Query-driven compaction in LSM-trees

Shubham Kaushik
Boston University
USA

****
Boston University
USA

****
Boston University
USA

ABSTRACT

In modern data systems, particularly for write-intensive workloads, log-structured merge (LSM)[1, 2] trees have become the most widely used technique. The idea of out-of-place updates, which logically invalidate keys that may exists on multiple levels, helps LSM-trees excel for write heavy workloads. However, current LSM-tree designs do not capitalize on the sort-merge operations performed for the range queries, which leads to carrying out almost the same amount of work for each range query, even when they are same or overlapping. This inefficiency can result in the wastage of CPU cycles and unnecessary I/O operations in worst-case scenarios due to presence of logically invalid keys.

In this paper, we present a query-driven compaction strategy that removes the invalid (logically deleted) keys from the LSM-tree and flushes back the valid keys filtered by sort-merge performed during a range query. We conduct performance experiments with the help of a custom implementation in one of the popular LSM-based data stores, RocksDB.

ACM Reference Format:
Shubham Kaushik, ****, and ****. 2023. Query-driven compaction in LSM-trees. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Write heavy workloads with LSM. The rising research in the field of robotics and IoT devices, combined with the integration of artificial intelligence and machine learning, has resulted in the generation of vast amounts of data. This data often requires real-time processing and exhibits a write-heavy nature. The data stores like RocksDB and LevelDB have been designed to efficiently handle such workloads. These data stores rely on the technique of **log-structured merge (LSM)** trees, an efficient data structure tailored for managing write-heavy workloads. The fundamental concept underlying LSM trees is of out-of-place updates, which logically invalidate keys instead of performing in-place updates.

Compaction. The LSM trees are composed of multiple levels, each of which is a sorted run of key-value pairs. The compactions are performed to merge the sorted runs from the lower levels into the higher levels. The process is triggered when the size of a level exceeds a certain threshold. It helps in removing the stale data from LSM and making room for new data in lower levels.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Range queries. The LSM design is optimized for write-heavy workloads, but it also supports range queries. The range queries are performed by merging the sorted runs from multiple levels and filtering out the keys that have been logically invalided. This process is also called **sort-merge**, which is similar to the compaction.

Problem. When executing a range query, the sort-merge operation is performed on sorted runs from multiple levels, leading to the retrieval of both valid and invalid keys from higher levels. Consequently, more data is read than actually required. This situation is acceptable when performing a single range query for a specific range. However, when the same query or an overlapping one is executed repeatedly, a nearly identical amount of work is executed. Furthermore, when a compaction is triggered within that specific range, some of the invalid keys that were previously read, sorted and filtered during the range query are revisited. This results in redundant CPU cycles and I/O operations, where the same data bytes are read repetitively until reaching the last level.

1.1 Motivation

The repetition of work involving invalid keys can be effectively mitigated by redirecting valid keys, filtered through the sort-merge operation during a range query, back to the higher levels. This approach can be termed as **query-driven compaction**. By minimizing the presence of invalid keys within the LSM tree, the compaction process gains efficiency, subsequently leading to less number of I/O operations and a more optimal utilization of CPU cycles.

1.2 Problem Statement

The state-of-the-art LSM-based data stores performs range queries by retrieving multiple files from various levels and filter out invalid keys. Once the range query is complete, the work done via the sort-merge operation for query is discarded. The results are neither cached nor flushed back to the LSM tree except returning it to the application. This approach can give rise to two problems: (1) *Redundant Work*: when the same range query or an overlapping one is executed again, and (2) *Increased Write Amplification*: when a compaction is triggered for the files containing keys within the range of a previous range query. During these process, the system re-reads the invalid keys and subsequently either drops them or replaces them with new values. As a result, the same data bytes are read and written during both the range query and compaction processes, leading to higher read, write, and space amplification.

1.3 Contributions

In this paper, we present a query-driven compaction strategy that involves writing the valid keys back to the higher levels of the LSM tree. While this approach may slightly increase the flush write bytes during a range query, it substantially reduces the presence of invalid keys in the LSM tree. As a result, there is a notable reduction in read, write, and space amplification during both compaction processes

and future range queries. We have also implemented our approach in RocksDB, a popular LSM-based data store and conducted a series of experiments to evaluate its performance.

2 BACKGROUND

LSM-based data stores are designed to efficiently manage write-heavy workloads. The data is ingested through a memtable, an in-memory data structure that stores keys and their corresponding values. Once the memtable reaches its capacity, it is flushed to disk in the form of a sorted run. These sorted runs are stored across multiple levels in files that are sorted by key. The compactions are also triggered in the background when the size of a level exceeds a specific threshold.

Updates in LSM. Updates are executed in an out-of-place fashion, where new values are written to lower levels while retaining old values in higher levels.

Deletes in LSM. Deletions are performed by adding a special marker called a *tombstone*. Tombstones are utilized to filter out logically deleted keys during range queries or compactions. They also serve as a form of soft deletion for point queries. Tombstones are removed when they reach the lowest level of the LSM structure.

Range queries in LSM. LSM supports range queries by merging keys from multiple levels and filtering out invalid keys.

In the context of range queries, LSM-based data stores typically employ a straightforward approach: they create an iterator for each level and perform a K-way merge. This merge process involves pulling files sequentially or asynchronously from each level into memory and executing a sort-merge operation. The K-way merge eliminates the need to load all files from each level into memory simultaneously, thus reducing memory footprint.

Leveled Compaction. Leveled compaction uses "merge with" strategy, where each level is merged with next level, which is usually much larger as per the configured size ratio.

The compactions are triggered to move data from lower levels to higher levels. The process involves selecting a file based on a compaction score from the lower level using a configured compaction policy. This file is then merged with overlapping files from the higher level. Invalid keys are filtered out during this merge, and the valid keys are written to the higher level. This compaction process occurs only between adjacent levels. Once a file is compacted, it is removed from the lower level, and the new compacted file is written to the higher level. This process can repeat multiple times until the level size falls below a specified threshold.

3 PROBLEM

In the preceding background section, the compaction process (specifically, the Leveled compaction) was discussed, highlighting its occurrence between adjacent levels. Now, let's delve into a scenario involving an LSM tree with N levels and a set *size ratio* of 10. In this context, the compaction process selects files randomly from lower levels based on the configured compaction policy.

Let's consider a scenario where each level shares at least one common key across all levels. Now, envision the initiation of compaction from the lowest level (level 0) to the highest level, sequentially. In this process, a specific key would be read N times and written N times. This repetition arises as the key is initially read from level 0 and subsequently written to level 1, then read from level 1 and written to level 2, and so on. This simple illustration serves to demonstrate the concept of read and write amplification. However, it's important to recognize that in actuality, the compaction process is significantly more intricate, involving multiple files across multiple levels. As a result, the read and write amplification is notably higher in complex cases.

For a more tangible understanding, consider the ideal scenario where the *size ratio* is 10. In this case, to propagate a single byte from the i^{th} level to the $(i+1)^{th}$ level, a maximum of 11 bytes must be read. Applying this principle to the earlier example, with N levels each containing the same key, we effectively encounter the need to read and write 11 bytes a total of N times due to the presence of N levels sharing the common key.

4 SOLUTION NAME

Query-driven compaction. The solution to the problem of redundant work and increased write amplification is query-driven compaction. This approach involves writing the valid keys back to the higher levels of the LSM tree.

5 EVALUATION

6 RELATED WORK

7 CONCLUSION

REFERENCES

[1] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal navigable key-value store. In *SIGMOD Conference*, pages 79–94, New York, NY, USA, 2017. Association for Computing Machinery.

[2] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.