GravelDB: Introducing Fragmented Log-Structured Merge Trees to RocksDB

Chris Sun and Carolyn Tran

Abstract

Widespread adoption of RocksDB as a key-value store for fast storage environments has made the open-sourced project an appealing target for further improvement. The introduction of Fragmented Log-Structured Merge Trees can provide users of RocksDB with an optimization from a write amplification perspective. The primary concern with high write amplification beyond even performance considerations is the fact that it quickens the degradation of solid state drives, which is especially concerning given the current time, when users are looking to store extremely large volumes of data. By implementing the concept of guards, a structure that adds a level of abstraction to organizing key-value data and loosens previous invariants enforced by the Log Structured Merge Tree data structure, the compaction process that organizes data on disk can then be modified to perform less write IO. GravelDB applies these high level concepts of PebblesDB onto RocksDB and successfully reduces write amplification by 10%.

I. INTRODUCTION

The popularity of key-value stores as the backbone of data intensive services and organizations reveals a primary concern for these systems going forward. Storing the volume of data demanded by users nowadays necessarily requires transferring data from memory to caches to disks. With these changes in storage devices comes the need to rewrite information, leading to high write amplification. Write amplification is defined as the volume of write IO performed by the system divided by the volume of data the user wrote to the system. High write amplification is of particular concern when systems use solid state drives, as these storage devices have limited write cycles before the high bit error rate renders them unusable. As such, decreasing write amplification for popular key-value stores is of significant concern for many services and organizations going forward.

RocksDB, one such key-value store, was initially released in May 2012 and has been maintained by Facebook since its onset. The database has a very broad user base, including Facebook, LinkedIn, Netflix, Airbnb, Uber, and more. It is also used with larger database management systems, such as in MyRocks (which uses RocksDB as the backend storage for all MySQL functionality) and MongoRocks (which is a storage engine for MongoDB), both of which are maintained by Facebook. The broad applications of the key-value store makes RocksDB an especially compelling target for improving write amplification.

II. BACKGROUND

A. Key-Value Stores

Key-value stores are databases that contain mappings from an identifying key to information associated with that key, collectively referred to as the corresponding value. Whereas relational databases rely on pre-defined schemas that specify the number and type of fields associated with a given key, key-value stores treat values as whole entities, without further consideration as to the specific fields contained within. In doing so, key-value stores allow users to have various internal structures of values associated with keys within the same database. Furthermore, this principle also allows for potential benefits on the storage side. Rather than reserve set numbers of bytes for every field in a relational database, key-value stores can take advantage of variation in value size and condense data storage when possible.

Distributed systems today rely on key-value stores for storage, which means that as distributed systems become more popular and widespread, key-value store performance is also becoming increasingly important. Key-value stores frequently support the following operations:

1) **`Get`**: This allows a user to retrieve the value associated with a given key.
2) **`Put`**: This allows a user to add a key-value pairing to the database.
3) **`Range Query`**: This allows a user to retrieve all key-value pairs associated with keys in a given range.
4) **`Iterator`**: The returned iterator allows a user to traverse the keys in the database. In some cases, the user can specify how keys should be ordered for his iterating purposes.

B. Log-Structured Merge Tree Databases (LevelDB and RocksDB)

RocksDB derives a significant portion of its code base from LevelDB, Google's key-value storage library. RocksDB was originally built to best accommodate server workloads on fast storage (such as flash memory), although it has thereafter expanded its support so that it is configurable for a wide variety of production environments and workloads.

Since RocksDB originally branched off from LevelDB's code base, both use the log-structured merge tree to organize its key-value pairs. In a log-structured merge tree, keys are organized sequentially in *sstables* (sorted string tables) and written to storage. These sstables are then grouped into levels, with older insertions being pushed into levels denoted by larger numbers through a process referred to as compaction. Compaction compresses the data in a database by removing any unused data and reorganizing the data in an efficient way depending on the database. Below, we discuss how databases using the log-structured merge tree data structure (such as LevelDB and RocksDB) support the previously mentioned key-value store operations.

1) **`Get`**: When asked to retrieve the value for a given key k, RocksDB will first query the memtable data structure - an in-memory collection of the most recently inserted / updated key-value pairs. It is necessary to check this in-memory data structure first because the memtable necessarily contains the most up-to-date information. If the key exists in the memtable, the database will return its associated value.

   If k is not in the memtable, RocksDB will then search in each level in ascending level number order. This ordering is necessary to ensure that the most recent information is returned. For each level, RocksDB will first perform a binary search on the key ranges of the sstables in order to determine which sstable's key range includes k. Because the sstables must contain disjoint key ranges, it must be the case that at most one sstable is selected for consideration per level. The selected sstable, if there exists one, will then be binary searched for k, and the associated value will be returned if k is found. If k is not found in the selected sstable or there does not exist an sstable that covers a key range containing k at the current level, RocksDB will move down to the subsequent level for consideration.
2) **`Put`**: Similar to the read operation, writing new values to the database also involves the memtable data structure. The data of the key-value pair specified by the `Put` operation is stored in the memtable. Once the memtable is fully populated, its contents are consolidated into an sstable and moved to level 0 of the log-structured merge tree. The compaction process (described in §VIII) moves sstables to level $n+1$ when level n becomes full.
3) **`Range Query`**: This operation will perform the same sequence of steps as a Get but will return a range, not just a single value. The implementation often relies on the `iterator` described below. The database will iterate through all keys in the range query to return these results.
4) **`Iterator`**: Because there is no guarantee about the ranges of keys at each level (only that the keys in each level are sorted), an iterator over all keys in the database is implemented via a combination of iterators, one at each level. With each `next()` call, the client's iterator will return the lowest value across all the level-specific iterators and increment that level's iterator in anticipation of the subsequent `next()` call. Most databases also give the client the option of providing his own ordering across keys.

Write Amplification Concerns Two primary invariants of the log-structured merge tree data structure are that (1) all sstables on the same level n must contain disjoint sets of sorted keys, and (2) all keys

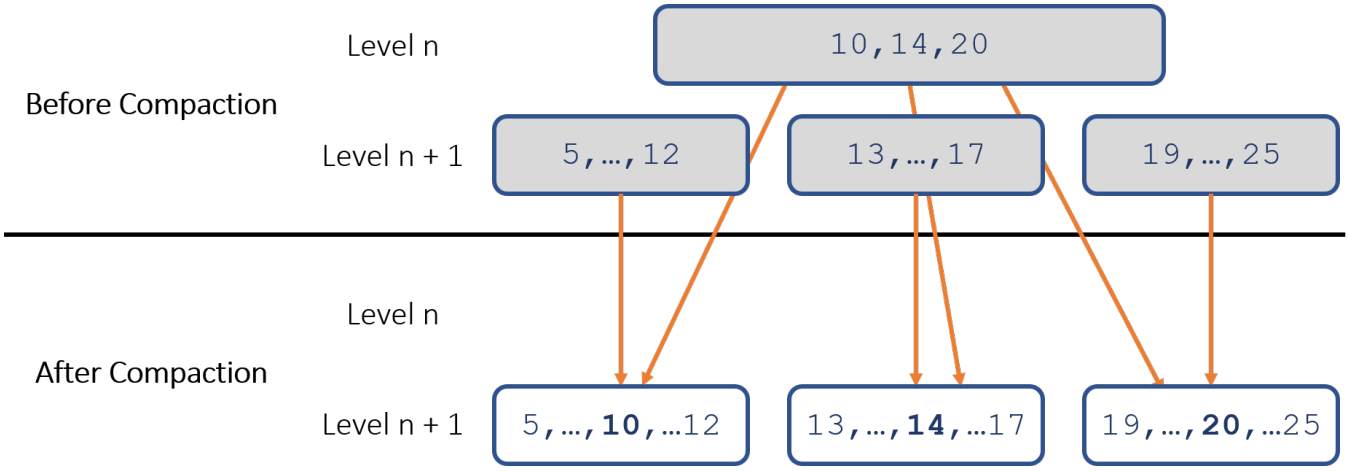


Fig. 1: Log-Structured Merge Tree Compaction. [Grey boxes denote the sstables that must be rewritten to facilitate this process.]

within an sstable are in sorted order. From our previous discussion of key-value store operations, we see that the latter is crucial for maintaining the high read performance of the database. However, these two requirements are also the root causes of high write amplification in the log-structured merge tree structure.

For example, consider an sstable *s* on level *n* with the set of keys $[10, 14, 20]$. If level *n+1* contains an sstable with a key range of $[9, ..., 21]$, the keys in both these sstables would need to be rewritten during compaction in order to ensure that no two sstables contain overlapping key ranges and that keys are always sequential when organized in an sstable. Furthermore, we observe that the amount of rewriting necessary is highly dependent on the number of sstable ranges that are overlapped. If level *n+1* contains three sstables with key ranges of $[5, ..., 12]$, $[13, ..., 17]$, and $[19, ..., 25]$, all three of these sstables would need to be rewritten in order to merge in the keys from *s*. This is purely for the requirements of the data structure; it is especially important to note that no data is being added, only reorganized. An illustration of this compaction process is provided in Figure 1.

III. FRAGMENTED LOG-STRUCTURED MERGE TREES

Fragmented log-structured merge trees have two primary structural differences from log-structured merge trees: (1) sstables fragments with non-disjoint key ranges are allowed on the same level, and (2) guards are used to organize sstables on the same level.

