CS561 : Query-driven compaction in LSM-trees

Ye Tian
Boston University

Peiying Ye
Boston University

Li Xi
Boston University

ABSTRACT

Log-structured merge-trees (LSM-trees) are widely used for efficient and scalable storage of key-value entries. However, the current implementation of LSM-trees does not fully exploit the opportunities for optimization, which can result in suboptimal performance and wasted resources. Specifically, the current compaction method does not utilize the merge-sort results of range queries, leading to redundant operations. In this paper, we propose query-driven compaction algorithms for LSM-trees as an optimization to leverage the merge-sort results of range queries, thereby improving performance and efficiency. We implement the proposed solutions on top of storagelayoutemulation, a simplified LSM-trees and range queries project provided by Professor Subhadeep. Based on our work, we demonstrate its effectiveness in reducing the write and space amplification of compaction. Therefore, this work provides a foundation for further research on query-driven compaction as an optimization for LSM-trees.

1 INTRODUCTION

1.1 Motivation

Log-structured merge-trees (LSM-trees) are widely used data structures for persistent storage of key-value entries. They are particularly useful for write-intensive workloads, where they offer efficient insertion and fast lookup performance. LSM-trees store data on disk as immutable logs, also known as sorted sequence tables (SSTs), which are maintained in hierarchical levels of increasing capacity. This allows LSM-trees to scale efficiently to large data sets, while also optimizing write performance by minimizing disk I/O operations.

LSM-trees are prevalent in modern applications. They are used in a wide range of domains, including databases, file systems, search engines, and key-value stores. Their efficiency and scalability make them particularly useful for real-time applications and big data workloads, where fast write performance is crucial.

Despite their widespread adoption and numerous benefits, the current implementation of LSM-trees does not fully exploit the opportunities for optimization. Specifically, the current compaction method in LSM-trees does not take advantage of the sort-merge performed for range queries by writing the result to the tree. This leaves space for further improvement of the compaction.

1.2 Problem Statement

LSM-trees have been widely adopted due to their scalability and efficient write performance. However, the current implementation of LSM-trees does not fully exploit the opportunities for optimization. Specifically, while range queries merge all qualifying runs with an overlapping key range to identify and ignore older entries, the current compaction method in LSM-trees does not take advantage of the sort-merge performed for range queries by writing the result to the tree. Therefore, LSM-trees can benefit from query-driven compaction, which leverages the sort-merge results of range queries to further improve the performance and efficiency of LSM-trees.

The problem statement of this paper is to design query-driven compaction algorithms for LSM-trees that exploit the sort-merge results of range queries to reduce the number of sort-merges of compactions, thereby reducing the overall write and space amplification of LSM-tree. The proposed solutions should leverage the existing range query algorithms of LSM-trees and build upon the existing compaction process. The proposed solutions should be implemented on top of storagelayoutemulation, a simple version of open-source LSM-tree-based database, and their performance should be analyzed with respect to the state-of-the-art.

1.3 Contributions

The main contribution of this paper is the proposal of query-driven compaction algorithms for LSM-trees that leverage the sort-merge results of range queries. We present the workflow for the project, which includes analyzing the underlying mechanism for range queries and compaction, designing query-driven compaction algorithms, and implementing the proposed solutions on top of storagelayoutemulation. We will also analyze the performance of our proposed solutions with respect to the state-of-the-art. Our proposed solutions can optimize the performance and efficiency of LSM-trees by utilizing the merge-sort results of range queries during the compaction process.

2 BACKGROUND

In the vanilla implementation of the LSM-tree, compactions are triggered by data ingestion, which is inserted query entries. Even though this mechanism highly improved the efficiency of lookups, it doesn't take full advantage of the characteristic of range queries. Each time when performing a range query, it's actually processing sort-merge for keys within the range. The query, however, only returns the result as output without writing back to the LSM-tree. As for the next compaction of the LSM-tree, it's highly likely to perform the sort-merge of this range again, which becomes a duplicated overhead. Given that, it's desirable to trigger compaction in range queries by writing their result back to the LSM-tree to improve the performance.

