Benchmarking RocksDB: Exploring Compaction Options

David Shen
Boston University

Xiaoyan Ge
Boston University

ABSTRACT

In the era of big data, the ability to gather and analyze large quantities of data is critical to gain key insight. Database Management Systems (DBMS) are a core component of data-intensive applications as they facilitate the analysis and organization of large amounts of data. RocksDB is an example of a storage engine used by numerous database management systems. As data systems manage large amounts of data, achieving good performance is critical, which is why vendors provide many tuning parameters and configuration knobs. Unfortunately, tuning a database is a difficult problem. Therefore, the purpose of our research is to benchmark different configuration methods of RocksDB and collect data on their effects. Specifically, we chose to look at compaction methods, as compaction options are sparsely documented within RocksDB. We found that although conventional wisdom favors tiered compaction for write heavy workloads, this is not necessarily the case for RocksDB. In fact, we found leveled compaction within RocksDB to be better in almost all scenarios. We conclude that users are better off tuning leveled compaction for writes, rather than using tiered compaction.

1 INTRODUCTION

In the era of big data, the ability to gather and analyze large quantities of data is critical to gain key insight. Database Management Systems (DBMS) are a core component of data-intensive applications as they facilitate the analysis and organization of large amounts of data [1]. These systems are often built upon an underlying storage engine. An example of one such storage engine is RocksDB. RocksDB is an embedded key-value storage system developed by Facebook and is forked from Google's own LevelDB [9]. RocksDB is used as the backend storage system for numerous data systems such as CockroachDB and MyRocks, the latter being an integration with MySQL. Systems such as RocksDB are often responsible for managing large amounts of data. As a result, optimizing the performance of these data systems is essential. Fortunately, database engineers and vendors expose many different tuning parameters and configuration knobs of the data system. These parameters allow users to optimize a database for their workload by running benchmarks and taking measurements. Metrics such as throughput and response time are often used to measure how well a DBMS performs.

Unfortunately, tuning a database is a difficult task [10]. The reason why tuning is so difficult is because databases are complex systems that are used for a wide range of applications. The conventional process of finding the best configuration for a workload is often through manual trial-and-error. We run our workload, measure the performance, tweak the configuration of our storage engine, and repeat. Although it sounds simple, this approach has many subtle challenges. For one, tuning parameters are often not independent. Changing one knob may affect the results of another, creating a complex web of dependencies. Because of these complex dependencies, the effects of a specific tuning parameter is often hard to document. Most of the time, we need to benchmark a specific tuning configuration in order to know how it will impact performance. To complicate matters even further, configurations are difficult to reuse. Even if we have arrived at an optimal configuration for our workload, what works well for one workload may not be viable for another. Lastly, the complexity of a storage system is ever changing as more features and tuning parameters are constantly being added. Despite these difficulties, benchmarking and tuning a database is still an essential task.

1.1 Motivation

The difficulty of database tuning has made it clear that benchmarking is the most dependable way to fully optimize a database for a workload. While more mature database systems often have large amounts of data and benchmarks to draw upon, a younger storage system such as RocksDB does not have quite the wealth of knowledge. However, many established database systems such as MySQL and MongoDB have started to offer RocksDB as an alternative storage engine. Some have even replaced their current storage engines for RocksDB. This paper attempts to capitalize on the growing popularity of RocksDB by studying and benchmarking certain aspects of the storage engine. As RocksDB is a relatively new storage engine, it is not as well documented or benchmarked as other storage engines. And being an open source storage engine, RocksDB experiences frequent updates and feature releases. As a result, existing documentation becomes quickly outdated. To add to this, RocksDB has poor default tuning parameters, which have been muddled due its young age and due to reasons such as backwards compatibility. To address these issues, in this paper we explore different features of RocksDB in order to collect data and to further understanding.

1.2 Problem Statement

The problem is it is quite difficult to know what certain tuning parameters within RocksDB will do. Furthermore, parameters such as compaction priority often undergo changes, which obsoletes previous benchmarks and research. Therefore, the goal is to provide up to date benchmarks on the effects of compaction and compaction tuning parameters within RocksDB. We will measure their effect among three different metrics, response time, throughput, and write amplification. In doing so, we also want to understand and explain how RocksDB has implemented its compaction methods and its compaction tuning parameters.

