Query-driven compaction in LSM-trees

Nishil Agrawal
Boston University

Shubham Kaushik
Boston University

Konstantinos Karatsenidis
Boston University

ABSTRACT

Log-structured merge-trees (LSM-trees)[1, 2] have become a widely used data structure for persistent storage of key-value entries, particularly in write-intensive workloads. However, current LSM-tree designs do not capitalize on the sort-merge operations performed for range queries, leading to redundant work for range query-heavy workloads. In this research project, we aim to investigate and develop a range query-driven compaction approach for LSM-trees, focusing on enhancing the efficiency of range query processing and improving overall system performance for specific types of workloads. The proposed solution will involve understanding and configuring RocksDB's internal tunable parameters, designing innovative query-driven compaction algorithms, implementing a proposed solution, and benchmarking its performance. Eager compaction can be performed based on incoming range queries to improve the performance of subsequent range queries. By implementing a smart range query-driven compaction strategy, it is possible to reduce I/O operations, CPU computation cycles, and improve space, write, and read amplification, leading to better system performance.

1 INTRODUCTION

Log-structured merge-trees (LSM-trees) are a widely used data structure for the persistent storage of key-value entries, excelling in write-intensive workloads. However, current LSM-tree designs do not fully exploit the potential of range query-heavy workloads, leading to redundant work and suboptimal performance. This research project aims to address this issue by investigating and developing a range query-driven compaction approach for LSM-trees to enhance range query processing efficiency and improve overall system performance for specific types of workloads.

Our approach involves understanding RocksDB's[3] internal tunable parameters, designing innovative query-driven compaction algorithms, implementing the proposed solutions, and benchmarking their performance. By implementing a smart range query-driven compaction strategy, we can reduce I/O operations, CPU computation cycles, and improve space, write, and read amplification, resulting in better system performance.

This report is organized as follows: Section 1 introduces the motivations and the problem statement of this research project. Section 2 talks about the potential benefits of range query based compaction. Section 3 discusses potential approaches to solving the problems and introduces our proposed range query-driven compaction algorithm that attempts to solve the problem. Section 4 describes the experimental setup and results, followed by a discussion on the implications of our findings. We discuss the challenges we faced while working on this project in section 5. Finally, we discuss future work and conclude the report in Sections 6 and 7.

1.1 Motivation

LSM-trees store data on the disk as immutable logs or sorted sequence tables (SSTs), arranged in hierarchical levels of increasing capacity. In order to limit the number of logs that a lookup must probe, LSM-trees periodically perform sort-merge operations on logs of similar sizes, a process known as compaction.

Compaction in LSM-trees is primarily driven by the ingestion of new entries into the tree. Meanwhile, range queries merge all qualifying runs with overlapping key ranges, excluding older and logically invalidated entries. However, state-of-the-art LSM-tree designs do not capitalize on the sort-merge operations performed for the range queries by incorporating the results into the tree. This oversight can lead to redundant work for range query-heavy workloads, especially when the same range queries are executed multiple times.

By introducing a smart range query-driven compaction strategy, it is possible to reduce the number of I/O operations and CPU computation cycles required for the subsequent range queries, particularly those with overlapping ranges. This approach offers several potential benefits, including:

- **Reduced space amplification**: By removing logically invalidated entries, the overall storage footprint can be decreased.
- **Amortized write amplification**: By avoiding unnecessary writes during compaction triggered by policy-driven approaches, the number of write operations can be minimized.
- **Diminished read amplification**: By leveraging the improvements in space and write amplification, the number of read operations for point lookups can be reduced.

This research project proposal aims to investigate and develop a query-driven compaction approach for LSM-trees, focusing on enhancing the efficiency of range query processing and improving overall system performance for specific types of workloads.

1.2 Problem Statement

The primary objective of this project is to enable range query-driven compaction in LSM-trees, a topic that has not been extensively researched thus far. While existing research has primarily focused on optimizing compaction by modifying compaction policies, it has not specifically addressed range query-driven compaction. To explore the feasibility and potential benefits of this approach, we began by examining the RocksDB source code and analyzing its implementation of range queries.

