NTNU
Norwegian University of
Science and Technology

Auto-tuning RocksDB

Hans-Wilhelm Kirsch Warlo

Master of Science in Computer Science
Submission date: June 2018
Supervisor: Svein Erik Bratsberg, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

RocksDB is one of the most widely used embeddable persistent key-value stores available open-source. Its configurability, performance and workload flexibility have been essential factors that differentiate it from contenders. The data structure, Log Structured Merge Trees (LSM-trees), used in RocksDB differs from the more traditional B+ tree especially by offering better write throughput. However, the LSM-trees themselves do not provide a full-grown solution to all workloads, hence why there exist so many different databases implementing their own versions of the data structure.

The amount of data being handled is only increasing, and this makes a case for write-optimised databases. Enduring high write load has been an issue for engineers over many years, and there are numerous examples of ticket sale servers crashing when opening popular events. Distribution techniques and horizontal database scaling is the regular way to mitigate these issues today, however being able to handle more writes per node could be very efficient in terms of resources required.

Auto-tuning databases is in the wind, with examples like Oracle Autonomous Database and Peloton offering next to no configuration. RocksDB has also recently received tuning features like dynamically changing level sizes for Leveled Compaction[1] and an auto-tuning rate limiter[2]. However, RocksDB is known for dominating background activity by default and requires configuration for optimal performance for different workloads. This thesis evaluates an implementation of a compaction auto-tuner for RocksDB and presenting positive write performance gains during high write load. The research did also attract positive attention from the RocksDB developers at Facebook.

Sammendrag

RocksDB er en av de mest brukte innbyggbare persistente nøkkel-verdi datalager som er tilgjengelig i åpen-kildekode. Konfigureringsevnen, ytelsen og lastfleksibiliteten er det som skiller RocksDB fra konkurrentene. Datastrukturen, Log Structured Merge-trees (LSM-trær), brukt i RocksDB skiller seg fra de mer tradisjonelle B+ trærne ved å tilby høyere skriveytelse. Men LSM-trær er en datastruktur som ikke løser alle lastsituasjoner av seg selv, og kan tilpasses for å oppnå optimal ytelse. Dette er grunnen til at det finnes så mange databaser som implementerer deres egen versjon av LSM-treet.

Datamengdene som håndteres er økende, og behovet for skrive-optimaliserte databaser kommer deretter. Å tåle høy skrivelast har vært en utfordring for ingeniører i mange år, og det er utallige eksempler på billettsystemer som har krasjet ved åpning av populære arrangementer. Distribueringsteknikker og horisontal skalering av databaser er den vanlige måten å håndtere dette problemet i dag, men å kunne tåle flere skrivinger per node kunne vært veldig effektivt og redusere behovet for systemresursser.

Automatisk justerende databaser er i vinden, med eksempler som Oracle Autonomous Database og Peloton tilbyr ytelse med nærmest ingen konfigurasjon. RocksDB har også nylig fått funksjoner som dynamisk forandrende nivå-størrelser i Leveled Compaction[1] og en automatisk innstillende skrivebegrenser (rate limiter)[2]. På en annen side er RocksDB kjent for å ha dominerende bakgrunnsaktivitet som standard, og krever derfor konfigurasjon for optimal ytelse for forskjellige lasttyper. Denne avhandlingen evaluerer en implementasjon av automatisk justering av vedlikeholdsoperasjoner (compactions) for RocksDB, og presenterer positive resultater av skriveytelse under høy last. Forskningen vakte også positiv oppmerksomhet blant RocksDB utviklerne i Facebook.

Preface

This Master's Thesis is written for the Department of Computer Science (IDI) at Norwegian University of Science and Technology. The research is conducted by Hans-Wilhelm Kirsch Warlo, under the supervision of Professor Svein Erik Bratsberg.

I would like to thank Svein Erik Bratsberg for his helpful assistance. Svein Erik's support, technical advice, and feedback on the project report have been incredibly valuable. Throughout the project, he showed great interest in my research which has been an especially motivating factor.

Lastly, I would also like to thank the RocksDB developers and community for their support and feedback on the developer forum.

Contents

Chapter 1

Introduction

In this chapter we introduce the motivation behind the project, present research goals and discuss potential implications of an successful outcome. Lastly we summarise the structure of the thesis.

1.1 Purpose

The motive for the research is to enable systems to utilise system resources more efficiently for write-intense workloads. Companies like Facebook experience tremendous load, and utilising system resources more efficiently could mean major cost reductions. Furthermore, single node systems are especially prone to load issues. An example is ticket sales systems when releasing tickets for popular events, these systems surprisingly often tend to time out. Companies deploy distribution techniques and horizontal scaling[3] of their systems to handle the load; however, this can be costly and challenging to implement.

Automatically configuring databases is in the wind[4], examples like Peloton[5] and Oracle Autonomous Database[6] optimise themselves using different techniques like Artificial Intelligence. However there are many factors to evaluate, and different workloads benefit from different configurations. Researching configurations to improve databases' ability to handle higher write load could be valuable for several use-cases.

Databases implementing the data-structure LSM-tree (Section 2.3) performs maintenance tasks called *Compactions*. These maintenance tasks improve space usage and read performance at the cost of having to do extra writes. The idea is that by disabling these compactions when the load is high we can improve the write performance, and by enabling them when the load is low we could compact the database when there are resources available. If able to handle this automatically it could exploit the best of both worlds, and be especially applicable for periodic write-heavy workloads. Though, under the assumption that space usage and read performance is not critical during a period of high write load. Nevertheless, the database would reach similar performance after a period of low load.

1.2 Research Goals

G1: Make RocksDB able to disable and enable compactions automatically.

By making RocksDB able to toggle compactions automatically, we intend that it should be able to disable compactions based on I/O or other statistics to achieve higher insertion rate. Additionally, it should enable compactions when it is no longer required.

G2: Increase RocksDB's write performance and provide a tuning baseline for other LSM-based databases.

G2 intends to determine how RocksDB might be able to improve write performance using compaction tuning and other compaction related parameters.

RQ1: How can we implement a compaction auto-tuner that dynamically toggles compaction, how does it benefit RocksDB and at what cost does it come?

Answering **RQ1** will enable us to reach **G1** by implementing a compaction auto-tuner in RocksDB, and hopefully lead to **G2** through evaluation and optimisation of the implementation. Answering **RQ1** could address **G2** by providing a baseline and proof-of-concept of auto-tuning compactions. The result could turn out to be an interesting contribution to database research in general.

1.3 Thesis structure

The thesis is structured into three main parts: The background chapter introduces RocksDB, features of its underlying data structure, and terms related to performance evaluation. The implementation chapter presents the hypothesis, explores the proof-of-concept implementation and optimising its configuration using an experimental methodology. Benchmarks and results are presented in the Benchmark chapter with multiple configurations. Lastly, we sum up the research goals, present a conclusion for the project and discuss future work.

Chapter 2

Background

This chapter focuses primarily on providing background knowledge related to the data structure behind RocksDB, as well as discussing RocksDB's features. Parts of this chapter is taken from my previous specialisation project[7].

2.1 NoSQL

NoSQL, non SQL, is a generic term that comprises a series of database systems that are modelled in other means than relational databases. Especially through the development of distributed systems, many NoSQL databases have proved to be suitable. The fact that one can horizontally scale a system by just adding another node is valuable. Moreover, since one does not have to throw expensive hardware at a single server, it becomes cheaper to build scalable systems. However, NoSQL itself does not exclusively promote distributed systems, systems like Redis[8], LevelDB[9] and RocksDB[10] are examples of NoSQL without any included distribution abilities. At later stages in NoSQL's existence it has by some been referred to as "Not only SQL", since the concepts and databases are more commonly being integrated into relational database systems and supporting SQL-queries, e.g. MyRocks[11].

2.2 RocksDB

RocksDB[10] is a database system created at Facebook, which was created as a replacement and improvement over LevelDB. It is based on LevelDB[9], but given the differences that have been introduced – it is considered a fork. The primary focus of RocksDB has been performance, specifically for low latency flash storage and it is written entirely in C++. However, given its modularity it provides many configuration possibilities that can influence the performance in different scenarios. RocksDB is a lot more featured and tailored than LevelDB, and thus the general performance is drastically improved. This encompasses multithreading, bloom filters, lock optimisations, modular data structures, and tunability.

2.3 LSM-trees

A Log Structured Merge-tree[12] is a file-organisation that offers a lot higher throughput for write-intensive tasks compared to others, like B+-trees. The structure of the tree maintains key-value pairs, but differs from other key-value structures by batching them and optimising the batches for an underlying storage medium. The LSM-tree exploits the fact that, in current storage hardware, sequential I/O is a lot cheaper than random I/O operations. This is done by sorting and batching data streams in memory before writing them sequentially to disk. The fact that sequential I/O is cheaper than random I/O holds for flash storage as well. So even with flash storage becoming a more viable data-centre storage medium, LSM-trees provides a well-suited file structure. Also, the write throughput for flash storage is order-of-magnitude higher than alternatives and optimisations have been done in RocksDB to exploit such storage mediums even better.

2.3.1 SSTables and Memtables

The first part of the LSM-tree structure is sorting and storing data as so-called SSTables. An SSTable, Sorted String Table, is just a file with key-value pairs sorted by the key. Initially, the data is written to memory in "Memtables", and when they exceed a given size, they are marked as immutable and called an SSTable. When the memory starts to fill up, the in-memory SSTables are "flushed" to a lower storage layer. Meaning that they are written to bigger and cheaper storage devices, and it is in these flushes that the LSM-structure allows for sequential writes to disk. This structure provides an O(1) worst case write time due to the sort-append nature in memory, but also an O(N) worst case read time.

2.3.2 Compaction

As mentioned above, the SSTables are marked as immutable. This implies that one cannot directly update a key-value pair directly, and could pose a storage problem if one were to store the all the old data, the LSM structure handles this issue by doing so-called compactions. Compaction merges multiple SSTables into single, bigger, SSTables, skipping all the overwritten data to avoid storing old entries. It is important that this process performed efficiently because there is a trade-off between achieving continuously high read and write throughput and space efficiency. By having well-compacted SSTables, reads will be faster, since there are fewer entries to search for. To achieve high write throughput, the compaction process should avoid too much I/O that throttles incoming writes. Lucky enough, the LSM structure blesses us with sorted data, allowing for sequential reads and it is therefore not required to load the entire SSTable into memory during compactions.

There have been created multiple compaction strategies that are customised for different amplification factors. Amplification factors for databases are described in more detail in section 2.4. Below we will give a brief explanation of the most known compaction strategies.

Leveled Compaction

The default compaction strategy in RocksDB is Leveled Compaction. Leveled Compaction organises the files in multiple levels usually with increasing sizes (Fig. 2.2). It guarantees that each SSTable in a level is non-overlapping (except level-0), meaning that it never exists both an old and an updated entry in the same level. The SSTables in each level is also always sorted by keys, and this allow binary search to be used to determine the position of a specific key quickly. This ability massively increases the read performance; however, it comes at a write amplification cost.

Compactions are a necessary evil in LSM-based databases; and as mentioned, it is required to ensure that the read performance is satisfying. Since the files in the level-0 have overlapping key-ranges, the database has to do linear seeks when reading data, instead of utilising the level hierarchy. This makes the level-0 → level-1 compaction especially important, but is quite heavy when there are many level-0 files to compact. The level-0 → level-1 compaction cannot be parallelised by default in RocksDB, but the flag `max_subcompactions` might be set to make RocksDB try to partition and execute it in parallel.

Level-0 → Level-0 compaction is a clever compaction method that allows reduction of available Level-0 files when other compaction threads have locked partitions of files for other compactions like Level-0 → Level-1. This improves the read amplification (Section 2.4) while sustaining burst of writes since it reduces the amount of linear file-seeks required in level-0. The RocksDB team wrote a blog post concerning this feature at blog.rocksdb.org[13].

The merging-strategy works by taking one or more SSTables from a full level and merging it with SSTables that matches the key range. The compaction then merges all the SSTables involved into multiple new ones, to ensure that the key-range is always sorted. This SSTable key-range matching is illustrated in Figure 2.1, where an SSTable from Level-2 is getting compacted into Level-3. After that merge, four SSTables are written to Level-3.

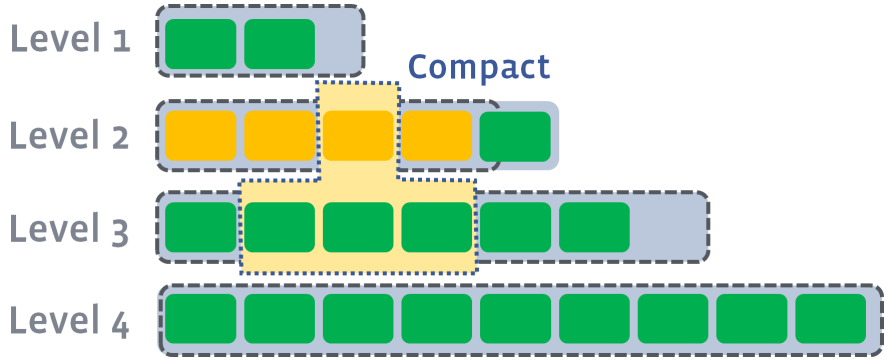

Figure 2.1: Leveled Compaction: Matching key-range strategy [14]

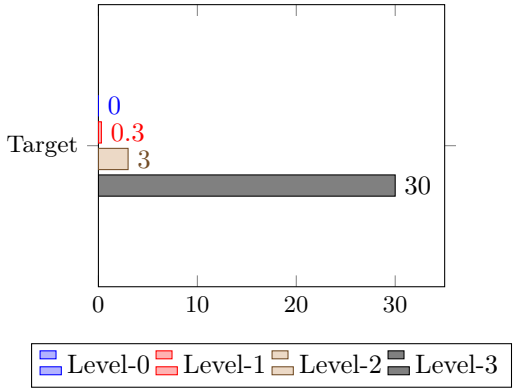

Figure 2.2: Leveled Compaction hierarchy default target sizes in GB

Universal Compaction

Universal Compaction, also called Size-Tiered Compaction, is a compaction strategy created to improve performance for write-intense workloads. In contrast to LevelDB and RocksDB, Universal Compaction is the default compaction strategy for Apache Cassandra[15]. To improve the write-performance this strategy avoids the costly key-range compaction that is executed in Leveled Compaction – and lets each SSTable contain the full key-range instead. The SSTables are still sorted in the compaction runs but avoid overlaps in time-ranges. This is done by merging SSTables that are adjacent in regards of time, and the output is a single sorted SSTable that does not overlap any of the other SSTables time-range.

This time-range compaction provides better write performance, but also limitations. The space used can percentage wise become much higher than the database actually is. E.g. when the space used is above 125% of the actual data; one might configure a trigger a full compaction to reduce the size used. During that full compaction, one has to store two database copies while merging this is called the "Double Size Issue". This issue implies that the disk has to have a lot of free space available to allow this compaction. Additionally, universal compaction may experience problems in databases, like in RocksDB if one chooses to use a single level. Then a compaction may potentially result in a single big SSTable, and RocksDB does not support file sizes above 4GB due to uint32 size limit[16].

FIFO Compaction

RocksDB implements a FIFO-compaction strategy, which differs from leveled and universal in use case. The FIFO style deletes the oldest entry periodically, which means it might delete data without a user's interaction. By default, FIFO does only use a single level, L0, and reads becomes slow if the number of L0 files

increases. Due to the files eventually being deleted, the number of files allowed in L0 affect the Time-To-Live (TTL) for the data proportionally. In RocksDB, among others, the number of files allowed is a configurable option to enable shorter or longer TTL times. The strategy requires little overhead, and given this property – it is best suited for storing log data and similar. It is recommended to use the FIFO strategy with caution since files might be deleted and read times might become slow.

2.4 Amplification Factors

When constructing or configuring databases, one often considers the amplification factors instead of only benchmarks themselves – since benchmarks often tend to show that they are better than the competition. The amplification factors describe the number of operations that the disk would have to process to satisfy queries, and thus also affects the lifetime expectancy and speed requirements of the storage devices used. In a real-life situation, the prices of the storage is also an important factor, because high speed and endurance flash are expensive.

2.4.1 Read Amplification

Read Amplification is the multiple of disc reads that are required to carry out a query. If a user queries a page, and the disk has to read three pages to satisfy it – the read amplification is 3. This also includes the reads that hit the cache, since they also affect the performance.

$$RA = \text{number of queries} \cdot \text{disc reads} \tag{2.1}$$

2.4.2 Write Amplification

Write Amplification is the ratio between data written to disk, and data written to the database. If the data written to disk is twice the data written to the database, the write amplification factor is 2. Having a high write amplification is especially disk wearing. Also, if the disk's throughput is 500MB/s and the write amplification is 10, the maximum throughput to the database is 50MB/s. The write amplification is, therefore, an especially important factor for write-intense workloads.

$$WA = \frac{\text{data written to disc}}{\text{data written to database}} \tag{2.2}$$

2.4.3 Space Amplification

Space Amplification denotes the ratio between the size of the database files and the size of the data these files use on disk. A 100MB database might use 150MB

on disk. This amplification is often due to old versions are stored and not removed, as well as metadata like transaction ids. E.g. InnoDB allocates 13bytes per row for metadata, even though none may exist. Compression, on the other hand, reduces space amplification which makes it an important factor to utilise the hardware resources available. By having a high space amplification, it would require the user to buy more storage.

$$SA = \frac{\text{size of database files}}{\text{size of database files used on disk}} \tag{2.3}$$

2.4.4 Trade-offs

When optimising databases regarding amplification factors, one has to choose what factors to tweak. This decision should be based on what kind of workload the database endures and the importance of either read, write or storage for the system. Due to limitations in the database structures, it is impossible to keep them all low.