1.3 Contributions

The contribution of this paper is twofold. First, we provide background information on the basics of compaction within RocksDB. In doing so, we also provide information on sparsely documented tuning parameters within RocksDB, namely compaction priority options. Secondly, we provide benchmarking data on a number of

compaction tuning parameters. In particular, we explore leveled versus tiered compaction, and how they perform for write heavy workloads. Ultimately, we demonstrate the concrete benefits of using leveled compaction within RocksDB.

2 BACKGROUND

2.1 LSM Trees

Log Structured Merge Tree(LSM-tree) is a structure that has been widely used by multiple databases nowadays. For example, Apache Casandra, RocksDB, LevelDB, etc. In this section, we want to briefly introduce how exactly LSM trees are implemented in RockDB by going through the data workflow, memtable, Write Ahead Log, Sorted Sequence Table, compaction, and Manifest File.

As the data is trying to be written into the LSM tree in RocksDB, first, the data gets stored both in an in-memory write buffer named memtable for in-memory storage and in a Write Ahead Log(WAL) for fault tolerance. The memtable and WAL serves as a receiver that will turn over the data to a Sorted Sequence Table(SST) file for disk storage and will discard the data in memtable and WAL when the process is completed. The system will assign a new pair of memtable and WAL while the "data handover" process is happening thus will not slow things down(in other word, concurrently). SST files are the basic storage "unit" that are sequentially stored in sequential levels of increasing size, from L0 to LN(N is user specified), with an index block for quick binary search. L0 is a little bit special since it might include an overlapping range of keys where all other levels only contain distinct key ranges.

As the SST files kept flowing in to the point that reached the "number of file" restriction in L0, a merging process called "compaction" that will merge the L0 files with L1 files, if there is any. This compaction process will be further explained in the next section. When the storing is done, the files will be deleted and replaced by the new files in L1,as the Figure 1 showed. Same rules apply to the rest of the levels, since each level stores N time more than the previous level(N is defined by user again), more data(also elder data) will be stored in the bottom level of LSM.

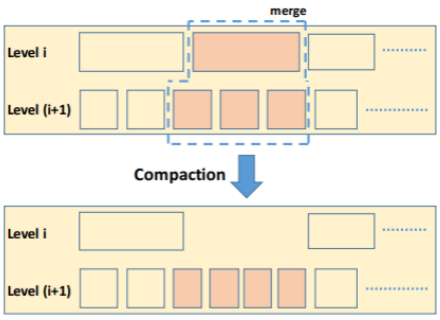

Figure 1: A Example of how an LSM Tree accomplishes merging

A Manifest File was also maintained as a log at each level(hence the Log in LSM). The file is in the form of a log that any changes will be added to, instead of edited. It contains mostly some metadata information such as key ranges. The main contribution of the file is to allow quick identification of SSTs that might have the target key while performing lookups. The search of a key will be performed at each level, from newest(e.g. L0) to oldest(e.g. L5). At each level we need to conduct 3 binary searches. First we need to search for the target SST files. Then in the SST files, we need to locate the target block. In the end, we need to locate the key within the data block. There is also a bloom filter that provides false positives(if the key is definitely not in the data block) for each data block we can reduce the number of unnecessary SST searches.

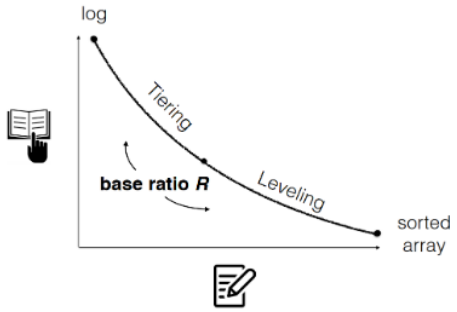

Figure 2: How Tiering and Leveling Compaction acts differently as base ratio R change

2.2 LSM Tree Design Space

In this section we are going to talk about the LSM tree design space, and more specifically, different trade-offs and how tuning parameters compete with each other in the LSM design space. In short description, LSM design is flexible enough from working like a write-optimized log to perform like a read-optimized sorted array. As it was shown in Figure 2, the read cost and write cost were affected by 4 main tuning parameters: (1) the compaction style (Leveling vs Universal), (2) the base ratio R between levels, (3) the allocation of memory between write buffer and bloom filters, and (4) allocation of filters in different bloom filters [3].

According to the paper and their experiments, when the size ratio was set to 2, the write and read cost for both Leveled and Universal compaction became identical. As size ratio increases, universal compaction's read cost decreases and write cost increases. Leveled compaction performed the other way around. As the size ratio becomes more extreme, the LSM starts to perform like a log or sorted array. For example, when size ratio is 1, the Universal compaction style LSM tree becomes a log which favors write, while leveled compaction LSM tree becomes more like a sorted array, which favors read.

