

# MVCC GARBAGE COLLECTION

---

## Definition

DBMS needs to remove reclaimable **physical versions**

- No active txn in the DBMS can see that version.
- The version was created by aborted txns.

## Problems:

In OLTP the queries/txn will complete in a short amount of time -> lifetime of an obsolete version is short as well.

**HTAP** may have long running queries that access old snapshots. Such queries block the traditional GC.

- Increased Memory Usage
- Memory Allocator Contention
- Longer Version chains
- Garbage Collector CPU Spikes
- Poor time-based version locality

## MVCC DELETES

When perform **delete** operation, the system will remove the tuple logically. But it is still physically available.

DBMS **physically deletes a tuple only when all versions are deleted logically**

How to denote a tuple has been logically deleted?

- **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

- **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.

## GC DESIGN DECISIONS

- 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.

How DBMS discover reclaimable tuples depends on the version tracking protocol

### Approach 1: Tuple-level

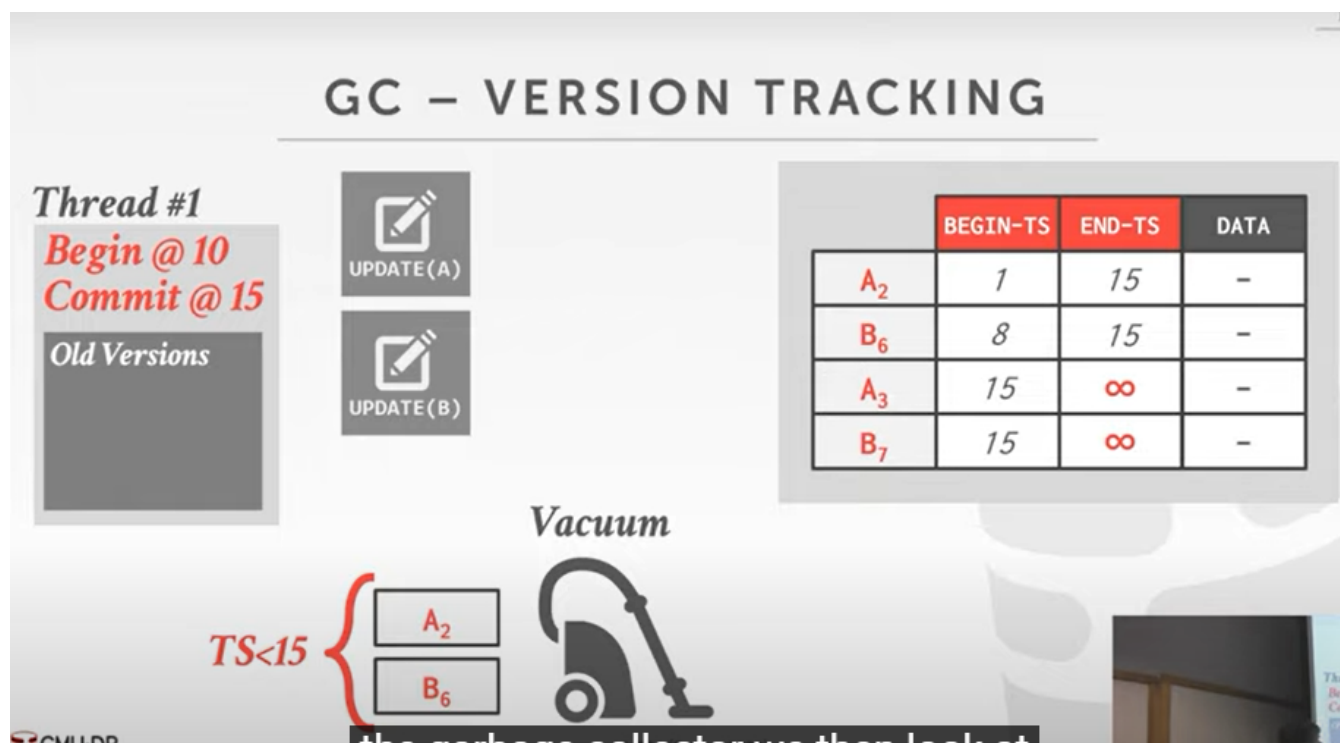
1. Find old versions by examining tuples directly.

2. **Background Vacuuming** vs **Cooperative Cleaning**

### Approach 2: Transaction-level

1. Each transaction keeps track their old versions in thread-local storage

2. does not have to scan tuples



The old versions are stored in thread local storage. When txn commits, pass the old versions to GC thread.

- GC-FREQUENCY

There's trade-off between:

1. Too frequent -> waste cycles and slow down txns.
2. Too infrequent -> cause storage overhead to increase and increase the length of version chains

### Approach1: Periodically

- Run the GC at fixed intervals or when some threshold has been met (like epoch, memory limits, like in JVM the heap size reach to some portion it will kick off GC)
- Some DBMSs can adjust this interval based on load.

