Acheron: Persisting Tombstones in LSM Engines

Zichen Zhu
Boston University
zczhu@bu.edu

Subhadeep Sarkar
Boston University
ssarkar1@bu.edu

Manos Athanassoulis
Boston University
mathan@bu.edu

ABSTRACT

Modern NoSQL storage engines frequently employ log-structured merge (LSM) trees as their core data structures because they offer high ingestion rates and low latency for query processing. Client writes are captured in memory first and are gradually merged on disk in a level-wise manner. While this *out-of-place* paradigm sustains fast ingestion rates, it implements delete operations via inserting *tombstones* which *logically invalidate older entries*. Thus, obsolete data cannot be removed instantly and may be retained for an arbitrarily long time. Therefore, out-of-place deletion in LSM trees may, on the one hand, violate data privacy regulations (e.g., *the right to be forgotten* in EU's GDPR, *right to delete* in California's CCPA and CPRA), and on the other hand, it hurts performance.

In this paper, we develop `Acheron`, which demonstrates the performance implications of out-of-place deletes and how our method achieves timely persistent deletes. We integrate both prior state-of-the-art compaction policies and our recently presented method, FADE, into `Acheron` and visualize the life cycle of tombstones in LSM trees. Using the `Acheron` visualization, users can observe that the state of the art does not provide guarantees on when obsolete entries can be physically removed and also observe that FADE can achieve timely persistent deletes without full tree compaction. Users can further customize the workload, LSM tuning knobs, and disk parameters to investigate their impact on tombstones and performance. This demonstration provides key insights into the impact of tombstones on LSM-interested researchers and practitioners.

CCS CONCEPTS

• **Information systems** → **Point lookups**; **Unidimensional range search**; **Record and block layout**; **Key-value stores**; • **Security and privacy** → *Social aspects of security and privacy*; *Privacy protections.*

KEYWORDS

LSM trees; data deletion; privacy

ACM Reference Format:
Zichen Zhu, Subhadeep Sarkar, and Manos Athanassoulis. 2023. `Acheron`: Persisting Tombstones in LSM Engines. In *Companion of the 2023 International Conference on Management of Data (SIGMOD-Companion '23), June 18–23, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3555041.3589719

SIGMOD-Companion '23, June 18–23, 2023, Seattle, WA, USA

ACM ISBN 978-1-4503-9507-6/23/06…$15.00
https://doi.org/10.1145/3555041.3589719

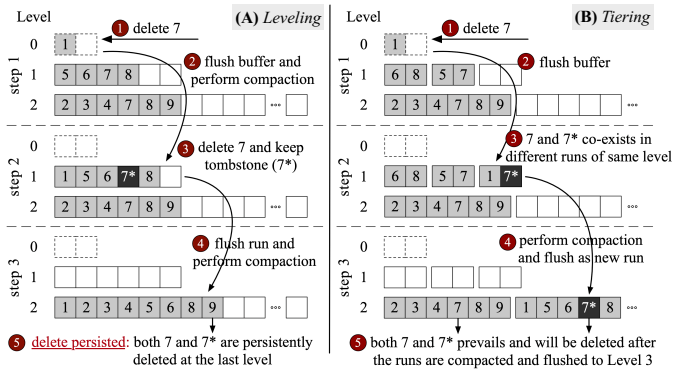

Figure 1: In an LSM tree, for every tombstone, there can be (A) one matching entry per level for leveling or (B) one matching entry per tier per level.

1 INTRODUCTION

LSM-based Key-Value Stores. Data-intensive applications (e.g., Internet-of-things, edge computing, 5G communications, and autonomous vehicles) generate a huge amount of data at unprecedented rates. Several modern NoSQL storage engines, including LevelDB, RocksDB, WiredTiger, and Cassandra, rely on Log-Structured Merge (LSM) trees [1–3] to sustain efficient ingestion for OLTP workloads. Many relational database systems (e.g., MyRocks and CockroachDB) also use LSM trees as the basic data structure to build their key-value storage engines. In LSM trees, incoming entries are batched in an in-memory write buffer, and once the buffer is full, it is flushed to storage as an immutable sorted run. When the number of accumulated similarly sized runs exceeds a predefined threshold, they are merge-sorted to form a larger immutable run (this process is also termed *compaction* [3, 5, 7]). The immutability of sorted runs does not allow for in-place updates/deletes; rather, all updates lead to sequential writes during flushing and compaction.