Next, we want to evaluate memory allocation. First we want to talk about the memory allocation between bloom filters and write buffers. The more memory we allocate for bloom filters, the better chance of false positive rates we can get from it. However, it majorly depends on the type of word load the user is having. If the user is writing way more data then performing lookups, then assigning memory for bloom filters does not seem very reasonable. We also

want to look at how memories for bloom filters are allocated to different bloom filters. By reallocating the memory, we can adjust the false positive rates between different filters respectively [3]. When benchmarking RocksDB, we will tune these LSM tree parameters in order to achieve better write performance.

2.3 Compaction

The purpose of compaction is to remove multiple copies of the same key and also process deletions of keys. The entire database is stored on disk in a set of SST files, based on SSTables. [8]. These types of files are immutable. SST files on disk are organized in multiple levels, as seen in Figure 3. Each level is one data sorted run, with the exception of level 0, as seen in Figure 4. Because SST files are immutable, writes to the database cannot directly overwrite existing data. For insertions and updates in the database, rather than overwrite existing rows, RocksDB writes new timestamped versions of the inserted or updated data in new SST files. And for deletes, RocksDB does not perform deletes by removing the deleted data. Rather, RocksDB marks it with tombstones. Because SSTables and SST files by extension are immutable, overtime RocksDB will accumulate many versions of a row in different SST files. As SST files pile up, the distribution of data can require accessing more and more files to retrieve data for a query. As a result, compaction in RocksDB is responsible for merging SST files and discarding old data. The merge process is performant, because rows of the SST files are sorted by key. The new versions of each row are written to a new SST file. The old versions, along with any rows that are ready for deletion, are deleted as soon as pending reads are completed.

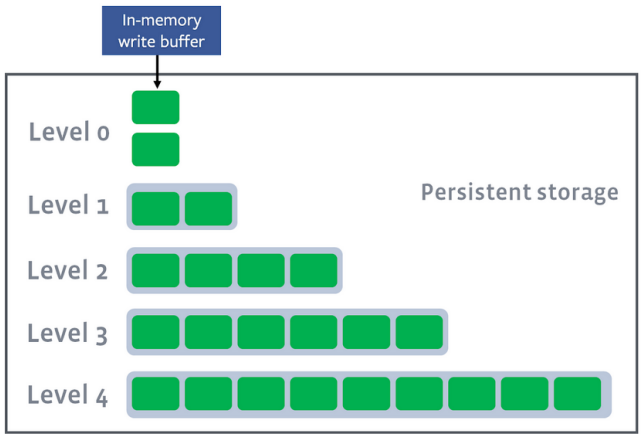

Figure 3: Each level of an LSM tree is made of SST files

Compaction constrains the LSM tree shape. The specific compaction algorithm used will determine which sorted runs can be merged and which sorted runs need to be accessed for a read operation. An LSM tree has two main types of LSM-tree compaction strategies, leveled compaction and tiered compaction [11]. The key difference between the two algorithms is that leveled compaction tends to greedily merge a smaller sorted run into a larger one, while tiered compaction uses a lazy merge strategy. Tiered compaction waits for several sorted runs with similar sizes and merges them together. Both strategies will flush SST files to the next level of the LSM tree once a certain capacity is reached.

Leveled compaction is generally regarded to have good read and space amplification but worse write amplification [3]. Tiered compaction on the other hand provides far better write amplification with worse read amplification. RocksDB uses leveled compaction by default. The overall write throughput of an LSM database directly depends on the speed at which compactions can occur.

