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) 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, 7 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

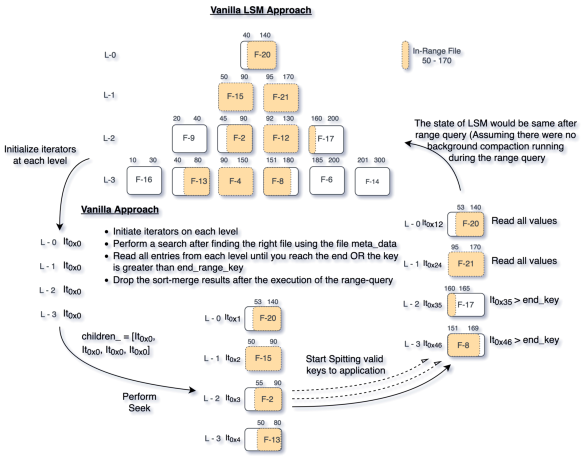


Figure 1: Vanilla range query flow in LSM

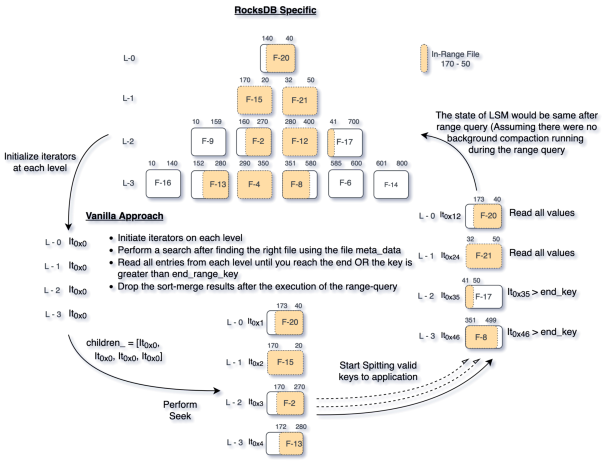


Figure 2: Range query flow in lexicographically sorted files

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.

