Seeking most efficient data cleaning policy for different data-stores workloads

Victor Livernoche
McGill University

Abstract. As we produce increasing amounts of data, write-intensive workloads have become commonplace in database systems. In such write-intensive workloads, the same data items can be frequently modified (i.e., updated), leading to stale versions that need to be cleaned up. RocksDB is an open source database system designed for write-heavy workloads, developed by Facebook. Data cleaning in huge databases is required to have efficient reading and writing performances. RocksDB implements two compaction policies: Leveled and Tiered compaction [9]. The goal of this project is to determine which one is better in a read or write-heavy workload with Uniform and Zipfian (1%) data access distribution. It is known theoretically that Leveled provides better read amplification while Tiered provides better write amplification that should lead to better performance in each respective read-heavy and write-heavy workloads [7]. An efficient compaction policy can save time and money for the owner of the database. Using a slightly modified RocksDB db_bench, we used database microbenchmarks to evaluate both compaction policies. We show that for a Uniform distribution, Leveled is in fact more efficient for a read-heavy workload while Tiered is better for a write-heavy workload. For a Zipfian distribution, there are no significant performance differences between both compaction policies. Our findings further solidify the use of each compaction policy for their respective strength and allow researchers to further the search to find an efficient way to switch the use of each of them in real-time when necessary.

1. Introduction

In the age of modern computing and online services, an efficient database is required to save time and money for the company that uses them. Facebook developed their own database RocksDB with a better write and space amplification than the state-of-the-art [1]. A good data-cleaning policy enables the database to have faster key-value reads and writes while keeping a relatively low space utilization. Facebook generates a few petabytes of data per day [1], an optimization of only a few microseconds per operation could save a decent amount of CPU server operating time, and thus electricity consumption and money. Two compaction policies were originally implemented in RocksDB: Leveled and Tiered. Tiered provides a better write amplification while Leveled has a better read and space amplification [2]. The goal of this project is to find exactly how these compactions affect the efficiency of the database on a write-heavy and a read-heavy workload with a Zipfian or Uniform distribution, two common types of key distribution we can find in benchmarks such as YCSB [3]. The prebuilt benchmark of RocksDB with some modifications was used to get results. These results could help developers to find the compaction policy to use with their database and even, maybe, combine

them with a key and operation analysis to have the most efficient use of an LSM tree database.

We show that Leveled achieves 87.2% lower throughput and 96.6% lower written bytes than Tiered compaction under Uniform write-heavy workload. Under a Uniform read-heavy workload, Leveled has 102.3% higher throughput and a similar 102.3% higher number of written bytes than Tiered. Under a Zipfian distribution, both compaction styles achieved similar performances.

2. Background

We will cover the basic reads and writes operation of a Log-structured merge (LSM) database. Given a key, we want to read or write the value stored in the database associated with it. RocksDB uses this kind of database to keep the key-values on disk [1].

2.1 Log-structured merge databases.

When writing into an LSM database, the key-value is first put into an in-memory table, called the memtable [4]. If the key is already in this table, the value is simply updated. When the memtable has a greater or equal size than the given parameter *write_buffer_size,* the memtable is flushed into an SST file on disk and replaced to a new one [4]. We only append the data to disk, this is why we need to clean the database as multiple SST files can contain the same key but only one is the most recent. It provides an efficient database for writing data. When reading a key, we first check if it is in the current memtable as it always contains the most recent data otherwise, we need to scan through the files on disk to find the most recent update of the key. If the data is never cleaned, then there are more files to search when trying to read a key.

The files on disk are separated into multiple levels of an LSM tree which is not a data structure per se, there is no real relation between each level of the tree other than time. The key-values that are in the memtable are sorted using a red-black tree and then flushed into the LSM tree at level 0 as an SStable file. There are thus multiple SStable accumulating in this level and the same key can be found multiple times in different files if we overwrote a key. An LSM tree compacts the SStables of a level into another at a time chosen by the compaction policy to clean the data. The merges are efficient since the data in each file is sorted. Each subsequent level contains fewer sorted files that were previously merged with another level.