Space amplification and write amplification are inversely related, this means one would have to choose to use an algorithm that optimises for lower space usage at the cost of requiring more writes, or vice versa. An example of this is shown in Figure 2.3 where block cleaning is used to reduce disk usage, but at the cost of having to rewrite the live pages.

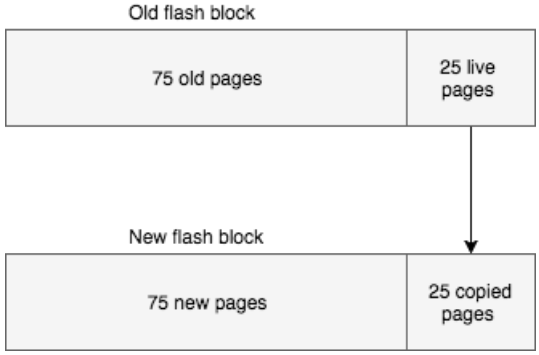

Figure 2.3: Write amplification during block cleaning

Compaction also affects this relation, since compaction reads, modifies and writes SSTables to remove old pages. Without compaction, the write amplification would be lower. However, at a major cost to space usage and slower reads, making the database much less practical in use.

2.5 Bloom filters

An efficient way to determine whether a key exists in a file or not is using *Bloom filters*. *Bloom filters* are generated using algorithms, often a hash functions, which creates probabilistic bit arrays. When looking up keys, we test the algorithm on the key and verify the result with the bit array. The result guarantees not to produce false negatives, meaning the key is not in the file if it says it is not. However, it may produce false positives – "the key might be in the file". *Bloom filters* can thus be used to increase read performance drastically, and the need for compactions can, therefore, be reduced for some workloads. The filters do not remove the compaction requirement since the number of files to test is only increasing without compactions. By increasing the number of elements, we also increase the number of false positives and reduce its efficiency. A way to increase the bloom filters hit rate, is to increase the number of bits in the array generated. Still, this is not a recommended and optimal way to scale.

2.6 Compression

With the increasing number of CPU-cores and improvement of processing speed in general, compression has shown to be an interesting factor to further assess in database-relation. Compression's main application is to store more using less space, even though lower space usage often is a good thing it comes at the cost of having to decompress when rereading the data. In 2018 it is no longer necessarily a problem to use more space; however, we often need to ingest data quickly. Compression has become a more viable feature to implement to increase the insertion rate. The reasoning behind its improved abilities is that disk throughput is not increasing at the same rate as processing power. Since processing power outweighs the I/O speeds, it can be beneficial to utilise those resources while waiting. A reduction of space usage decreases the writing time, thus potentially increasing the throughput.

2.6.1 Snappy & Zlib

There exists a lot of different software and algorithms that offer different rates of compression and speeds of compression for both lossy and lossless applications. So in a database system, there is a fine line of how much time you can spend to compress. By compressing too extensively, the performance benefit is reduced. One want to find the sweet-spot where both CPU and I/O is fully utilised.

Snappy is the default compression engine in RocksDB and LevelDB. It provides both fast compression and decompression and reasonable compression rates. It features compression speeds of ∼200MB/s and 500MB/s on a single core of a 64bit Intel i7 processor [17].

Zlib is another supported compression engine in RocksDB, and is widely used in general. As with Snappy, Zlib is a lossless compression engine; however, it does not focus as much on compression speeds. Zlib is known for its compression ability and the fact that it is completely free without any restricting license or patents. For database purposes, it is not as widely used. Mainly due to its inferior compression speed.

Even though Snappy provides fast compression, compression is still most viable for very powerful CPU's because of the overhead introduced. Despite being defaulted in RocksDB we decided to skip using compression totally in this research, to avoid potential irregularities in the benchmarking results.

Chapter 3

Implementation

This chapter is dedicated to the exploration and implementation of a proof-of-concept compaction auto-tuner. First, the hypothesis is presented following a section on the actual implementation including inspirations taken from RocksDB's rate limiter. To be able to evaluate and gain results, RocksDB's benchmarking tool `db_bench` was extended with new features for this project. Lastly, a section of optimising configurations using an experimental methodology for the compaction tuner is presented.

3.1 Hypothesis

Compacting LSM-based databases is a necessity to ensure acceptable space usage and read performance, but it requires a lot of system resources. We hypothesise that by disabling these compactions when I/O is increasing, and enabling compactions when it decreases – we can increase write performance. If this proves to be possible without too much overhead, it could prove especially valuable for periodic write intense workloads. The hypothesis is deduced from the fact that many systems experience load issues during, e.g. ticket sales. Even though it exists solutions that can manage these kinds of loads in 2018, a database designed to handle these periodic peaks could reduce system requirements and handle higher load.

3.1.1 Industry comments

As a part of the development process, I posted the hypothesis on the RocksDB's Developer forum[18] to try to get some feedback on the project as well as technical guidance.
As shown in the screenshot 3.1, Mark Callaghan the author of the smalldatum blog[19] and Facebook employee quickly liked the post and responded that he thought the project sounded interesting. Furthermore, the post receive multiple likes by other RocksDB core developers and Facebook employees. Despite personally believing in the potential of the project, these responses were a real motivation boost and gave the project merit.

Figure 3.1: Feedback from RocksDB Developer Public

3.2 Parsing statistics

To quickly determine different statistics from the log files produced by RocksDB's `db_bench` a log file parser was implemented in python for this purpose. The complete source code is available open-source under the MIT Licence[20] at Github[21] and in Appendix A.

3.2.1 Log file format

The output from RocksDB's `db_bench` is piped directly to the terminal. The statistics output is piped to stderr and captured to log files using UNIX shell `2>` operator, which redirects stderr to a designated file.

An example excerpt of the log files are given below.

```
Cumulative compaction: 2.09 GB write, 106.48 MB/s write, 1.19 GB read,
    60.66 MB/s read, 14.4 seconds
Interval compaction: 1.85 GB write, 130.27 MB/s write, 1.19 GB read, 83.86
     MB/s read, 13.2 seconds
```

```
Cumulative writes: 10K writes, 10K keys, 10K commit groups, 1.0 writes per
     commit group, ingest: 0.93 GB, 47.57 MB/s
Cumulative WAL: 10K writes, 0 syncs, 10000.00 writes per sync, written:
    0.93 GB, 47.57 MB/s
Cumulative stall: 00:00:0.000 H:M:S, 0.0 percent
Interval writes: 7201 writes, 7201 keys, 7201 commit groups, 1.0 writes
    per commit group, ingest: 686.97 MB, 47.36 MB/s
Interval WAL: 7201 writes, 0 syncs, 7201.00 writes per sync, written: 0.67
     MB, 47.36 MB/s
Interval stall: 00:00:0.000 H:M:S, 0.0 percent
```

3.2.2 Retrieving statistics

The parser uses regular expressions[22] to match the different values in the log files. Example code used to retrieve values for `Interval writes` is given below.

```
interval_regex = 'Interval\swrites.*?(\d*\.\d*)\sMB\/s'
compiled_regex = re.compile(interval_regex)
matches = compiled_regex.findall(file)
```

This code returns a list of the values matched by the group `(\d\.\d*)`. The regex searches for lines starting with 'Interval writes' and the last occurrence of digits with a . between them with a `MB/s` suffix. We use these values to automatically generate coordinates for pgfplots[23] used in this thesis.

3.3 Implementing Auto-Tuner

Implementing auto-tuning compactions for RocksDB felt initially like a daunting task, especially given the magnitude of the database project. Containing tens of thousands of code lines written in C++, a low-level language working close to the metal for optimal performance.

3.3.1 Compaction parameters

During preliminary research of RocksDB, it became clear that RocksDB uses an internal option, `disable_auto_compactions`, to decide whether to do compactions or not. In addition to `disable_auto_compactions`, the other parameters in the `PrepareForBulkLoad()`[24] function were subject to evaluation, since the function's intention is to configure optimal insertion rate for bulk loads. Most of the parameters set in `PrepareForBulkLoad()` are evaluated for the Auto-tuner in Section 3.5. The obvious parameter to change dynamically for this proposed auto-tuner is the `disable_auto_compactions`, but should the auto-tuner work properly it would require increasing the `level0`-slowdown parameters, as done in `PrepareForBulkLoad()`. This because we want to disable compactions, but we also need to avoid stalling writes when having a high number of files in level-0.

Additionally, we figured that `level0_file_num_compaction_trigger` should also be dynamically changed. `PrepareForBulkLoad()` increases the trigger since the trigger is used in slowdown calculations. However, since it is used to determine the compaction score and set a trigger threshold for compactions including Level-0 $\rightarrow$ Level-0 compactions (Section 2.3.2), it should be lower when compactions are enabled.

3.3.2 Design choices

The `disable_auto_compactions` flag is exposed through RocksDB's SetOptions-API[25], and thus this API seemed like the appropriate way to toggle compactions. However, when trying to access this API, it became clear that it is a method only accessible through the database instance. Meaning one would have to have access to the database instance to set new options – `db->SetOptions()`. This led to a design assessment:

1. Should we implement a higher-order plugin that runs above RocksDB, that can access the database instance and toggle compactions using the SetOptions-API?

2. Should we pass a reference to the SetOptions API further down to the core and call the SetOptions function from within the core itself?

3. Or should we create a new interface for getting and setting the compaction flags and in a shared environment?

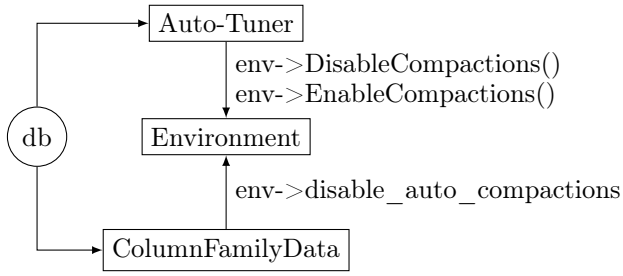

Figure 3.2: Design of getter and setters within RocksDB core using shared environment

Alternative one proved difficult to accomplish regarding detection of background I/O, with less access to database internals. The second alternative was strenuous because having to pass a reference to the function through all the complex initialisation layers of RocksDB. The last option turned out to be the easiest to implement since RocksDB already provided a shared environment for all threads and database instances in the running process.

3.3.3 Environment

Interface
The compaction interface implementation is shown in Fig. 3.3. It is located in the `env.h` header file, and holds different compaction related variables (Section 3.3.1). To trigger the disable/enable compactions, we simply execute the related method `DisableCompactions()` or `EnableCompactions()` then the variables are updated accordingly. For the `DisableCompactions()` we set the `disable_auto_compactions` variable to false, and the `level0_file_num_compaction_trigger` very high to `1<<30`. The reason we used `1<<30`, 1 bitshifted 30 times to the left, is because it is done in `PrepareForBulkLoad()` (Section 3.3.1). Still, this does only update the variables so the next step is to make RocksDB use them instead of the default options.

```
bool disable_auto_compactions;
int prev_level0_file_num_compaction_trigger;
int level0_file_num_compaction_trigger;

void DisableCompactions() {
  if (!disable_auto_compactions) {
    prev_level0_file_num_compaction_trigger =
        level0_file_num_compaction_trigger;
    disable_auto_compactions = true;
    level0_file_num_compaction_trigger = (1<<30);
  }
};

void EnableCompactions() {
  if (disable_auto_compactions) {
    disable_auto_compactions = false;
    level0_file_num_compaction_trigger =
        prev_level0_file_num_compaction_trigger;
  }
}
```

Figure 3.3: Compaction interface implementation in the shared environment

Column Family Data

ColumnFamilyData is the class that handles the compaction trigger and stalling factors within RocksDB. As mentioned, the `disable_auto_compactions` flag was read from initialised options. Having created the interface, we wanted to make RocksDB use this interface instead of the options. The issue was that the `RecalculateWriteStallConditions` method in `ColumnFamilyData` (Fig. 3.5) did not have a reference to this environment. By latching the `Env::Default()` provided from `env.h` onto the *ColumnFamilyData* class (Fig. 3.4), we were able to read the compaction variables directly in the `RecalculateWriteStallConditions` method. Despite being functioning, the environment is shared across multiple database instances in the same process, meaning that this implementation will disable compactions for every instance – even if not opting into tuning. It should, however, not be a big issue for a proof-of-concept.

```
ColumnFamilyData* new_cfd = new ColumnFamilyData(
  id, name, dummy_versions, table_cache_, write_buffer_manager_, options,
  *db_options_, env_options_, this, Env::Default());
```

Figure 3.4: Extending ColumnFamilyData with a reference to the shared environment

This implementation was not the preferred way of achieving the result. The intention was actually to pass a pointer from the `mutable_cf_options`, which hold all the options, to the environment instead of duplicating the specific options. When pursuing this approach, circular imports posed issues when importing the `cf_options`-type from its header file. Despite having `pragma once`[26], it still was not able to compile. This could have been circumvented by refactoring the files, but since the files had a lot of dependencies and 100-1000+ lines in each, we chose a pragmatic solution to duplicate them for this proof-of-concept.

Fig. 3.5 shows how we use the environment to retrieve and sets the environment variables to the `mutable_cf_options` variable. the `disable_auto_compactions` flag for every calculation of write stalls, the function that evaluates

```
-WriteStallCondition ColumnFamilyData::RecalculateWriteStallConditions(
-      const MutableCFOptions& mutable_cf_options) {
+WriteStallCondition ColumnFamilyData::RecalculateWriteStallConditions() {
   auto write_stall_condition = WriteStallCondition::kNormal;
   if (current_ != nullptr) {
+    if (mutable_cf_options_.auto_tuned_compactions) {
+      mutable_cf_options_.level0_file_num_compaction_trigger =
+          env_->level0_file_num_compaction_trigger;
+      mutable_cf_options_.disable_auto_compactions =
+          env_->disable_auto_compactions;
+    }
+    const MutableCFOptions& mutable_cf_options = mutable_cf_options_;
```

Figure 3.5: Retrieval of environment compaction variables when determining write stalls and compactions

3.3.4 Existing rate-limiter

Despite rate-limiting RocksDB not being the purpose of this project, the rate limiter can be used to efficiently determine disk I/O based on write requests to the database. RocksDB features a generic rate limiter used to manage throttling of maximum write speeds. It uses a classic token bucket technique[27] to perform the actual rate-limiting, by draining a bucket of tokens which gets refilled in a specified interval. If the drains the bucket empty, the following requests are rejected until it is refilled.

This rate limiter is configured with three different flags:

As described by the RocksDB wiki[28]:

- `rate_limit_bytes_per_sec`: this is the only parameter you want to set most of the time. It controls the total write rate of compaction and flush in bytes per second. Currently, RocksDB does not enforce rate limit for anything other than flush and compaction, e.g. write to WAL.

- `refill_period_us`: this controls how often tokens are refilled. For example, when `rate_bytes_per_sec` is set to 10MB/s and `refill_period_us` is set to 100ms, then 1MB is refilled every 100ms internally. Larger value can lead to burst writes while smaller value introduces more CPU overhead. The default value 100,000 should work for most cases.
- `fairness`: RateLimiter accepts high-pri requests and low-pri requests. A low-pri request is usually blocked in favor of hi-pri request. Currently, RocksDB assigns low-pri to request from compaction and high-pri to request from flush. Low-pri requests can get blocked if flush requests come in continuously. This `fairness` parameter grants low-pri requests permission by 1/`fairness` chance even though high-pri requests exist to avoid starvation. You should be good by leaving it at default 10.

3.3.5 Auto-tuned Rate Limiter

In December 2017, RocksDB core developer Andrew Kryczka released an auto-tuned rate-limiter[2] that tunes the configuration of the rate limiter based on the background I/O demand. The release of an auto-tuned rate limiter relying on the background I/O to tune was very convenient and interesting to explore in terms of this project.