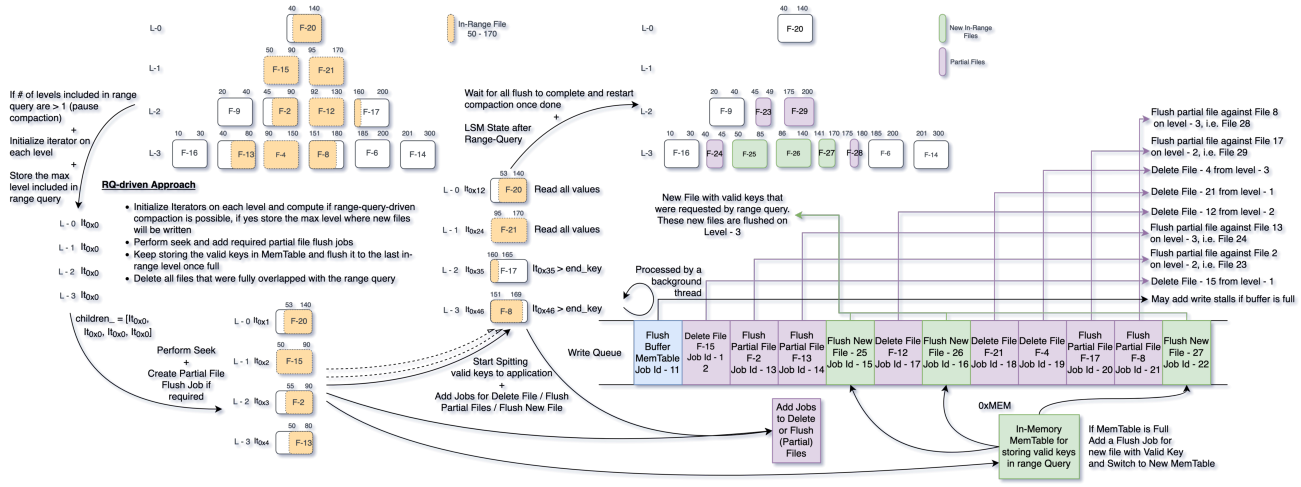


Figure 3: Query-driven compaction flow in LSM

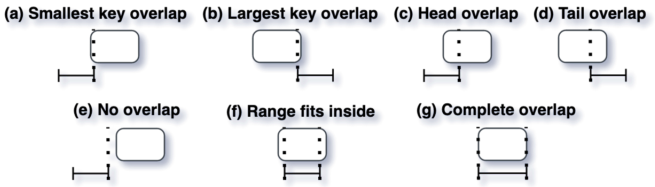


Figure 4: File range-query overlaps

4 SOLUTION NAME

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.

4.1 Vanilla Approach

The flow of range query in vanilla approach is shown in Fig-1 and Fig-2. The query runs by initiating the iterators for each level of LSM tree and then perform a seek operation on each iterator to find the first key that is greater than or equal to the start key of the range query. It then iterates through the SSTables in the LSM tree and returns the values that fall within the key range. At the end of the range query execution the state of LSM tree would be same as before the query execution.

4.2 Query-driven Compaction

The flow of range query in query-driven compaction is shown in Fig-3 and Fig-5. The initial setup of level iterators goes the same as in the state-of-the-art LSM range query. Once all the iterators are initialized, it performs a seek operation on iterators using the range-query start_key for each level. The seek operation in query-driven compaction performs an extra operation to create a partial file flush job and add it to the write_queue_. The partial file flush will only happen if the file does not completely overlap with the range-query start_key and end_key. We can have three scenarios for the partial file flush, which (as shown in Fig-4) are as follows.

(1) **No overlap** In this scenario, the partial file flush will not happen as the file does not overlap. Fig-4 (e)
(2) **Smallest or Largest key overlap *(Partial Flush)*** In this, the smallest or largest key of the file overlaps with the range query start or end key. Fig-4 (a) and (b)
(3) **Head or Tail overlap *(Partial Flush)*** In this, the partial part (having more than one key) overlaps with the range query. Fig-4 (c) and (d)
(4) **The range fits inside file overlap *(Partial-Partial Flush)*** Both the start and end keys of the range query fit completely in the file's smallest and largest keys. Fig-4 (f)

These partial flush jobs, that are added to the write_queue_ are executed by the *Priority :: HIGH* background thread, parallel to the range query. It flushes the partial part of the file that does not fall in the range query to the same level in the LSM tree.

The files that are completely overlapping with the range query start_key and end_key will be deleted from the lower levels and the valid keys will be added to the higher levels in new files. We can have one scenario for the file deletion from the lower levels which is as follows.

(1) **Complete overlap i.e. start_key <= smallest key and end_key >= largest key *(Just Delete)*** All files in the lower level, as well as in higher level, that are completely overlapping with the range query will be deleted by the query-driven compaction. Fig-4 (g)

The query-driven compaction also initiates an in-memory buffer of the default size configured in db_options to keep copies of the valid keys and flush them back to the LSM when it is full. The main thread, that is executing the range query will keep on performing the sort-merge operation and return the valid keys back to the application. Whenever it returns a valid key to the application, the query-driven compaction makes a copy and stores it in a memtable. Once the memtable is full, it creates a new flush job and adds it to the write_queue_. The query-driven compaction also adds write