### Approach2: Continuously

- Run GC as part of the regular txn processing, like GC everytime you commit a txn or during query execution.
- GC-GRANULARITY

How should the DBMS internally organize the expired versions that it needs to check to determine whether they are reclaimable.

**Trade-off between** ability to reclaim versions sooner and computational overhead.

## GC – GRANULARITY

### Approach #1: Single Version

- Track the visibility of individual versions and reclaim them separately.
- More fine-grained control, but higher overhead.

### Approach #2: Group Version

- Organize versions into groups and reclaim all of them together.
- Less overhead but may delay reclamations.

### Approach 3:

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 need to determine whether versions 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.


- **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.

## GC – COMPARISON UNIT


**Thread #1**  
*Begin @ 10*

  
READ(A)

**Thread #2**  
*Begin @ 20*  
*Commit @ 25*

  
UPDATE(A)


**Thread #3**  
*Begin @ 30*  
*Commit @ 35*

  
UPDATE(A)

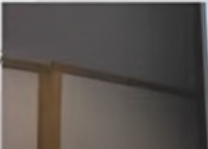
	BEGIN-TS	END-TS	DATA
$A_1$	1	25	–
$A_2$	25	35	–
$A_3$	35	$\infty$	–

**Timestamp**  
→ GC cannot reclaim  $A_2$  because the lowest active txn TS (10) is less than END-TS.

**Interval**  
→ GC can reclaim  $A_2$  because no active txn intersects the interval [25,35].



15-721 (Spring 2020)

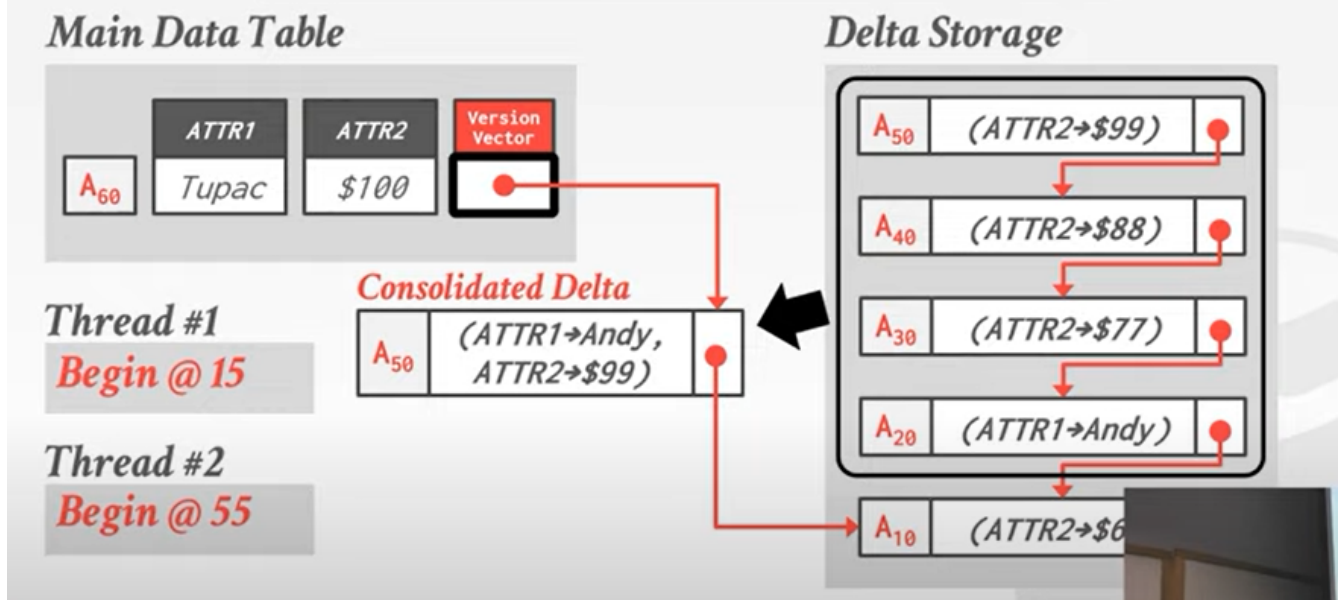


If using timestamp approach, only transaction with END-TS  $\leq$  current txn TS can be reclaimed.

If using Interval approach, if active txn TS not lie between tuple's BEGIN-TS and END-TS, then the tuple can be reclaimed.

What if you do **delta-storage**? There's no TS info in the tuple. GC in delta storage usually needs consolidate the tuple data.

# GC – INTERVAL DELTA RECORDS



## 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.

说白了就是tuple删除之后原来位置是否能够重用？

- **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。重用会破坏局部性

- **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。

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 (物以类聚 不常用的数据放在同一个block)

## Approached to identify blocks that needed to be compacted

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 there are schema hints available (e.g., **foreign keys**).