The auto-tuned rate-limiter uses a MIMD-algorithm, Multiplicative Increase Multiplicative Decrease, typically used for congestion control in network protocols[29]. Background I/O is detected using upper and lower threshold for the rate-limiter to kick in. Auto-tuning happens when the time since the last tuning is higher than `kRefillsPerTune` timed with the `refill_period_us`. Meaning auto-tuning happens every 100th `refill_period_us`, shown in Fig. 3.6.

```
if (auto_tuned_) {
  static const int kRefillsPerTune = 100;
  std::chrono::microseconds now(NowMicrosMonotonic(env_));
  if (now - tuned_time_ >=
      kRefillsPerTune * std::chrono::microseconds(refill_period_us_))
          {
    Tune();
  }
}
```

Figure 3.6: Code executed for every write request determining when to Tune() intervally

The Tune() method in the RateLimiter recalculates and sets a new rate limit and causes too much overhead to be run for every write request. Therefore it is only

re-evaluated intervally to reduce the amount of overhead, without an affecting the rate-limiting appreciably. It works by first determining four constant factors:

1. kLowWatermarkPct = 50
   - A threshold determining background I/O below 50%.
2. kHighWatermarkPct = 90
   - A threshold determining background I/O above 90%.
3. kAdjustFactorPct = 5
   - A factor used to adjust potential rate limit value by 5%.
4. kAllowedRangeFactor = 20
   - A factor used to determine lowest rate limit value in the range [max_bytes_per_sec/kAllowedRangeFactor, max_bytes_per_sec]

To determine the I/O percentages, the number of rate limiter bucket drains and elapsed rate limit intervals are used. The number of drains increments for every write request, which means the previous number of drains have to be subtracted to get the drain difference between the previous tune. Additionally, we need the total number of refill periods carried out to calculate a percentage.

$$drained\_pct = \frac{(num\_drains - prev\_num\_drains) \cdot 100}{elapsed\_intervals} \tag{3.1}$$

The `elapsed_intervals` is determined similarly to the drains, by taking the current time, subtracting it with the time at the previous tune and dividing it with the refill period. Lastly, by timing the drain difference with 100 and dividing on the `elapsed_intervals` – we get an acceptable I/O estimate. Both the I/O percentage and elapsed intervals are calculated in Fig. 3.7.

```
Status GenericRateLimiter::Tune() {
  const int kLowWatermarkPct = 50;
  const int kHighWatermarkPct = 90;
  const int kAdjustFactorPct = 5;
  // computed rate limit will be in
  // `[max_bytes_per_sec_ / kAllowedRangeFactor, max_bytes_per_sec_]`.
  const int kAllowedRangeFactor = 20;

  std::chrono::microseconds prev_tuned_time = tuned_time_;
  tuned_time_ = std::chrono::microseconds(NowMicrosMonotonic(env_));

  int64_t elapsed_intervals = (tuned_time_ - prev_tuned_time +
                               std::chrono::microseconds(refill_period_us_) -
                               std::chrono::microseconds(1)) /
                              std::chrono::microseconds(refill_period_us_);
  // We tune every kRefillsPerTune intervals, so the overflow and division
  // by-zero conditions should never happen.
  int64_t drained_pct =
      (num_drains_ - prev_num_drains_) * 100 / elapsed_intervals;

  int64_t prev_bytes_per_sec = GetBytesPerSecond();
  int64_t new_bytes_per_sec;
  if (drained_pct == 0) {
    new_bytes_per_sec = max_bytes_per_sec_ / kAllowedRangeFactor;
  } else if (drained_pct < kLowWatermarkPct) {
    // sanitize to prevent overflow
    int64_t sanitized_prev_bytes_per_sec = std::min(prev_bytes_per_sec,
        port::kMaxInt64 / 100);
    new_bytes_per_sec = std::max(max_bytes_per_sec_ / kAllowedRangeFactor,
        sanitized_prev_bytes_per_sec * 100 / (100 + kAdjustFactorPct));
  } else if (drained_pct > kHighWatermarkPct) {
    // sanitize to prevent overflow
    int64_t sanitized_prev_bytes_per_sec = std::min(
        prev_bytes_per_sec, port::kMaxInt64 / (100 + kAdjustFactorPct));
    new_bytes_per_sec = std::min(max_bytes_per_sec_,
        sanitized_prev_bytes_per_sec * (100 + kAdjustFactorPct) / 100);
  } else {
    new_bytes_per_sec = prev_bytes_per_sec;
  }
  if (new_bytes_per_sec != prev_bytes_per_sec) {
    SetBytesPerSecond(new_bytes_per_sec);
  }
  num_drains_ = prev_num_drains_;
  return Status::OK();
}
```

Figure 3.7: Tune() method for the auto-tuned rate-limiter, where total background I/O is used to determine a new rate limit

3.3.6 I/O detection

Foundation for efficient I/O detection had already been laid in the auto-tuned rate limiter. This made the process to achieve similar behaviour in the compaction tuner easier. However since the I/O determined was the total background I/O this encompassed both flush and compaction requests, meaning it was not possible to differentiate between flushes and compactions. For the rate-limiter, this was ok since its purpose is to throttle new requests regardless what kind of request it is. However, for a compaction tuner how can one determine when to enable or disable compactions using this percentage value? With a watermark to disable at 90% and enable at 50%, it works for disabling compactions when increasing. But once compactions are enabled below 50% compactions jobs starts and increases I/O substantially, and once over 90% it disables compactions again. Thus high compaction pressure disables compactions.

To differentiate flushes and compaction requests, RocksDB makes it quite easy for us. Compactions are considered low-priority requests and flushes high-priority requests. By exploiting this priority we can hold a drain variable for both of the priority levels and using these to avoid the issue with compactions disabling compactions. The code splitting up `num_drains` to low and high is provided in Fig. 3.8

The TuneCompaction() method used to auto-tune compactions is shown in Fig. 3.9. It uses a similar I/O detection approach as the rate limiter's `Tune()` method (Fig. 3.7). The disparities is the differentiation between high and low priority requests, the trigger conditions and the usage of the environment interface (Fig. 3.3).

3.3.7 Trigger conditions

Having taken inspiration from RocksDB's rate limiter, it was natural to consider the same watermarks from the rate-limiter for the compaction tuner. However, we needed to factor in having both high and low priority drain variables. After some experimentation and issues with compactions disabling compactions (Section 3.3.6), we figured that it was most natural to disable compactions when the flush I/O is above 50% and total I/O is above 90%. If flush I/O increase above 50%, we then assume that it is subject to further increase. We enable when we see that total I/O is below 90% and flush I/O is below 50%. These assumptions are not optimal; a workload could potentially disable compactions with flush I/O just above 50% and continue hovering at that percentage. In this case it would never enable compactions and waste potential resources that could have been used to compact the database. Though, for a proof-of-concept tuner, these assumptions will suffice to show the potential.

```
    bool timedout = false;
    // Leader election, candidates can be:
    // (1) a new incoming request,
    // (2) a previous leader, whose quota has not been not assigned yet
        due
    //    to lower priority
    // (3) a previous waiter at the front of queue, who got notified by
    //    previous leader
-   if (leader_ == nullptr &&
-       ((!queue_[Env::IO_HIGH].empty() &&
-         &r == queue_[Env::IO_HIGH].front()) ||
-        (!queue_[Env::IO_LOW].empty() &&
-         &r == queue_[Env::IO_LOW].front()))) {
+   bool leader_isnull = leader_ == nullptr;
+   bool io_high = leader_isnull ? (!queue_[Env::IO_HIGH].empty() && &r ==
      queue_[Env::IO_HIGH].front()) : false;
+   bool io_low = !io_high ? (!queue_[Env::IO_LOW].empty() && &r == queue_
     [Env::IO_LOW].front()) : false;
+   if (leader_isnull && (io_high || io_low)) {
      leader_ = &r;
      int64_t delta = next_refill_us_ - NowMicrosMonotonic(env_);
      delta = delta > 0 ? delta : 0;
      if (delta == 0) {
        timedout = true;
      } else {
        int64_t wait_until = env_->NowMicros() + delta;
+       if (io_high) {
+         ++num_high_drains_;
+         RecordTick(stats, NUMBER_RATE_LIMITER_HIGH_PRI_DRAINS);
+       } else if (io_low) {
+         ++num_low_drains_;
+         RecordTick(stats, NUMBER_RATE_LIMITER_LOW_PRI_DRAINS);
+       }
+       num_drains_ = num_high_drains_ + num_low_drains_;
        RecordTick(stats, NUMBER_RATE_LIMITER_DRAINS);
-       ++num_drains_;
        timedout = r.cv.TimedWait(wait_until);
      }
```

Figure 3.8: Request election extended with high and low priority rate limiter drains

```
Status GenericRateLimiter::TuneCompaction(Statistics* stats) {
  const int kLowWatermarkPct = 50;
  const int kHighWatermarkPct = 90;

  std::chrono::microseconds prev_tuned_time = tuned_time_;
  tuned_time_ = std::chrono::microseconds(NowMicrosMonotonic(env_));

  int64_t elapsed_intervals = (tuned_time_ - prev_tuned_time +
                               std::chrono::microseconds(refill_period_us_) -
                               std::chrono::microseconds(1)) /
                              std::chrono::microseconds(refill_period_us_);
  // We tune every kRefillsPerTune intervals, so the overflow and division
      -by-
  // zero conditions should never happen.
  assert(num_drains_ - prev_num_drains_ <= port::kMaxInt64 / 100);
  assert(elapsed_intervals > 0);
  int64_t drained_high_pct =
          (num_high_drains_ - prev_num_high_drains_) * 100 /
              elapsed_intervals;
  int64_t drained_low_pct =
          (num_low_drains_ - prev_num_low_drains_) * 100 /
              elapsed_intervals;
  int64_t drained_pct = drained_high_pct + drained_low_pct;

  if (drained_pct == 0) {
    // Nothing
  } else if (drained_pct <= kHighWatermarkPct && drained_high_pct <
      kLowWatermarkPct) {
    env_->EnableCompactions();

  } else if (drained_pct >= kHighWatermarkPct && drained_high_pct >=
      kLowWatermarkPct) {
    env_->DisableCompactions();
    RecordTick(stats, COMPACTION_DISABLED_COUNT, 1);
  }
  num_low_drains_ = prev_num_low_drains_;
  num_high_drains_ = prev_num_high_drains_;
  num_drains_ = prev_num_drains_;
  return Status::OK();
}
```

Figure 3.9: TuneCompaction() method for the compaction auto-tuner that differentiates between flushes and compactions to toggle compactions

3.4 Extending db_bench

`db_bench` is the provided benchmarking tool for RocksDB; it features default benchmarks for many different operations like filling random data, overwriting existing data, compacting, deletions and more.

3.4.1 Auto-tuner flag

To be able to conduct benchmarks using the compaction tuner, `db_bench` need a command line flag for this purpose. `db_bench` uses gflags[30] to process command line arguments, and allows easy extension by adding a single gflags `DEFINE`. This flag is then accessible using a FLAGS prefix, `FLAGS_auto_tuned_compactions` which becomes a boolean.

```
DEFINE_bool(auto_tuned_compactions, false,
           "Enable dynamic disabling of compactions when I/O is high");
```

3.4.2 Sine wave

To benchmark RocksDB in terms of a periodic workload, a sine wave was a natural choice. Implementing a waveform simulation of write load would, therefore, be an interesting way of visualising the write performance. To enable the sine wave write rate limiting, we define a bool `sine_write_rate` and `sine_write_rate_interval_milliseconds` to set a recalculation interval.

```
DEFINE_bool(sine_write_rate, false,
           "Use a sine wave write_rate_limit");

DEFINE_uint64(sine_write_rate_interval_milliseconds, 10000,
              "Interval of which the sine wave write_rate_limit is
                  recalculated");
```

Sine wave function used to determine the `write_rate_limit` is provided below. It has an amplitude of 150, with peaks at 50 and 200 (MB/s). The period is set to 350seconds by taking $\frac{2\pi}{350} = \frac{\pi}{175}$. The wave is plotted in Fig. 3.10.

$$S = 75 \cdot sin(\frac{\pi}{175} \cdot x + \frac{3\pi}{2}) + 125 \tag{3.2}$$

Initially this sine function was hard-coded in `db_bench` for the purpose of this project, but later made generic by adding flags for the constants of $f(x) = Asin(bx + c) + d$. The flags are accessible in the code using `FLAGS_sine_a`, `b` etc.

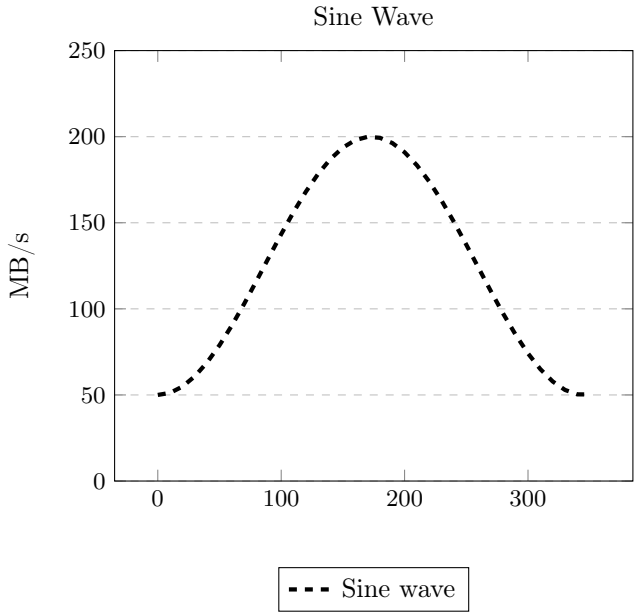

Figure 3.10: Sine wave for benchmark write rate limit

```
DEFINE_double(sine_a, 1,
            "A in f(x) = A sin(bx + c) + d");

DEFINE_double(sine_b, 1,
            "B in f(x) = A sin(bx + c) + d");

DEFINE_double(sine_c, 0,
            "C in f(x) = A sin(bx + c) + d");

DEFINE_double(sine_d, 1,
            "D in f(x) = A sin(bx + c) + d");
```

Figure 3.11: Flags for sine wave constants

The `SineRate` function in Fig. 3.12 is making use of these constants and calculates a new point at the sine wave given a double value X, where X is the number of seconds after the benchmark was initiated.

```
double SineRate(double x) {
  return FLAGS_sine_a*sin((FLAGS_sine_b*x) + FLAGS_sine_c) + FLAGS_sine_d;
}
```

Figure 3.12: Sine wave function of X seconds used to calculate new rate limit

3.4.3 Benchmark write rate

`db_bench` executes a DoWrite method when conducting write benchmarks. The primary purpose of this method is to write and update related statistics. In this method, we have access to the current write thread and all thread shares some variables and functions including a write rate limiter used to handle benchmark writes. Through this rate-limiter, we can set a new rate limit value, chose a value calculated using the SineRate (Fig. 3.12) function and the number of seconds since the start. The `write_rate_limiter` is reset every 10 seconds and allows a `sine_finished` bool to opt out of the sine wave rate limiting during the benchmark, shown in Fig. 3.13.

```
if (FLAGS_sine_write_rate) {
  if (!sine_finished && usecs_since_last > (
      FLAGS_sine_write_rate_interval_milliseconds * uint64_t{1000})) {
    thread->stats.ResetSineInterval();
    uint64_t write_rate = static_cast<uint64_t>(
        SineRate(static_cast<double>(usecs_since_start) / 1000000.0)
    );
    thread->shared->write_rate_limiter.reset(
        NewGenericRateLimiter(write_rate)
    );
  }
}
```

Figure 3.13: Excerpt of dynamic rate limiting in db_bench according to sine wave function

For the 700 and 1000sec benchmarks in Chapter 4, we used the `sine_finished` flag to set a static `write_rate_limit` at 10MB/s after an initial sine wave for 350sec. Fig. 3.14 shows the code that executes `SetSineFinished()` flipping the `sine_finished` flag, opting out of the sine wave rate limit.

```
DEFINE_int32(sine_finished_seconds, 0,
            "Number of seconds to opt out of sine wave and use
             rate_limiter_bytes_per_second instead");

DEFINE_int32(sine_finished_write_rate_limit, 0,
            "Write rate limit when opting out of sine wave");
```

```
if (!sine_finished && usecs_since_start > (FLAGS_sine_finished_seconds *
    1000000)) {
  thread->stats.SetSineFinished();
  thread->shared->write_rate_limiter.reset(
         NewGenericRateLimiter(FLAGS_sine_finished_write_rate_limit));
}
```

Figure 3.14: Flags handling opting out of sine wave rate limit

3.5 Configuring the Auto-Tuner

The following section evaluates multiple existing RocksDB options that influences performance. Options evaluated are taken from `PrepareForBulkLoad()`[31] which is RocksDB's method to optimise write-insertion for bulk loads. We consider the flags from this method specifically relevant to this tuner. Additionally, we consider rate-limiting (Section 3.5.1) and subcompactions (Section 3.5.4). Given the many factors affecting performance in different ways, we chose an experimental methodology. Meaning we conducted multiple experiments with different values for the most relevant options.