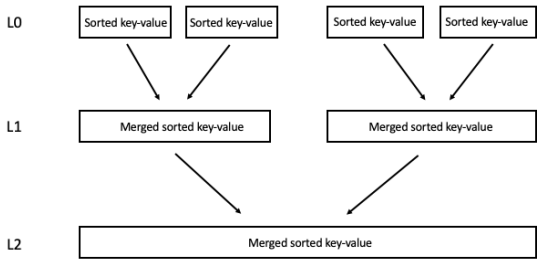

Figure 1: Level merging evolution of LSM tree

In Figure 1, we can see that the sorted files in L0 are merged together into bigger files that contain the new values from L0 while keeping the older values that were not overwritten. Thus the higher the level number is, the older the keys it contains are [2]. To compact, the files are read into memory and merged together with the sorted keys. The newer files overtakes the keys from the older one. Then, a new compacted file is written back to disk. How it is decided when to compact is the subject of this project. We will analyze the two compaction policies implemented in RocksDB that are Leveled and Tiered.

2.2 Read and Write amplification

Read and write amplification are the amount of logical work per read and write operation respectively [5]. For a read operation, we often need to compare keys and search through multiple files. For a write operation, when a memtable is full and L0 also becomes full after the memtable is flushed, the operation takes more time to do as it has to wait for compaction and thus has a higher write amplification. In short, these terms help to describe the relative time a read or write operation should take while not replacing the big-O notation.

2.3 Leveled compaction

The level-0 still holds the SStables that were just flushed from the memtable with this policy. Each upper level contains a definite partition of keys in different files [6]. These files, as they are partitions, do not have overlapping keys. A level contains one sorted run, namely, files that span a timed interval with sorted keys. When the database compacts, it chooses the file in the lower level to merge with the corresponding file with overlapping key range needed [6]. For example, as seen in Figure 2, the first file in L3 could have a key range between 0 and 16 while the second could have a key range between 17 and 33. If we were to compact L2, we would merge the first file of L2 with range 0 to 20 with the two first files of L3. When does the database merge? The Leveled compaction policy defines a variable *level0_file_num_compaction_trigger* that specifies how many files L0 can hold before compacting [6]. When this number of files is reached, an L0 compaction with L1 is triggered. Each level has a target size, if a level has a greater size than the target, a compaction is triggered with the upper level.

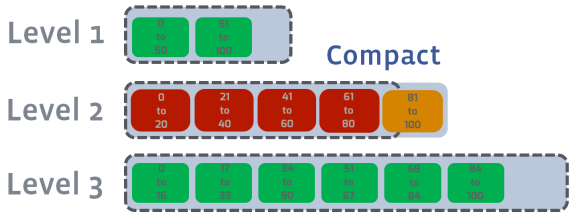

Figure 2: Leveled Compaction target size example [6]

In the example in Figure 2, the target size of L2 has been passed. Thus, a compaction between L2 and L3 will be triggered. Multiple compactions can be done simultaneously, limited by the parameter *max_background_compactions* [6].

2.4 Tiered compaction

The Tiered compaction policy, also called universal compaction in RocksDB, is different from Leveled as a level contains multiple sorted runs instead of one [2]. Each sorted run is spanning the whole key range for a certain timed interval sorted in order of key. All sorted runs are a sequence of non-overlapping time intervals. Compaction only occurs between two or more subsequent sorted runs so that we keep the property that each level has a subsequent time interval [7]. For example, as seen in Figure 3, a run could contain the data from 40 to 44 minutes while another could contain 44 to 46 minutes. When compacting these two sorted runs, the resulting sorted run would then span the time between 40 to 46 minutes and contain the compacted values between the two runs as seen before.