Deletes in LSM Trees. However, the immutability of sorted runs comes at the cost of *out-of-place* deletes. Every delete is implemented by inserting a *tombstone*, as shown in Figure 1. Within the memory buffer, a tombstone eagerly deletes any older matching entries, and it is maintained to invalidate further matching entries in the tree. When the buffer is full, all the entries (including tombstones) in the buffer are flushed to disk as a file. As more data is ingested, this file will be involved in compactions, during which older entries with the same key will be *physically* discarded.

Problems. When a tombstone is inserted, older entries may exist in deeper levels, and thus, we can only safely remove the tombstone when it reaches the last level of the LSM tree through iterative compactions. Tombstones and obsolete entries may co-exist for an arbitrarily long time until all tombstones are compacted to the last level, which has the following implications:

(a) Tombstones and obsolete entries increase *space amplification*.
(b) Before obsolete entries are physically discarded, they are likely to be involved in other compactions, which may lead to increased *write amplification*.
(c) Invalid entries may pollute the indexes and filters, and thus hurt range and point query performance.
(d) Out-of-place deletion does not guarantee when older records will be physically removed, which may *violate data privacy regulations* [4] such as *the right to be forgotten* in EU's GDPR, *right to delete* in California's CCPA and CPRA.

Our approach. The approach [6] that we demonstrate proposes a new compaction policy that prioritizes files containing older tombstones. Compared to full tree compaction, this approach retains the LSM tree's benefits of an amortized merging cost with no latency spikes and has less write amplification and shorter write stalls. Specifically, compactions are more eagerly triggered based on the maximum age of tombstones for each level, so that tombstones can reach the last level within a user-defined threshold.

In this paper, we develop `Acheron`[1], which demonstrates the performance implications of tombstones, and how our method achieves timely persistent deletes. To our knowledge, there are no previous systems or tools that reveal tombstones' impact on new data privacy regulations and system performance in LSM trees. LSM-interested researchers and practitioners can benefit from `Acheron` to gain more useful insights into this impact.

Demonstration. Participants in the conference can interact with `Acheron` to compare and analyze the life cycle of tombstones in LSM trees under different scenarios. The visual interface allows participants to see (i) when tombstones move from shallower levels to deeper levels and are physically deleted in the last level, (ii) how our method purges obsolete data compared to other compaction policies, and (iii) how tombstones affect the system performance. `Acheron` also offers the opportunity for participants to vary the workload composition and LSM tunings (e.g., delete percentage, size ratio, storage read/write speed) to explore more scenarios. The demo is available at https://disc-projects.bu.edu/acheron/research.html.

2 ENABLING TIMELY PERSISTENT DELETES

To enable timely persistent deletes, in prior work, we introduced FADE, a new family of delete-aware compaction policies. FADE (short for *FAst DEletes*) piggybacks the task of timely delete persistence to the LSM tree's compaction routine while retaining the LSM tree's benefit of amortized merging cost and predictable performance. `Acheron` taps into the key design of FADE and presents an interactive framework that highlights how users can navigate the performance-privacy trade-off between delete persistence and desired performance. In this section, we present the technical details of FADE and the metrics we use in `Acheron`.

2.1 FADE

Delete Persistence Threshold (DPT). FADE ensures that all the tombstones are persisted within a user-specified delete persistence threshold (DPT). DPT is formally defined as the worst-case time required, following the insertion of a tombstone, for the tree to be void of any entry (including tombstones) with a matching (older) key to that of the inserted tombstone. Typically, DPT is specified as part of the service level agreement (SLA) concerning data retention. The complete algorithm of FADE is presented in Algorithm 1. Compared to the state-of-the-art LSM compaction policy, FADE augments it in the following two aspects.

Compaction Trigger. In FADE, a compaction is triggered not only when a level is saturated but also when tombstones are close to expiration. To support the additional trigger, FADE maintains the age of the oldest tombstones per file and assigns every file a *time-to-live* (TTL). If TTL is fixed as DPT for all the files, all the expired tombstones/files in shallow levels can result in cascading compactions and, thus, high write amplification. Instead, FADE assigns a smaller TTL, d_i, for every file in level i such that $\sum_{i=1}^{L-1} d_i =$ DPT, where L is the number of levels in the current LSM tree. The allocation strategy for d_i proposed by FADE follows a geometric sequence with an increasing ratio T (the same as the size ratio) because the exponential assignment is coherent with the capacity in each level and thus leads to fewer concurrent compactions.