In RocksDB, the ***Seek(const Slice& target)*** method of the ***Iterator*** [4] class performs a random walk on each level for every range query, returning an iterator for each level to initiate the sort-merge process. During this process, RocksDB merges the data with same keys from different levels and returns to the user valid records that fall within the queried range.

Two potential optimization opportunities arise when a range query is received from the application:

(1) An efficient algorithm can be executed with every range query, filtering out all files that can be compacted using the meta-data stored for SST files across all *levels*, including *memtables & immutable memtables*) and calling a manual

compaction using *CompactFiles(vector<string> input_file_names)*, an in-built function provided by RocksDB. This filtering process can be based on several factors, such as the size of the files and the amount of key overlap between them. This approach will reduce *space & write amplification* by compacting files and storing them in lower levels.

(2) An optimization opportunity arises after RocksDB has merged the keys within all the files of a subrange for the results of a range query. We can utilize the results of the range query to perform compaction and write them back to a higher level, thereby reducing the overhead of reading multiple files for other similar range queries. By intelligently leveraging the output generated during the range query, we can optimize the compaction process and minimize the work required for subsequent overlapping range queries.

Achieving this optimization requires a comprehensive understanding of the state-of-the-art compaction policies, the mechanisms that trigger compaction, and the steps involved in compaction between two or more levels. Additionally, it demands a more fine-grained approach to compaction that can accommodate different types of workloads. In summary, the problem to be addressed in this research project is developing and evaluating a range of query-driven compaction strategies for LSM-trees, aimed at improving compaction efficiency and overall system performance for specific workloads.

1.3 Contributions

The following are the key contributions of this research project on query-driven compaction in LSM-trees:

- **Understanding RocksDB's internal tunable parameters**: Exploring RocksDB's internal tunable parameters, focusing on how to optimize them for a range operation-heavy workload. This will involve delving into the intricacies of RocksDB's implementation and exploring ways to enhance its efficiency for range queries.
- **Design of query-driven compaction algorithms**: Developing innovative query-driven compaction algorithms that use the metadata of files to perform manual compaction or take the advanatge of the sort-merge operations performed for range queries. These algorithms aim to help reduce the overhead of range queries because of the amount duplicate keys by minimizing their occurrence.
- **Implementation of proposed solutions**: Implementing the proposed query-driven compaction solution on top of RocksDB and conducting analysis on it. This implementation will involve integrating one of the proposed algorithms into the existing RocksDB architecture and ensuring their compatibility with the database's internal tunable parameters. The implementation will be designed to enhance the efficiency of range queries related to duplicates and provide a seamless experience for users.
- **Benchmarking and optimization**: Performing extensive benchmarking tests using range operation-heavy workloads to evaluate and refine the implemented solutions. These tests will involve comparing different approaches, tunable parameters, and their effects on the overall performance of the query-driven compaction algorithms. The benchmarking results will provide valuable insights into the strengths and weaknesses of the proposed solutions and guide future refinements and enhancements.

2 BACKGROUND

Although there has been limited research on query-driven compaction in LSM trees, it is essential to understand the potential benefits of eliminating duplicates to improve the read cost of range queries. We assume that all duplicates in a range are removed after the first range query is performed, and the subsequent range query is a subset of the previous range query. This analysis considers the following parameters for an LSM tree:

- T: Size ratio between adjacent levels
- L: Number of levels
- P: Number of disk pages in the buffer
- D: Amount of duplicate keys
- S: Selectivity of Range Query

By analyzing the number of pages accessed per level, the number of pages accessed due to duplicates, and the total number of pages accessed (including duplicates), we can estimate the potential improvement in range query performance after removing duplicates.