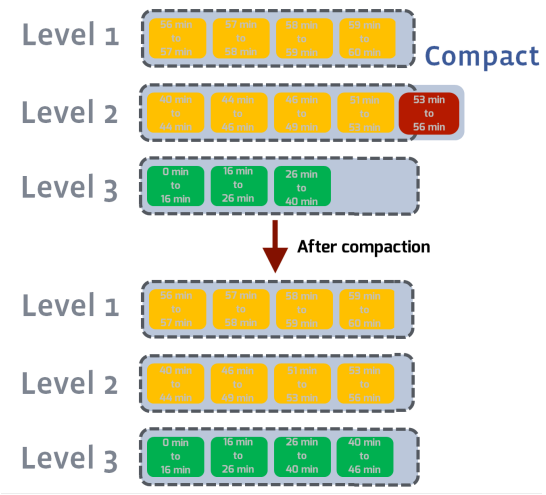

Figure 3: Sorted runs Tiered Compaction example

A compaction is triggered when we have reached the maximum number of sorted runs in a level, the parameter *options.level0_file_num_compaction_trigger* in RocksDB. When a level is compacted, the resulting sorted run is placed in the next level [7]. Once the number of sorted runs is greater or equal to the target parameter, three conditions can trigger a compaction. The first one is by space amplification. The space amplification ratio is estimated using the sum of the size of the sorted runs from 1 to n-1 on the size of the nth sorted run. If this ratio is bigger than the parameter *options.compaction_options_universal.max_size_amplification_percent / 100* all runs are compacted into one sorted run [7]. The second trigger method is by individual size ratio and is used if the first trigger method did not pass the test. The size ratio trigger is calculated from *(100 + options.compaction_options_universal.size_ratio) / 100*. For a level, if the size of R2 on R1 is smaller or equal to the size ratio trigger, then they can be compacted to a single run. If the size of R3 on R2 + R1 is smaller or equal to the trigger, the database adds R3 to the compaction. We do this for each run until it is bigger than the trigger. If the two first methods did not trigger a compaction, whatever runs will try to compact together so that there are less sorted runs in the level than *options.level0_file_num_compaction_trigger* [7]. These compaction methods will be important to understand the difference in time by operation between the two compaction policies. With the sorted runs getting larger and larger, compactions of older runs require more memory and take longer to complete.

3. Contribution

The Uniform distribution is the simplest form of key distribution for a benchmark. With a total of n keys, each key has a probability to get picked of 1/n at each operation. Each key should roughly get picked the same amount of time over a significant number of operations over n. The Zipfian distribution is very useful to use as a key distribution in a benchmark as it simulates the use of some keys a lot more than others. Every key has a rank that gives him a frequency of selection based on the alpha parameter given and the harmonic series. We can thus simulate the 1% rule, a rule of thumb in social media where most of the data comes from 1% of the users [8]. This distribution was added to modify the *readrandomwriterandom* benchmark in RocksDB with a parameter alpha of 0.99 to analyze both compaction policies [9]. A hypothesis was made using the theory in section 2 by looking at the read and write amplification of each compaction algorithm. Leveled should provide a better read amplification to favor a read-heavy workload while Tiered should have a better write amplification such that it performs better in a write-heavy workload. RocksDB was enhanced to give more statistics during the execution of the implemented benchmark, i.e. printing an histogram with read and write times for a specified interval (we used 1 second) as well as the number of written bytes during compaction. In addition, we included a Zipfian distribution to test how

compaction policies perform with it. Note that the write-heavy workload used in the benchmark had 50% writes for 50% reads while the read-heavy workload had 95% reads for 5% writes which is consistent with the widely used YCSB benchmark [3]. Each test was made multiple times on a 6 cores 3.2 GHz CPU to find the best parameters and the final results came from a one-hour benchmark test with 100,000 different keys unless said otherwise. It should also be said that when the benchmark was run on a much higher number of keys, the time per operation increases drastically. For a billion keys, many more keys than the total number of operations, the average reading time iss 393.2457 µs for Leveled compared to 382.9171 µs for Tiered for a write-heavy workload while it is 27.7755 µs and 21.1946 µs for a hundred thousand keys. That is about 18 times slower.

3.1 Write heavy Uniform distribution

We can see in Figure 4 and 5 below the 99th percentile of the read and write operations in a 1-second interval for a write-heavy workload with a Uniform distribution.