On the first point, our previous discussion of log-structured merge trees indicates that the root cause of high write amplification when using this data structure is the need to rewrite keys in an sstable from level *n* when that sstable is compacted to level *n+1* due to the structure's invariant that sstables must contain disjoint key ranges. Allowing non-disjoint key ranges is a natural solution to lowering write amplification. Under the same scenarios of the example drawn for log-structured merge trees, the FLSM structure would therefore allow an sstable containing the set of keys $[10, 14, 20]$ to coexist on the same level with sstables with key ranges of $[5, ..., 12]$, $[13, ..., 17]$, and $[19, ..., 25]$. However, this could dramatically decrease read performance because the assertion that a read operation only requires two binary searches would no longer hold true. In this example, when the database may have to binary search two sstables to find the key value of 11. The more overlapping key ranges allowed, the worse read performance will be.

Guards are the primary tool that FLSM introduces to lower write amplification while maintaining strong read performance. Guards act as dividers within a level and can contain multiple sstables. While two sstables within a guard can have overlapping key ranges, any two guards on the same level must contain disjoint key ranges (where the key range of a guard is defined as spanning from the minimum

key contained in any sstable in the guard to the maximum key contained in any sstable in the guard). Assuming two adjacent guards g_1 and g_2 in level *n* have guard values of v_1 and v_2, respectively, g_1 will contain all sstables whose key ranges fall in the range $[v_1, v_2)$. The first guard at each level is referred to as a sentinel guard and does not have a distinct guard value so as to not impose a lower bound on allowed keys. It encompasses keys in the range $(-\infty, v_1)$, assuming v_1 is the value of the first guard in that level that is not a sentinel guard. The last guard g_i in a level will contain all keys in the range $[v_i, \infty)$. Next we consider the effects that these two structural differences together have on write amplification and read performance.

Write Amplification From a write amplification standpoint, the benefit of FLSM is the loosening of the disjoint sstable constraint. Because there is no longer a requirement that all sstables must be disjoint, fewer sstables are rewritten. Rewriting data may still be necessary when moving sstables from level *n* to level *n+1* due to the disjoint key ranges imposed by the guard structure. An sstable at level *n* that overlaps the key ranges of two different guards at level *n+1* will need to be fragmented into two separate sstables so that there is no longer an overlap across guards. Nevertheless, this rewriting process does not involve merging with other sstables already at the lower level. FLSM's lower write amplification when compared to LSM is due to the fact that any key will only be written once to any level, as moving an sstable from level *n* does not require any rewriting of data already in level *n+1*.

Read Performance Due to the disjoint key range invariant imposed by guards, the FLSM structure maintains strong read performance. Using LSM, reads require up to 2 binary searches at each level. Using FLSM, reads at level *n* require up to $1 + m$ binary searches, where m is the maximum number of sstables in a guard at that level. One binary search will be required to isolate the guard at that level that could contain a given key. Once the guard has been isolated, the database would proceed to perform a binary search on each sstable associated with the guard. The maximum number of additional binary searches then is limited to the maximum number of sstables associated with one guard. By probabilistically selecting guards and triggering compaction based on guard size, read performance can be further improved. More discussion on guard selection can be found in §VII, and more discussion on triggering compaction can be found in §VIII.

An illustration of the compaction process with both structural changes is provided in Figure 2. The sstables before compaction are the same as those in Figure 1 for ease of comparison. Of particular note is the fact that, unlike in Figure 1, only the sstable that is changing levels is shaded, indicating that only this sstable needs to be rewritten.

IV. PEBBLESDB

The first implementation of FLSM was with PebblesDB, which was built on top of HyperLevelDB. HyperLevelDB is a more mature and much smaller codebase compared to RocksDB, so it was the perfect candidate to implement and test FLSM for the first time. By understanding their design and results, we used PebblesDB as a model for beginning our work on RocksDB. In this section, we discuss their relevant findings and how they influenced our work.

A. Write Amplification

For 500M keys, PebblesDB reduced write amplification by 2.5x compared to RocksDB and HyperLevelDB and 1.6x compared to LevelDB. The successful outcome of FLSM implemented on top of HyperLevelDB led us to rely heavily on the high-level design of PebblesDB such as guard selection (Section §VII) and guard level compaction (Section §VIII).

B. Impact of Empty Guards

One potential concern with the effectiveness of guards is the impact of empty guards on performance. Empty guards can occur based on the range of keys inserted and the number of key deletions there

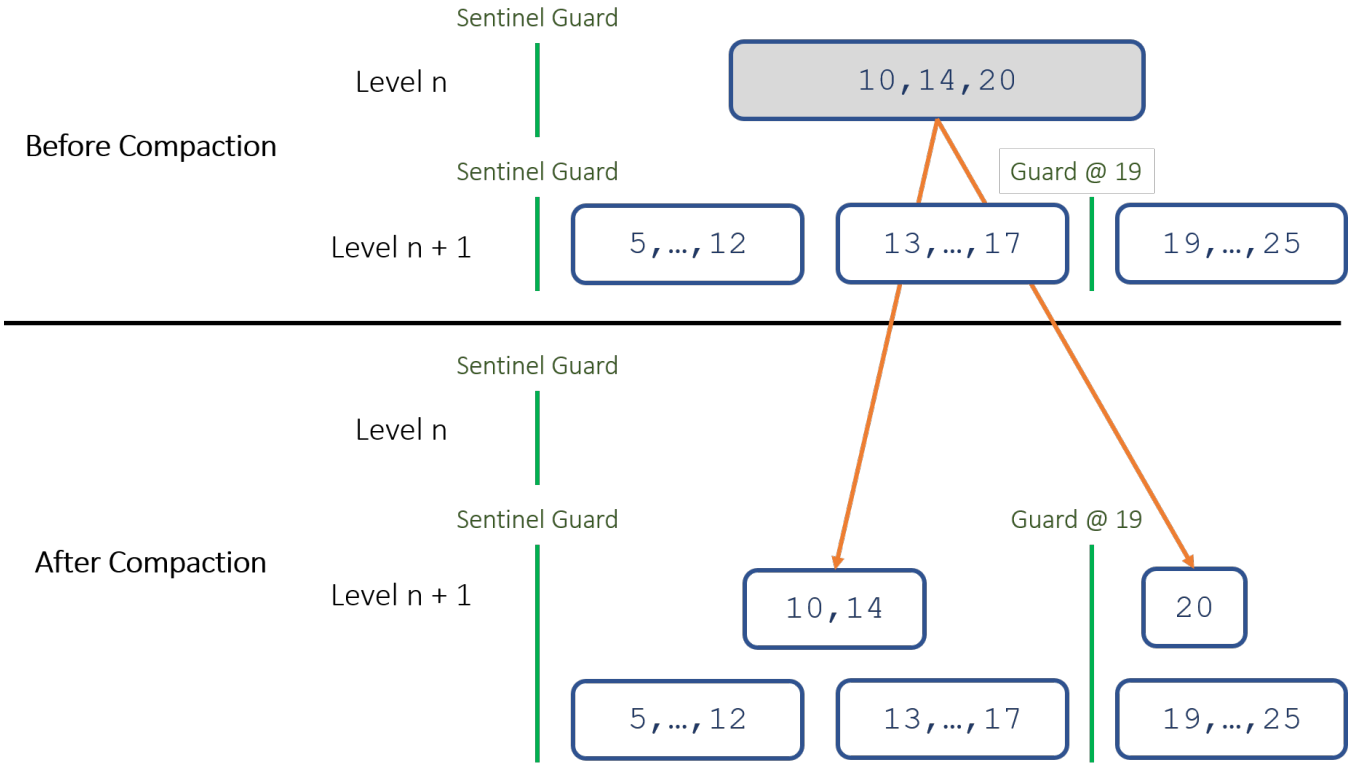


Fig. 2: Fractured Log-Structured Merge Tree Compaction. [Grey boxes denote the sstables that must be rewritten to facilitate this process.]

are. If empty guards proved to significantly reduce `Get` and `Range Query` performance, we would have placed a higher priority on implementing and testing the deletion of guards. However, work from PebblesDB concluded that read throughput did not reduce even if there were more empty guards. They ran an experiment that repeatedly inserted 20M keys, performed 10M reads, then deleted all the keys. That pattern was repeated 20 times, and that resulted in 9000 empty guards at the beginning of the last iteration. Based on this experiment, GravelDB also did not handle empty guards yet because it would require more work on compaction.

V. DESIGN

A. *RocksDB Architecture*

RocksDB manages state by organizing changes to the database in terms of Versions. A Version consists of all the sstable files that currently exist in the LSM tree. A new Version is created whenever there is a compaction or memtable flush (the movement of the contents in a memtable to storage) and all new `Get` calls and iteration will read from it. In addition to an update getting added to a memtable, the update will be stored in a Write Ahead Log on disk for recovery purposes. In the case of a failure, all the data in memory can be rebuilt based on these logs.