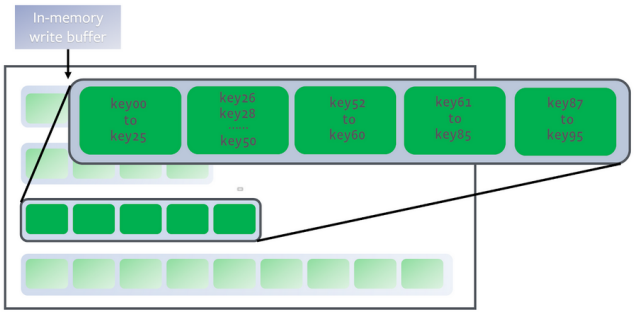

Figure 4: Data is sorted within each level

2.4 Leveled Compaction

We will describe how leveled compaction works within RocksDB. We note that despite its naming, leveled compaction in RocksDB is a hybrid compaction approach that uses tiered for the smaller levels and leveled for the larger levels. Leveled compaction is the most widely used compaction style in RocksDB, because it is the default style [7]. As mentioned above, data is stored in files, which are organized into multiple levels. Each level is a sorted run, as the keys within the SST files of a level are in sorted order as evident in Figure 4. The sorted nature of these keys allows for reads on the data to be implemented via binary search. In addition to data being stored in levels, each level has a target size, as seen in Figure 5. This target size can be thought of as the capacity of the level. One of the goals of compaction is to restrict the data on each level to be under the target size. The target size of each level is usually exponentially increasing. The factor at which it increases by is an available tuning parameter that defaults to 10.

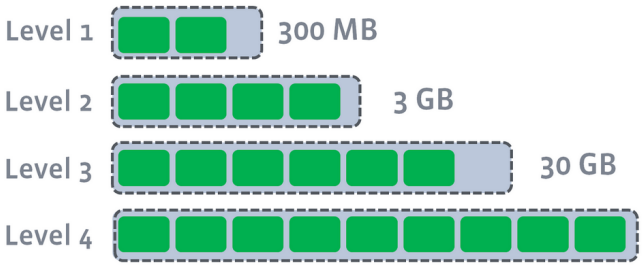

Figure 5: The target size of each level increases by a factor of k

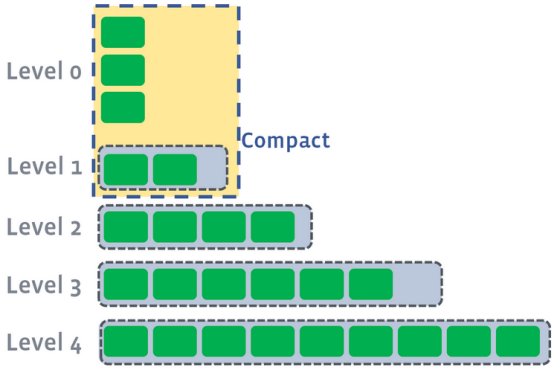

Figure 6: Level Compaction: L0 into L1

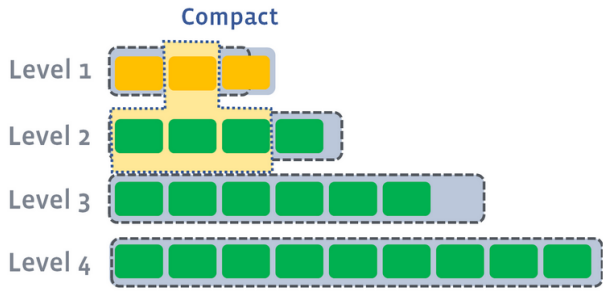

Figure 7: Level Compaction: Li-1 into Li

Compaction starts at the lowest level (level 0) and propagates to the bigger levels as data is compacted and flushed. When the number of files in level 0 (L0) reaches or exceeds the level0_file_num_compaction_trigger, files of L0 will be merged into L1. The process merges the range of data in L0 with the overlapping range of L1. Normally all L0 files are picked for compaction, because data in L0 usually overlaps as seen in Figure 6. L0 is special in that its SSTs may have overlapping key ranges, while the SSTs of higher levels have distinct non-overlapping key ranges [5]. After L0 files are merged with L1, the new data may cause L1 to exceed its target size. In this case, we will pick at least one file from L1 and merge it with the overlapping range of L2 as seen in Figure 7. The specific file picked for merging is not a trivial choice, and is explored more in the later section on compaction priority. After L1 is merged with L2, the size of L2 may now exceed the target size, and the same compaction process would be triggered between L2 and L3. This compaction continues to propagate up the levels of the LSM tree.

2.5 Compaction Priority in Leveled

The SST files of all levels above L0 have distinct non-overlapping key ranges. During level compaction of a non-L0 level, the question of which file should be chosen for compaction arises. Whereas storage engines such as LevelDB select files in a round robin fashion, RocksDB has multiple built in options for picking which file to compact [4]. This choice is determined by the compaction priority tuning parameter. The idea behind compaction priority is to assign a priority to each file on the level to be compacted, and the file with the highest priority is selected for compaction. RocksDB currently offers four built in compaction priority options, and they are kByCompensatedSize, kOldestLargestSeqFirst, kOldestSmallestSeqFirst, and kMinOverlappingRatio.

