform a write.
3. **Application-level Semantics:** The DBMS determines how tuples from the same table are related to each other according to some higher-level construct. This is difficult for the DBMS to figure out automatically unless their are schema hints available (e.g., foreign keys).

Truncate Operation

During block compaction process, the DBMS might want to free up large amount of space with small overhead. `TRUNCATE` removes all tuples in a table, which is the same as `DELETE` without a `WHERE` clause.

The fastest way to execute `TRUNCATE` is to drop the table and then create it again. This invalidates all versions within that table. We would not need to track the visibility of individual tuples. The GC will free all memory when there are no active transactions that exist before the drop operation. If the catalog is transactional, then this is easy to do since all the operations on metadata are atomic.

References

[1] J. Böttcher, V. Leis, T. Neumann, and A. Kemper. Scalable garbage collection for in-memory mvcc systems. *Proc. VLDB Endow.*, 13(2):128–141, Oct. 2019. doi: 10.14778/3364324.3364328.

[2] J. Lee, H. Shin, C. G. Park, S. Ko, J. Noh, Y. Chuh, W. Stephan, and W.-S. Han. Hybrid garbage collection for multi-version concurrency control in sap hana. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1307–1318, 2016. doi: 10.1145/2882903.2903734.