B. *GravelDB Architecture*

We take advantage of the RocksDB structure to efficiently implement the architecture needed for FLSM. Guards are stored in a Set and organized in terms of committed and uncommitted guards. When a key-value pair is inserted, it gets written into the memtable and the Write Ahead Log (illustrated in Figure 3) in addition to potentially getting selected as a guard. The path of a key inserted into GravelDB is illustrated in Figure 4. If it is selected as a guard, it will get added into a set of New Guards. When

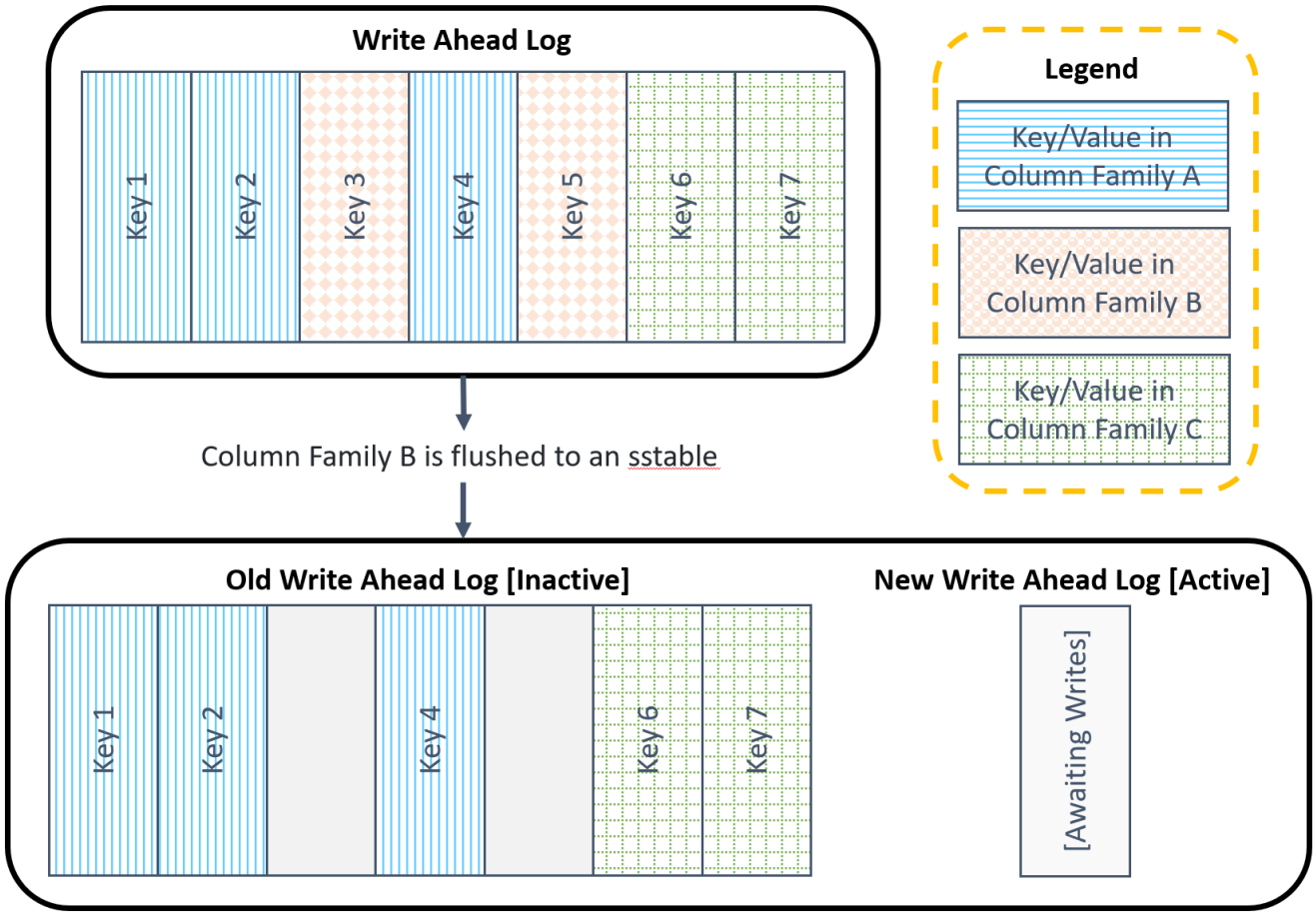


Fig. 3: Write Ahead Log Architecture in RocksDB

compaction gets triggered, new output sstables are generated and will then the used to populate all guards, new and complete, by level. The result of the compaction will then be committed along with all of the uncommitted guards.

VI. COMPARING HYPERLEVELDB AND ROCKSDB

In this section, we point out key features of RocksDB that are not present in HyperLevelDB. We had to take into account these additional features and work around them in order to properly implement FLSM.

A. Column Families

The primary differentiator between Google's LevelDB and Facebook's RocksDB is the introduction of column families. Column families allow users to set up subgroups of key-value pairs among the data they insert into the same database, thereby logically partitioning the database. In RocksDB, each key-value pair is associated with exactly one column family. If the user has not specified a particular column family, then the data is automatically placed into the `default` one.

Column families are implemented almost like separate databases. Every column family has their own Version and VersionEdits. Versions were mentioned previously in §V. Therefore, when implementing FLSM, we made sure to implement guards at the Version level. In that way, every column family will have their own setup of guards.

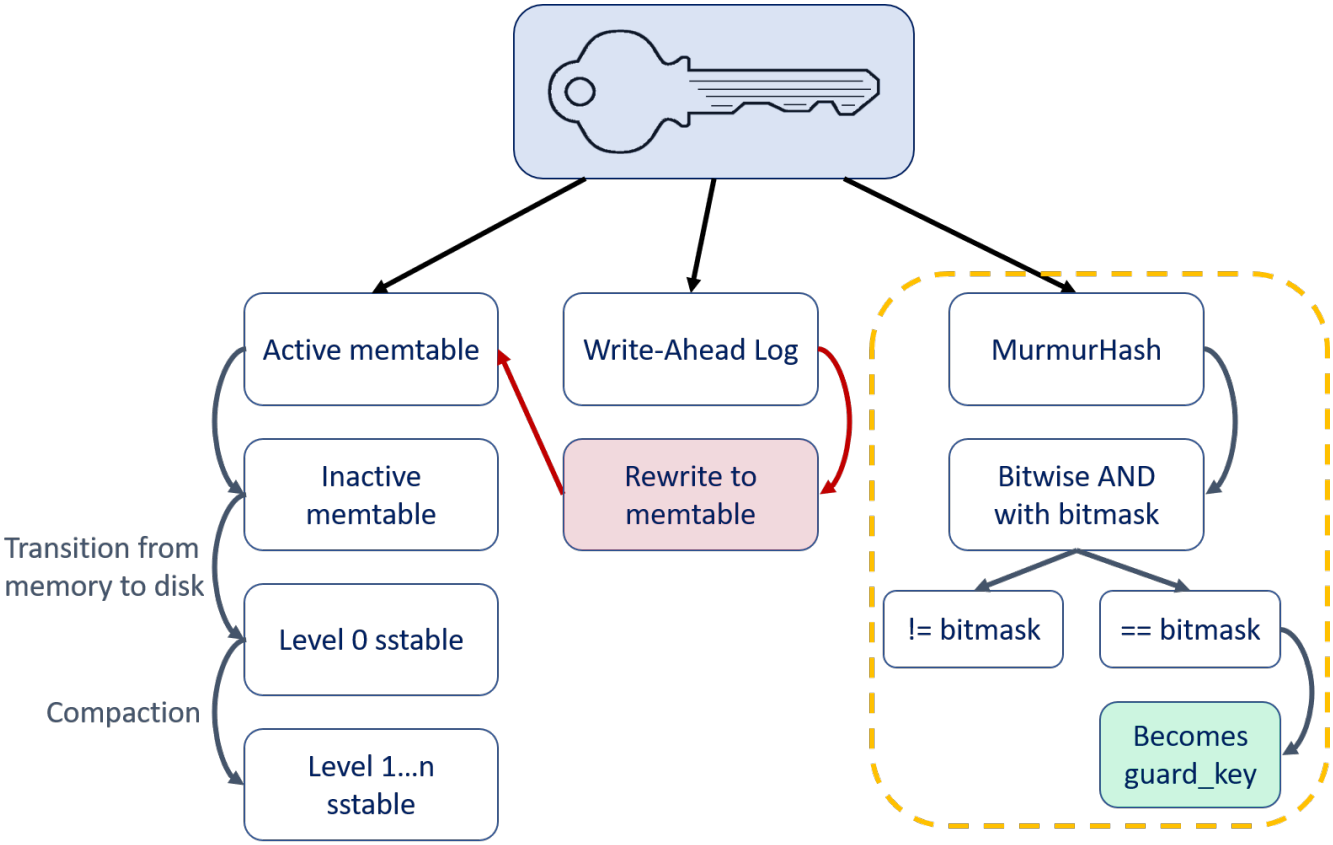


Fig. 4: Life Cycle of an Inserted Key. [The dashed box highlights the change that GravelDB introduces on top of the RocksDB architecture.]

B. *Multiple styles of compaction*

RocksDB introduces two extra styles of compaction, Universal and FIFO.

1) Universal: All sstables in universal compaction are sorted in order of time and called ”sorted runs” instead of ”levels”. The compaction process will select two or more sorted runs that are chronologically adjacent to each other, merge them together, then replace the original sorted runs with the output. This style of compaction is suited for write-heavy workloads because data can be ingested in higher rates at the cost of the possibility of needing to do a full compaction over many or all sstables, which is expensive.
2) FIFO: As the name implies, this compaction style will allow sstable files to be added until the size of all the data in the database exceeds a configured size. After that threshold is reached, the oldest sstable will be deleted. The concept also allows for a time-to-live (TTL) parameter to be set and will delete the sstables that exceed the TTL.

The primary goal of our work is to improve Leveled Compaction, so we have not tuned or modified these styles of compaction. Leveled Compaction was described in depth in terms of LSM and FLSM in §II-B and §III respectively. The design of our changes allows these styles of compaction to still operate as expected because guards is a layer of abstraction that can be ignored. Still, it was challenging to work around these styles because they added to the overall complexity of the codebase.

C. *Breadth of Change*

Although changing a database from an LSM data structure to an FLSM data structure was previously implemented (from HyperLevelDB to PebblesDB), the difference in scale between HyperLevelDB and

RocksDB makes this work a significant undertaking. The HyperLevelDB code base is approximately 38,000 lines of code. On the other hand, the release of RocksDB that we built our changes on had 387,000 lines of code, with 6,662 commits and 367 contributors. The 10x change in code base size, along with the added complexity of the new features that RocksDB introduced, made this project a significant undertaking.