3.5.1 Rate-limiting

RocksDB suggest setting the `rate_limit_bytes_per_sec` to the disk write rate on dedicated hosts for the auto-tuned rate limiter[2]. The database write-throughput hovers around the half of the maximum write speed, so the rate limiter internally halves the `rate_limit_bytes_per_sec` for configuration. For the compaction tuner, it is also necessary to set this flag as it is used as the 100% I/O cap per interval. It calculates the number of bytes allowed to be written per interval, called `refill_bytes`. These refill bytes are required to generate the drain variables – used to determine I/O efficiently. Despite the drain variables, actual rate limiting of writes is not required for the compaction tuning. However, it makes the proof-of-concept more stable concerning disabling and enabling compaction triggers, which also makes the results more stable and reproducible.

Given the benchmarks conducted without any rate-limit cap results in $\sim$ 190MB/s flush rate, due to the RateLimiter halving explained above, it was natural to cap the benchmarks at 380MB/s. This value was chosen instead of the disk write rate naively since the database is not able to achieve a higher flush rate with RocksDB compiled in debug mode. Though, this rate limits compactions as well, and compactions are actually able to achieve higher throughput than flushes – up to $\sim 225 - 250$MB/s. Benchmarks without the compaction tuner and rate-limiting are, therefore, able to compact with a higher rate, which influences the results presented in the Chapter 4.

3.5.2 Pending compaction bytes

When flushing decreases below ∼100-125MB/s compactions gets enabled, and post-enabling RocksDB can behave differently using the auto-tuner. By having soft and hard pending compaction bytes flag set to the default, the flush rate will decrease close to 0, and the database will compact all the way down to the limit before continuing allowing further writes. The outcome of this will be that the database will have better read throughput as quickly as possible after a peak; however, it spends much time compacting and does somewhat contradicts the intention of the tuner – increasing write throughput. By setting it unlimited, flushing can continue as usual. Naturally, then the database will not prioritise to be as compacted right after the peak. However, the database will continue to compact for an extended period afterwards – thus having "deferred compactions". This means that after some time with less load the database will get compacted similarly. This is the chosen approach for this proof-of-concept. For future work, it could be interesting to take a more in-depth look at how to handle these post-peak writes.

Figure 3.15 are excerpts of two benchmarks that shows the different behaviours. The grey area paints the intervals that have compactions disabled. Once the compactions are triggered, new writes are stalled for almost 200 seconds by default until compactions catch up. The writes also struggle once below the compaction bytes threshold, since it starts accumulating new bytes to compact.

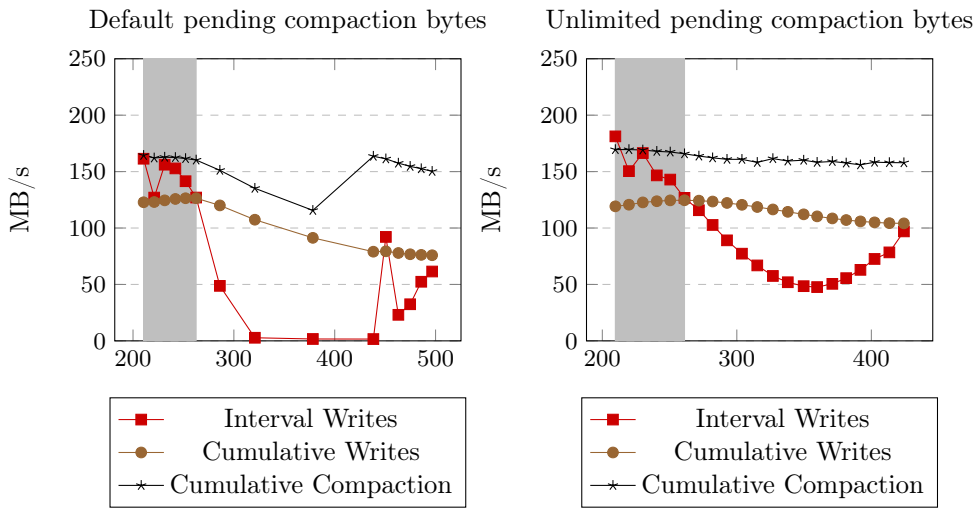

Figure 3.15: Difference between limited and unlimited pending compaction bytes

3.5.3 Threads

When RocksDB is configured to utilise a high amount of compaction threads, it queues and starts more compaction jobs. This leads to a delayed ramp of insertion rate during the peaks, meaning that the compaction threads throttle flush threads despite having lower priority. Thus to ensure that RocksDB can quickly respond to increasing load, experimentation has shown that one should use more flush threads than compaction threads to overcome this throttle. An interesting observation was that by using 1x flush thread, the compaction disabling did not trigger at all – hovering around 25% of the max flush rate.

In Fig. 3.16 the *Interval Writes* denotes different flush rates for different amount of threads. For the plot with 4x compaction threads disables compaction almost 50 seconds later than with 2x compaction threads. Note that these plots were configured with 4x flushing threads.

As for future work, one should consider investigating the impact of pending compactions when compactions are disabled. It might be interesting to clear the compaction queue and cancel running compactions. In this proof-of-concept, running compactions are allowed to finish after compactions have been disabled. RocksDB does not have a straightforward way of clearing the queue nor stopping running compactions and this has not been prioritised in this thesis. However in the benchmarks conducted this has not been a significant issue, but it is not wrong to imagine that this might be a throttling factor for some workloads.

3.5.4 Subcompactions

By default RocksDB allocates only one thread for level-0 → level-1 compactions, despite the number of threads allocated using `max_background_compactions` or `max_background_jobs`. The auto-tuner struggles massively with this compaction after the number of level-0 files accumulates during a peak. By increasing the `subcompactions` flag, we can somewhat increase the parallelism by allowing RocksDB to range-partition the compaction. This allows multiple level-0 → level-1 compactions to happen simultaneously and reduces the time spent for the compaction by about 30-50seconds. A few quick benchmarks revealed that increasing the flag did not reduce the time spent further. The efficiency of compactions could most likely be improved even further, however, given the flush vs. compaction threads impact on triggering the compaction disabling (Fig. 3.16) – this was left at 2.

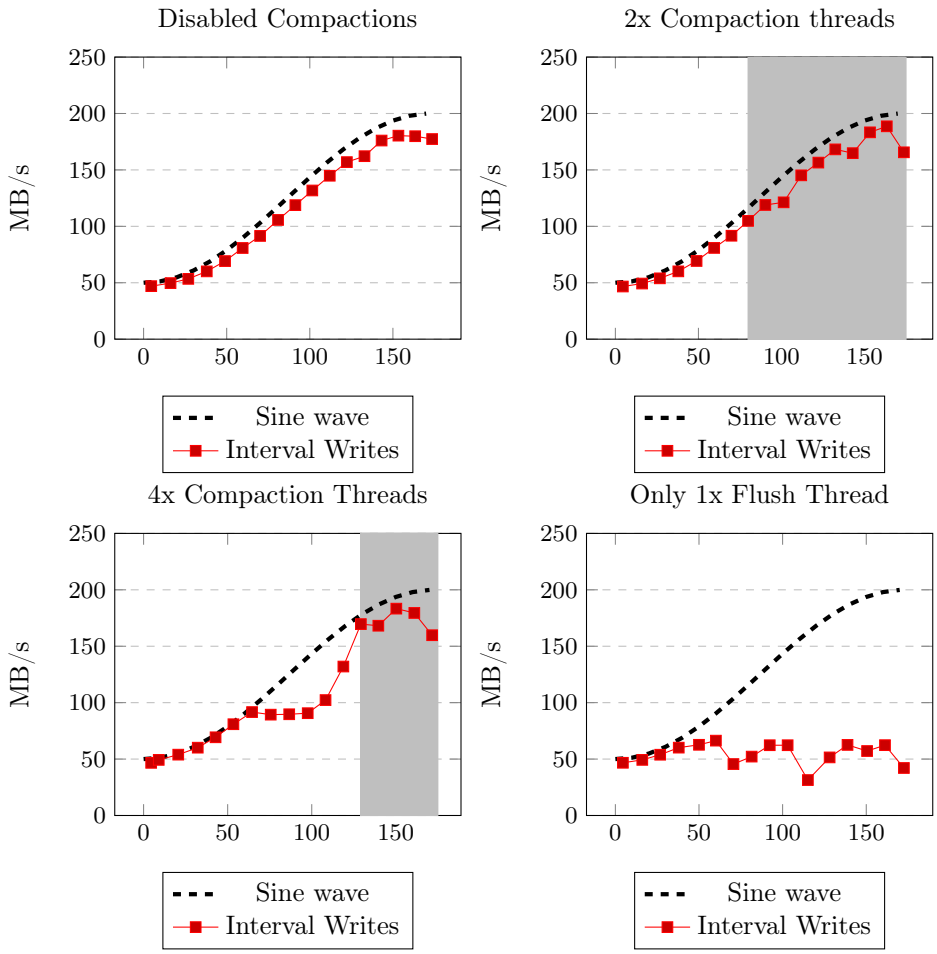

Figure 3.16: Comparison of thread allocation

3.5.5 Write buffer size

Conducting benchmarks with different write buffer sizes showed a tight correlation between the number of threads and buffer size. The write rate is exhibiting the same trend with a buffer size of 4 as with increased compaction threads. When increasing the buffer to 6, the issue is pretty much gone.

The correlation between the buffer size and the number of threads is a result of potential starvation[32]. By not having memtables ready for flushing, the flush threads have to wait for new writes to memory before initiating a new flush.

Increasing the buffer beyond six does not affect the flush rate substantially for the system used for these benchmarks. Thus, the memory requirement is not necessarily that high. Smaller devices like smartphones could therefore make use of an auto-tuner like this.

Figure 3.17 shows the improvement in writes using 6x buffers over 4x. The blue shaded area shows the difference between optimal and actual writes. The grey shaded area denotes where compactions gets disabled.

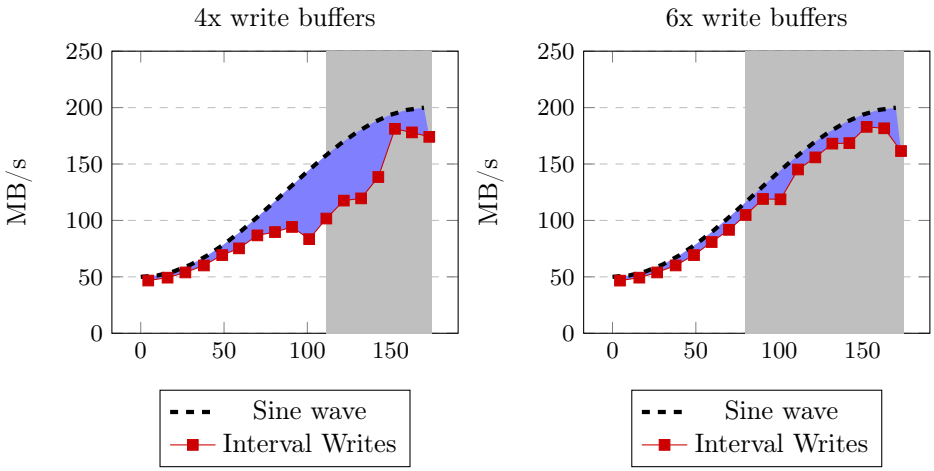

Figure 3.17: Comparison of 4x and 6x memtable buffers

3.5.6 Resulting configuration

Through the configuration experimentation, we chose to set a few parameters initially as default by making a `PrepareAutoTunedCompactions()` method since the tuner would not behave correctly without these for any system.

```
if (env_ && mutable_cf_options_.auto_tuned_compactions) {
  env_->disable_auto_compactions =
      mutable_cf_options_.disable_auto_compactions;
  env_->level0_file_num_compaction_trigger =
      mutable_cf_options_.level0_file_num_compaction_trigger;
  mutable_cf_options_.PrepareAutoTunedCompactions();
}
```

```
  void PrepareAutoTunedCompactions() {
    level0_slowdown_writes_trigger = (1<<30);
    level0_stop_writes_trigger = (1<<30);
    soft_pending_compaction_bytes = 0;
    hard_pending_compaction_bytes = 0;
  }
```

Figure 3.18: Compaction interface implementation in CFOptions

We did not hard code all parameters since they could depend on the system running the database. Hence we set the parameters below manually for each of the auto-tuned benchmarks. Commands for each benchmark conducted is provided in Chapter 4.

- `max_background_flushes = 4`
- `max_background_compactions = 2`
- `write_buffer_size = 6`
- `compression = None`
- `subcompactions = 2`
- `sine_a` $= 75000000$
- `sine_b` $= \frac{\pi}{175} \approx 0.017942857$
- `sine_c` $= \frac{3\pi}{2} \approx 4.712388980$
- `sine_d` $= 125000000$
- `rate_limiter_bytes_per_sec` $= 380000000$

Chapter 4

Benchmarks

The following chapter is divided into sections that present and compare results of the auto-tuned, enabled and disabled compactions for a periodic workload and maximal throughput using leveled compaction (Section 2.3.2). The benchmarks are conducted with options determined through experimentation presented in Section 3.5. Results are evaluated in terms of insertion rate to the database, write amplification, compaction rate, random read rate and time spent compacting. In order to a evaluate the different benchmarks conducted using a periodic workload, a synthetic sine wave workload is used. Since this did not exist in the benchmarking tool, `db_bench`, this was implemented for the purpose of this project. The extension of `db_bench` is further elaborated in the Section 3.4.

4.1 System and Installation

System

System specifications:
Ubuntu 17.10 (Artful Aardvark) Kernel 4.13.0-38-generic
Dell OptiPlex 9020
Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
16GiB DIMM DDR3 Synchronous 1600 MHz
Lite-On IT LCS-256L9S-11 2.5 7mm 256GB

SSD speed

Lite-On IT LCS-256L9S-11 2.5 7mm 256GB
Sequential Read: 530MB/s and Sequential Write: 430MB/s
Up to 74,000/70,000 IOPS Interface: SATA III 6.0 GB/s

A few benchmarks were conducted to verify the sequential write rate in Fig. 4.1.

```
time sh -c "dd if=/dev/zero of=testfile bs=100k count=200k && sync"
```

The commands executed to retrieve statistics for the performance of the internal SSD is provided in Fig. 4.1. What it does is using `dd`[33] to write batches of 100KB for 200k times. In total, this gives 20 GiB of data written. The volume divided on the time spent gives us the average throughput. The command was run three times, and the results showed an average of 465MB/s, which is a bit higher than provided by the manufacturer.

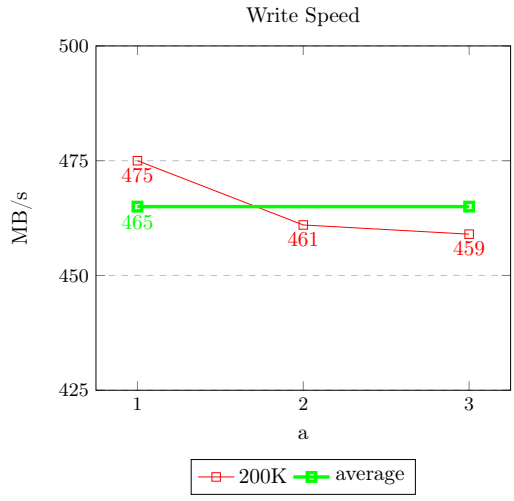

Figure 4.1: Write Speed of internal SSD

Installation

RocksDB requires `C++11` and gcc should be version 4.8 or higher. This can be verified by running `gcc -v`.

1. `sudo apt install gcc build-essential`
2. `sudo apt install libgflags-dev libsnappy-dev zlib1g-dev libbz2-dev liblz4-dev libzstd-dev`
3. `git clone git@github.com:hanswilw/rocksdb.git`
4. `cd rocksdb && git checkout auto-tuned-compactions`
5. `make db_bench`

4.2 Explanation of data

4.2.1 Plots

In the plots we present three different benchmark versions and four different graphs for each version:

- **Enabled** is a default configured RocksDB instance, except allowing 2x compaction threads (Section 3.5.3), 6x write buffers (Section 3.5.5) and disabled compression (Section 2.6).
- **Disabled** is similarly configured as the Enabled, with the exception of disabling compactions and increasing `level0`-slowdown triggers (Section 3.3.1).
- **Tuned** is the auto-tuned version, it is configured accordingly to the experiments conducted in Section 3.5. The most significant configurations is of course enabling the tuner, rate limiting at 380MB/s (Section 3.5.1), slowdown triggers including pending compaction bytes (Section 3.5.2) and subcompactions (Section 3.5.4).
- **Sine wave** is the waveform presented in Section 3.4.2, which is the write rate limit cap.
- **Interval writes** is the value of the current write rate for new incoming writes to the database. This means that the statistic tells us at what rate it ingests new data, without considering any database internals.
- **Cumulative writes** is the average accumulated write rate statistic for all the *Interval Writes* since the start. It is an interesting statistic in addition to the intervals since it tells us the total write rate combined from start to end of the benchmarks.
- **Cumulative compaction**, on the other hand, is the combined sum of *Cumulative Writes* and (cumulative) compactions' write rate. This is a bit

ambiguous since one often distinguish between flushes and compactions, but flushes are included in this statistic. Thus, *Cumulative Writes* are considered a part of the *Cumulative Compaction* rate, and their difference is data written through the compaction threads.