The number of pages accessed per level is given by:

$$\sum_{l=0}^{l=L} S * P * T^{l} \tag{1}$$

The number of pages accessed because of duplicates is:

$$D * \sum_{l=0}^{l=L} S * P * T^{l} \tag{2}$$

The total number of pages accessed, including duplicates, is:

$$(1 + D) * \sum_{l=0}^{l=L} S * P * T^{l} \tag{3}$$

The improvement is calculated as the relative reduction in cost between the old and new cost, as follows:

$$Improvement = \frac{OldCost - NewCost}{OldCost} \tag{4}$$

The old cost is $1 + D$, and the new cost is 1. Therefore, the improvement can be expressed as a ratio:

$$Improvement = \frac{(1 + D) - 1}{(1 + D)} = \frac{D}{(1 + D)} \tag{5}$$

To convert the improvement into a percentage, multiply the ratio by 100:

$$ImprovementPercentage = \frac{D}{(1 + D)} * 100\% \tag{6}$$

For example, if the duplication is 20% of the entire data, we can estimate the potential improvement in range query performance after removing duplicates. Using the formula:

$$ImprovementPercentage = \frac{0.2}{(1 + 0.2)} * 100\% = 16.67\% \tag{7}$$

Therefore, if the duplication is 20% of the entire data, removing duplicates can potentially improve the range query performance by

16.67%. This percentage of improvement is relatively small but can still be significant in scenarios where range queries are performed frequently or on large datasets. It is important to note that the improvement percentage will vary depending on the amount of duplication in data. As the amount of duplication increases, the potential improvement in range query performance will also increase.

3 ARCHITECTURE

In LSM-trees, a single range query performs numerous operations before providing results to the user. The LSM-tree first hops to each level, identifies all keys within the queried range $[k_{\min}, k_{\max}]$, and performs a sort-merge to exclude logically invalidated keys or older entries. For large ranges and repeated range queries, this process involves redundant work. Our goal is to perform this task once for the entire range and subsequently only for valid keys in later queries.

Since we already perform the sort-merge operation after reading all the relevant pages, we can implement a smarter approach to writing them back after returning the results to the user. This requires an efficient algorithm to identify overlapping files between two or more levels.

We propose two approaches to tackle this problem:

(1) **Eager Compaction based on Range Query**: This approach involves eagerly triggering compaction by providing the files and levels that can be compacted based on the query range. This may require pulling these files back into memory according to the state-of-the-art compaction strategy and writing them back to the lower level. This approach can be tuned based on multiple parameters like range size, and the number of overlapping files. The can be achieved using *CompactFiles(vector<string> input_file_names)* functionality after running an algorithm.

(2) **Intelligent Iterator**: When in-range files are pulled into the memory during a range-query operation, we can use the metadata of each file to know how much overlapping is occurring. If a file at level x (L_x) overlaps entirely with a file at level $x + 1$ (L_{x+1}), the result of the range query can be written to the file at level L_{x+1}, and the file at level L_x page can be marked for garbage collection. This process can be applied across multiple levels, depending on the overlapping ranges. The iterator algorithm for the range query can be made to be worked intelligently to enable this set of tasks efficiently based on multiple parameters.

The second algorithm is more complex than the first one. In the first algorithm, we are not using the results of the range query directly but rather marking the files, that have been accessed to generate the output of the range query, for compaction. While, in the second algorithm, we aim to directly use the iterators that generate the output for the range query to perform a set of functions on the tree for compaction while marking duplicates for garbage collection.

3.1 Solution Design

As discussed in Section 3, we have two approaches for query-driven compaction. The high-level algorithms for these approaches are described below.

Eager Compaction based on Range Query

(1) Execute the range query $[k_{\min}, k_{\max}]$ on the LSM-tree.
(2) Identify all the files and corresponding levels that were accessed during the range query.
(3) Calculate the degree of overlapping between the accessed files across levels.
(4) Set a threshold for the range size and the number of overlapping files. If the overlapping degree exceeds the threshold, proceed with the eager compaction.
(5) For each pair of overlapping files, read the data from the higher-level file and merge it with the lower-level file according to the state-of-the-art compaction strategy.
(6) Write the merged data back to the lower level and delete the compacted file from the higher level.
(7) Repeat steps 5-6 for each pair of overlapping files until all files have been compacted.
(8) Update the LSM-tree metadata to reflect the changes.