2.5.1 kByCompensatedSize: If a particular SST file contains a lot of delete markers, iterating over this file will face significant slowdown, as we have to iterate over the deleted keys. Furthermore, the longer these deleted keys live, the more they impact performance. Whereas the sooner we compact these delete markers into the last level, the sooner the disk space is reclaimed. This is the motivation behind kByCompensatedSize. If the number of delete markers in an SST file exceeds the number of inserts, then this file is assigned a higher priority. The more the number of deletes exceed inserts, the higher the priority and the more likely it will be picked for compacted. The optimization is added to avoid the worst performance of space efficiency and query performance when a large percentage of the DB is deleted. kByCompensatedSize is the default compaction priority method, for backward compatibility reasons [4].

2.5.2 kOldestLargestSeqFirst: In certain workloads, a subset of keys are frequently updated, while other key ranges are barely touched. In these scenarios, we want to keep the frequently accessed (hot) keys on the same level if possible. And we want to compact the keys that are infrequently accessed (cold). Keeping hot keys from compacting will improve write and space amplification [4]. This is because as files are compacted from the prior level, most of the keys will overlap, meaning the size of the level will increase minimally. On the other hand, if we compacted the hot keys, then future compactions will have little overlap, leading to an increased level size, which may lead to more compactions. And more compactions generates more writes.

2.5.3 kOldestLargestSeqFirst: is designed for the case where we only update a small subset of keys. This strategy prioritizes files whose latest update is the oldest. As in, if all other files have a more recent update than this file, then this file will be picked for compaction next. Usually, this file corresponds to the file that contains the coldest range.

2.5.4 kOldestSmallestSeqFirst: In kOldestLargestSeqFirst, we considered the case where updates occurred on a small subset of keys. This compaction priority option will address the opposite scenario, where updates are uniformly distributed across the key space. kOldestSmallestSeqFirst will prioritize files whose key range hasn't been compacted to the next level for the longest. This ensures that the distribution of keys stays uniform on each level. If a workload has updates that are random across the key space, then this compaction priority option will improve write amplification.

2.5.5 kMinOverlappingRatio. kMinOverlappingRatio is the recommended compaction priority option, said to optimize write amplification in many cases [6]. This strategy prioritizes files that overlap less between levels. Specifically, it compacts files whose ratio between overlapping size in next level and its size is the smallest. kMinOverlappingRatio also accounts for the number of delete

markers of a file, similar to kByCompensatedSize. Although this strategy is said to optimize write amplification, it is not the default.

2.6 Universal Compaction

Universal compaction, traditionally known as tiered compaction, acts quite differently in how it merges files. We will be using the terms "tiered" and "universal" interchangeably in this paper. Tiered compaction basically minimizes the write amplification at the cost of read and space amplification. In other words, tiered compaction is optimized for write heavy workloads.

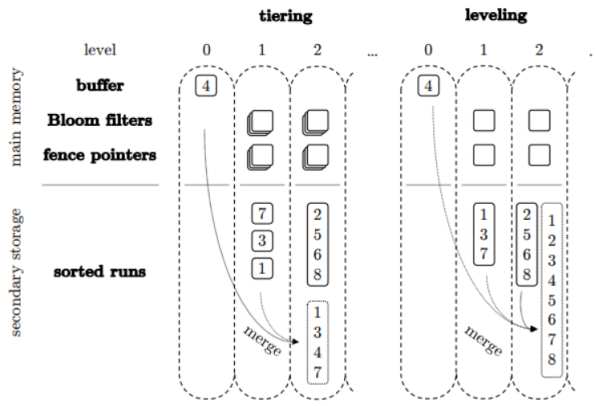

Figure 8: How Leveling and Universal Compaction differ with each other

The key strategy of tiered compaction is that it waits until the level reaches its capacity before merging [3]. With tiering, all runs in the previous level will be merged into a new run and placed to the next level. Unlike leveled compaction, universal compaction does not read or update sorted runs in the level when it got merged into it, which was also shown in Figure 8. As evident, the runs were all saved in the same run of L2 and there is no merging and sorting performed at which time with previous runs [2].

3 ARCHITECTURE

After we defined our problem at hand, which is to develop some potential relations between data throughputs and different compaction parameters, we first want to see how greatly reading and writing performances were affected in different compaction styles after we tune some fundamental parameters. The parameters are: write_buffer_size, max_bytes_for_level_base, target_file_size_base, max_bytes_for_level_multiplier, bloom_bits, and bytes_per_sync.