VII. IMPLEMENTATION: GUARDS

A. GuardMetaData

We implemented guards as a structure containing relevant metadata. This was modeled off of both the existing FileMetaData structure that RocksDB uses for storing information about sstables as well as the GuardMetaData structure that PebblesDB used when implementing FLSM into LevelDB. "Sstable" and "file" will be used to refer to the same structure.

1) Guard key
2) Smallest and largest keys in range
3) Vector of pointers to FileMetaData objects, where each FileMetaData includes information about an sstable (for example: FileDescriptor, smallest and largest keys, smallest and largest sequence numbers)
4) Level

For debugging and evaluation purposes, the GuardMetaData structure also has functions for calculating the minimum, median, and maximum file sizes.

B. Guard Selection

Balancing decreased write amplification with strong read performance while using the FLSM data structure, as discussed in §III, relies on careful guard selection. Firstly, the distribution of guards across the set of all keys is an important consideration of guard selection. Skewed guards could lead to some guards having significantly more sstables than others and increase variation of read performance. Read performance in more bloated guards will be worse due to the larger number of binary searches that need to be performed (compared to a key contained in a leaner guard).

We chose to use the method of guard selection that PebblesDB used, which probabilistically selected guards from inserted keys. It would be unreasonable for the database to provide its own guards without any knowledge of how keys in the client's data are distributed. Instead, we draw randomly from the set of all known keys. When a new key-value pair is added to the database, the key value is hashed. We then use the bit-wise AND operation with a pre-determined bitmask, and if the result matches the bitmask, the key is chosen as a guard. The bitmask changes at each level; at the top-most level, the bitmask would be `top_level_bits` ones. With each increase in the level number, the length of the bitmask decreases by `bit_decrement`. This increases the probability that a key is selected as a guard at as the level number increases so that there ends up being more guards at the lower levels. Every key that is a guard at level n will also be a guard at all levels greater than n.

To show the probabilistic nature of our guard selection method, we demonstrate the distribution of the keys selected as guards compared to the distribution of keys inserted using a 10 million key insertion workload in Figure 5. The guard key distribution uses the guards at the lowest level in the database. The 10 million keys inserted were based on choosing values from the Zipfian distribution. From the box plots, we see that the guards selected also demonstrate the long right-tail relationship characteristic of the Zipfian distribution.

C. Sentinels

Sentinel guards are used to encompass the guard range from the minimum key inserted to the smallest guard key at a particular level. In order to make no claims about the values client write

	Minimum	First Quartile	Median	Third Quartile	Maximum
Guard Keys	0	523503	1537911	4555174	9856991
Keys Inserted	1	36	2370	153762	9999980

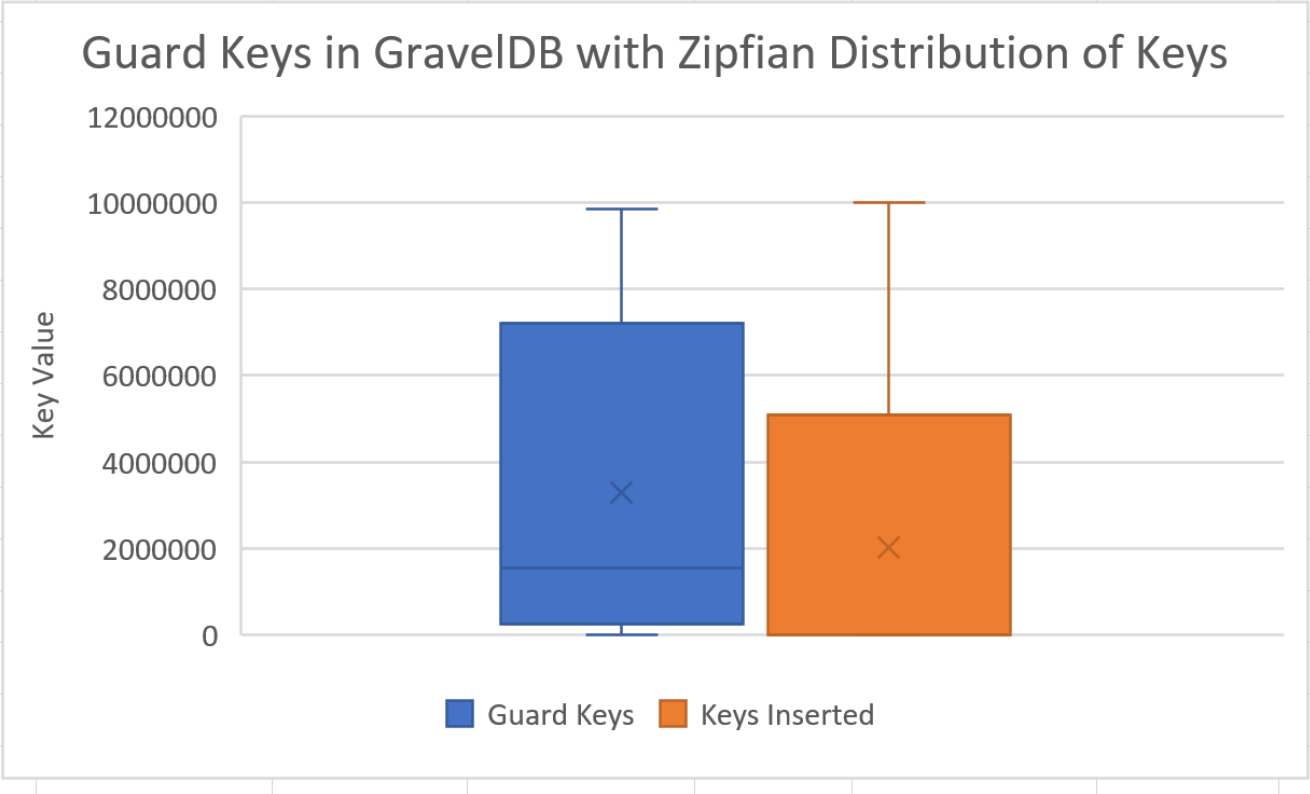


Fig. 5: Box plots for Guard Key values versus Keys Inserted using a skewed distribution for 10 million key insertions.

for his/her keys, we represent sentinel guards using GuardMetaData objects with zero-sized keys. The sentinels differ from typical guards because they do not require the database to have seen any key-value insertions by the client for their creation. They are stored in the VersionStorageInfo data structure as a mapping of `<level, GuardMetaData>` pairs. This vector is populated in the constructor of the VersionStorageInfo structure.

D. GuardSet

The GuardSet data structure is used primarily for iterating over a given set of guards. Closely tied with this structure is also the GuardSetComparator, which is primarily a wrapper around the pre-existing comparator for InternalKey objects. The GuardSet structure is especially crucial for facilitating `AllGuardsAtLevel`, which returns a GuardSet object that allows the caller to easily iterate over all guards (including sentinel) at a level in ascending key order.

VIII. IMPLEMENTATION: COMPACTION

As previously mentioned, compaction is the process by which sstables at level *n* are moves to level *n+1*. This increases read efficiency because older data (which is less likely to be accessed) will be farther from consideration and will not negatively impact reads at level *n*.

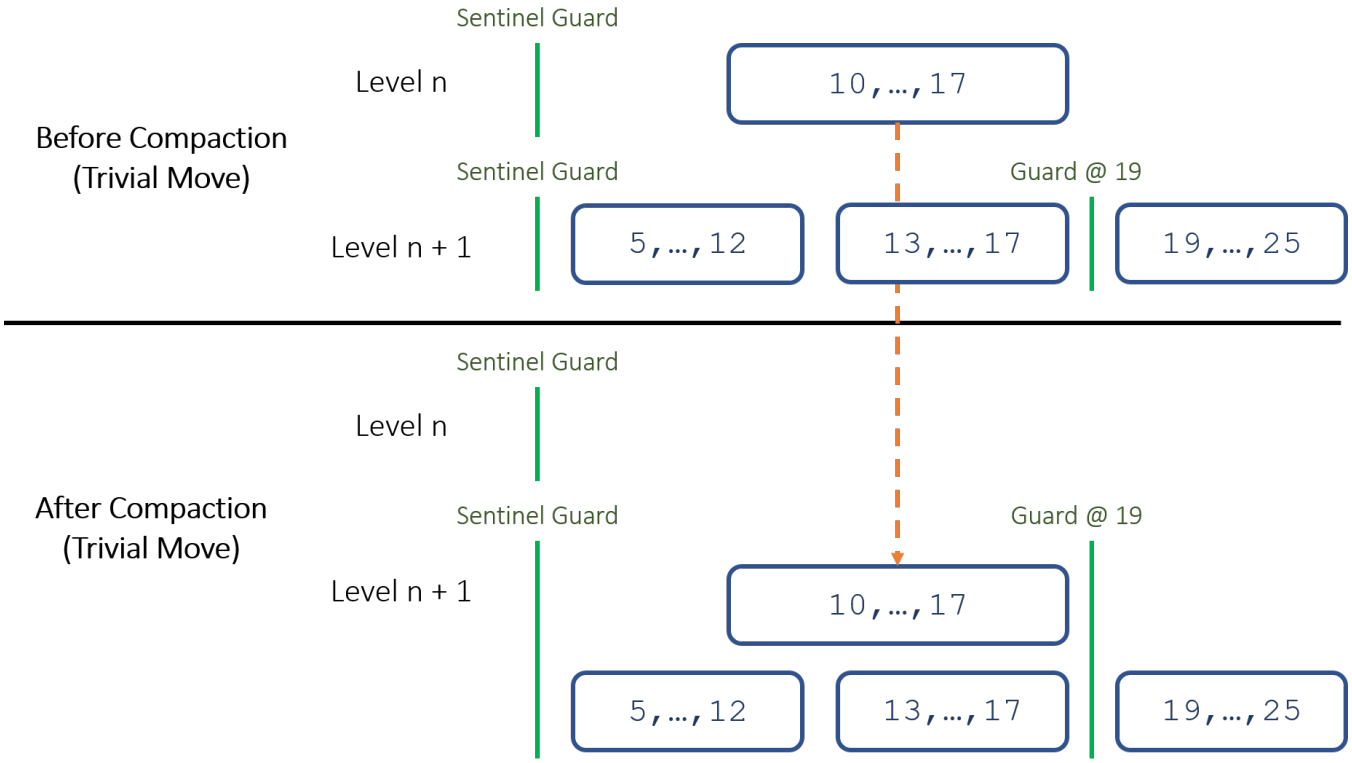


Fig. 6: Fractured Log-Structured Merge Tree Trivial Compaction.