Algorithm 1: FADE

```
Input: delete persistence threshold (DPT); levels in tree (L_old); size ratio (T);
       size of memory buffer (M)
FADE():
begin
    L_new = getCurrentTreeLevel(), d_0 = 0
    if L_new > L_old then
        for i ∈ [1 : L_new] do
            d_i = d_{i-1} + DPT·(T − 1)/(T^L − 1) · T^{i−1}
    for i ∈ [1 : L_new] do
        csize(i) = 0, ttl_i = 0, cap_i = M · T^i
        for j ∈ [1 : getFileCountInLevel(i)] do
            csize(i) += size(j)
            if d_i < age_j then
                ttl_i++
        score[i] = csize(i)/cap_i + ttl_i
    compact_level = getLevelToCompact(score[])
    compact_file = getFileToCompact(compact_level)
    initiate compaction with compact_file
getLevelToCompact(score[])
begin
    c_level = score[0]
    for i ∈ [1 : L_new] do
        if score[i] > score[i − 1] then
            c_level = i
    return c_level
getFileToCompact(compact_level)
begin
    files = getFilesInLevel(compact_level)
    for i ∈ [1 : files.size() − 1] do
        if d_{compact_level} ≥ files[i].age then
            return files[i]
    sort files by overlapping ratio in an ascending way
    return files[0]
```

File Picking Policy. The state-of-the-art LSM engines use the picking policy of selecting the file with the smallest overlap to reduce the write amplification (while there are many other picking policies [7], this is the most common one and thus selected to compare with FADE in `Acheron`). However, in FADE, files with

[1] *Acheron* is the Greek mythological river used to transport souls to the underworld (persist tombstones).

Figure 2: The Acheron UI visualizes the life cycle of tombstones under different compaction policies.

expired TTL are prioritized over files with the least overlapping rate to enforce the timely physical deletion. In other words, compactions triggered by TTL expiration only pick expired files to compact, while compactions that are only triggered by saturation still pick the file with minimum overlapping ratio.

2.2 Demonstration Metrics

FADE ensures that all the tombstones are persisted by the time their lifetime reaches DPT by triggering more compactions, which naturally introduces a trade-off between tombstones and compactions. To better capture this trade-off, Acheron benchmarks the following metrics during the emulation.

Tombstone-related Metrics. After every flush and every compaction, Acheron records *the number of deletes*, *the number of existing tombstones*, *the number of expired tombstones* (with respect to the specified DPT), and *the maximum age of existing tombstones*.

Compaction Metrics. Compaction metrics include *the number of compactions*, *the average compaction (input) size*, *the average compaction latency*, and *the worst-case compaction latency*. These metrics are updated every time compaction finishes.

Performance Metrics. Many performance metrics can also be affected by extra tombstones and compactions. For example, more tombstones lead to more disk space, larger Bloom Filters (BFs), and indexes of the LSM tree, and thus Acheron also records *storage space* and *memory footprint* for auxiliary structures. On the other hand, extra compactions can significantly affect *the average ingest I/O cost* and *write amplification*, which is also tracked in Acheron.

3 THE ACHERON DEMONSTRATION

Overview. The overall structure of Acheron UI is summarized in Figure 2, which consists of an input panel, a progress bar control panel, an emulation panel, and a performance panel. Users interact with Acheron through the following operation flow: (i) specify the configuration in the input panel, (ii) click the "Play" button in the control panel (after which Acheron starts emulating the ingestion process), (iii) watch the animation of how tombstones propagate through iterative compactions in the emulation panel, (iv) compare the metrics presented in the performance panel.

Input and Control Panels. Users can customize the workload by specifying *the number of total ingests*, *the key size*, *the entry size*,

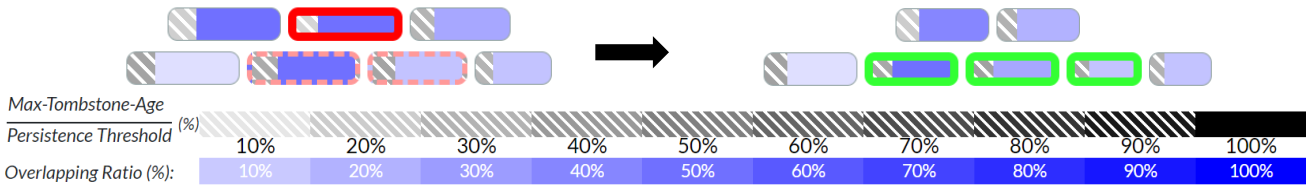

Figure 3: Acheron UI and visualizations. Each rectangle represents a file. During compaction, the file with a red solid border indicates that it is being selected to compact to the next level, while the files with a red dashed border overlap with the selected file. After the compaction finishes, newly generated files are marked with a solid green border.

and *the percentage of deletes* **(1)**. Acheron emulates a real workload by proportionally scaling down the number of ingests. In addition to the workload specification, Acheron allows users to configure the main memory parameters (i.e., *the memory size for MemTable*, *the bits per key for BFs*) **(2)**, *the size ratio* **(3)**, disk parameters like *read latency per I/O* and *write latency per I/O*) **(4)**, and *persistence threshold (DPT)* **(5)**. After setting the input, users can control the emulation progress via buttons *Play*, *Pause*, *Finish*, and even drag the progress bar in the control panel.