4.2.2 Statistics

- **DB size** gives an insight into how much data the database holds, and is interesting to compare with the total written data.
- **Total written** is the combined amount of data flushed and data written through compactions.
- **Write Amplification** is discussed in greater detail in Section 2.4.2.
- **Insertion Rate** denotes the rate of new data inserted to the database.
- **Compaction Rate** is the sum of the *Insertion Rate* and data written through compactions. This means that the difference between the *Compaction Rate* and the *Insertion Rate* is the write rate of compactions threads exclusively.
- **Leveled Compaction Stats** is the hierarchy statistics in leveled compaction explained in Section 2.3.2.
- **Random reads**: An easy way of determining the compaction efficiency is to run a random read benchmark on the resulting database left by the write benchmarks. The read benchmarks are run with compactions disabled because RocksDB starts to compact while reading by default – if required. Throughputs by the read benchmarks are closely correlated with the leveled compaction stats, meaning that well-compacted databases should and will provide faster reads than less-compacted. Fig. 4.2 shows the command run to determine the random read rate for the benchmarks presenter later in this chapter.

```
./db_bench -benchmarks="readrandom,stats" --num=100000 -statistics -
    use_existing_db=1 -disable_auto_compactions=true
```

Figure 4.2: Random read benchmark

4.3 Sine wave

The benchmark conducted for the following sections are `db_bench`'s fillrandom. It is described as "write N values in random key order in async mode"[34].

4.3.1 350sec

Benchmark commands

Enabled Compactions:

```
./db_bench -benchmarks="fillrandom,stats" -statistics -duration=350
    -value_size=100000 -compression_type=None -sine_write_rate=true
    -sine_a=75000000 -sine_b=0.017942857 -sine_c=4.71 -sine_d=125000000
    -stats_interval_seconds=10 -stats_per_interval=1
    -max_write_buffer_number=6 -max_background_flushes=4
    -max_background_compactions=2
```

Disabled Compactions:

```
./db_bench -benchmarks="fillrandom,stats" -statistics -duration=350
    -value_size=100000 -compression_type=None -sine_write_rate=true
    -sine_a=75000000 -sine_b=0.017942857 -sine_c=4.71 -sine_d=125000000
    -stats_interval_seconds=10 -stats_per_interval=1
    -max_write_buffer_number=6 -max_background_flushes=4
    -level0_slowdown_writes_trigger=10000 -level0_stop_writes_trigger
        =10000
    -level0_file_num_compaction_trigger=10000 -disable_auto_compactions=
        true
```

Auto-tuned Compactions:

```
./db_bench -benchmarks="fillrandom,stats" -statistics -duration=350
    -value_size=100000 -compression_type=None -sine_write_rate=true
    -sine_a=75000000 -sine_b=0.017942857 -sine_c=4.71 -sine_d=125000000
    -stats_interval_seconds=10 -stats_per_interval=1
    -max_write_buffer_number=6 -max_background_flushes=4
    -max_background_compactions=2 -subcompactions=2
    -auto_tuned_compactions=true -rate_limiter_bytes_per_sec=380000000
```

The auto-tuner was evaluated in terms of the sine wave presented in Section 3.4.2 for three different configurations; a pretty standard with enabled compactions, with disabled compactions and using the auto-tuner. Benchmarks conducted for the following sections are done using the commands given above, and the plots are presented in Fig. 4.3.

In the first plot showing the *Enabled Compactions*, we see that the write rate is not able to cope with the sine wave write rate limit. *Interval Writes*, being the red plot with squares, starts diverging already at 50 seconds after start. At X = 270 seconds, we see that the *Interval Writes* drops substantially. This is due to level-0 files exceeding `level0_slowdown_writes_trigger` at 20, combined with the increasing amount of pending compaction bytes. That means RocksDB slows down new writes until these files are compacted to level-1 (or level-0). The compaction threads have at this point already started compacting a part of level-0, but new level-0 $\rightarrow$ level-1 compactions are waiting for the running compaction to finish. It will, however, be selected first once available, due to the compaction score for level-0 is way higher than the other levels given the write-pressure. As a side note, RocksDB could benefit from a higher number of compaction threads, a thread per processor core is often recommended. However, to compare the results with similar conditions, we chose two (Section 3.5.3). Also, experiments with enabled compactions with increased slowdown triggers are provided in Section 4.5.

By disabling compactions entirely and increasing the slowdown triggers never to occur, we achieve a pretty much optimal write rate. The second plot shows a close to perfect correlation between the sine wave write cap and the actual write rate. In comparison to enabled compactions, this plot results a *Cumulative Write* rate at 116MB/s while the enabled results in 62MB/s. So we can conclude that by disabling compactions we can almost double the write rate for this workload, at the expense of not compacting the database.

Thirdly, the *Auto-tuned Compactions* results are presented. As explored throughout Chapter 3, the goal is to exploit the best of both worlds. The plot shows that the auto-tuned version can cope with write rate better than the enabled up to $\sim$100MB/s, and this is partly due to the rate limiting affecting the compaction rate, as well as having increased the slowdown triggers – similarly to the disabled version. Indications from the rate-limiting were that it stabilised the writes at the cost of throttling the max compaction rate, making the tuner toggling compactions more reliably. The grey area shows where the compactions are disabled, and here is where the tuner shows its strength. It can toggle compactions both when it is increasing and decreasing, making it able to handle higher write throughput during the peak and toggling compactions back on afterwards.

As discussed in Section 3.5.1, the compaction rate with compactions enabled is higher than with the auto-tuned compactions due to the rate limiting. Since the rate limiting is unnecessary for the tuner, the potential is not fully exploited. However, it gives a clear indication that it is possible to improve write-performance by toggling compactions based on I/O since the combined flush and compaction rate is $\sim$40MB/s higher for the auto-tuner.

Plots 350sec

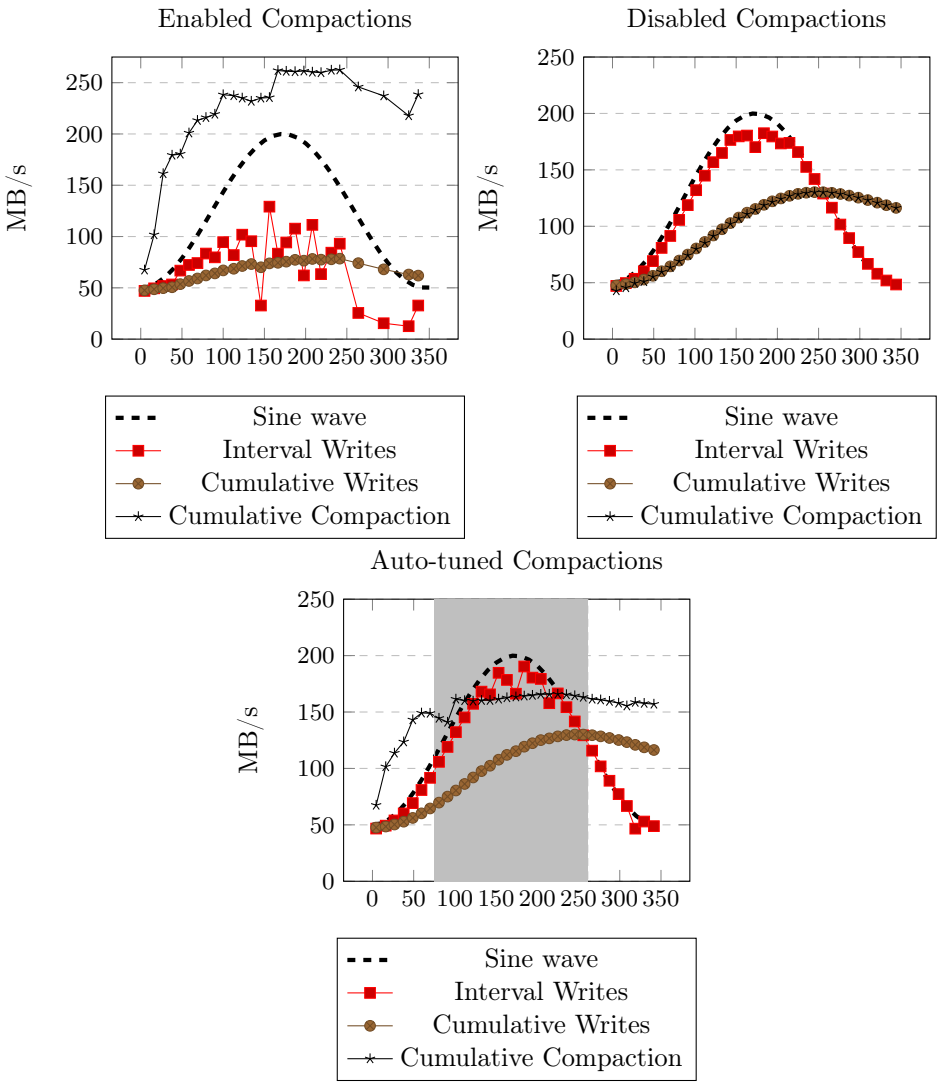

Figure 4.3: Plots of enabled, disabled and tuned compactions for 350 second sine wave

Statistics 350sec

Statistics from the 350sec sine wave peak benchmarks are presented in Fig. 4.4.

The random read rate shows the fundamental requirement of compactions for LSM-based databases. When compactions are disabled, the random read rate is about 3.4MB/s but enabled it is 74.3MB/s. That is a ridiculous factor of 2185%, and even 74.3MB/s is a bit slow in this context. Even though the level hierarchy for *Enabled Compactions* is far off the target, it was still compacting at the end of the benchmark. The reason that compactions provides such an improvement in terms of reads are discussed in better detail in Section 2.3.2, but the primary reason is avoiding many linear seeks when having many files in level-0.

The performance achieved by the auto-tuner is satisfying, despite the low read rate. The read rate is expected since we continuously insert a lot of data >50MB/s and most of the data is inserted while compactions are disabled during the peak. However it initiates compactions on basically all files in level-0 once enabled, but it does not have time to finish. The write results, on the other hand, are great; the insertion rate is similar to the disabled version, while the *Cumulative Compaction* rate is 40MB/s higher. These 40MB/s are accumulated in the small windows where compactions are enabled. Hence being able to exploit the disk speed better, prioritising new writes and compacting when able to. In total it allows $1 - \frac{423K}{222K} = 90.5\%$ more writes than the *Enabled Compactions* over a 350second period and just 3K writes off the "write-optimal" *Disabled Compactions*.

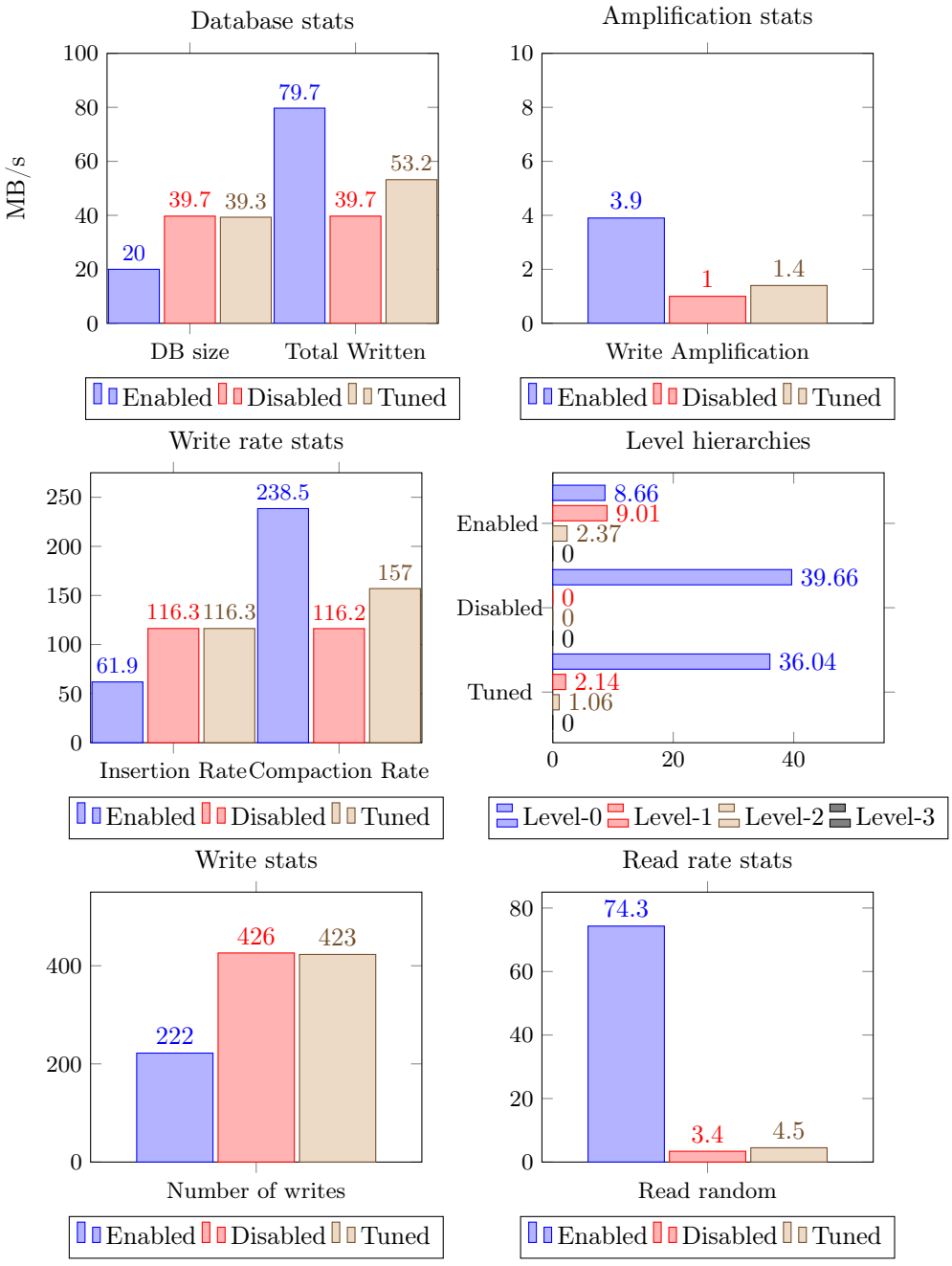

Figure 4.4: Statistics post 350sec

4.3.2 700sec

Benchmark commands

Enabled Compactions:

```
./db_bench -benchmarks="fillrandom,stats" -statistics -duration=700
    -value_size=100000 -compression_type=None -sine_write_rate=true
    -sine_finished_seconds=350 -sine_finished_write_rate_limit=10000000
    -sine_a=75000000 -sine_b=0.017942857 -sine_c=4.71 -sine_d=125000000
    -stats_interval_seconds=10 -stats_per_interval=1
    -max_write_buffer_number=6 -max_background_flushes=4
    -max_background_compactions=2
```

Disabled Compactions:

```
./db_bench -benchmarks="fillrandom,stats" -statistics -duration=700
    -value_size=100000 -compression_type=None -sine_write_rate=true
    -sine_finished_seconds=350 -sine_finished_write_rate_limit=10000000
    -sine_a=75000000 -sine_b=0.017942857 -sine_c=4.71 -sine_d=125000000
    -stats_interval_seconds=10 -stats_per_interval=1
    -max_write_buffer_number=6 -max_background_flushes=4
    -level0_slowdown_writes_trigger=10000 -level0_stop_writes_trigger
        =10000
    -level0_file_num_compaction_trigger=10000 -disable_auto_compactions=
        true
```

Auto-tuned Compactions:

```
./db_bench -benchmarks="fillrandom,stats" -statistics -duration=700
    -value_size=100000 -compression_type=None -sine_write_rate=true
    -sine_finished_seconds=350 -sine_finished_write_rate_limit=10000000
    -sine_a=75000000 -sine_b=0.017942857 -sine_c=4.71 -sine_d=125000000
    -stats_interval_seconds=10 -stats_per_interval=1
    -max_write_buffer_number=6 -max_background_flushes=4
    -max_background_compactions=2 -subcompactions=2
    -auto_tuned_compactions=true -rate_limiter_bytes_per_sec=380000000
```

Plots 700sec

Similar to 350sec benchmarks, the plots in Fig. 4.5 shows the different write performance during the sine wave for the three different versions.

The benchmarks were run first with a similar 350second peak, then with a capped `sine_finished_write_rate_limit` of 10MB/s continuing from 350 to 700seconds. Thus giving RocksDB time and resources to do compactions, despite having the small rate limit for the tuned version (Section 3.3.4 and 3.5.1).

With *Enabled Compactions*, we see that after the sine wave, the compaction rate sustains a similar rate of 200-250MB/s, despite having 10MB/s *Interval Writes*. This indicates that it still has many compaction jobs queued up throughout the whole run.