In this section, we first discuss the changes made in order for RocksDB compaction to support the introduction of the guard structures. We then discuss how we altered compaction triggering to exploit information at the guard granularity, as RocksDB previously relied on thresholds for the volume of data contained at each level to trigger when compactions occur.

A. FLSM in the Context of Compaction

Leveled compaction is the style of compaction that we hope the introduction of FLSM will improve. When a compaction is triggered, inputs are chosen for the Compaction job. Depending on the size of the inputs and how much they overlap, multi-threaded compaction on a level can occur. For each subcompaction, we process the key value pairs in the specified range and produce output sstables. All the output sstables are then written into the new Version and guards will be populated when the new Version is applied.

Trivial Moves An example of a trivial move in the FLSM structure is shown in Figure 6. The dotted orange line and the lack of shading in any of the boxes (unlike Figures 1 and 2) indicate that no rewriting is necessary. The only necessary change is for the pointer to the sstable being compacted to be removed from the sentinel guard at level n and added to the sentinel guard at level $n+1$.

B. Compactions at Guard Granularity

Motivation After ensuring that the FLSM structure was implemented correctly to accommodate compaction, we explored different ways of modifying the compaction algorithm to be at the guard granularity as opposed to the file granularity. Without any change in this scope, compactions would operate exactly the same as they had been in RocksDB because the algorithm would ignore the guard abstraction that we presented. The initial evaluation in Section §IX-C indicates that FLSM does not demonstrate its full benefits without any mechanisms to take advantage of it.

Considering Compaction Input/Output There were two designs we were considering in terms of how to handle the input/output changes that taking into account guard granularity would entail. Conceptually speaking, we thought that the most all-encompassing way to change compaction such that it took advantage of the guard structure would be for the compaction jobs to take in and produce GuardMetaData objects as output. However, this method would have been a more circuitous route that did not take advantage of the current compaction triggering setup in the RocksDB codebase.

Changing Compaction Score RocksDB already used a scoring system to choose the input for compaction jobs, and altering the scoring methodology would be a sufficient method to change the compaction process to take advantage of knowledge at the guard granularity. RocksDB originally computes the compaction score of a level that is not level 0 by comparing a level's data size to the max data size recommended for a level. To compute the compaction score of level 0, they base the fraction on number of files instead of size of data. The user can specify level0_file_num_compaction_trigger and the score will be in terms of that. Level 0 considers number of files instead of size of level because too many level 0 compactions aren't desirable, especially with larger write-buffer sizes. In other words:

$$\text{Level 0 score} = \frac{\text{number of files}}{\text{number of files to trigger}}$$

$$\text{Level 1 .. n score} = \frac{\text{non-compacting bytes in level}}{\text{threshold bytes for level}}$$

We wanted to tune the compaction scoring algorithm to take advantage of the guard structure to further enforce similarity in guard sizes (due to the previously discussed disadvantages associated with bloated guards). Borrowing the scheme from PebblesDB, we set an expected size for each level's guards and scored each guard by the fraction of its size and the expected size. We then take those guard compaction scores into account when computing the level's compaction score. The level's compaction score is the max of all the guard compaction scores in the level. This would then prioritize guards that are extremely full. We chose to leave level 0's compaction score the same.

$$\text{Level 0 score} = \frac{\text{number of files}}{\text{number of files to trigger}}$$

$$\text{Level 1 .. n guard score} = \frac{\text{total guard size}}{\text{threshold guard size}}$$

$$\text{Level 1 .. n score} = \max \text{guard score}$$

IX. EVALUATION

We ran our experiments on a Dell PowerEdge R230 with 4 * Intel(R) Xeon(R) CPU E3-1220 v5 @ 3.00GHz. It has 16GB RAM and runs Ubuntu 16.04 LTS with the Linux 4.4 kernel. It has one single SSD (1.2 TB) running an Ext4 filesystem.

A. Microbenchmark Setup

Given the extensive testing frameworks that RocksDB already had, we chose to rely on the provided db_bench suite to benchmark the performance of our implementation. We initially ran three different workloads - `fillrandom`, `readrandom`, and `seekrandom`, but eventually chose to focus on `fillrandom` for the time being, as that is the workload that should yield reflect the largest difference based on our changes for improved write amplification performance. This workload inserts keys in a randomized order so as to ensure the database does not depend on any specific ordering of key insertions. After these insertions, we also collected the results of the `stats` and `sstables` workloads, which yield the performance metrics and sstable tree structure, respectively.

In configuring the flags for the db_bench program, we were careful to use arguments that would yield results with a common basis of comparison to RocksDB.

1) `write-buffer` was set to 64MB, a standard value for RocksDB usage
2) `slowdown_writes_trigger` was set to 20, a standard value for RocksDB usage
3) `stop_writes_trigger` was set to 24, a standard value for RocksDB usage
4) `open_files` and `value_size` were set to 1000 and 1024, respectively, which were the values used for db_bench measures for HyperLevelDB

The `stats` workload provided information about volume of read/write IO and other performance-related metrics, such as database uptime. In our evaluation, we focused on the total amount of data written by the database given the same number of keys for GravelDB versus RocksDB. We did also examine the uptime results to ensure that GravelDB performance did not prohibitively trail that of RocksDB. However, our primary consideration was still on volume of data written, particularly as the `fillrandom` workload increased (number of insertions increased).

While the default `sstables` workload revealed the general LSM structure, we had to adapt the output to reflect the structural changes in the database that came with implementing FLSM. We changed the `sstables` output to print the following for every level (in addition to the information already provided, such as the associated version number of the sstables printed):

1) Number of guards in this level
2) Information about each guard (discussed below)
3) Median number of files in a guard on this level
4) Total data contained in this level (computed as the sum of all files associated with all guards in this level)

Every guard outputs the following information:

1) Guard key
2) Smallest key in guard
3) Largest key in guard
4) Number of files in this guard
5) Minimum, median, and maximum file size of the files associated with this guard
6) Information about each file - this was part of the original output and includes information about the file such as file size and smallest and largest key values in the file
7) Total data contained in this guard (computed as the sum of all files in this guard)

These additions to the `sstables` output were crucial for identifying potential changes that would better the performance of our initial implementation.

We used varying sizes of workloads in our experiments and attempted to adjust the guard selection variables accordingly. We generally veered towards having more guards rather than fewer guards than necessary, as having insufficient guards leads to skew among the guards and negatively affects performance.

	Write(GB) - Sum	Wnew(GB) - Sum	W-Amp - Sum	Comp(sec) - Sum	Comp(cnt) - Sum	Uptime(secs)
GravelDB	51.1	12.12	10.42	199.6	1447.2	181.7
RocksDB	50.57	12.17	10.3	181.7	1447.3	165.04
Rocks / Gravel	98.96%	100.41%	98.85%	91.03%	100.01%	90.83%

Fig. 7: GravelDB performance prior to compaction score changes vs. PebblesDB performance; 5M insertions

B. MyRocks Setup

RocksDB is an embeddable database that can be used as the storage engine within larger database management systems. One such use of RocksDB is MyRocks, an open-sourced software that allows the use of MySQL features with RocksDB. To see if the changes introduced through FLSM could be apparent in MyRocks, we set up a TPC-C benchmark to compare RocksDB and GravelDB. The benchmark model executes New Order transactions against the database and records the average response time of operations as well as throughput.

C. Results

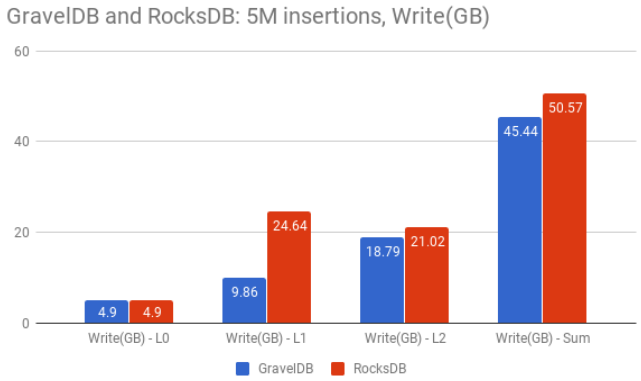
In our evaluation, we focused on the following metrics that the `db_bench` benchmarking program provides in its `stats` workload. For all metrics except `Comp(sec)` and `Uptime`, we examined the values for individual levels as well as the sum across all levels.