At first we conducted the benchmarking on our own benchmarking scripts. The scripts allow us to perform write, read, update, and delete to the database and output the number of seconds each function took. The script is also capable of tuning parameters such as compaction style, compaction priority, etc for us to compare the results. As we increased the dataset's size for better output, we ran into some problems that due to time constraint we did not have the time to debug. More of this will be discussed in the result section. In order to resolve this issue, we decided to turn to rocksDB's default benchmarking, named db_bench, where it pretty much included what the benchmark we were trying to do. In addition, we could specify the key and value's size for larger data input.

So how to improve writing performance(write amplification)? One of the ways to buffer writing performance in rocksDB is to reduce the amount of compactions in different levels. According to RocksDB's wiki page, write_buffer_size stated the number of bytes to buffer in memtable before compaction. Max_bytes_for_level_base stated the max amount of bytes rocksdb could store in L1. Target_file_size_base specified the file size at L1. By doubling the size of each 4 parameters listed above from default, it allowed us to read more data to RocksDB and essentially perform less compactions per level in both leveled compaction and Universal Compaction. Furthermore, we changed the max_bytes_for_level_multiplier from 10 to 8. Max_bytes_for_level_multiplier tells the user how many times more data could the next level store than this level. For example, Max byte of L1 is set to be 8 GB of data and the multiplier is set to be 8, L2 is able to store 64 GB, and L3 is able to store 512 GB data, and so on. In the end, we also specified 10 bits of bloom filter bits to each key by passing the bloom_bits parameters, since the default value was not specified in the RocksDB's wiki page. The result of this experiment was discussed in the result section. The bytes_per_sync allowed the operating system to incrementally sync SST files while the data were writing in. In the end, As we mentioned before, since we are now using db_bench, we specified the key size to be 20 bytes and value size to be 400 bytes for larger input data, thus allows the leveling compaction to produce levels more efficiently. Furthermore, we want to see how parameters in compaction, specifically compaction priority, can affect rocksDB performance.

4 RESULTS

In this section, we will be talking about the results we get from our own benchmarking tests and the benchmarking tools that RocksDB implemented called DBbench. First, I should introduce the experimental setup for the benchmarkings. The tests were run in the environment of a sub linux system(centos 7) under windows 10. The windows 10 has the CPU of 8 * Intel Core i7-6700K at 4.00GHz and 16GB of RAM. The rocksDB version we used for the test was 6.8, which was the newest version at the time we computed the experiments.

```
[root@DESKTOP-3L0D8JT rocksdb]# ./rocksdb_test
500000 Writing Take Time: 3.29601s
500000 Reading Take Time: 0.446203s
500000 Update Take Time: 3.47655s
500000 Delete Take Time: 3.79079s
```

Figure 9: The response time of 500K operations

In the beginning stage of the benchmarking, we developed a simple c++ script that allowed us to perform multiple inserts(writes), reads, updates, and deletes to the rocksDB system. Also, we are able to pass different parameters such as compaction style, compaction priorities, and fundamental parameters we discussed in section 3. In the picture 3, it is an example of the response time of each function (Write, Read, Update, and Delete) where the number of key-value

pairs is 500,000, where the type of key is integer and value has the type of string. However, As you can see in Figure 9, Since the dataset is rather small for Leveling compaction to actually produce anything meaningful levels in the database, and the response time between different parameters are rather insignificant for us to observe, our only way is to increase the number of key-value pairs or increase the size of each key-value pair in the hope that we can increase the total amount of data to process.

We then ran into the Segmentation fault (core dumped) problem as we increased the size of key-value pairs, one of the most common bugs that exists in programming languages. There are some potential explanations for this. First, it might be caused by the fact that we try to assign an array of strings for the keys values before we write data to the database, so the array was getting too big for 16GB of RAM. If this is the case, we could simply edit our array from current to a "assembly line" fashion code. We could simply allocate memory for a much smaller array in size and input that in the database. We then can discard this array and free the memory and repeat the process of creating the array and writing to the system. However, this approach is more than likely to corrupt the output response time because the operation is happening as the data writes in RocksDB, so the time for writing potentially is much longer than it is now. Secondly, this could be caused by the fact we are trying to conduct the experiences in the sublinux system of Windows 10, so there is a quite high chance that the sublinux system does not have full access to the system memory hence causing the error. If this is the case, there is not much we could do other than install full linux on the computer and redo the computation(which we did not do because of the time constraint). In the next section, we will be introducing what the result looks like in db_bench.