Having disabled compactions entirely, we see no unique information despite inserting the 10MB/s cap after the sine wave. The auto-tuner shows a bit different behaviour. It bumps the *Cumulative Compaction* rate from 100 to 150MB/s at X=575sec, at that point a major Level-0 $\rightarrow$ Level-1 compaction finishes. At this point, the read performance increases significantly, presented in the following Statistics section. In total, we see that the *Cumulative Writes* for the *Auto-tuned Compactions* is similar *Disabled Compactions*, but at the same time utilising free system resources to do compactions at a rate of $\sim$ 95MB/s. The *Cumulative Compaction* rate is, however, $\sim$ 75MB/s lower than the enabled, due to rate-limiting and the tuning overhead (Section 3.5.1).

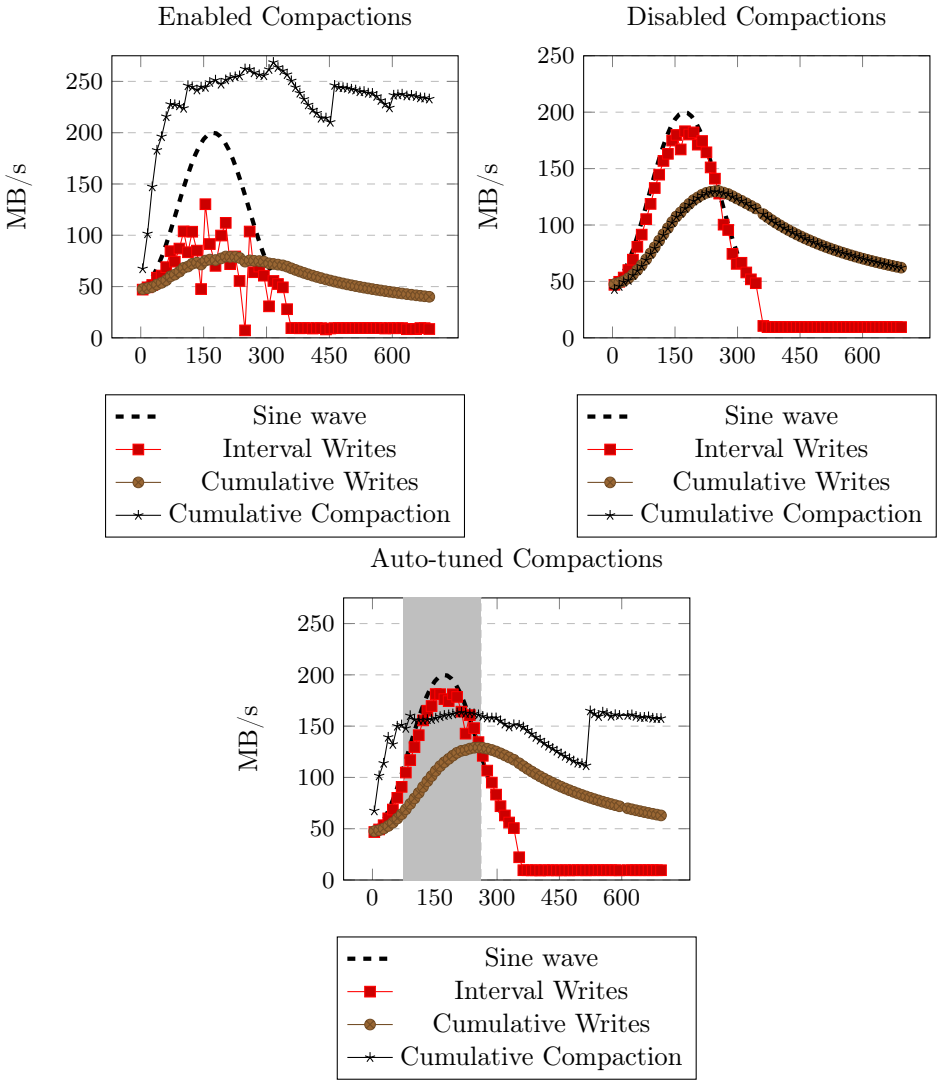

Figure 4.5: Plots of enabled, disabled and tuned compactions for 350 second sine wave, followed by 350sec of low load

Statistics 700sec

Having presented the statistics for the 350second peak, it was natural to identify compaction stats and read rates for the same configurations after the period of low load. Questions like how much time RocksDB uses to catch up compacting after a period with disabled compactions should prove helpful in terms of verifying the usefulness of the auto-tuner.

The results after 700seconds were satisfying. In comparison to the 350sec benchmark the write results were similar, but the random read rate of *Auto-tuned Compactions* vastly improved and achieved 53.6MB/s which is about 20% of the enabled with 35% more data. It should be noted that since the tuned version inserts more data, it subsequently increases the need for compactions. This means that it need use more time than the enabled to achieve the same read performance. Another interesting observation is that the read rate for *Enabled Compactions* at this point was over three times higher than after 350sec.

Since the `sine_finished_write_rate_limit` was capped at 10MB/s, the enabled benchmarks were able to cope with the new writes meaning that the percentage difference between the tuned, disabled and enabled was reduced. The tuned version managed to ingest 462K writes, only 2K less than disabled. Enabled did only manage 291K, which is the reason for the DB size being 13.3GB smaller than the auto-tuned. Despite the auto-tuned having improvements in most areas, the level hierarchy was not completely optimal. As discussed earlier (Section 2.3.2) the level-1 target size is 300MB, level-2 3GB and level-3 30GB, and the tuned was not able to compact down to level-3. However, it is order-of-magnitude better than the completely disabled version, where files just keep accumulating in level-0 and random read rate is catastrophic 3.3MB/s.

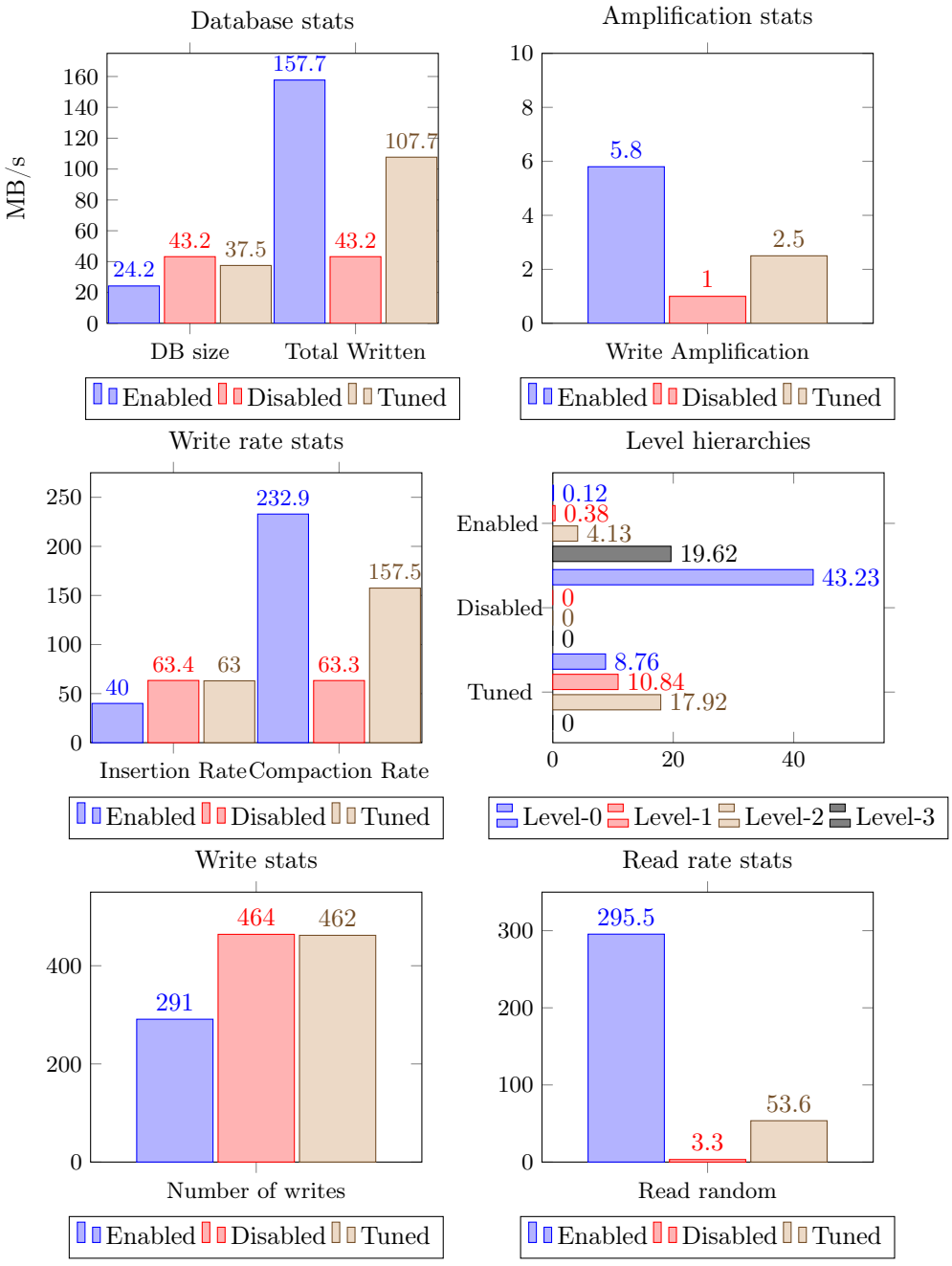

Figure 4.6: Statistics post 700sec

4.3.3 1000 sec

Benchmark commands

Enabled Compactions:

```
./db_bench -benchmarks="fillrandom,stats" -statistics -duration=1000
    -value_size=100000 -compression_type=None -sine_write_rate=true
    -sine_finished_seconds=350 -sine_finished_write_rate_limit=10000000
    -sine_a=75000000 -sine_b=0.017942857 -sine_c=4.71 -sine_d=125000000
    -stats_interval_seconds=10 -stats_per_interval=1
    -max_write_buffer_number=6 -max_background_flushes=4
    -max_background_compactions=2
```

Disabled Compactions:

```
./db_bench -benchmarks="fillrandom,stats" -statistics -duration=1000
    -value_size=100000 -compression_type=None -sine_write_rate=true
    -sine_finished_seconds=350 -sine_finished_write_rate_limit=10000000
    -sine_a=75000000 -sine_b=0.017942857 -sine_c=4.71 -sine_d=125000000
    -stats_interval_seconds=10 -stats_per_interval=1
    -max_write_buffer_number=6 -max_background_flushes=4
    -level0_slowdown_writes_trigger=10000 -level0_stop_writes_trigger
        =10000
    -level0_file_num_compaction_trigger=10000 -disable_auto_compactions=
        true
```

Auto-tuned Compactions:

```
./db_bench -benchmarks="fillrandom,stats" -statistics -duration=1000
    -value_size=100000 -compression_type=None -sine_write_rate=true
    -sine_finished_seconds=350 -sine_finished_write_rate_limit=10000000
    -sine_a=75000000 -sine_b=0.017942857 -sine_c=4.71 -sine_d=125000000
    -stats_interval_seconds=10 -stats_per_interval=1
    -max_write_buffer_number=6 -max_background_flushes=4
    -max_background_compactions=2 -subcompactions=2
    -auto_tuned_compactions=true -rate_limiter_bytes_per_sec=380000000
```

Plots 1000 sec

Fig. 4.7 presents plots after 1000sec, which basically is just an extended version of the 700sec run (Fig. 4.5) with additional 300sec with a write rate at 10MB/s.

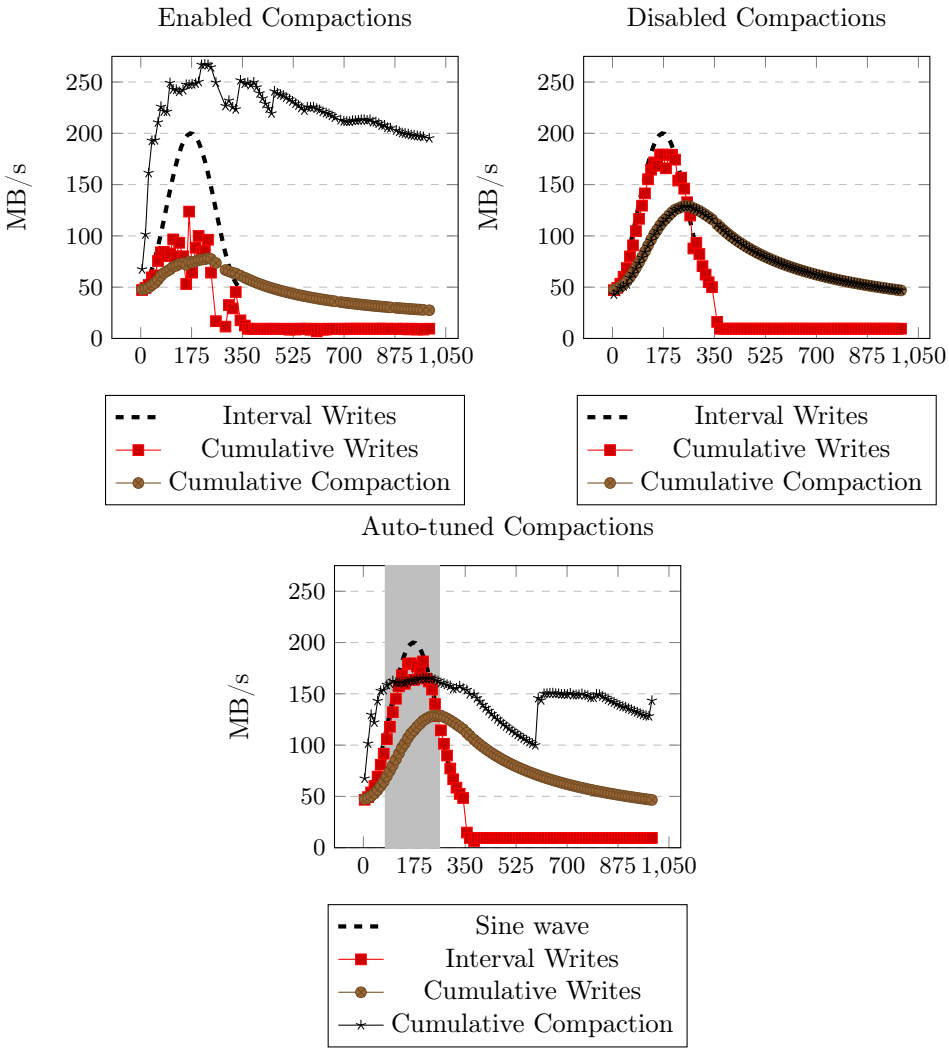

Figure 4.7: Plots of enabled, disabled and tuned compactions for 350 sec sine wave load following a 650seconds of low load

Statistics 1000sec

The statistics for the 1000sec run are consistent with the 700sec run, and as expected the level hierarchy for the auto-tuned and enabled is better distributed. However, the tuned does not have a completely optimal hierarchy since Level-2 consists of 11.55GB data with a target at 3GB.

As with the prior benchmarks, the random read rate is improved. At this point we achieve 200MB/s with auto-tuned compactions, which is a satisfying throughput rate. Having enabled compaction and optimal level hierarchy we achieve almost 400MB/s. Despite the better read rate, it still has a much higher write amplification.

In total, these results live up to the intention of the project and prove that this proof-of-concept has merit. These benchmarks show that it provides overall disk efficiency improvements for periodic write-intensive workloads, but it could most likely be applied to many other workloads.

On a side note; when performing benchmarks it is important to ensure that they are reproducible, and thus we conducted them multiple times. With the auto-tuner, the compaction trigger points are not identical for every benchmark run. It generally triggers at similar points (∼100-120MB/s), but each run is always a bit different and some discrepancies are observed. This means that for a small portion of the runs the *Number of writes* were lower, due to different compaction trigger points, but that had the effect that the read rate was better since more time was spent doing compactions – and vice versa.