1) `Write(GB)`: the total amount of data during compactions from level *n* to level *n+1*
2) `Wnew(GB)`: the amount of new data written during compactions from level *n* to level *n+1*
3) `W-Amp`: write amplification of compactions from level *n* to level *n+1*, calculated as (total data written to level *n+1*) / (total data read from level *n*)
4) `Comp(sec)`: the amount of time (in seconds) spent performing compactions
5) `Comp(cnt)`: the number of compactions performed
6) `Uptime`: the number of seconds the database was up to perform the requested number of insertions

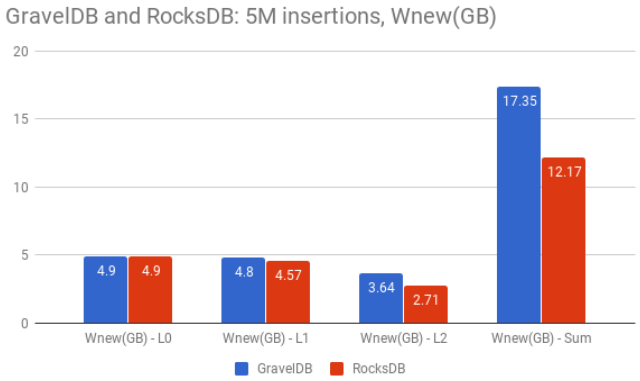
Initial Evaluation of Section §VIII-A This initial implementation was focused on making sure that the original compaction process still worked following the introduction of guards. However, there were no changes at this point to adjust the compaction process to take advantage of the guard structure; rather, the guard structures accommodated the original setup. The results of initial testing (summarized in Figure 7, however, suggested that we would need to change the implementation of compaction to be more guard-centric in order to take full advantage of the write amplification benefits that FLSM provides. All of the primary metrics that we focused on indicated that the initial implementation performed only equivalently (if not slightly worse) than the original RocksDB implementation.

5 Million Key Insertions We again chose to ran experiments using 5 million key insertions in order to observe whether our changes to the compaction scoring process impacted the performance of GravelDB in comparison to RocksDB. Since the volume of inserts was the same, we continued to use 22 and 2 for the `top_level_bits` and `bit_decrement` values, respectively. Data presented here for each of the databases represents averaged data from 10 trials for each database.

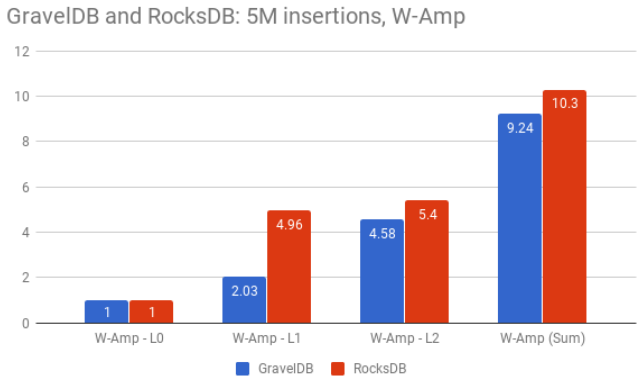
Across 10 runs averaged together, we see that GravelDB writes less data overall but writes more new bytes. On average, RocksDB writes 111% of the data the GravelDB writes (50.57 GB with a standard deviation of 1.19 for RocksDB, and 45.44 GB with a standard deviation of .89 for GravelDB), although it only writes 70% of the amount of new data the GravelDB writes. However, we also observed that GravelDB would have data as deep as level 5, whereas RocksDB never had data past level 2 for the 5 million inserts. The graphs provided only extend for as many levels as both databases had data in order to facilitate more meaningful comparisons Therefore, we attribute much of this new data writing to the fact that GravelDB had to write the data into deeper levels. As FLSM guarantees that data is



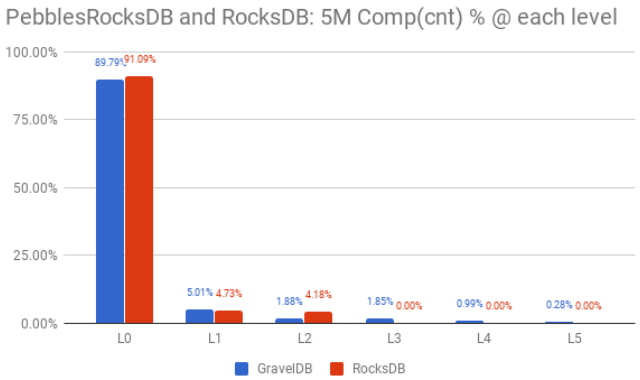
(a) Amount of data (GB) written during compactions



(b) Amount of new data (GB) written during compactions



(c) Write Amplification



(d) Number of compactions at each level as a percent of total compactions

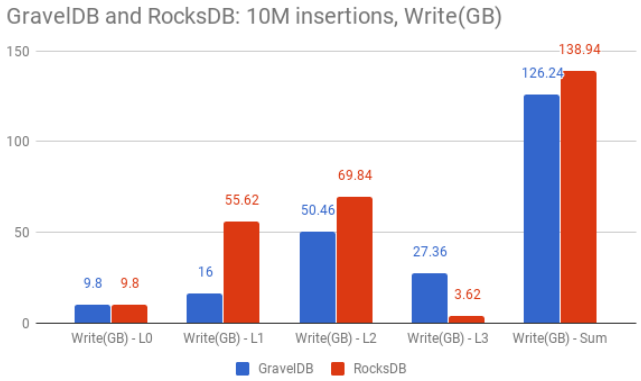
Fig. 8: GravelDB vs. RocksDB at L0, L1, L2, and sum across 5M insertions. Only levels in which both databases had data were shown for figures (a), (b), and (c)

only written once to any level, the fact that these new writes occurred should not be of concern. This is because the absolute upper bound on these new writes is the number of levels multiplied by the number of keys inserted for FLSM, whereas there is not an intuitive upper bound on this value for LSM.

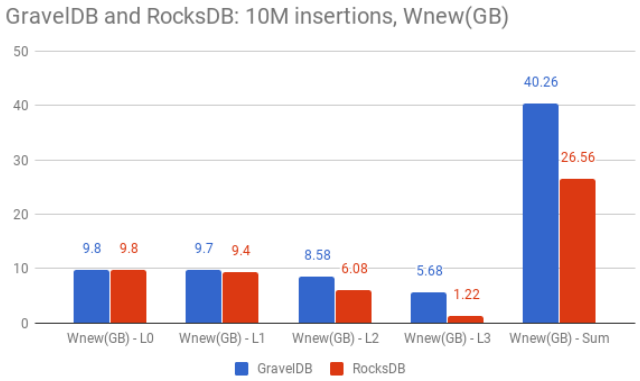
Beyond the amount of data written into each of the databases, we also observe that RocksDB has higher write amplification at the 5 million key insertion level. Finally, the last graph presented in that figure indicates that a strong majority of compactions for RocksDB are occurring from level 0 to level 1. This number is slightly depressed for GravelDB, which pushed data deeper into its structure than Rocks does. A visual representation of GravelDB performance versus RocksDB performance can be found in Figure 8

10 Million Key Insertions To accommodate the larger number of inserts, we changed the `top_level_bits` and textttbit_decrement values to 23 and 2, respectively. Besides that, the structure of the experiment is roughly equivalent. For this part of the experiment, we chose to use 5 trials from each database instead. The results with these 10 million key insertion workloads reflect the same patterns observed for the 5 million key insertion workloads. In terms of total writes to the database, RocksDB writes an average of 138.94 GB with a standard deviation of 1.94. GravelDB, on the other hand, writes an average of 126.24 GB with a standard deviation of 1.84. A visual representation of GravelDB performance versus RocksDB performance can be found in Figure 9.

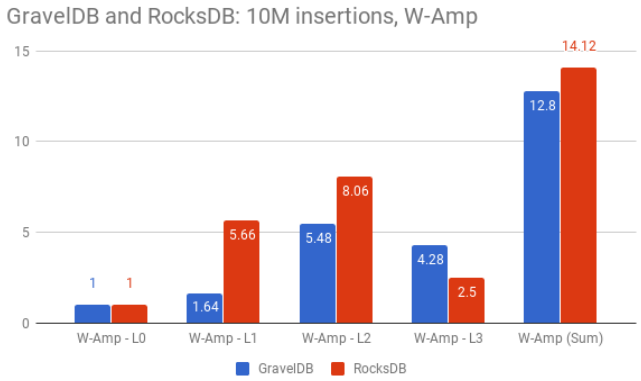
A summary of the primary metrics across the two databases for both 5 million and 10 million key insertions is available in Figure 10. For brevity, the summary only includes averaged summation data,



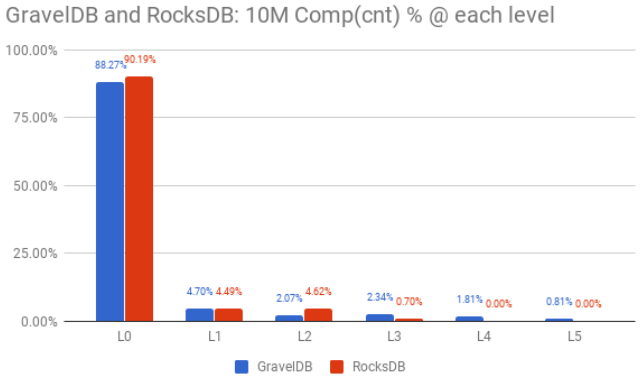
(a) Amount of data (GB) written during compactions



(b) Amount of new data (GB) written during compactions



(c) Write Amplification



(d) Number of compactions at each level as a percent of total compactions

Fig. 9: GravelDB vs. RocksDB at L0, L1, L2, and sum across 10M insertions. Only levels in which both databases had data were shown for figures (a), (b), and (c)

rather than data broken down by level. More detailed information about specific trial results and the standard deviations of these summed data are available in the Appendix, Section §XI.