A few recent works have attempted to optimize the LSM tree with different compaction strategies. For example, Sarkar [3] constructed and analyzed the vast design space of LSM-compactions, and evaluated the state-of-art compaction strategies with respect to key performance metrics. Chai [1] added adaptive policy to adjust key compaction threshold to optimize LSM tree key-value stores. Sarkar [2] extended their previous work and discussed further about LSM trees, especially upon optimizing data stores, including data ingestion and compaction optimization. Despite these research findings, there is a lack of efforts in improving LSM-tree performance by employing a query-driven compaction strategy. Therefore, this project aims to fill this

research gap and leverages the current LSM-tree implementation with query-driven compaction solution, algorithms, and benchmarks.

3 ARCHITECTURE

3.1 Solution Design

When designing algorithms for range query driven compaction in the context of an LSM tree, there are three key problems that must be addressed.

- How should the sort-merge result of the range query be stored in memory prior to compaction?
- Given the sort-merge result, how can compaction be performed across all levels of the LSM tree?
- When should the query driven compaction take place to minimize the impact on performance?

To address the first problem, it may not be feasible to write the entire range query result to main memory if it is unbounded. One potential solution is to store the result in a fixed-size memory page and flush the page once it is full, before continuing to write to it. While this approach may impact range query speed, it allows for efficient use of a fixed amount of memory.

The second problem arises from the fact that the boundary of the sort-merge result for the range query may not align with the boundaries of SST files at each level of the LSM tree. To address this, our proposed solution is to fragment the SST files along the boundary of the sort-merge result. This enables the data unrelated to the range query to remain untouched while ensuring that only data overlaying with the sort-merge result is compacted.

Finally, the third problem concerns the optimal time to initiate the range query driven compaction process to minimize its impact on system performance. If we initiate compaction after each range query, it may slow down the subsequent operations. Future improvements may be possible to address this issue.

In this project, we will mainly focus on the second problem since it determines the feasibility and the other 2 problems mainly affect efficiency. In the following section, we provide further details regarding our designed algorithm for solving the second problem and ignore the other 2 problems for now.

3.2 Algorithm

Suppose we have an LSM tree with layers L_0 to L_n, and we're performing a range query. As we process the query, we store the result P in memory. When the range query is complete, we use result P to do a compaction in the LSM tree. We only conduct compaction to layers L_2 to L_n since L_0 and L_1 are changed frequently in practice.

To perform the compaction, we need to determine which SST files in the LSM tree overlap with the range of keys in P. We do this by finding the fence pointers for P_{min} and P_{max}, which are the minimum and maximum keys in P, respectively.

For each level i in the LSM tree (starting from L_1), we partition the SST files that overlap with P_{min} and P_{max} into two parts. For example, P_{min} is in SST file f_1. Then we partition f_1 into 2 smaller files separated by P_{min}.

If we're not at the bottom level of the LSM tree (i.e., $i < n$), we can safely drop all the files that fall within the range of P_{min} and P_{max}, since their contents are already represented in P. If we're at the bottom level (i.e., $i = n$), we replace the contents of these files with P instead. Since we already separate the files at boundaries of P, we will only compact data overlapping with P.

Algorithm 1 Query-Driven Compaction in LSM-Trees

```
P ← range query result in memory
if LSM tree layer number n > 3 then
    P_min ← smallest key in P
    P_max ← largest key in P
    for i ← 2 to n do
        find SST file f_1 with fence pointers p_1^i and p_2^i such that P_min ∈ [p_1^i, p_2^i]
        find SST file f_2 with fence pointers p_3^i and p_4^i such that P_max ∈ [p_3^i, p_4^i]
        partition f_1 into two parts f_1' and f_1'' such that f_1' ∈ [p_1^i, P_min] and f_1'' ∈ [P_min, p_2^i]
        partition f_2 into two parts f_2' and f_2'' such that f_2' ∈ [p_3^i, P_max] and f_2'' ∈ [P_max, p_4^i]
        if i < n then
            delete all files within [P_min, P_max]
        else
            replace all files within [P_min, P_max] with P
    empty P
```