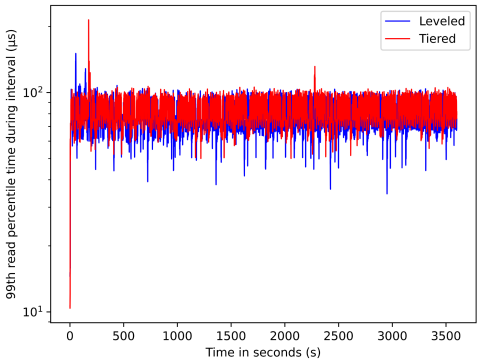

Figure 4: 99th percentile read times of 1-second intervals with Tiered and Leveled compaction with a Uniform write-heavy workload

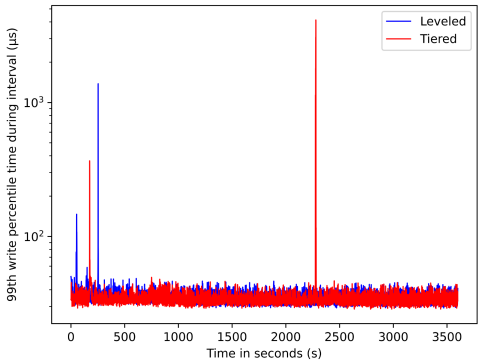

Figure 5: 99th percentile write times of 1-second intervals with Tiered and Leveled compaction with a Uniform write-heavy workload

Figure 6 represents the number of operations per second with the same workload. Notice the two phases for both compaction styles, many more slow spikes at first for Tiered while it is after for Leveled.

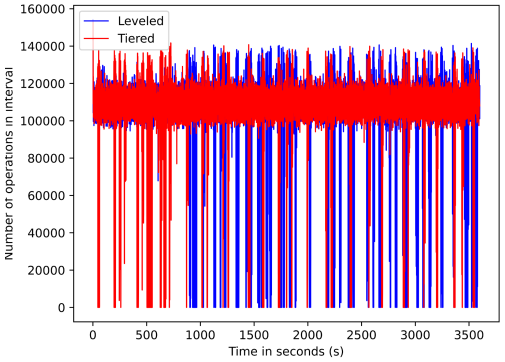

Figure 6: Number of operations per 1-second interval with Tiered and Leveled compaction with a Uniform write-heavy workload

Figure 7 represents the number of bytes written to the database while compacting. This is useful to see how much time is spent compacting and how whether or not the database can continue writing (no stall).

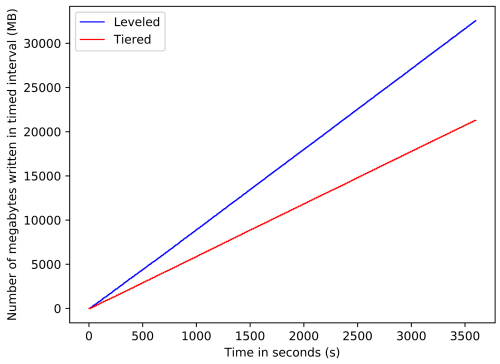

Figure 7: Total number of bytes written during compaction for Tiered and Leveled with a Uniform write-heavy workload

Leveled has a read times average of 17.3701 µs for Leveled and 18.1649 µs for Tiered and the write times average of 27.7755 µs and 21.1946 µs. While, at first, the final results read time average might not give an advantage to Tiered, the write average latency gives a significant edge for Tiered. This can give a good insight on which compaction is better for this workload but it is clear that looking at the 99th percentile can be helpful as it shows which one can have the worst-case performance. The global read 99th percentile for Leveled is 75.25 µs while Tiered is 82.92 µs while write for Leveled is 33.45 µs and Tiered is 33.25 µs.

It is unclear from Figure 4 that Leveled has worse performance than Tiered for reading times while the writing times are similar. We can see spikes at the same time in both Figures that are due from stalls of the database to allow compaction. The benchmark test used options that reduced the stalls the most so that we would have clearer results but in those runs, there were stalls for approximately 11% of the time for Leveled and 7% for Tiered. These stalls are definitely something to take into account when analyzing the results. Leveled compacts more and can spend more time compacting. This explains the slower writing times while the readings are faster for the same reason (fewer files to search a key). Tiered can have more compactions and thus more stalls if we decrease the parameter *level0_file_num_compaction_trigger* or *level0_slowdown_writes_trigger*. Both were tunes to have as few stalls as possible while keeping a decent reading time. The writing times should always be similar for both compaction policies if there are no stalls since a write does not depend on the configuration of the database other than stalls, this Figure 5 is there to show this. In the next sections, there are no stalls so we can ignore the writing times.

The total number of operations further shows the advantage of Tiered compaction for this write-heavy workload with a total of 115% more throughput. There are much more intervals in which Leveled compaction makes fewer operations than Tiered, especially in the end where the database is getting bigger and many writes are stalled by the increasing number of compactions. The many spikes we can see are mostly due to the threads compacting with stalls. It is still overall consistent in the number of operations with approximately 110,000 operations/second.