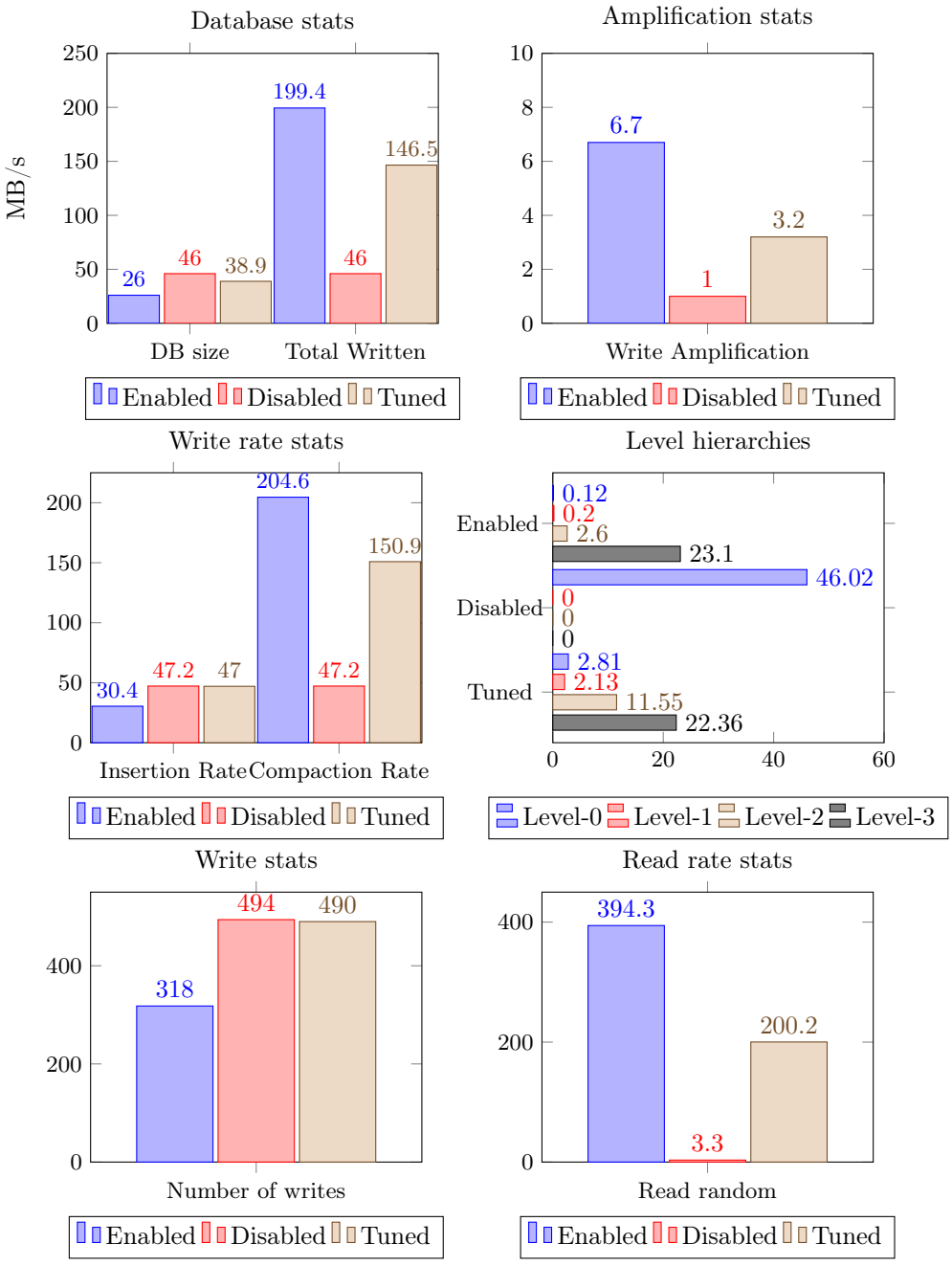

Figure 4.8: Statistics for sine wave after 1000sec

4.4 Bloom filters

As discussed in Section 2.5, using bloom filters increases the read performance despite having many files in the levels. Also, Siying Dong, a RocksDB core developer, mentions the possibility to rely on bloom filters for acceptable read performance for insert heavy workloads with compactions disabled[35]. Since our random read rates for the auto-tuner and disabled compactions was ∼3-4MB/s, it was interesting to see what performance gain we could achieve by enabling them.

We configured RocksDB to use bloom filters by adding `-bloom_bits=10`. The benchmarks conducted are identical to the 350sec in Section 4.3.1 and the read benchmark in Fig. 4.2, only with this bloom bits flag added.

Fig. 4.9 reveals that the read improvements are quite impressive and that bloom filters are very advantageous. The auto-tuner achieves a read performance of 108.9MB/s, which only strengthens the value of auto-tuning compactions. Even with compactions disabled entirely, the bloom filters shows acceptable read throughputs – achieving 79.7MB/s. This means that by using bloom filters, we can get both great read performance during the periods of high write load and performing compactions when the load is low.

Statistics

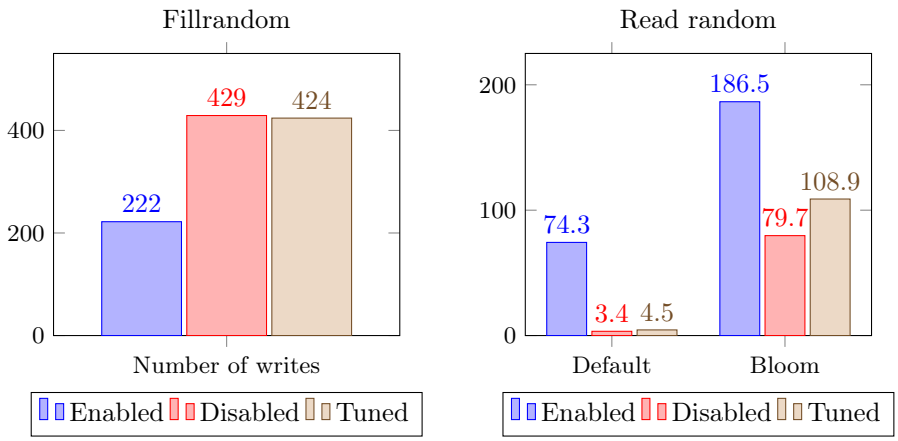

Figure 4.9: Statistics 350sec with bloom filters enabled

4.5 Enabled Compaction slowdowns

Since the *Enabled Compaction* configuration was without the increased `level0` parameters and pending compaction bytes, two quick benchmarks were performed to compare how much they affect the performance. The intention of this section is to show that the improvements in the auto-tuner is not because of these parameters.

As shown in Fig. 4.10, the primary difference lies in the the write rate X = 250sec. With the level0 stalling parameters increased, it is able to avoid the slowdown which results in cumulative write rate at 81.38MB/s, with stalling it is 61.92MB/s. An improvement, however, still less than the 116.34 MB/s achieved by the auto-tuner (Fig. 4.4). The slowdown parameters' intention is to throttle incoming writes to allow better read performance while writing, meaning that increasing these reduces the expected read performance.

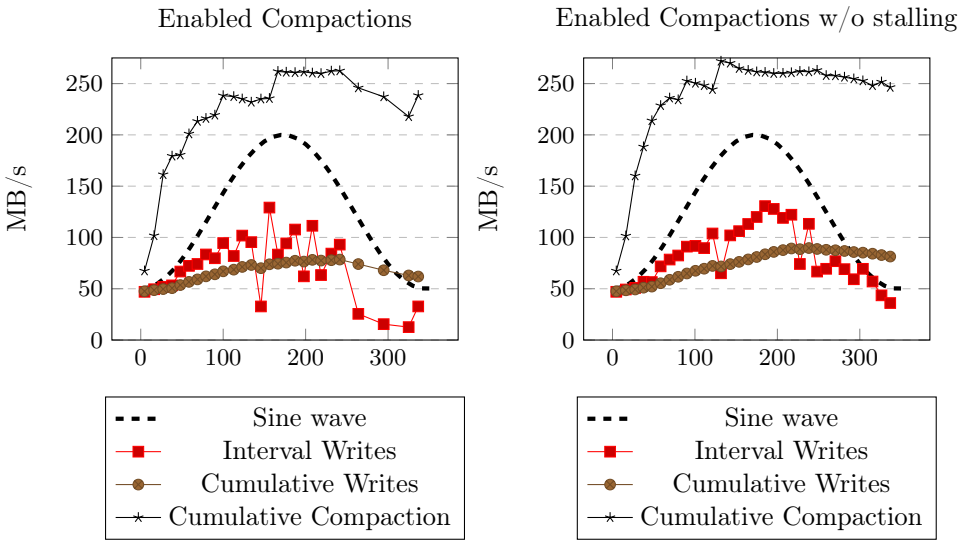

Figure 4.10: Benchmarks that shows how level-0 and pending compaction bytes stalls

4.6 Max throughput

4.6.1 250K fillrandom

Benchmark commands:

Enabled Compactions:

```
./db_bench -benchmarks="fillrandom,stats" -statistics
    -writes=500000 -value_size=100000 -max_write_buffer_number=6
    -compression_type=None -stats_interval_seconds=10 -stats_per_interval
        =1
    -max_background_flushes=4 -max_background_compactions=2
```

Disabled Compactions:

```
./db_bench -benchmarks="fillrandom,stats" -statistics
    -writes=500000 -value_size=100000 -max_write_buffer_number=6
    -compression_type=None -stats_interval_seconds=5 -stats_per_interval=1
    -max_background_flushes=4 -max_background_compactions=2
    -level0_slowdown_writes_trigger=10000 -level0_stop_writes_trigger
        =10000
    -level0_file_num_compaction_trigger=10000 -disable_auto_compactions=
        true
```

Auto-tuned Compactions:

```
./db_bench -benchmarks="fillrandom,stats" -statistics
    -writes=500000 -value_size=100000 -max_write_buffer_number=6
    -compression_type=None -stats_interval_seconds=10 -stats_per_interval
        =1
    -max_background_flushes=4 -max_background_compactions=2
    -auto_tuned_compactions=true -rate_limiter_bytes_per_sec=380000000
```

Benchmarks were also conducted using maximal throughput for 250K `fillrandom` writes. As shown in Fig. 4.11, especially with *Disabled Compactions* we are able to continuously hold a stable high write rate of ∼200MB/s.

Enabled Compactions is more unstable, due to the stalling mechanisms like `level-0` slowdowns described in Section 3.3.1. It does maintain a high *Cumulative Compaction* rate at ∼250MB/s, which is ∼50MB/s higher than the compaction rate for the disabled. Still, it has issues with its own *Cumulative Write* rate.

For the *Auto-tuned Compactions*, it starts similar to the *Enabled Compactions* when compactions are enabled initially. This is visible in the grey shaded area of the Fig. 4.11. Afterwards it converges against the same pattern like the *Disabled Compactions*, only differing with a few MB/s due to rate-limiting and tuning overhead.

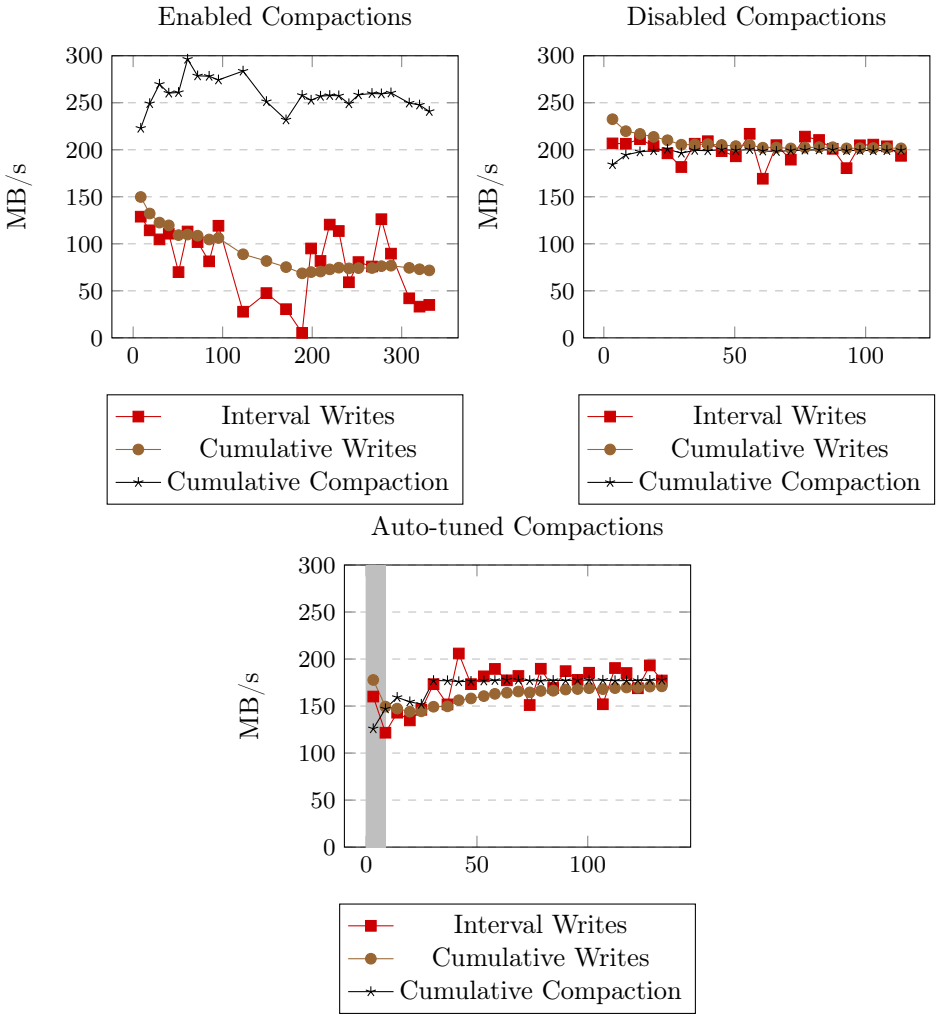

Figure 4.11: 500K random writes at max throughput

4.6.2 250K overwrite

Benchmark commands:

Command used for filling DB with 250K entries:

```
./db_bench -benchmarks="fillseq" -statistics
    -writes=250000 --value_size=100000 --max_write_buffer_number=6
    -level0_slowdown_writes_trigger=10000 -level0_stop_writes_trigger
        =10000
    -level0_file_num_compaction_trigger=4 -compression_type=None
    -max_background_flushes=4 -disable_auto_compactions=true
```

Enabled Compactions:

```
./db_bench -benchmarks="overwrite,stats" -statistics
    -writes=250000 -value_size=100000 -max_write_buffer_number=6
    -compression_type=None -stats_interval_seconds=10 -stats_per_interval
        =1
    -max_background_flushes=4 -max_background_compactions=2
    -use_existing_db=true
```

Disabled Compactions:

```
./db_bench -benchmarks="overwrite,stats" -statistics
    -writes=250000 -value_size=100000 -max_write_buffer_number=6
    -compression_type=None -stats_interval_seconds=10 -stats_per_interval
        =1
    -max_background_flushes=4 -max_background_compactions=2
    -level0_slowdown_writes_trigger=10000 -level0_stop_writes_trigger
        =10000
    -level0_file_num_compaction_trigger=10000 -disable_auto_compactions=
        true
    -use_existing_db=true
```

Auto-tuned Compactions:

```
./db_bench -benchmarks="overwrite,stats" -statistics
    -writes=250000 -value_size=100000 -max_write_buffer_number=6
    -compression_type=None -stats_interval_seconds=10 -stats_per_interval
        =1
    -max_background_flushes=4 -max_background_compactions=2
    -auto_tuned_compactions=true -rate_limiter_bytes_per_sec=380000000
    -use_existing_db=true
```

We also conducted benchmarks where the database is filled with 250K writes without compacting anything. The primary intention is to see whether the auto-tuner is able to disable compactions when the compaction pressure is very high initially.

In the *Enabled Compactions* it is very unstable since different stalling factors are affecting the write rate. It does not insert anything new into the database until after 142 seconds, due to the compaction pressure from `level-0`. It allows new writes once the compactions have caught up, but drops quickly once the compaction pressure increases again. Hence, new writes suffers massively when having a lot of data to compact.

Disabled Compactions and *Auto-tuned Compactions* shows similar behaviour to the 250K fillrandom in Fig. 4.11. The positive result here is that the tuner ignores stalling factors and lets the flushes continue without prioritising compactions. Our tuner would not be able to trigger if it stalled writes.

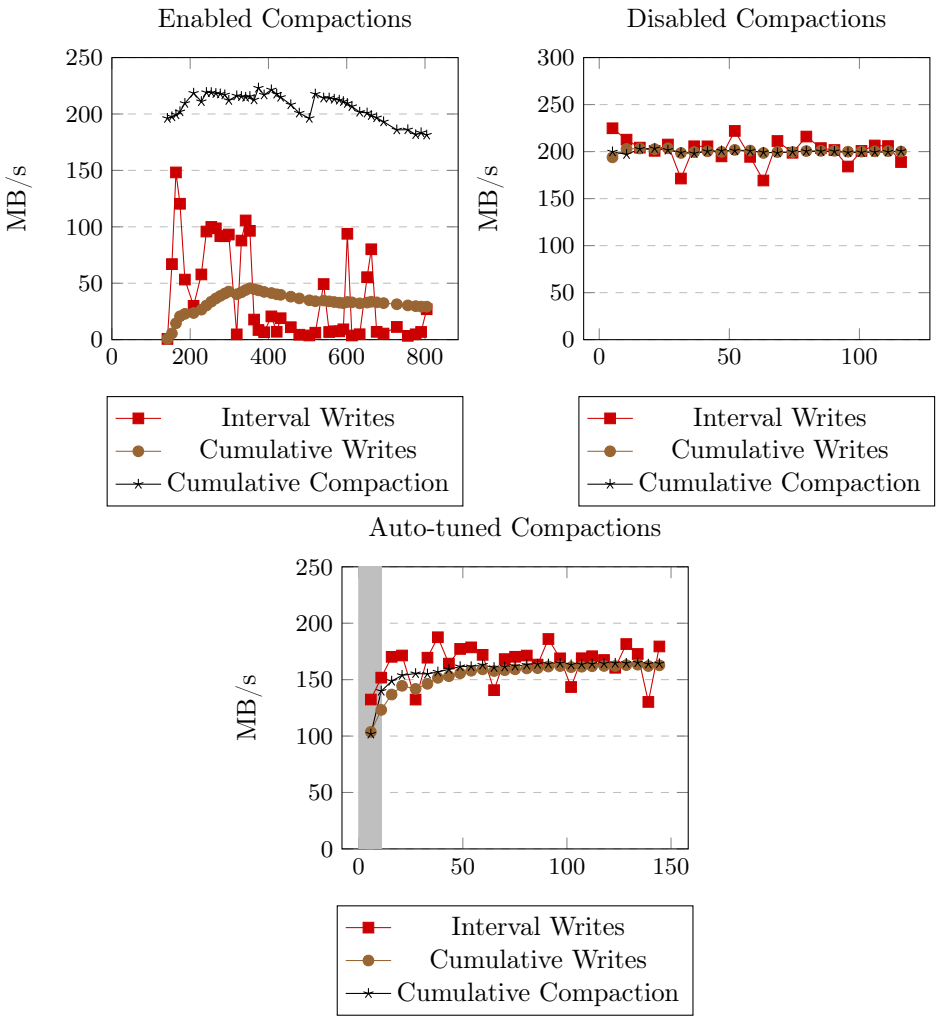

Figure 4.12: Overwriting 250K values

Chapter 5

Conclusion and Future Work

The following chapter evaluates the research goals presented in Section 1.2, the outcome of the implementation and discusses potential areas of improvement for future work.