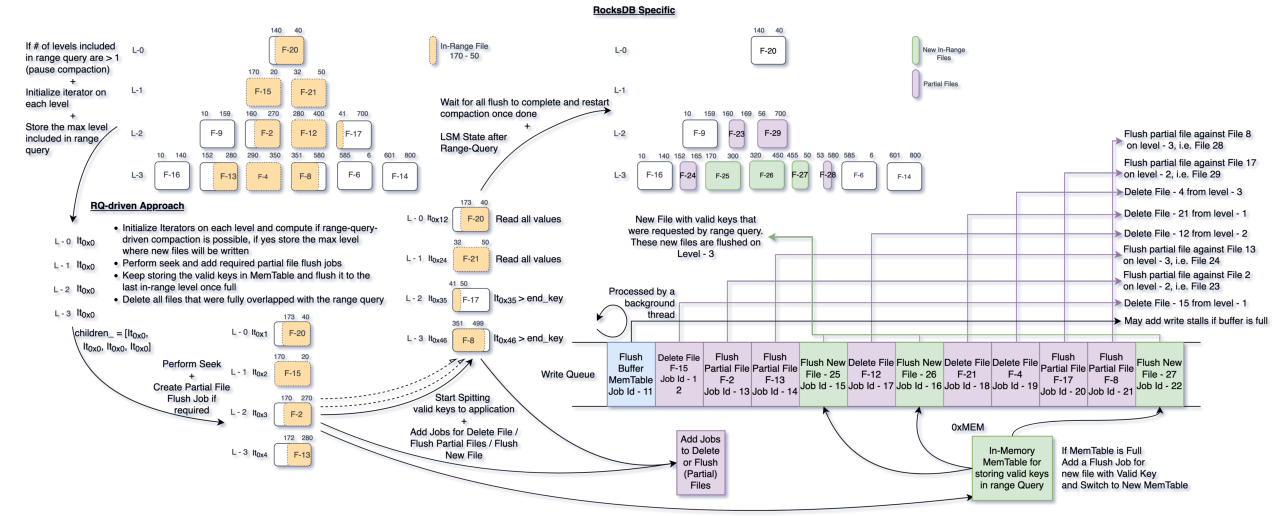


Figure 5: Query-driven compaction flow in lexicographically sorted files

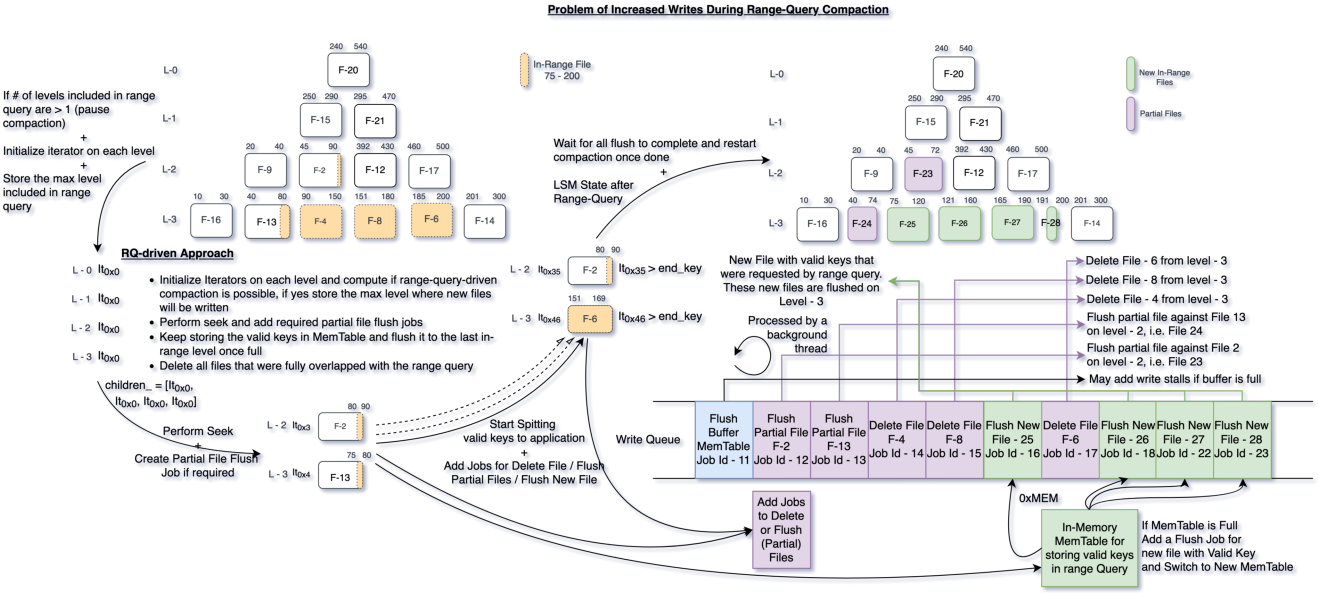


Figure 6: Query-driven compaction with increased writes problem due to less overlap

stalls for each range query to flush all the jobs initiated during this operation. This happens by stopping the background work during the start of the range query and waiting for all the flush jobs to execute successfully at the end of the range query.