Intelligent Iterator

(1) Execute the range query $[k_{\min}, k_{\max}]$ on the LSM-tree and store the accessed files and their metadata.
(2) For each file at level L_x, check if it overlaps entirely with a file at level L_{x+1}.
(3) If there is complete overlap, identify the lowest level among consecutive levels with overlapping files (e.g., L_{x+n}).
(4) Combine the in-memory iterators for the range query result and the overlapping files from levels L_{x+1} to L_{x+n}.
(5) Write the merged iterator result back to the file at the lowest level, L_{x+n}.
(6) Mark the files at levels L_x to L_{x+n-1} for garbage collection and update the LSM-tree metadata.
(7) Repeat steps 2-6 for all files across multiple levels.
(8) Remove the marked files and reclaim storage space by periodically performing garbage collection.

3.2 Pseudocode

In this section, we present the Range Query Driven Compaction algorithm, an efficient approach to manage data in a multi-level storage system. This algorithm operates on file metadata to identify the relevant files that can be compacted within a specific key range and triggers a manual compaction process accordingly.

The algorithm[5] iterates over all files in each level, comparing them against the provided start and end keys for range query. By executing this process, the algorithm ensures that only the necessary files are compacted, leading to optimized storage and improved query performance. An overview of this operation can be observed in Figures 1 and 2. To utilize this algorithm, it must be explicitly called by the application or user, providing them with greater control over the compaction process.

One of the key benefits of this algorithm is that developers who are aware of the potential gains from its implementation can easily integrate it into their applications. By understanding the underlying queries and advantages, developers can make informed decisions about when to trigger the algorithm, leading to a better overall system performance.

Algorithm 1 Range Query Driven Compaction Pseudocode

```
Result: Range Query Driven Compaction
RangeQueryDrivenCompaction($start\_key, end\_key$)
Initialize column family handle, data, and version
Create a vector of CompactionInputFiles for each non-empty level
Combine level and file metadata collection loops
$filter\_files$ ← FilterFileThatCanBeCompacted($all\_files$,
  $start\_key, end\_key, cfd\_$)
if $filter\_files.size() > 0$ then
    $level\_to\_write$ ←
      GetHighestLevelFromCompact($filter\_files$)
    Create a list of input file names
    Perform compaction with the list of input file names at
      $level\_to\_write$
end

GetHighestLevelFromCompact($files$)
$max\_level \leftarrow -1$
foreach $cif \in files$ do
    Update $max\_level$ if $cif.level > max\_level$
end

FilterFileThatCanBeCompacted($all\_files, start\_key, end\_key$,
  $cfd\_$)
Initialize variables and data structures
Iterate files from last level to first
Determine the largest and smallest key ranges
Handle edge cases and track levels
return Files with the highest size across levels

HighestFilesSizeAcrossLevels($files\_to\_be\_compacted$)
$max\_size \leftarrow 0$
$index\_having\_largest\_size \leftarrow -1$
foreach $vec\_cif \in files\_to\_be\_compacted$ do
    Calculate the total size of files
    Update the index with the largest size
end
return Files at the index with the largest size

FilterOnlyInRangeFiles($all\_files, start\_key, end\_key, cfd\_$)
Initialize filtered files vector
foreach $cfi \in all\_files$ do
    Filter files with overlapping key ranges
end
return Filtered files

FilesToBeCompactedAcrossLevels($all\_files, track\_levels$)
Initialize files across levels vector
Create CompactionInputFiles objects and add to the files across
  levels vector
Handle duplicate levels and update current level
return Files across levels
```

Additionally, the algorithm will eventually support adjustable threshold settings. This feature, once implemented, can allow developers to set a threshold for range query driven compaction, ensuring that the process is only initiated when the specified conditions are met. This flexibility enables more precise control over the compaction process, ultimately contributing to a more efficient and effective storage system.