Emulation and Performance Panels. The emulation panel displays iterative compactions in three LSM trees with different compaction policies: FADE **(7)**, MinOverlappingRatio **(8)**, and Round-Robin **(9)**. To better highlight the difference, Acheron shows the compaction progress in detail, as shown in Figure 3. In the visualization, each rectangle represents a file (where the length is proportional to the file size). Additionally, the length of the gray/white striped part in a rectangle represents the proportion of tombstones in this file. The darkness of the striped part indicates the maximum tombstone age, and when the oldest tombstone expires, the color turns completely dark. The darkness of the blue part indicates the overlapping ratio of this file. The LSM tree structure is updated every flush and every compaction. For illustration, every flush is forced to be synced across different policies, but the actual time could differ. When watching the animated compaction process, users can also observe that the performance metrics and plots are updated in real-time**(10)**, with the x-axis representing the flushed data. If users want to investigate the impact of different DPT in FADE, they can also switch to the *DPT Analysis in FADE* mode **(11)**.

4 DEMONSTRATION SCENARIOS

The participants can fully interact with Acheron to understand the life cycle of tombstones, how the life cycle differs across different compaction policies, and also the impact of DPT along with twelve system metrics. We use the following three example scenarios to demonstrate possible interactions between users and Acheron.

Scenario 1: Visualize the Life Cycle of Tombstones Through Iterative Compactions. We consider the default setting: 10M 128-byte entries with 25% deletes are ingested into an LSM tree with 16MB buffer size, 10 bits per key for BFs, a 50-second DPT, and a size ratio of 2. After users click the *Play* button, they can see the whole tree construction progress, which includes how tombstones move from shallower levels to deeper levels, how tombstones age (the color gradually changes from white to gray), and persist (no gray strips/tombstones in the last level).

Scenario 2: Compare Compaction Policies for Deletes. When comparing different compaction policies, users can observe how tombstones expire (gray strips are replaced with pure black) under the MinOverlappingRatio and Round-Robin policies. In contrast, pure black tombstones never appear in FADE since DPT is applied as a hard TTL constraint. Besides, users can also observe the higher compaction cost in FADE via the compaction metrics since FADE achieves timely persistence by performing extra compactions.

Scenario 3: Examine the Performance Impact of Delete Timeliness. Users can switch to *DPT Analysis in FADE* mode to explore how different DPT affects performance. When playing with this mode, users can observe from the performance panel that the oldest tombstones strictly follow DPT configuration (tombstones never become older than the specified DPT) no matter what DPT users specify. Users can also examine the performance impact by comparing other metrics (e.g., #compactions and write amp).

5 CONCLUSION

In this paper, we introduce Acheron, which visualizes the life cycle of tombstones in LSM trees under different settings. Acheron can help users to understand why tombstones can stay for an arbitrarily long time under state-of-the-art compaction policy and how FADE achieves timely persistent deletes without full tree compaction. Acheron also allows users to explore the impact of DPT on tombstones and performance.

ACKNOWLEDGMENTS

This work was funded by NSF grants IIS-1850202 and IIS-2144547, a Facebook Faculty Research Award, and a Meta gift.

REFERENCES

[1] Chen Luo and Michael J. Carey. 2020. LSM-based Storage Techniques: A Survey. *The VLDB Journal* 29, 1 (2020), 393–418.
[2] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.
[3] Subhadeep Sarkar and Manos Athanassoulis. 2022. Dissecting, Designing, and Optimizing LSM-based Data Stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2489–2497.
[4] Subhadeep Sarkar and Manos Athanassoulis. 2022. Query Language Support for Timely Data Deletion. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 429–434.
[5] Subhadeep Sarkar, Kaijie Chen, Zichen Zhu, and Manos Athanassoulis. 2022. Compactionary: A Dictionary for LSM Compactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2429–2432.
[6] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A Tunable Delete-Aware LSM Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 893–908.
[7] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2021. Constructing and Analyzing the LSM Compaction Design Space. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2216–2229.