As expected, the total number of compacted bytes is much bigger for Leveled. In Leveled compaction, a compaction does not necessarily compact a smaller file into a file close to the final sorted run which means that it contributes less to this final sorted run in which it is easier to find the wanted key when reading. The advantage of Leveled in reading times is what was expected since in Tiered compaction, when we compact as seen in section 2, a file is compacted to a bigger sorted run. This final sorted run contains most of the data such that every compaction makes the data

closer to the final sorted run. There is one final thing to note: Leveled compaction created a database more than twice as small as Tiered with 240.40 MB versus 522.25 MB. The more efficient Tiered compaction with this workload might not be the way to go if space is an issue.

3.2 Read heavy Uniform distribution

With no stalls, there is no point showing the writing times since they are the same for both compaction styles. Figure 8 represents the 99th percentile reading times for a read-heavy workload.

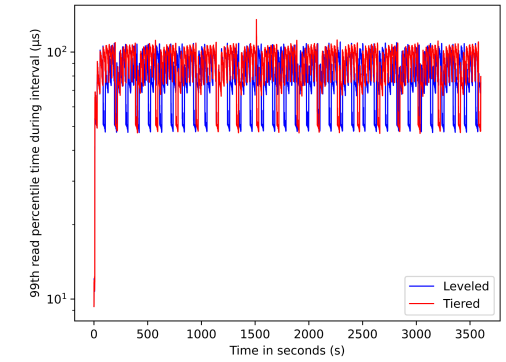

Figure 8: 99th percentile read times of 1-second intervals with Tiered and Leveled compaction with a Uniform read-heavy workload

Figure 9 shows the number of operations per second for both compaction policies with the same workload.

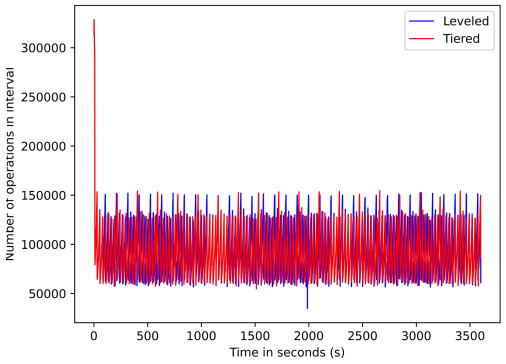

Figure 9: Number of operations per 1-second interval with Tiered and Leveled compaction with a Uniform read-heavy workload

Figure 10 is the number of bytes written during compaction.

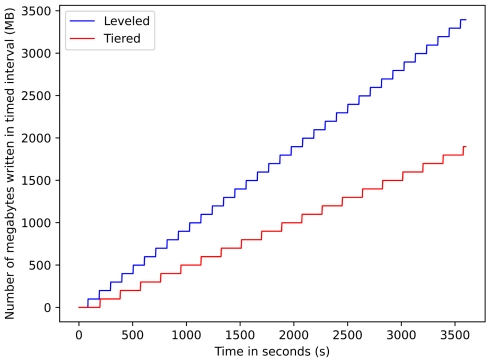

Figure 10: Total number of bytes written during compaction for Tiered and Leveled with a Uniform read-heavy workload

The read-heavy workload, as in all tests made, started with an empty database which particularly makes a difference on this workload since most of the reads at the beginning of the benchmark test did not find any value in the database. It thus is slower since it needs to check many files. It is not a bad thing because it simulates the worst-case scenario in which we are quite interested. The global reading time average is 21.6315 µs for Leveled and 22.1360 µs for Tiered. The global 99th percentile reading time is 78.47 µs for Leveled and 89.14 µs for Tiered. These first results show that reading times are better for Leveled compaction as expected since it has a lower read amplification.

The 99th percentile reading times show how Leveled compaction has better performance than Tiered with this read-heavy workload. We see much faster spikes of 99th percentile reading times for this compaction policy. However, the number of operations is not significantly larger for Leveled with 102,2% more throughput and Figure 9 does not give any indication that there could be a major

problem somewhere other than that small spike around 2000 seconds.