Read Performance As previously discussed, we anticipated that the introduction of the FLSM data structure into RocksDB would decrease read performance of the database. In order to evaluate how much this performance would be impacted, we performed 10 million key insertion experiments with varying expected numbers of guards and evaluated the read performance and write volume at each number. The results are presented in Figure 11. Each pair of bars is associated with a different `top_level_bits` value on the x-axis, and the higher this variable value, the fewer expected number of guards at each level. The y-axis represents the GravelDB Write(GB) and readrandom (micros/op) performances as percentages of the corresponding variables in RocksDB. Write(GB) measures the amount of total data written during compaction, so lower values here indicate lower write amplification. The other metric, readrandom (micros/op), measures the amount of time the database takes to perform a read of a random key. Therefore, lower values here are also desirable. Given that the y-axis presents values as percentages of RocksDB performance, values less than 100% are desirable for all metrics, as they indicate that GravelDB has either lower write amplification or faster writes.

As expected, we see that there is no value of `top_level_bits` tested that yields faster reads in GravelDB than in RocksDB. However, we see that the read performance is best when the `top_level_bits` variable is set to 23. We also observe that this is the value that presents the largest decrease in write amplification for 10 million key insertions. Therefore, we come to the conclusion that although read

	Write(GB) - L0	Write(GB) - L1	Write(GB) - L2	Write(GB) - Sum
GravelDB	4.9	9.86	18.79	45.44
RocksDB	4.9	24.64	21.02	50.57
Rocks / Gravel	100.00%	249.90%	111.87%	111.29%
	Wnew(GB) - L0	Wnew(GB) - L1	Wnew(GB) - L2	Wnew(GB) - Sum
GravelDB	4.9	4.8	3.64	17.35
RocksDB	4.9	4.57	2.71	12.17
Rocks / Gravel	100.00%	95.21%	74.45%	70.14%
	W-Amp - L0	W-Amp - L1	W-Amp - L2	W-Amp (Sum)
GravelDB	1	2.03	4.58	9.24
RocksDB	1	4.96	5.4	10.3
Rocks / Gravel	100.00%	244.33%	117.90%	111.47%

(a) 5 million key insertions

	Write(GB) - L0	Write(GB) - L1	Write(GB) - L2	Write(GB) - L3	Write(GB) - Sum
GravelDB	9.8	16	50.46	27.36	126.24
RocksDB	9.8	55.62	69.84	3.62	138.94
Rocks / Gravel	100.00%	347.63%	138.41%	13.23%	110.06%
	Wnew(GB) - L0	Wnew(GB) - L1	Wnew(GB) - L2	Wnew(GB) - L3	Wnew(GB) - Sum
GravelDB	9.8	9.7	8.58	5.68	40.26
RocksDB	9.8	9.4	6.08	1.22	26.56
Rocks / Gravel	100.00%	96.91%	70.86%	21.48%	65.97%
	W-Amp - L0	W-Amp - L1	W-Amp - L2	W-Amp - L3	W-Amp (Sum)
GravelDB	1	1.64	5.48	4.28	12.8
RocksDB	1	5.66	8.06	2.5	14.12
Rocks / Gravel	100.00%	345.12%	147.08%	58.41%	110.31%

(b) 10 million key insertions

Fig. 10: Summary metrics for GravelDB performance vs. RocksDB performance

performance is likely to suffer in GravelDB compared to RocksDB, the negative impact on performance can be lessened with a proper choice in frequency of guard selection. This further emphasizes the importance of carefully tuning the guard selection variables in alignment with the volume of expected key insertions.

TPC-C Benchmark The results of MyRocks (Figures 12 and 13) did not definitively show GravelDB outperforming RocksDB or vice versa. Averages and standard deviations of the results are as follows:

	Throughput	95%	99%
GravelDB avg	1448.40	91.80	142.83
RocksDB avg	1465.43	94.28	150.57
GravelDB std dev	105.65	7.05	68.94
RocksDB std dev	122.62	14.44	123.09

Throughput indicates the number of transactions completed within a set time frame, so the higher RocksDB average indicates stronger performance. The 95% and 99% columns indicate the response times of the databases at the 95th and 99th percentiles, respectively, and GravelDB outperforms RocksDB in these metrics by having shorter response times. However, none of these differences are particularly large, hence why it is difficult to conclude whether one database performed better than the other.

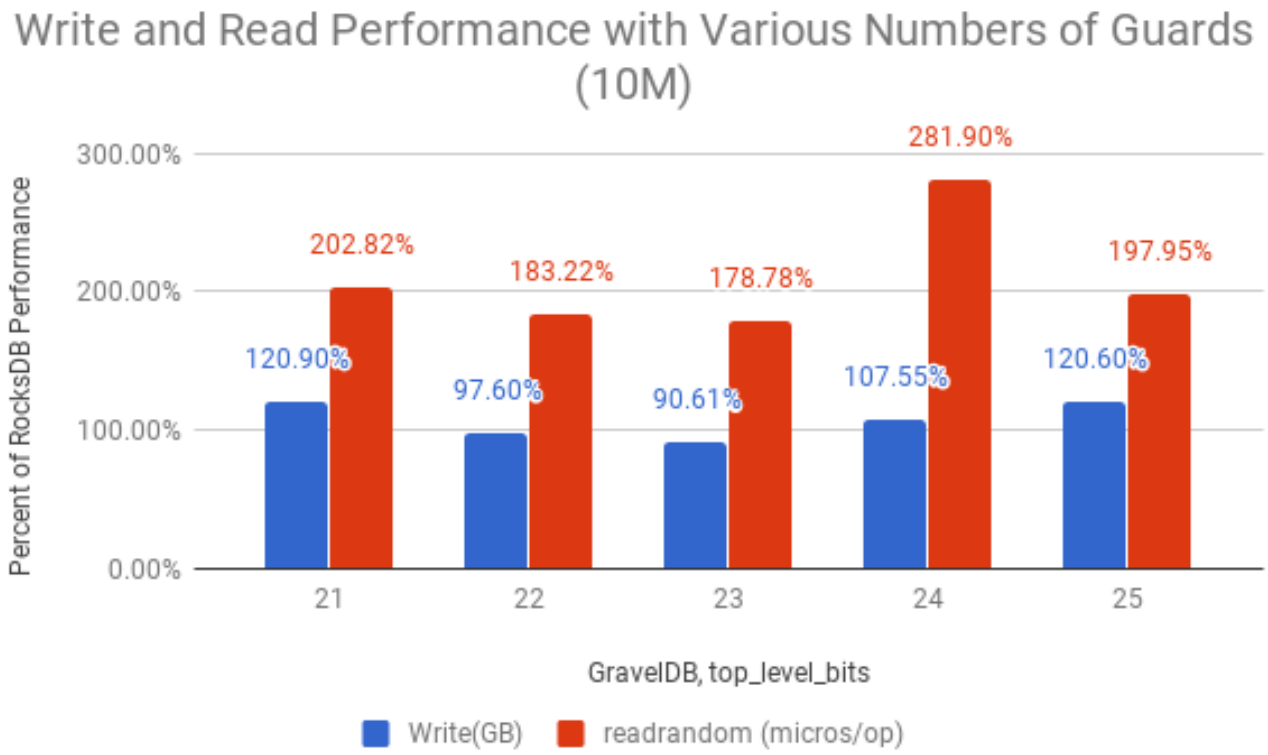


Fig. 11: GravelDB read performance and write amplification with changes in the guard selection variable value.

X. FUTURE WORK

A. *Customizing MyRocks*

Given the ambiguous results from our evaluation of GravelDB on the TPC-C benchmark, more work needs to be done in tuning either MyRocks or GravelDB in order for the response times and throughput to improve. Factors to consider may include:

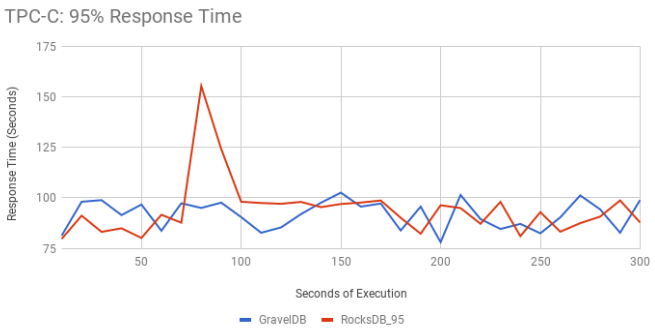
1) Experimenting with the percentage of guards
2) Optimizing reads and writes with FLSM

B. *Further Minimizing Rewrites*

The benefit of FLSM in comparison to traditional log-structured merge trees is the diminished need for rewriting data that comes from the potential overlap within guards. For example, a file f at level n whose key range falls completely within the bounds of a guard g at level $n+1$ can be transferred during compaction simply by adding a pointer to f in g's list of FileMetaData objects.

Even under the FLSM scheme, however, rewriting data cannot be eliminated. If f's key range included values falling into two different guards in level $n+1$, the system would still be forced to rewrite all of the data in f into two files, splitting the original file's contents based on the delineating guard value. One further optimization that could be performed here is to mix the two approaches - changing the ownership of a pointer and rewriting data. Note that under the current setup, keys are guaranteed to be in sequential order within a file. As such, it is guaranteed that a file f that must be written into two files f_1 and f_2 (where f_1 encompasses all keys less than the delineating guard key and f_2 encompasses all keys greater than or equal to the delineating guard key) changes its write target only one time, assuming

Fig. 12: TPC-C: Transactions executed across time (measure of throughput)



(a) 95% response times

(b) 99% response times

Fig. 13: TPC-C: 95% and 99% Response times

the system writes in order of the keys in the original file. Therefore, write amplification could be further decreased by deleting the keys in f associated with the key range for f_2 and rewriting those in a new file. The original f would then contain exactly the keys for f_1 and could be moved to level $n+1$ by moving the pointer, without further rewrites.