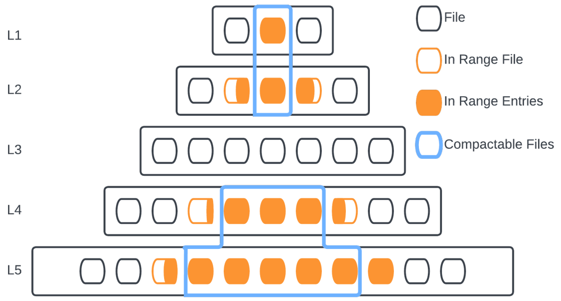

Figure 1: Range query driven compaction (a)

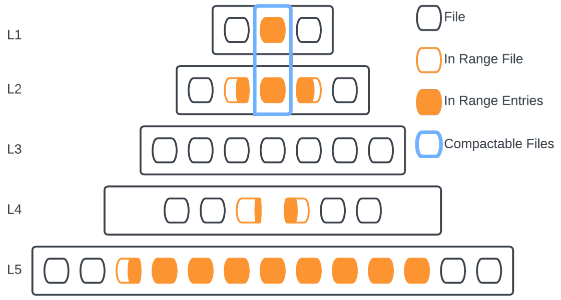

Figure 2: Range query driven compaction (b)

This Range Query Driven Compaction algorithm provides a powerful tool for managing data in multi-level storage systems. By offering customizable controls and leveraging file metadata for targeted compaction, this algorithm enables developers to optimize their applications' performance and storage efficiency.

4 BENCHMARK EXPLANATIONS

The experiments were conducted on a machine with the following specifications:

- Intel Core i7-7700HQ CPU @ 2.8GHz x 4
- 16GB RAM
- 512GB HDD

The options for RocksDB were set as follows:

- Compaction style: Level
- Write buffer size: 512KB
- Memtable factory: SkipListFactory
- Level 0 file num compaction trigger: 2
- Target file size base: 512KB
- Target file size multiplier: 2
- Max background jobs: 4
- Max bytes for level base: 512KB
- Max bytes for level multiplier: 2

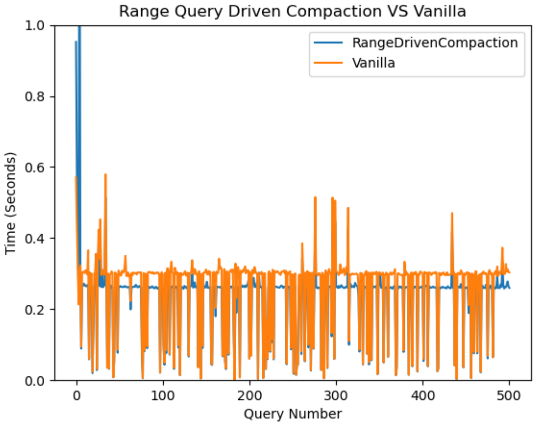

Figure 3: Query driven compaction v/s Vanilla approach (a)

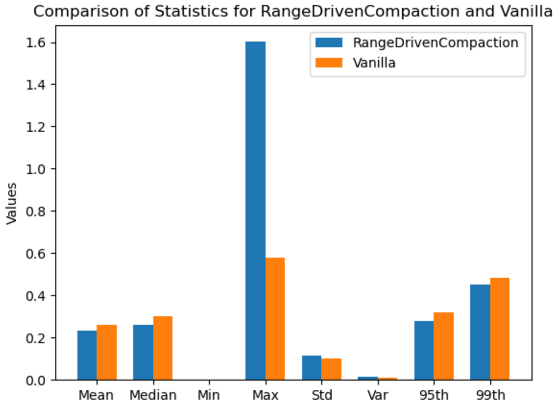

Figure 4: Query driven compaction v/s Vanilla approach (b)

4.1 Results

Our experiments were conducted using several different workloads. For one particular workload involving 4 million inserts, 6 million updates, 10,000 deletes, 500 range queries, and 40% selectivity, we present the results in Figures 3 and 4. As shown in Figure 3, there is an initial spike due to manual compaction, but the graph then shifts downwards, indicating improved performance. This leads to subsequent range queries becoming more efficient compared to the vanilla approach.