In Figure 10, the number of compaction is greater for Leveled as Tiered waits more before compacting and has linear steps which shows that the number of written values in the database were constant in time. This time, Leveled had a final database size of 240.73 MB while Tiered had 170.32 MB. Tiered managed this time to use less space in the database.

The overall slightly better time performance of Leveled over Tiered in a read-heavy workload was in fact expected since Leveled is compacting more often (see Section 2) and thus, when reading often, there are slightly fewer files to search into in general to find a key.

3.3 Write heavy Zipfian distribution

We first have in Figure 11 the 99th percentile reading time for a write-heavy workload with a Zipfian distribution. Figure 12 represents the number of operations per seconds for both compaction policies with this workload.

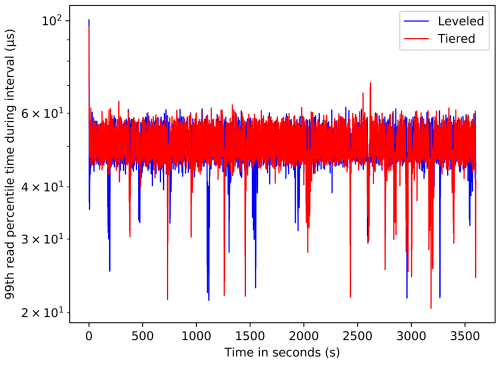

Figure 11: 99th percentile read times of 1-second intervals with Tiered and Leveled compaction with a Zipfian write-heavy workload

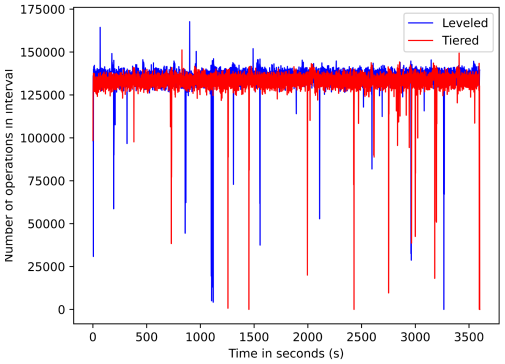

Figure 12: Number of operations per 1-second interval with Tiered and Leveled compaction with a Zipfian write-heavy workload

Figure 13 shows the number of bytes written during compaction for each compaction algorithm. Notice that Tiered is this time over Leveled, with more written bytes while compacting.

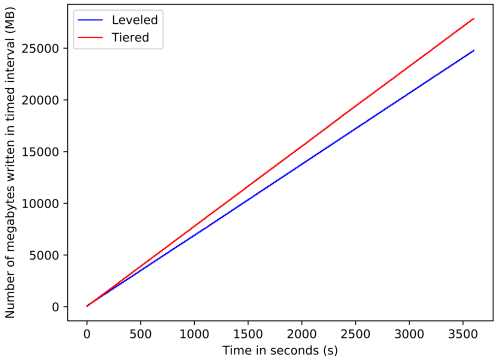

Figure 13: Total number of bytes written during compaction for Tiered and Leveled with a Zipfian write-heavy workload

The Zipfian distribution benchmark aimed to replicate the 1% rule with an alpha parameter of 0.99. The tests were this time made on an existing database run with Uniform distribution first so that there would not be too many misses for the keys not part of the 1%. The average read time for Leveled is 10.9416 µs with a 99th percentile of 49.45 µs and an average of 11.0471 µs for Tiered with 99th percentile of 49.59 µs.

From Figure 11 and 11, the global average, and the 99th percentile, it is hard to see any difference in performance between the two compaction policies for a Zipfian distribution with this workload. We can easily explain this lack of difference by the fact that many operations are hitting in either the memtable or L0. Since the picked keys are often the same, there is often no compaction needed to quickly find a key. We have some stats to back this up. For the Leveled test, there were about 7 times more memtable hits than misses and 19.4 times more L0 hits than L1. Similarly for Tiered, there were about 7 times more memtable hits than misses and 16 times more L0 hits than L6 hits.