Whenever query-driven compaction is triggered during the range query, it will remove all the logically invalid keys and tombstones from the LSM tree that fall in that range, which also results in reduced space amplification. This way whenever background compactions are triggered after successful query-driven compaction, it will increase the chances of trivial moves of files between levels as per the minimum overlapping strategy. It will also reduce the write amplification for any compaction that has overlapping keys with the previous query-driven compactions.

Keys from Level-0 are not pushed to the last level and there would be no partial flush for the same to keep the hot data in lower levels.

4.3 Informative Compaction

When the write buffers is full and more ingestion request hits the database, it will keep stalling the new writes till the buffer is not flused on to the disk. The query-driven compaction is fruitful for future queries but it may stall the newer writes for the compactions

Level	# of E_{useful}	# of $E_{unuseful}$	min key	max key	# of Entries
L-0	-	-	-	-	-
L-1	$E_{useful\ 1}$	$E_{unuseful\ 1}$	x_1	y_1	z_{01}
L-2	$E_{useful\ 2}$	$E_{unuseful\ 2}$	x_2	y_2	z_{12}
L-3	$E_{useful\ 3}$	$E_{unuseful\ 3}$	x_3	y_3	z_{23}
⋮	⋮	⋮	⋮	⋮	⋮

Table 1: Decision making data per level for each range query

Level	L-1	L-2	L-3	⋯
L-1	L_{11}	L_{12}	L_{13}	⋯
L-2	×	L_{22}	L_{23}	⋯
L-3	×	×	L_{33}	⋯
⋮	⋮	⋮	⋮	⋱

Table 2: Decision matrix created based on Table-1

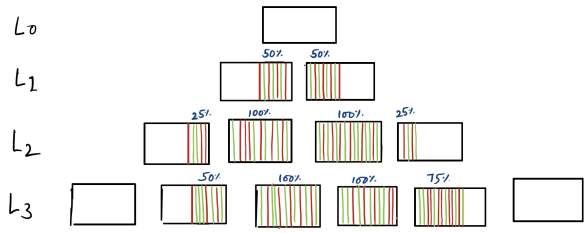


Figure 7: Example to perfrom query-driven compaction

Level	# of E_{useful}	# of $E_{unuseful}$	min key	max key	# of Entries
L-0	-	-	-	-	-
L-1	2K	2K	x_1	y_1	2K
L-2	5K	3K	x_2	y_2	5K
L-3	6.5K	1.5K	x_3	y_3	6.5K

Table 3: Decision making data for example shown in Fig-7

Level	L-1	L-2	L-3
L-1	[0.5, -] = T	[0.58, (-, 0.4)] = T	[0.67, (-, 0.4, 0.76)] = T
L-2	×	[0.62, -] = T	[0.71, (-, 0.76)] = T
L-3	×	×	[0.81, -] = T

Table 4: Decision matrix based on Table-3 (Cell: $[R_{utu}, R_{ete}]$)

happening during the range query. This needs a better decision making strategy to decide whether we should go for compaction or just do the vanilla approach while the execution of range query.

The range query could be anything and it can read one or more files or entries from each level. If the query reads highly overlapping data across multiple levels then the query-driven compaction would remove more invalid keys but if the query only has fewer keys that are overlapping, it may add more overhead of writing valid keys than removing invalid ones. Once a query-driven compaction is done, it writes all the data for the range into one level. Now, if a new range query is triggered after some ingestion (assuming the intersection of previous range query with new entries is not null) and it overlaps 90% with previous range query with remaining 10% that is uniformly distributed across whole range query. It will end up with a scenario like shown in Fig-6.