3.3 Implementation

We cloned the storage-layout-emulation repository and implemented our proposed solutions based on its work. To begin with, we added a variable write-file-count to record the number of SST files writes during the range-query-driven compaction. We then added QueryDrivenCompaction function in workload-executor.cc to implement the compaction during range query which is constrained by a lower limit and upper limit. If there are only 2 levels in the LSM tree, no query-driven compaction is needed. If there are more than 2 levels, The query-driven compaction initiates from level 2. For each level, we create a pointer to traverse through SST files and search for files within the query range by checking the fence pointers of each SST file. Once we find any overlap, we merge the files. We call sortMergeRepartition between level 2 to level N-1 and sortAndWrite for level N. Both of them will return the number of file writes during their execution.

In addition, the sortMergeRepartition function mentioned above is used to implement the repartition of the boundary region of query-driven compaction. The boundary of the sort-merge result may not align well with the SST file boundaries in each LSM tree level as we discussed above. We fragmented the SST files along the sort-merge results' boundaries. This method splits the partially-overlapped-with-query SST files into half with the data unrelated to the query unaffected while ensuring only the overlapped data is compacted. In detail, the sortMergeRepartition function sort-merges the key-value pairs within the range query range. Only the key-value pairs overlapped with the range query result are compact, while the rest of the pairs in the same SST file are written back to the tree in an unchanged manner. those unchanged key-value

pairs are written to newly created SST files and these SST files are inserted back. This write-back also adds to the overall write count as a cost of query-driven compaction.

Finally, in order to understand the write I/O impact of the query driven compaction on range query, we do not always compact all of the range query results. Instead, we introduce a new parameter called QueryDrivenCompactionSelectivity which determines the percentage of range query results used for compaction. As a high percentage leads to more compactions but slower speed for range query, it may not always be optimal to compact all or large part of the range query.

3.4 Benchmark Explanations

We implemented a loop in the main method of emu-runner.cc to generate workloads, the workloads, perform the range query based on the function of Query-range-query-compaction-experiment(selectivities [i]), and clear the database using DiskMetaFile-clearAllEntries(). Specifically, Query-range-query-compaction-experiment(selectivities[i], "output.csv") performs range queries for each selectivity value in the selectivity array and writes output into a csv. For current testing workloads, we set the QueryDrivenCompactionSelectivity variable to be 1 which indicates compactions are triggered upon 100% of the queries. If the variable has a value less than 1, then compactions will be triggered on less than 100% queries. Moreover, every time the experiment is triggered, the csv will be created with a list of header rows including SRQ count, selectivity, range start, range end, occurrences, and write file count. Then we perform the range query and calculate the write file count using Query-rangeQuery(range-query-start, range-query-end, QueryDrivenCompactionSelectivity). We continue to update the csv output file.

We define a sequential workload as a sequence of inserts and then a sequence of range queries. We define a non-sequential workload as interleaving sequences of inserts and range queries. We generated both sequential and non-sequential workloads and designed our experiments to perform 100000 inserts and a number of range queries with varying selectivity choices. In total, we tested for 5 selectivity choices, including 1%, 25%, 50%, 75%, and 95%. We performed both sequential and interweaving workloads to evaluate our approach.

4 RESULTS

For the sequential workload, we observed that our total I/O decreases over time and this trend becomes more apparent at larger selectivity choices (Figure 1, 2). Although the total I/O cost is always higher than the total vanilla I/O cost in our 50 Range Query experiment, it becomes lower than the vanilla I/O cost in our 500 Range Query experiment. The reason is that the write I/O cost becomes amortized over time since the write costs are the highest for the start runs without any new data inserted (Figure 3). As we only checks the range query cost, in 25 selectivity, we can also see that there's a spike for compaction, but later get amortized and in total smaller than vanilla implementation (Figure 4).