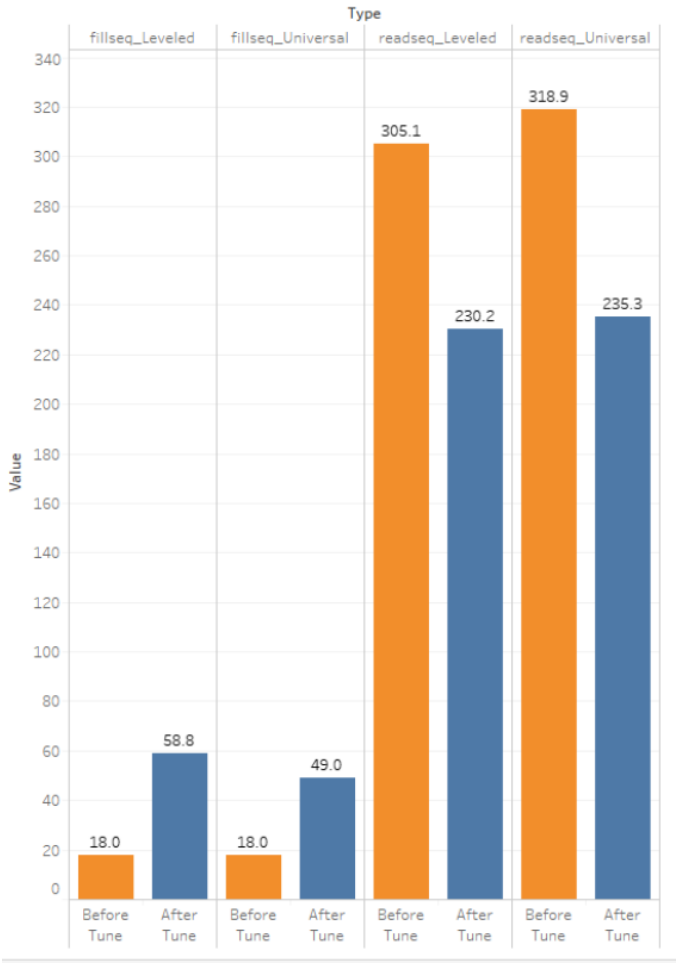

Figure 10: Comparison between before tuning and after tuning in Leveled and Universal Compaction

4.1 Optimizing For Writes

First, In the scenario where write performance is much more needed rather than read performance or space, users are more than happy to trade reading performance for writing performance. As it was stated in Section 3 about different parameters we used for the tuning, we performed 5 tries of 10 million key-value for each compaction style, Leveled and Universal compaction. The results are shown in the below. In the X axis, we have the writing and reading in different compaction styles(from left to right, writing for leveled, writing for Universal, Reading for Leveled, Reading for Universal) as it was marked on the top of the chart. On the y axis, we have the data throughputs, where the unit is MB/s. The chart is also color coded, where in yellow is the reading and writing performance before tuning, and In blue is the reading and writing performance after tuning.

From Figure 10, we can see that after tuning the writing throughputs has been increased to 2 -3 times faster than before tuning, while we lost about 25% of reading throughputs. So we can conclude that the number of compactions are indeed reduced and does contribute to the writing throughputs. On the other hand, we observed that even though both Universal and Leveled Compaction gained significant amount of writing performance, Leveled Compaction actually gained more writing performance while sacrificing less reading performance, which is quite different from our initial thoughts(since Universal is marked as write optimized and leveling is marked as read optimized). There are some potential reasons for this phenomenon. First, since the parameters are targeting leveled, potentially there might be some parameters that are ignored when compaction style is universal. Secondly, since in the rocksDB wiki page, Universal Compaction Style is actually a hybrid compaction of Tiered + leveled, so the parameters we passed in are not as effective as leveled only compaction.