4.3.1 Solution. A small amount of work before starting a range query can help in making more informative decisions for the above scenarios. Lets say for every range query we have few useful and unuseful entries that will be read from each level. *Useful entries (E_{useful})* are those which are read from the level to serve the range query and can be moved down to remove logically invalid entries from lower levels. *Unuseful entries ($E_{unuseful}$)* are those that are read from the level and will be written on the same level in smaller (partial) files. The *useful* entries will also have *min_key* (x_i), *max_key* (y_i), and *Total # of entries* (z_{ij}) for that range query as shown in Table-1. The z_{ij} is the # of overlapping entries between two adjacent levels i and j from the range of entries. Once we have this decision making data for each level we can form our decision matrix as shown in Table-2. Each cell of decision matrix will represent the boolean value $L_{[start\ end]}$. $L_{[start\ end]}$ is the AND ($\wedge$) for the ratio of *useful* to *unuseful* entries (R_{utu}) and ratio of *number of useful entries in i* to *number of useful entries in $i+1$* (R_{ete}).

$$L_{[start\ end]} = \begin{cases} R_{utu} = \frac{\sum_{i=start}^{end} E_{\text{useful } i}}{\sum_{i=start}^{end} E_{\text{useful } i} + E_{\text{unuseful } i}} \\ R_{utu} > WC_{threshold} \\ \\ R_{ete} = \frac{z_i}{z_{i+1}} \forall\ i = start\ to\ end \\ R_{ete} >= UTL_{overlap}\ \&\ R_{ete} <= LTU_{overlap} \end{cases}$$

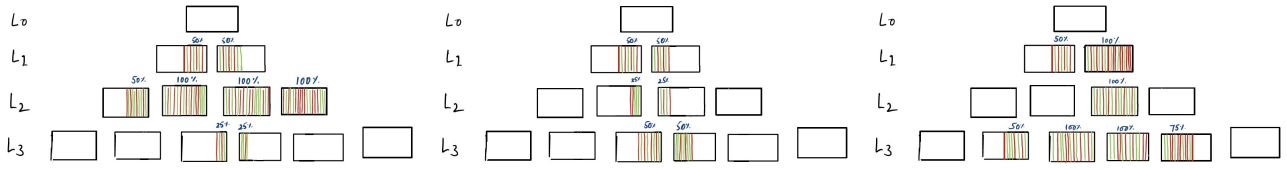
where $WC_{threshold}$ is the configurable write cost threshold and $UTL_{overlap}$ is the threshold for overlapping entries from upper to lower level, $LTU_{overlap}$ is the threshold of overlapping entries from lower to upper level. When $start == end$, the R_{utu} will represent the ratio for number of $useful$ to number of $useful + unuseful$ entries for same level (assuming R_{ete} true for $start == end$). The same level ratio will help query-driven compaction to not pick a level in which fewer entries are useful and it may write lot of unuseful at same level.

Let's take an example where we have an LSM with *size ratio* 2 (T), number of pages 512 (P), number of entries per page 4 (B), and entry size 1024 B (E). Now, assume we have 4 levels in current LSM state (see Fig-7).

- Page size : $B * E \approx 4KB$
- File size : $P * B * E \approx 2MB$
- Total entries in file : $P * B \approx 2K$
- $WC_{threshold}$: 0.5
- $UTL_{threshold}$: 0.4 & $LTU_{threshold}$: 1

All the levels upto L2 are full and L3 has 6 files. This means we have approx 4K entries in L1 (2 SSTs), 8K in L2 (4 SSTs) and 12K in L3 (6 SSTs) as shown in Fig-7.