Figure 13 brought up an interesting and unexpected difference between the compaction in a Zipfian and a Uniform distributed database. This time, Tiered surprisingly compacted more than Leveled. A possible explanation could be how the Tiered compaction is picked. Recall that there are three different compaction picking methods once the number of L0 files exceeds the target parameter. All three of them do not care about the content of the files, Tiered will compact only based on the size of the sorted or the number of sorted runs. Leveled, on the contrary, compacts based on the size of the level. Thus L1 would really need to get compacted since the sorted runs from L0 compacted to L1 contain almost always the same keys. Hence Leveled will almost only need to compact from L0 to L1 while Tiered acts normally.

3.4 Read heavy Zipfian distribution

We have in Figure 14 the 99th percentile reading time for a read-heavy workload with a Zipfian distribution. Again, the writing times are not shown since there are no stalls. Figure 15 represents the number of operations per seconds for both compaction policies with this workload.

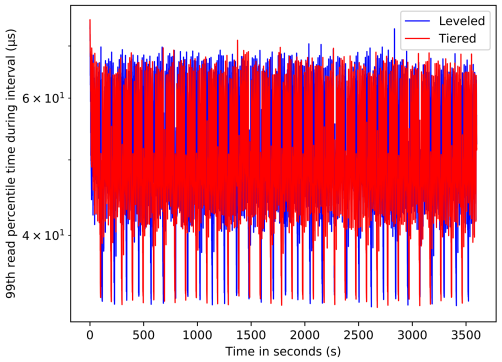

Figure 14: 99th percentile read times of 1-second intervals with Tiered and Leveled compaction with a Zipfian read-heavy workload

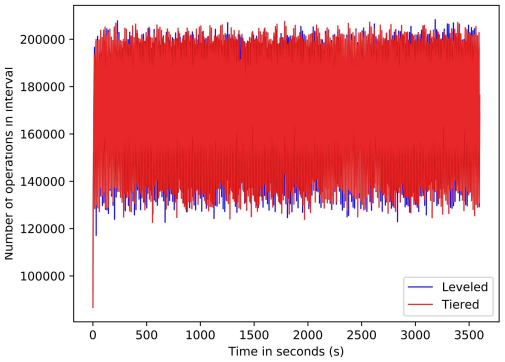

Figure 14: Number of operations per 1-second interval with Tiered and Leveled compaction with a Zipfian read-heavy workload

Figure 15 presents the number of bytes written when compacting for both compaction policies.

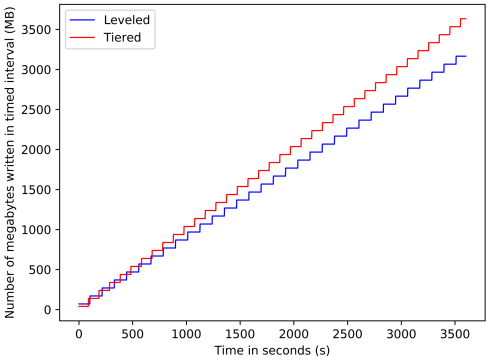

Figure 16: Total number of bytes written during compaction for Tiered and Leveled with a Zipfian read-heavy workload

This workload with a Zipfian distribution is hardly more interesting than the last one. The results are similar. We have an average reading time of 11.3134 µs for Leveled with a 99th percentile of 50.26 µs while Tiered has an average reading of 11.2634 µs and 99th percentile of 49.91 µs.

These two last figures do not give any performance advantage to one compaction style over the other. They both are really similar. This was to be expected knowing the results of the write-heavy workload. There is the same amount of hits and misses in the memtable and the lower L0 level.

There are not many compactions this time since there were not that many writes. Again, for the same reason as in the write-heavy, Tiered compacted more than Leveled. The results of the Zipfian distribution are not conclusive in terms of time performance. Nonetheless, Tiered compacted more and used, therefore, smaller storage space which would normally mean that reading should be a bit faster. On a larger scale of orders of terabytes, this might make a little difference that can not be shown with a small database. More tests would be required to determine the best compaction policy for a Zipfian distribution on a large database.