For the non-sequential workload, we have interweaving insert and range query workloads 5 times and 10 times for all selectivity choices. After each insert, we run a batch of range queries. We can

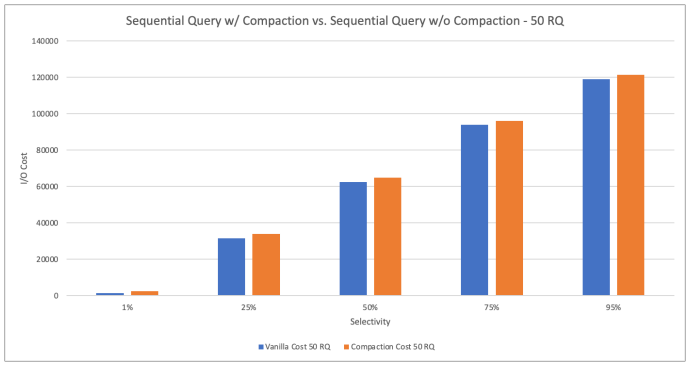

Figure 1: Sequential Query w: Compaction vs. Sequential Query w:o Compaction - 50 RQ

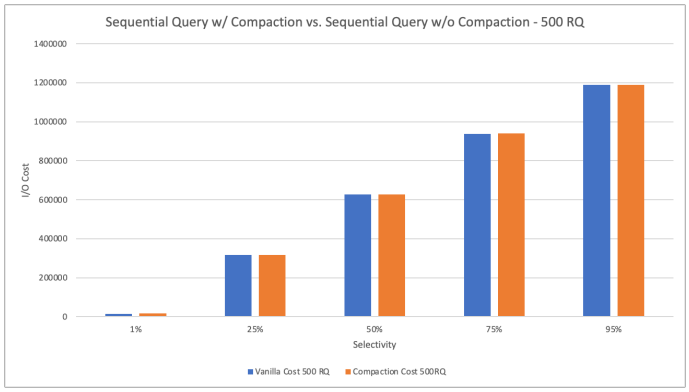

Figure 2: Sequential Query w: Compaction vs. Sequential Query w:o Compaction - 500 RQ%

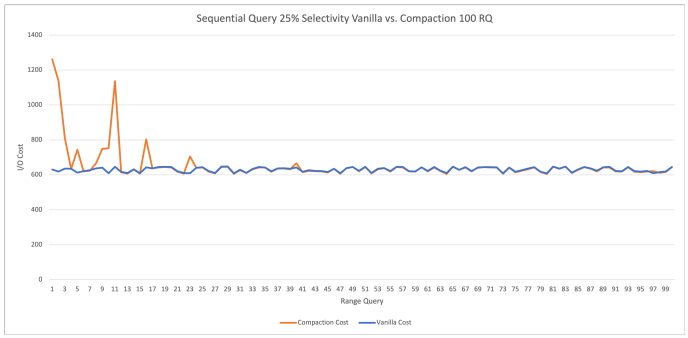

Figure 3: Sequential Range Query Compaction vs. Vanilla, Selectivity 25%

see that the insertion I/O remains stable with query driven compaction whereas in the vanilla version, it increases steadily(Figure 5 and 6).