Now, consider a range query (shaded portion in Fig-7) that will read entries from L1, L2, and L3. The E_{useful} and $E_{unuseful}$ would be as shown in Table-3. The *min_key* & *max_key* are the minimum and maximum keys of the range query from each level. The decision matrix can be formed based on Table-3 (as shown in Table-4). For this example we assume that the range of keys from every level are uniformly overlapping with the range of keys on the next level. The query-driven compaction will pick the first true entry from right



Level	# of E_{useful}	# of $E_{unuseful}$	min key	max key	# of Entries
L-0	-	-	-	-	-
L-1	2K	2K	x_1	y_1	2K
L-2	7K	1K	x_2	y_2	7K
L-3	0.5K	2.5K	x_3	y_3	0.5K

Level	L-1	L-2	L-3
L-1	[0.5, -] = T	[0.75, (-, 0.28)] = F	[0.63, (-, 0.28, 14)] = F
L-2	×	[0.87, -] = T	[0.68, (-, 14)] = F
L-3	×	×	[0.2, -] = F

Level	# of E_{useful}	# of $E_{unuseful}$	min key	max key	# of Entries
L-0	-	-	-	-	-
L-1	2K	2K	x_1	y_1	2K
L-2	1K	3K	x_2	y_2	1K
L-3	2K	2K	x_3	y_3	2K

Level	L-1	L-2	L-3
L-1	[0.5, -] = T	[0.37, (-, 2)] = F	[0.41, (-, 2, 0.5)] = F
L-2	×	[0.25, -] = F	[0.37, (-, 0.5)] = F
L-3	×	×	[0.5, -] = T

Level	# of E_{useful}	# of $E_{unuseful}$	min key	max key	# of Entries
L-0	-	-	-	-	-
L-1	3K	1K	x_1	y_1	3K
L-2	2K	0K	x_2	y_2	2K
L-3	6.5K	1.5K	x_3	y_3	6.5K

Level	L-1	L-2	L-3
L-1	[0.75, -] = T	[0.83, (-, 1.5)] = F	[0.82, (-, 1.5, 0.30)] = F
L-2	×	[1, -] = T	[0.85, (-, 0.30)] = F
L-3	×	×	[0.81, -] = T

Table 5: Examples for NOT performing query-driven compaction based on decision matrix

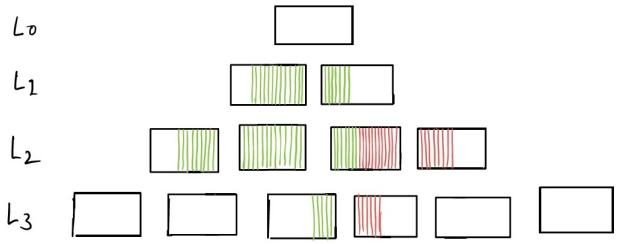


Figure 8: Extended Solution

top of the decision matrix checking every diagonal from left top to the right botton of the matrix. For the above example it would be L_{13}.

5 EVALUATION

5.1 Experimental Settings

- CPU :
- RAM :
- RocksDB version :

5.2 Experimental Results

Experiement results for workload

- # of Inserts - 500000
- # of Updates - 250000, 500000, 750000
- # of Range Queries - 100
- Selectivity for Range Queries - 0.2, 0.4, 0.8

Figure-9 shows the comparison between the number of times compaction is triggered for vanilla vs query-driven approach for the given workload. It appears that for different updates with different selectivity, the vanilla approach triggered more compactions as compared to the query-driven approach.

Figure-10 & Figure-11 shows the comparison between read and writes bytes during the compactions that were triggered in vanilla and query-driven comapactions. We can see that the bytes that were read and written during the vanilla are much higher than the newer approach

Figure-12 shows the number of keys that were dropped or re-written with newer values while compaction.