4. Discussion

All benchmark tests were made on a relatively small database and thus the gain made in time by using the right compaction policy for a workload was not as big as it could have been. Tests that were made with a billion keys often led to almost 50 µs in average time per operation saved. This is a huge boost in performance for big databases. The main points to take away from this project are which compaction policies are the best to use for the tested workloads if only time performances are in mind:

- Use Leveled for Uniform read-heavy workloads
- Use Tiered for Uniform write-heavy workloads
- Use either Leveled or Tiered for Zipfian distributed workloads

One thing to remember from the tests is that better time performances often cost storage space. This is clear when we compare the final size of the database for both compaction policies after each test with a Uniform distribution. Also, when trying to find the best parameter to use, with fewer stalls and more operations/second, increasing the buffer size always made better benchmark test results. This is a trade-off to think about before implementing a database in a system especially when smaller space storage is available. The bigger space amplification of Tiered compaction is also to take into account when choosing the policy.

5. Future work

Further work could be done by creating a database analyzer that checks which kind of distribution it is and the read/write percentage. This could even be a real-time analyzer to switch compaction policy when needed. For example, in an interval of one minute, the database could have made 90% reads such that Leveled compaction is picked and in the next minute, 60% could be writing operations such that the database switches to Tiered compaction. Some problems could come out of this since the levels are structured differently in the two compaction styles. Using one after the other is definitely possible, the structure will be changed over time, but this might affect performance. This is where new tests would be very useful.

There was already work done on a compaction policy that uses Leveled and Tiered together. It is actually implemented in some databases with Leveled in higher levels while Tiered is used in lower levels [10]. This compaction algorithm gives a mix of performance, with lower write amplification than Leveled and lower space amplification than Tiered [10]. While this method can be very useful to get other performance, it is an in-between solution. This is not the same as completely switching between compaction policies. One way it could be done is by merging all sorted runs in all levels but L0 together before switching to Leveled from Tiered. With enough compaction threads, this might be possible without slowing down too much the database but it definitely would stall the database on bigger levels. Again, testing would be required. There is also Leveled-N compaction that provides less write amplification but more read amplification by allowing N sorted runs per level [6]. These two other compaction policies could be subject to the same tests made in this project with maybe more read/write percentage to be tested and with more distributions. This would allow us to have a clear picture of which compaction policy is the best in which situation. We already know now that Leveled is better in read-heavy workloads while Tiered is better in write-heavy workloads. The Tiered+Leveled and Leveled-N would probably fall between the two in terms of performances which could be a great alternative when a developer is unsure what kind of workload their database will face. Either way, this project gave a good start in the search for the best usage of compaction algorithms. An automatic performance tuning algorithm would be the ultimate goal to provide the best performance in a database.

6. Conclusion

The goal of the project was to find in which database workload with different distributions the Leveled and Compaction policies perform the best. The theory and the benchmark tests are clear, in a uniform key distribution Leveled is more efficient in a read-heavy workload while Tiered is better in a write-heavy workload. There is no significant performance advantage on any of the compaction styles for a Zipfian distribution with an alpha parameter of 0.99. This will help developers choose which compaction algorithm to use for their database and, hopefully, allow more research to be done to provide the most efficient data-cleaning policy for an LSM tree-based database.

7. References

[1] Y. Matsunobu, S. Dong, H. Lee (2020). MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph.

[2] RocksDB Wiki. Compaction. https://github.com/facebook/rocksdb/wiki/Compaction

[3] B.F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears (2013). Benchmarking Cloud Serving Systems with YCSB. https://courses.cs.duke.edu//fall13/cps296.4/838-CloudPapers/ycsb.pdf

[4] RocksDB Wiki. MemTable. https://github.com/facebook/rocksdb/wiki/MemTable

[5] M. Callaghan (2015). Read, write & space amplification - pick 2. Small Datum. http://smalldatum.blogspot.com/2015/11/read-write-space-amplification-pick-2_23.html

[6] RocksDB Wiki. Leveled Compaction. https://github.com/facebook/rocksdb/wiki/Leveled-Compaction

[7] RocksDB Wiki. Universal Compaction. https://github.com/facebook/rocksdb/wiki/Universal-Compaction

[8] B. Carron-Arthur, J.A. Cunningham, K.M. Griffiths (2014). Describing the distribution of engagement in an Internet support group by post frequency: A comparison of the 90–9–1 Principle and Zipf's Law. Internet Interventions. 1 (4): 165–168

[9] RocksDB. https://github.com/facebook/rocksdb

[10] M. Callaghan (2018). Name that compaction algorithm. Small Datum. https://smalldatum.blogspot.com/2018/08/name-that-compaction-algorithm.html