Figure 7 shows the I/O costs for both insertion and range query processes for the non-sequential workload. It shows that the I/O cost for range query with query driven compaction is lower than

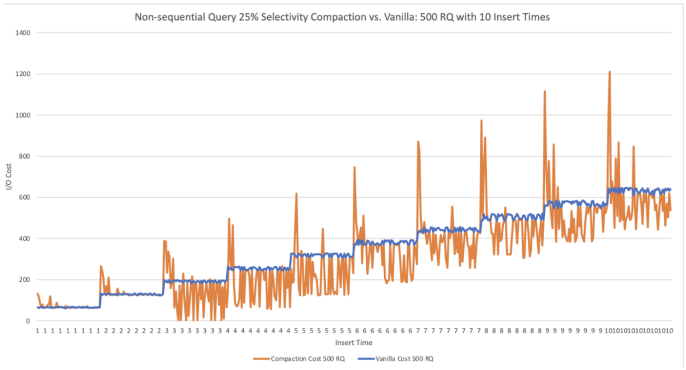

Figure 4: Non-sequential Query 25 Selectivity Compaction vs. Vanilla: 500 RQ with 10 Insert Times %

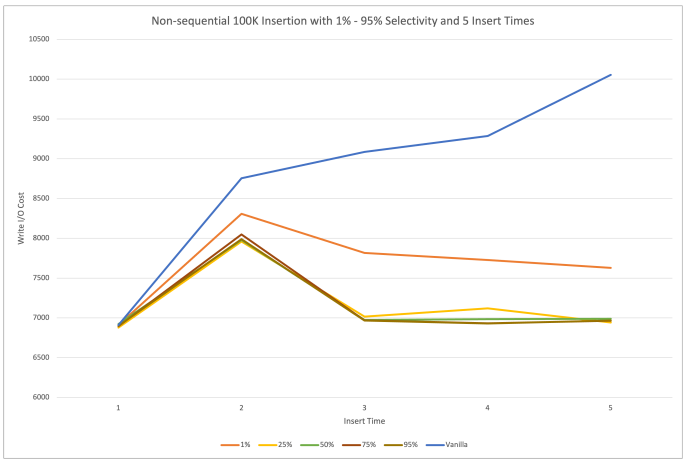

Figure 5: Non-sequential Inserts Compaction vs Vanilla, Selectivity 25%

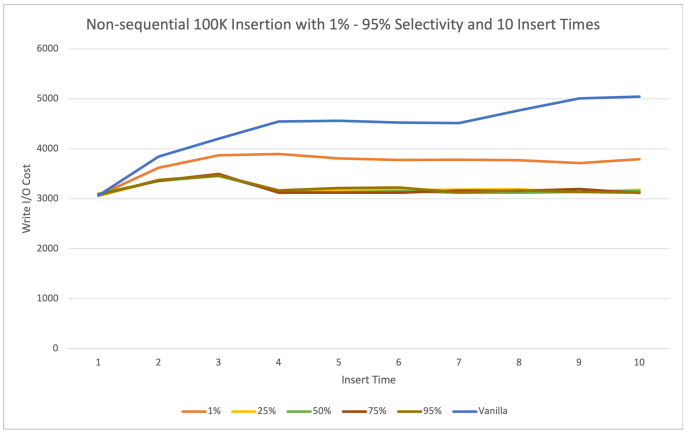

Figure 6: Non-sequential Inserts Compaction vs Vanilla, Selectivity 25%

vanilla range query for both inserts and range query. And the gap widens as more inserts and range queries run.

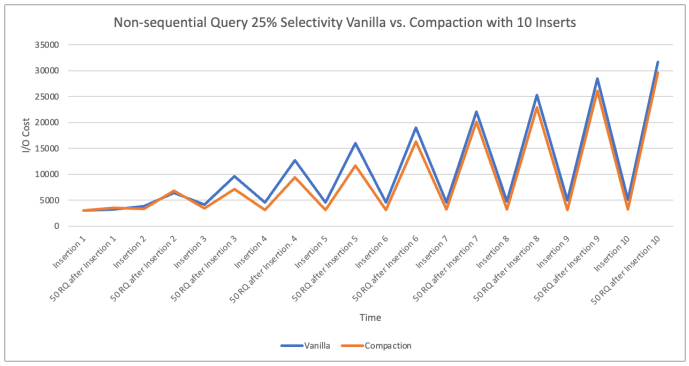

Figure 7: Non-sequential Query Compaction vs Vanilla with 10 Insert Times, Selectivity 25%

Finally, we run a number of point queries after 100000 inserts and 5000 range queries to show our method's impact on point queries. Figure 8 shows our sequential point query experiment result. Since many SST files were compacted during range queries, point queries only cost roughly half write I/Os compared with the vanilla version.