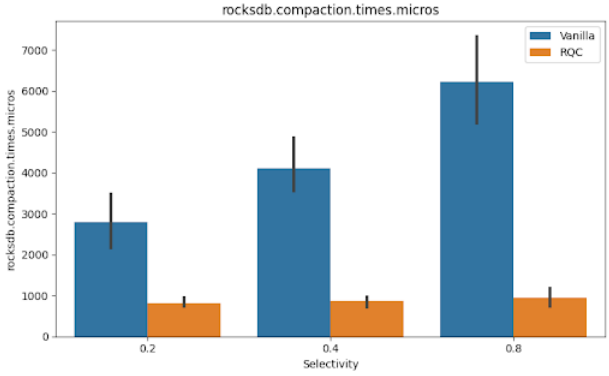


Figure 9: Number of compactions triggered during the workload execution

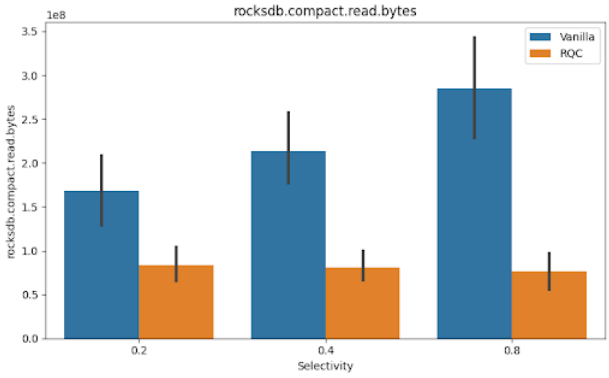


Figure 10: Number of bytes read during compactions

Figure-13 show that query-driven compaction writes more bytes while flushing partial files and in-range files during range queries. This factor can be tuned by computing the overlapping entries in the lower levels while performing the range query. If the lower levels have less number of entries that overlap with the higher level then we can perform the vanilla fashion range query but if the overlap is much larger then we can go for query-driven compaction.

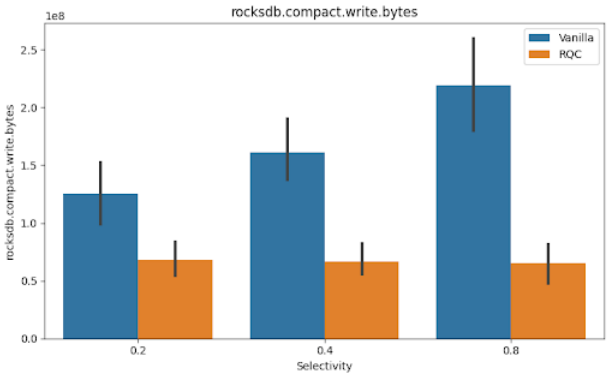


Figure 11: Number of bytes written during compactions

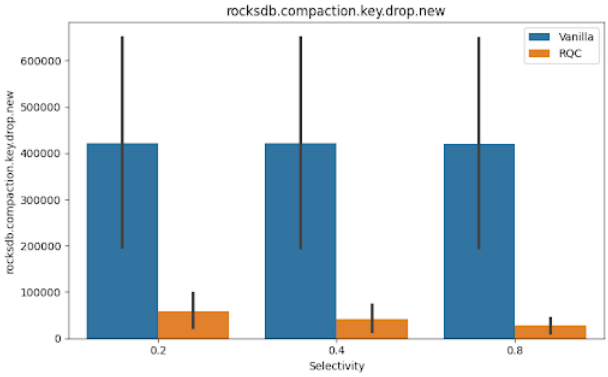


Figure 12: Number of keys dropped during compactions

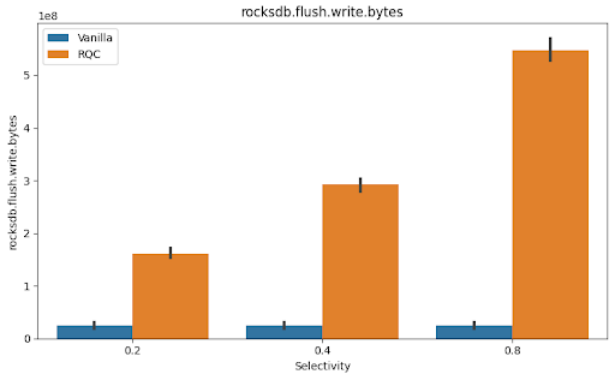


Figure 13: Number of bytes written during range query and buffer flush at level-0

6 RELATED WORK

7 CONCLUSION

REFERENCES