5.1 Conclusion

The purpose of this thesis was to enable RocksDB to tune compactions automatically and use it to achieve better write throughput. We presented two research goals and a research question in Section 1.2 that concretised the project. In addition, the hypothesis (Section 3.1) postulated the performance potential of auto-tuning compactions.

> **G1: Make RocksDB able to disable and enable compactions automatically.**
>
> **G2: Increase RocksDB's write performance and provide a tuning baseline for other LSM-based databases.**
>
> **RQ1: How can we implement a compaction auto-tuner that dynamically toggles compaction, how does it benefit RocksDB and at what cost does it come?**

Throughout the project we have answered **RQ1**; the *Implementation* Chapter 3 presents an implementation of a proof-of-concept auto-tuner – achieving **G1**, and the *Benchmark* Chapter 4 shows the results and discusses the pros and cons – achieving **G2**. In total, we are able to conclude that write-intense workloads can benefit from auto-tuning compactions, and with bloom filters we can avoid bad read performance when compactions are disabled as well. Improving write performance was a presumed outcome since we knew that compactions were resource demanding. On the other hand, we could not assume that we would be able to implement a proof-of-concept that gave such satisfying results. Nevertheless, the proof-of-concept is by no means a perfect implementation. There are a lot of areas that can be improved; especially the trigger conditions (Section 3.3.7), the rate-limiting (Section 3.5.1) and thread allocation (Section 3.5.3).

After finishing the implementation, I posted an excerpt of this thesis including the benchmarks and implementation details at the RocksDB Developer Forum[18]. The post attracted interest from multiple people, including many of the RocksDB core developers at Facebook. Siying Dong, one of the core developers, commented that the adaptive compactions and smooth write throttling were something they prioritised, and that the research gave a good perspective and could be an inspiration into solving the problem (Fig. 5.1). He also asked me to contribute the sine-wave benchmarking code from Section 3.4.2 to RocksDB, which was reviewed and approved 31. May 2018[36]. Being able to contribute to a major database system, combined with the positive feedback, is an accomplishment I am proud of.

Figure 5.1: Feedback from RocksDB Developer Public

5.2 Future Work

Rate-limiting Compactions

An especially interesting idea stumbled upon while working with compactions and rate-limiter was to rate-limit the compactions exclusively. By calculating a compaction rate-limit based on the flush rate, one could potentially gain better I/O usage than a static disable/enable threshold. The idea is somewhat similar to how active noise control works, but instead of cancelling sound to zero – we sum the rates to the maximum rate write rate. This could be done by intervally calculate the optimal compaction rate using determined flush rate and the maximum write rate. This way one could possibly optimise the I/O efficiency even more, and sustain a higher write rate combined with better read throughput.

Revisit rate-limiting

This proof-of-concept rate-limits the max cap (Section 3.5.1), achieving better write throughput. This is the case because the max rate limit is set to the max flush rate. However, the max compaction rate can utilise the I/O better than the flushes, meaning that the compactions are throttled since the cap is too low. This is shown in the plots where the *Cumulative Compaction* rate is much higher with enabled compactions than tuned compactions (Fig. 4.5). There are multiple approaches to tweak and avoid this from happening; one could lower the disable compaction trigger watermark and increase the rate-limiter, or drop the actual rate limiting and mimic the drain variables exclusively for the I/O detection. The main issue encountered in terms of further improvements was the stability of the compaction vs flush priority I/O percentages; thus the rate limiting proved useful to gain more stable results.

Regarding rate-limiting, the `fairness` variable (Section 3.3.4) could also be interesting to further evaluate since it is used to allow low priority requests to be completed. Our intention is to prioritise flushes over compactions, hence the fairness variable could be increased to reduce the amount of low-pri requests (compactions) being accepted above high-pri (flushes).

Cancel Compactions

An idea to improve the reaction time of the insertion rate when disabling compactions is to cancel initiated compactions. This way the insertion rate will be able to utilise I/O for flushes without waiting for the compactions to finish.

Manual Compactions

In the proof-of-concept implemented in this project, we enable compactions and let RocksDB handle the compaction picking itself. It could be applicable to tailor the compaction strategy post peak, and potentially determine a more optimal compaction to perform when enabling compactions again.

References

[1] Facebook. *Dynamic Level Size*. URL: https://rocksdb.org/blog/2015/07/23/dynamic-level.html (visited on 05/05/2018).

[2] Andrew Kryczka. *RocksDB: Auto-tuned Rate Limiter*. URL: https://rocksdb.org/blog/2017/12/18/17-auto-tuned-rate-limiter.html (visited on 05/05/2018).

[3] *Wikipedia: Horizontal Scaling*. URL: https://en.wikipedia.org/wiki/Scalability#Horizontal_and_vertical_scaling (visited on 29/05/2018).

[4] Michael Stonebreaker. *It's Time for a Complete Rewrite*. URL: http://nms.csail.mit.edu/~stavros/pubs/hstore.pdf (visited on 01/06/2018).

[5] CMU: Carnegie Mellon University. *Peloton*. URL: https://github.com/cmu-db/peloton (visited on 05/05/2018).

[6] Oracle. *Autonomous Database*. URL: https://www.oracle.com/database/autonomous-database/feature.html (visited on 05/05/2018).

[7] Hans-Wilhelm Kirsch Warlo. *TDT4501: Tuning RocksDB for peak performance*.

[8] Redis Labs. *Redis*. URL: https://redis.io/ (visited on 01/12/2017).

[9] Google. *LevelDB*. URL: https://github.com/google/leveldb (visited on 01/11/2017).

[10] Facebook. *RocksDB*. URL: http://rocksdb.org/ (visited on 01/11/2017).

[11] Facebook. *MyRocks*. URL: http://myrocks.io/ (visited on 08/12/2017).

[12] Patrick O'Neill. *LSM Tree*. URL: https://www.cs.umb.edu/~poneil/lsmtree.pdf (visited on 01/06/2018).

[13] Andrew Kryczka. *Level0-Level0 compactions*. URL: https://rocksdb.org/blog/2017/06/26/17-level-based-changes.html (visited on 20/05/2018).

[14] Facebook. *RocksDB: Leveled Compaction Strategy*. URL: https://github.com/facebook/rocksdb/wiki/Leveled-Compaction (visited on 11/05/2017).

[15] Datastax. *Apache Cassandra: Compaction configuration*. URL: https://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsConfigureCompaction.html (visited on 08/12/2017).

[16] Facebook. *RocksDB Wiki: Universal Compaction level issue*. URL: https://github.com/facebook/rocksdb/wiki/Universal-Compaction#db-column-family-size-if-num_levels1 (visited on 08/12/2017).

[17] Google. *Snappy performance*. URL: https://github.com/google/snappy#performance (visited on 01/12/2017).
[18] *RocksDB Developer Public*. URL: https://www.facebook.com/groups/rocksdb.dev/ (visited on 22/03/2018).
[19] Mark Callaghan. *Small Datum*. URL: http://smalldatum.blogspot.com/ (visited on 05/05/2018).
[20] *MIT License rocksdb-statistics*. URL: https://github.com/hanswilw/rocksdb-statistics/blob/master/LICENSE.
[21] Hans-Wilhelm Kirsch Warlo. *RocksDB statistics parser*. URL: https://github.com/hanswilw/rocksdb-statistics (visited on 26/04/2018).
[22] Wikipedia. *Regular Expressions*. URL: https://en.wikipedia.org/wiki/Regular_expression (visited on 31/05/2018).
[23] Christian Feuersänger. *Pgfplots*. URL: http://pgfplots.sourceforge.net/ (visited on 31/05/2018).
[24] Facebook. *RocksDB: PrepareForBulkLoad()*. URL: https://github.com/facebook/rocksdb/blob/04c11b867df9190da204e38357a14d20296fb244/options/options.cc#L342 (visited on 05/05/2018).
[25] Facebook. *RocksDB SetOptions() API*. URL: https://github.com/facebook/rocksdb/wiki/Basic-Operations#rocksdb-options (visited on 05/05/2018).
[26] *pragma once*. URL: https://en.wikipedia.org/wiki/Pragma_once (visited on 05/05/2018).
[27] Wikipedia. *Token Bucket rate-limiting*. URL: https://en.wikipedia.org/wiki/Token_bucket (visited on 31/05/2018).
[28] Facebook. *RocksDB Rate-Limiter Wiki*. URL: https://github.com/facebook/rocksdb/wiki/Rate-Limiter (visited on 05/05/2018).
[29] et al E. Altman K. Avrachenkov. *Multiplicative Increase Multiplicative Decrease*. URL: https://pdfs.semanticscholar.org/98e1/cce52f4c927bd2d2bd1a7c8a8c pdf (visited on 05/05/2018).
[30] Google. *gflags*. URL: https://github.com/gflags/gflags (visited on 05/07/2018).
[31] Facebook. *RocksDB: PrepareForBulkLoad*. URL: https://github.com/facebook/rocksdb/blob/492ab7c7d9425408b05b338eae5c581165af3463/options/options.cc#L336 (visited on 20/05/2018).
[32] Wikipedia. *Starvation (Computer Science)*. URL: https://en.wikipedia.org/wiki/Starvation_(computer_science) (visited on 31/05/2018).
[33] *dd*. URL: https://en.wikipedia.org/wiki/Dd_(Unix) (visited on 31/05/2018).
[34] Facebook and Google. *RocksDB's benchmarking tool: db_bench*. URL: https://github.com/facebook/rocksdb/wiki/Benchmarking-tools (visited on 24/05/2018).
[35] *Reddit AMA: RocksDB*. URL: https://www.reddit.com/r/IAmA/comments/3de3cv/we_are_rocksdb_engineering_team_ask_us_anything/ (visited on 24/05/2018).
[36] *Pull Request: Benchmark sine wave write limit*. URL: https://github.com/facebook/rocksdb/pull/3914 (visited on 02/06/2018).

Appendix A

rocksdb-statistics: Statistics Parser

```
#!/usr/bin/env python3
import re, argparse
import os
from itertools import accumulate


class Statistics:
    def __init__(self):
        self.uptime = 'Uptime\(secs\).*?(\d*\.\d*)\stotal'
        self.interval = {
            'name': 'Interval step',
            'regex': 'Uptime\(secs\).*?(\d*\.\d*)\sinterval',
            'suffix': '_intervals'
        }
        self.interval_stall = {
            'name': 'Interval Stall',
            'regex': 'Interval\sstall.*?(\d*\.\d*)\spercent',
            'suffix': '_interval_stall'
        }
        self.cumulative_stall = {
            'name': 'Cumulative Stall',
            'regex': 'Cumulative\sstall.*?(\d*\.\d*)\spercent',
            'suffix': '_cumulative_stall'
        }
        self.interval_writes = {
            'name': 'Interval Writes',
            'regex': 'Interval\swrites.*?(\d*\.\d*)\sMB\/s',
            'suffix': '_interval_writes'
        }
```

```
        self.cumulative_writes = {
            'name': 'Cumulative Writes',
            'regex': 'Cumulative\swrites.*?(\d*\.\d*)\sMB\/s',
            'suffix': '_cumulative_writes'
        }
        self.cumulative_compaction = {
            'name': 'Cumulative Compaction',
            'regex': 'Cumulative\scompaction.*?(\d*\.\d*)\sMB\/s',
            'suffix': '_cumulative_compaction'
        }
        self.interval_compaction = {
            'name': 'Interval Compaction',
            'regex': 'Interval\scompaction.*?(\d*\.\d*)\sMB\/s',
            'suffix': '_interval_compaction'
        }

        self.legend_list = []
        self.base_filename = ''

    def coordinates_filename(self):
        return self.base_filename + '_coordinates.log'

    def save_statistic(self, d, log, steps=None):
        matches = self.get_matches(d['regex'], log)
        new_filename = self.base_filename + f'{d["suffix"]}.csv'
        self.save_to_file(matches, new_filename)

        coordinates = self.generate_coordinates(matches, steps)
        self.save_coordinates_to_file(coordinates, self.
            coordinates_filename())
        self.legend_list.append(d["name"])

    def clean_log(self, log):
        regex = re.compile('(2018\S+).*\(([\d,\.]*)\).*\(([\d,\.]*)\)
            .*\(([\d,\.]*)\)')
        path = os.path.join(os.getcwd(), 'output', log)
        with open(path, 'r') as f:

            matches = regex.findall(f.read())
        return [','.join(match) for match in matches]

    def get_matches(self, regex, log):
        regex = re.compile(regex)
        path = os.path.join(os.getcwd(), log)
        with open(path, 'r') as f:
            matches = regex.findall(f.read())
        return matches

    def generate_coordinates(self, matches, steps):
        if not steps:
```

```
            return [f'({i*1},{match})' for i, match in enumerate(matches)]
        return [f'({key},{value})' for key, value in zip(steps, matches)]

    def save_to_file(self, data, filename):
        os.makedirs('output', exist_ok=True)
        with open(f'output/{filename}', 'w') as f:
            f.writelines('\n'.join(data))

    def save_coordinates_to_file(self, data, filename, last=False):
        os.makedirs('output', exist_ok=True)
        with open(f'output/{filename}', 'a') as f:
            str_data = ''.join(data)
            f.write('\\addplot\n\tcoordinates {{ {0} }};\n'.format(str_data
                ))
            if last:
                legend = ', '.join(self.legend_list)
                f.write(f'\\legend{{{legend}}}\n')

    def append_legend(self, filename):
        with open(f'output/{filename}', 'a') as f:
            legend = ', '.join(self.legend_list)
            f.write(f"""
\\legend{{{legend}}}
\\end{{axis}}
    \\end{{tikzpicture}}
    \\end{{subfigure}}
""")

    def initialize_coordinate_file(self, filename):
        axis = f"""   \\begin{{subfigure}}[t]{{0.5\\textwidth}}
    \\begin{{tikzpicture}}
\\begin{{axis}}[
    title={self.base_filename},
    xlabel={{}},
    ylabel={{MB/s}},
    ymin=0,
    ymax=250,
    ytick={{0,50,...,300}},
    width=\\textwidth,
    legend style={{
        at={{(0.5,-0.2)}},
        anchor=north,legend columns=1
    }},
    ymajorgrids=true,
    grid style=dashed,
]
"""
        with open(f'output/{filename}', 'w') as f:
            f.write(axis)
```

```
    def get_steps(self, regex, log):
        interval_steps = self.get_matches(regex, log)[::2]
        accumulated_steps = list(accumulate([float(step) for step in
            interval_steps]))
        rounded_steps = [round(step, 2) for step in accumulated_steps]
        return rounded_steps

    def save_all(self, log):
        self.base_filename = log.split('.')[0]
        interval_steps = self.get_steps(self.interval['regex'], log)
        uptime_steps = [float(step) for step in self.get_matches(self.
            uptime, log)[::2]]
        min_interval_step = uptime_steps[0] - interval_steps[0]
        steps = [round(step - min_interval_step, 2) for step in
            uptime_steps]
        s.initialize_coordinate_file(self.coordinates_filename())
        s.save_statistic(self.interval_writes, log, steps)
        s.save_statistic(self.cumulative_writes, log, steps)
        # s.save_statistic(self.interval_stall, log)
        # s.save_statistic(self.cumulative_stall, log)
        s.save_statistic(self.interval_compaction, log, steps)
        s.save_statistic(self.cumulative_compaction, log, steps)
        s.append_legend(self.coordinates_filename())


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument("log", type=str, help="logfile")
    args = parser.parse_args()
    s = Statistics()
    log = args.log
    s.save_all(log)
```