4.2 Analyzing Leveled Compaction Priority vs Universal Compaction

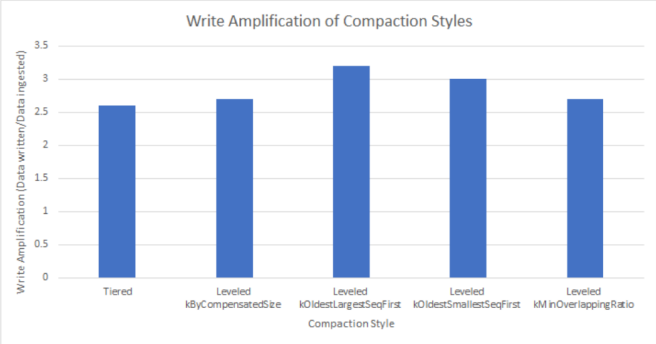

Figure 11: Result for write amplification with different Compaction Priority

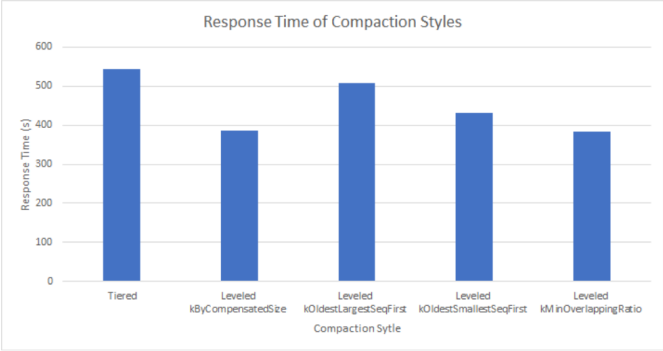

Figure 12: Result for Response Time with different Compaction Priority

In this experiment, we tested leveled compaction against universal compaction, varying the compaction priority for leveled compaction. For our workload, we ran a write-heavy query that inserted data sequentially into the database. We measured both the write amplification and the response time of the workload. Generating, running, and measuring the workload was done through RocksDB's db_bench tool. The results for write amplification can be seen in Figure 11, and the results for response time in Figure 12. In Figure 11, we can see that universal (tiered) compaction achieves the best write amplification among all options, which is expected. However, we note that the improvement in write amplification is not that much. In addition, we see that kOldestSmallestSeqFirst achieves better write amplification and response time than kOldestLargestSeqFirst, which is expected as kOldestSmallestSeqFirst is optimized for updates uniformly distributed on the key space.

The most surprising result is that universal compaction achieves a much worse response time than every level compaction option, despite generally being favored for write heavy workloads. We have two reasons for this result. Firstly, leveled compaction as it is known in RocksDB is a hybrid compaction scheme that uses tiered for the smaller levels and leveled for the larger levels. Furthermore, due to limitations with our machine hardware, we were unable to run extremely big workloads, meaning that our data mainly sat in the lower levels that are managed by tiering. However, this does not explain why true tiered compaction did not perform as well as leveled compaction. We hypothesize that universal compaction in RocksDB has a different implementation than traditional tiered compaction, which leads to worse write performance.

5 CONCLUSION

From our findings, we heavily discourage the use of universal compaction in favor of leveled compaction. Even for write-heavy workloads, we strongly recommend at least starting with the use of leveled compaction with kMinOverlappingRatio as the compaction priority. And instead tuning various parameters we've outlined above. However, we do note that universal compaction does offer slightly better write amplification.

REFERENCES

[1] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), 1009–1024. https://doi.org/doi/pdf/10.1145/3035918.3064029
[2] Mark Callaghan. 2018. *name-that-compaction-algorithm.* https://smalldatum.blogspot.com/2018/08/name-that-compaction-algorithm.html
[3] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), 79–94. https://doi.org/doi/pdf/10.1145/3035918.3064054
[4] Siying Dong. 2016. *Option of Compaction Priority.* https://rocksdb.org/blog/2016/01/29/compaction_pri.html
[5] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Stumm. 2017. Optimizing Space Amplification in RocksDB. *CIDR. Vol. 3. 2017.* (2017).
[6] Facebook. 2020. *Advanced_Options.* https://github.com/facebook/rocksdb/blob/master/include/rocksdb/advanced_options.h
[7] Facebook. 2020. *Choose-Level-Compaction-Files.* https://github.com/facebook/rocksdb/wiki/Choose-Level-Compaction-Files
[8] Facebook. 2020. *Multi-threaded-compactions.* https://github.com/facebook/rocksdb/wiki/RocksDB-Basics#multi-threaded-compactions
[9] Facebook. 2020. *RocksDB.* https://github.com/facebook/rocksdb/
[10] Facebook. 2020. *RocksDB-Tuning-Guide.* https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide#final-thoughts
[11] Facebook. 2020. *Universal-Compaction.* https://github.com/facebook/rocksdb/wiki/Universal-Compaction