In Figure 4, we present statistical measures of the range query performance, including mean, median, maximum, minimum, standard deviation, variance, 95th percentile, and 99th percentile. Our approach shows a higher maximum value compared to the vanilla approach, but the mean, median, 95th percentile, and 99th percentile metrics exhibit significantly better results. These findings suggest that our method delivers a more consistent and improved performance for the majority of range queries, making it a preferable choice for similar workloads.

4.1.1 Inferences. Based on the experimental results, the following inferences can be made about the performance of the implemented approach:

(1) The approach performs well when the number of updates is significantly larger than the number of inserts.
(2) If there are enough range queries with similar ranges in a workload, the cost of compaction that is performed during the initial range queries can amortized

On the other hand, the approach may not perform as well for workloads with a lower number of updates as compared to inserts, and a high number of range queries with really low selectivity.

5 CHALLENGES

In this section, we detail the challenges we faced and our experiences.

(1) Exploring and Understanding RocksDB: To implement an algorithm to improve the performance of RocksDB for certain workloads, we have delved into the intricacies of RocksDB, an open-source key-value store that already boasts high performance and efficiency for a variety of applications. In order to effectively harness the power of RocksDB, we have studied its core components such as the LSM tree data structure, the compaction process, the process of performing range queries, and the tunable parameters that influence performance. This exploration has proven to be a challenging yet enriching learning experience for us.
(2) Generating Optimal Workloads: One key aspect of our project is the generation of optimal workloads to evaluate the performance of our algorithm. Crafting these workloads has required us to gain an in-depth understanding of RocksDB's performance for different workloads. We have strived to ensure that our workloads are flexible so as to cover all possible scenarios we could. By employing a well-rounded workload generator, we have been able to identify performance bottlenecks and derive valuable insights for tuning and optimizing our algorithm in the future.

6 FUTURE WORK

While our algorithm has demonstrated improved results for update-heavy workloads with frequent range queries, there is still potential for further enhancements.

One area for future work is implementing threshold mechanisms that provide a more granular control over the range-driven compaction algorithm. This could enable more fine-tuned adjustments to the algorithm based on specific workload requirements.

Another area for improvement is optimizing the algorithm to reduce the amount of repeated operations it performs when selecting files for compaction. This could further enhance the efficiency and performance of the algorithm.

In addition, we can explore the possibility of designing an algorithm that uses iterators to perform optimal compactions on the LSM tree, with the aim of improving the performance of range queries. This could provide an alternative approach to the Range Query based Eager Compaction approach we have proposed.

Overall, there are several potential avenues for future research and development in the field of LSM-tree compactions for more efficient range query processing.

7 CONCLUSION

In this research project, we present a novel approach to perform LSM-tree compactions, focusing on improving the efficiency of range query processing in range operations heavy workloads. We have discussed two potential approaches for this, Eager Compaction based on Range Query and Intelligent Iterator, and presented an algorithm that follows the former approach. We demonstrated the benefits of the algorithm using experimental results. There is still some potential to improve upon these, to further enhance the performance in terms of I/O operations, CPU computation cycles, and storage efficiency, while reducing write and read amplification.

REFERENCES

[1] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

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

[3] Facebook. Rocksdb. http://github.com/facebook/rocksdb, 2012.

[4] Facebook. RocksDB Iterator. https://github.com/facebook/rocksdb/wiki/Iterator, 2023. [Online; accessed 5-May-2023].

[5] Forked: Rocksdb github repository. https://github.com/shubham-sudo/rocksdb, 2023.