This process can easily be extended for a file at level n whose key ranges overlap more than two keys as well. Regardless, a binary search would first be performed to identify the point at which to "cut" the file. The contents after this delineation would then be deleted and rewritten. Given the probabilistic manner in which guards are chosen and the default scheme for increasing the probability of a key value being a guard as the level number increases, a guard at level n should, on average, correspond to $2^{bit\_decrement}$ guards at level $n+1$. Because it is fairly unlikely for a single file to span the entire key range of a guard, it is likely that this is effectively an upper bound when considering this strategy.

C. File Metadata Manipulation

The intuition behind further minimizing rewrites can be taken a step further into purely manipulating data by changing pointers to desired storage areas. This would involve changing the contents of a file's metadata block to indicate where to read from so that no rewrites have to be performed to facilitate the FLSM structure.

The primary concern associated with this direction of research is that range queries can become significantly more expensive, as quantitatively adjacent keys would no longer be stored in physically similar locations on a particular level. Given the potential change in locality, if not storage media, reading a range of sequential keys would essentially amount to performing every read in the range individually. The number of times that such a change must occur within the range of key values associated with a range query could vary widely, making it difficult to approximate the degree to which range query performance would generally be affected. Nevertheless, workloads that are predominantly write operations and use few range query operations could benefit for the complete elimination of rewrites that comes from file metadata manipulation.

XI. CONCLUSIONS

In this study, we introduced the Fragmented Log-Structured Merge Tree (FLSM) to RocksDB, a popular key-value store owned by Facebook. This data structure serves primarily to reduce write amplification by loosening the absolute sorting of keys on each level and introducing the concept of guards, which hold collections of sstables (represented as files in the database) and obey the invariant of disjoint key ranges. After implementing the FLSM data structure, we made attempts to further optimize our resulting product, GravelDB, by altering the compaction scoring system to account for guards.

Our experimental results indicated that while a bare implementation of FLSM itself does not lower the write amplification of the database, changing compaction to better exploit the guard granularity of the FLSM data structure did produce results that decreased write amplification, just as the FLSM data structure promises. Not only is the compaction scoring change to take advantage of guard granularity sizing important, but the variables necessary to tailor the number of guards at each level is also crucial to optimal database performance.

Overall, we conducted tests of 5 million and 10 million key insertions, focusing on insertions because they represent the write functionality that we are aiming to improve. On the whole, RocksDB writes approximately 110% the amount of data that GravelDB does. We also observed that with these workloads, GravelDB tended to push data deeper into the database than RocksDB did. This allowed RocksDB to outperform GravelDB in terms of the amount of new data written during compaction, but it is important to note that the amount of new data written for compaction under the FLSM scheme is bounded by number_of_levels $\times$ number_of_keys, whereas no such guarantee can be similarly made for LSM.

Finally, we took steps to employ our implementation of GravelDB as the backend storage of MyRocks, capitalizing on the popularity of RocksDB for many applications. Although the work that we were able to complete does not yet indicate a significant benefit that accompanies the change from RocksDB to GravelDB, we believe that this would be a very approachable next step and that there are definitely ways to better configure the database to suit the needs of MyRocks.

DIVISION OF WORK

Each of our contributions to this project were interwoven, hence why there was no clear division to cohesively write two separate theses. The distinct aspects that we worked on individually are as follows:
Chris:

1) Introduced guards to RocksDB
2) Added guard granularity output to tests
3) Wrote test functions to confirm guard structure

4) Developed scripts to run experiments efficiently
5) Executed microbenchmark experiments

Carolyn:

1) Implemented trivial moves
2) Modified compaction from file granularity to guard granularity
3) Customized compaction scoring
4) Configured MyRocks TPC-C benchmark

ACKNOWLEDGMENT

We would like to thank Professor Vijay Chidambaram for advising us, Souvik Banerjee for working in a group with us last semester, Pandian Raju for answering many of our questions, and the UT SAS Lab for supporting our work.

REFERENCES

[1] Facebook. 2017. RocksDB Basics. https://github.com/facebook/rocksdb/wiki/RocksDB-Basics. (2017).
[2] Facebook. 2018. Basic Operations. https://github.com/facebook/rocksdb/wiki/Basic-Operations. (2018).
[3] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. SOSP17.
[4] Facebook. 2015. Column Families. https://github.com/facebook/rocksdb/wiki/Column-Families. (2015).
[5] Facebook. 2017. Manual Compaction. https://github.com/facebook/rocksdb/wiki/Manual-Compaction. (2017).
[6] LevelDB. 2016. LevelDB db_bench benchmark. https://github. com/google/leveldb/blob/master/db/db_bench.cc. (2016).
[7] PerconaLabs. 2017. tpcc-mysql. https://github.com/Percona-Lab/tpcc-mysql. (2018).
[8] MyRocks. 2017. Getting Started. http://myrocks.io/docs/getting-started/. (2017)
[9] Facebook. 2017. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide. (2017).

APPENDIX

This section includes more detailed information regarding the results previously discussed. The latter four tables include relevant output metrics from db_bench stats workload [10 runs of 5 million key insertions each followed by 5 runs of 10 million key insertions each]

GravelDB (prior): 5 million insertions						
Trial	Write(GB) - Sum	Wnew(GB) - Sum	W-Amp (Sum)	Comp(sec) - Sum	Comp(cnt) - Sum	Uptime(secs)
0	51.7	12.2	10.5	204	1448	185.9
1	50.9	12.1	10.4	199	1447	181.1
2	50.5	12.1	10.3	194	1445	176
3	52.4	12.1	10.6	203	1447	184.3
4	50	12.1	10.3	198	1449	181.2
Average	51.1	12.12	10.42	199.6	1447.2	181.7
Std Dev	0.9565563235	0.04472135955	0.1303840481	4.037325848	1.483239697	3.791437722

Fig. 14: GravelDB performance prior to compaction score changes

GravelDB: 5 million insertions						
Trial	Write(GB) - Sum	Wnew(GB) - Sum	W-Amp (Sum)	Comp(sec) - Sum	Comp(cnt) - Sum	Uptime(secs)
0	45.7	17.4	9.3	152	1472	145.5
1	44.2	17.3	9	164	1469	152.7
2	45.8	17.1	9.3	162	1470	154
3	45.5	17.6	9.2	168	1472	155.5
4	46.5	17.6	9.5	168	1471	158.4
5	45.5	17.3	9.3	159	1469	150.1
6	44.2	17.1	9	165	1469	153.9
7	46.9	17.2	9.5	161	1469	152.4
8	44.7	17.6	9.1	159	1470	149.5
9	45.4	17.3	9.2	179	1470	165.8
Average	45.44	17.35	9.24	163.7	1470.1	153.78
Std Dev	0.8871928263	0.1957890021	0.1776388346	7.180993432	1.197219	5.500868618

Fig. 15: GravelDB Performance: 5 million

RocksDB: 5 million insertions						
Trial	Write(GB) - Sum	Wnew(GB) - Sum	W-Amp (Sum)	Comp(sec) - Sum	Comp(cnt) - Sum	Uptime(secs)
0	51	12.1	10.4	150	1445	140.2
1	49.9	12.2	10.2	188	1447	169.7
2	50.7	12.2	10.3	185	1447	168.1
3	51.6	12.1	10.5	192	1449	173.1
4	48.6	12.1	9.9	177	1446	160
5	51.5	12.2	10.5	189	1449	172.4
6	49.8	12.3	10.1	179	1448	160.6
7	49	12.1	10	178	1443	160.8
8	51.5	12.2	10.5	184	1450	168
9	52.1	12.2	10.6	195	1449	177.5
Average	50.57	12.17	10.3	181.7	1447.3	165.04
Std Dev	1.188883697	0.06749485577	0.240370085	12.64954984	2.162817093	10.51107353

Fig. 16: RocksDB Performance: 5 million

GravevlDB: 10 million insertions						
Trial	Write(GB) - Sum	Wnew(GB) - Sum	W-Amp (Sum)	Comp(sec) - Sum	Comp(cnt) - Sun	Uptime(secs)
0	124.3	40.4	12.6	501	2993	469.3
1	124.5	40	12.6	497	2988	468.3
2	127.8	40.7	13	517	2993	484.8
3	128.3	39.9	13	523	2992	489
4	126.3	40.3	12.8	502	2992	474.1
Average	126.24	40.26	12.8	508	2991.6	477.1
Std Dev	1.83521116	0.3209361307	0.2	11.3137085	2.073644135	9.329790994

Fig. 17: GravelDB Performance: 10 million

RocksDB: 10 million insertions						
Trial	Write(GB) - Sum	Wnew(GB) - Sum	W-Amp (Sum)	Comp(sec) - Sum	Comp(cnt) - Sun	Uptime(secs)
0	141.5	26.5	14.4	516	2924	481.6
1	138.4	26.6	14.1	218	2924	482.2
2	138.3	26.6	14	523	2931	487.7
3	140.1	26.5	14.2	535	2923	497.1
4	136.4	26.6	13.9	521	2919	483
Average	138.94	26.56	14.12	462.6	2924.2	486.32
Std Dev	1.939845355	0.05477225575	0.1923538406	136.9134763	4.324349662	6.488220095

Fig. 18: RocksDB Performance: 10 million