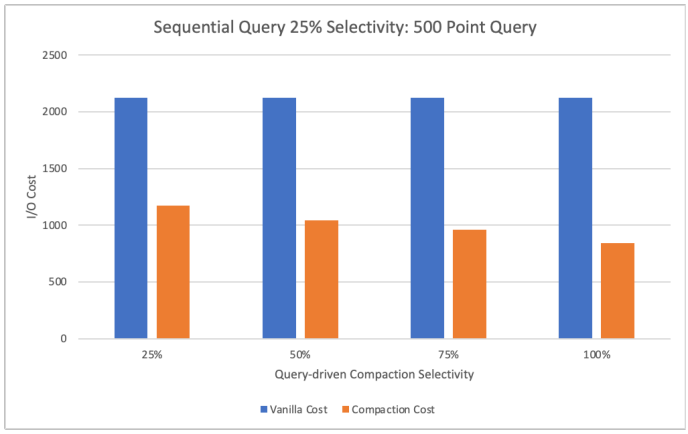

Figure 8: Sequential Point Query Compaction vs Vanilla, Selectivity 25%

These findings perfectly match our preliminary assumption that query-driven compaction can save I/O costs from duplicate merge-sort during both range queries and compaction. It can also reduce space amplification by dropping files more easily and reducing the LSM-tree expansion size. Since we separated files that are partially contained in the range queries into two files – one that's entirely in the range query and one that is not. This separation would cause some time consumption. Also, the time of flushing P to the last level of the LSM-tree would cause extra time. Nevertheless, our range-query-driven compaction decreases the total time of compaction because the part of the work that's done by regular insertion-triggered compaction is done here. In other words, we are using more partial compaction to save time amplification.

Consequently, in the duration of executing this strategy, the time amplification is amortized, and the total time saved by this range-query triggered compaction would be more obvious that using this strategy would be more time efficient

5 CONCLUSION

In conclusion, LSM-trees are a powerful data structure for handling large-scale, write-intensive workloads due to their efficiency, scalability, and fast lookup performance. Nevertheless, the current implementation does not fully exploit optimization opportunities arising from range queries. This paper addresses this research gap by proposing query-driven compaction algorithms for LSM-trees that leverage the sort-merge results of range queries to reduce the number of sort-merges during compactions, thereby improving overall write and space amplification.

Experimental results from sequential and non-sequential workloads indicate that our query-driven compaction approach effectively reduces I/O costs and space amplification, while introducing minimal time overhead. Although further research can explore potential improvements in compaction initiation and other optimization strategies, the findings presented in this paper demonstrate the potential of query-driven compaction in LSM-trees and contribute to the ongoing pursuit of more efficient data structures for handling large-scale, write-intensive workloads.

REFERENCES

[1] Y. Chai, Y. Chai, X. Wang, H. Wei, and Y. Wang. 2022. Adaptive Lower-Level Driven Compaction to Optimize LSM-Tree Key-Value Stores. *IEEE Transactions on Knowledge & Data Engineering* 34, 06 (2022), 2595–2609. https://doi.org/10.1109/TKDE.2020.3019264

[2] Subhadeep Sarkar and Manos Athanassoulis. 2022. Dissecting, Designing, and Optimizing LSM-based Data Stores. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. ACM, Philadelphia, PA, USA, To Appear. https://doi.org/10.1145/3514221.3522563

[3] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2021. Constructing and Analyzing the LSM Compaction Design Space. *PVLDB* 14, 11 (2021), 2216–2229. https://doi.org/10.14778/3476249.3476274