Data-driven optimizations for Log-Structured Merge trees

Arijit Pramanik
University of Wisconsin-Madison

Nithin Venkatesh
University of Wisconsin-Madison

ABSTRACT

Modern database systems leverage key-value stores based on Log-Structured Merge (LSM) trees in their storage layer for storing data requiring fast look-ups and updates. However, recent studies have shown that application throughput can be compromised by internal LSM tree operations like background compactions that periodically write data to disk. We aim at improvising the existing background work scheduler on Google's LevelDB for better application performance. We decouple the foreground writes from background in-memory component flushes and compactions and show that for real-life workloads with write bursts, we are able to obtain 2.24-2.34X improvement in terms of observed client write throughput by scheduling these background operations during idle periods or when there are very few or no writes to the database.

1. INTRODUCTION

Relational database models [25, 21, 18] come with an inbuilt-schema and hence any update to a relation must follow the schema. Hence, operations like insertions and deletions incur high overheads. NoSQL databases [12] are non-relational and can support fast insertions and lookups critical to latency-sensitive applications with unstructured or even semi-structured data. Recent trend has shown that many NoSQL databases like Apache Cassandra [14] and storage systems like Google Bigtable [5] use LSM trees to organize the data on disk (HDD or SSD).

Applications require data platforms to deliver low latency and high throughput. It is especially important for workloads with frequent updates in quick succession with bursty behaviour. Log-Structured Merge key-value stores (LSM KVs) [23] such as RocksDB [27], LevelDB [15], DynamoDB [9] are widely used in production today for such write-heavy workloads. These are used to store frequently accessed/updated metadata for a wide range of enterprise applications. LSM trees have gained widespread popularity owing to their ability to absorb writes quickly in an in-memory buffer [22] and provide high write throughput to applications without affecting read performance. A tree-like structure is maintained on storage. In addition to client operations, LSM KVs implement two types of internal operations: *flushing*, which persists the content of in-memory buffers to disk, and *compaction*, which merges data from the higher levels (near to level-0) into the lower levels of the tree to remove older versions of the key value pairs.

In this paper, we study the working of LSM trees by analysing a popular NoSQL database, LevelDB, which is a key-value store based on LSM trees with levelling merge policy (illustrated in Fig 2) optimized for reads. LevelDB was developed by Google in 2011. We demonstrate that it suffers from poor write throughput, especially under heavy and variable client write loads. There has been recent focus on improving the client throughput of LSM KVs [24, 19, 13, 10] by focusing on the cost of internal operations but this increases write latency. Client operations arriving during ongoing internal operations will experience high latencies because of interference with theses internal tasks, especially for bursty client loads.

A naive approach would be to tune several hyper-parameters related to the LevelDB tree structure [28, 29] to improve performance for each workload. This can help us to adapt to a variety of workloads, including bursty ones. However, a burst of client writes can trigger a burst of flushes. Due to client writes being logged to disk at the same time, flushes have to share the limited bandwidth and are slow. Writes can get stalled if this in-memory component fills up, introducing delays. This leads to the need for coordination between foreground client load and internal operations and hence, the need for redesigning the background work scheduler in LSM trees.

2. BACKGROUND

We now discuss the various components in an LSM KV store and various client and internal operations supported by them.

2.1 LSM KV Architecture

As shown in Fig 1, the main data structures used in LevelDB are the Memtable, SSTable and the commit log.

- **Memory component** : This is an in-memory sorted data structure C_m, also referred to as memtable. It is typically a B-Tree, skip list, hash table or even an array. The purpose of C_m is to temporarily absorb user updates. When C_m exceeds its size threshold, it is replaced by a new, empty component in memory.

Proceedings of the CS 764 course
.

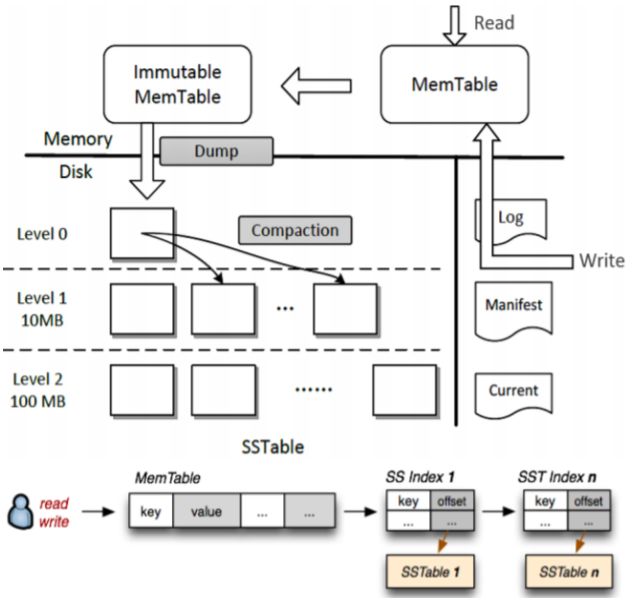

Figure 1: LevelDB Design

The previous one is then flushed as-is to level-0 L_0 of the LSM disk component in the background.

- **Disk component** : The disk component C_{disk} is structured into multiple levels $(L_0, L_1, ...L_n)$, where each level is larger than the previous level by a configurable factor. LevelDB uses a size ratio of 10. Each level contains multiple sorted files (sorted by key wherein key-value records maybe compressed), called SSTables (short for sorted string tables). The number of SSTables on a given level is limited by a configuration parameter, as is the maximum size of an individual SSTable for a given level. L_0 can store a maximum of 4 SSTables, while L_1 can store maximum 10 MB, L_2 100 MB and so on before triggering compactions. SSTables on levels L_i $(i > 0)$ have disjoint key-ranges. L_0 allows overlapping key-ranges between files.

- **Commit log** : The commit log C_{log} stores the updates that are made to C_m on stable storage in small batches. C_{log} is typically smaller than C_m. It is used if the application requires the data to be durable in case of a failure, but it is not mandatory. The heuristics/optimizations we propose apply regardless of whether C_{log} is active or not.

We also highlight the read and write paths in LevelDB as illustrated in Fig 1.

- **Read path** : The read first goes to C_m in memory. If the key k is not found in C_m, the read continues to $L_0, L_1, ...L_n$, until k is found. At most one SSTable is checked on each level L_i for $i > 0$ owing to non-overlapping key ranges. More than one SSTable in L_0 may need to be checked because L_0 SSTables may contain the entire key-range. Per-SSTable Bloom filters [11, 3] are used to address this issue. Therefore, in practice, only a single SSTable ends up being checked on L_0 most of the time.

- **Write path** : When the *Update(k, v)* or the *Write()* call for a key-value pair is made on the database, the key value pair is logged to commit log C_{log} and then appended to the memtable C_m in memory. The writes continue till C_m reaches a particular size and then, a new empty memtable is created to take the writes and the previous one is flushed. This can recursively trigger compactions until all levels are within their size thresholds.

2.2 LSM KV Operations

LSM KVs implement two main kinds of operations, which are executed within a single thread in the background for LevelDB.

- **Client operations** : The main client operations in LSM KVs are writes (*Update(k, v)*), point reads (*Get(k)*), and range scans (*Scan(k1, k2)*). *Update(k, v)* associates value v to key k. Updates are absorbed in C_m, to achieve high write throughput. *Get(k)* returns the most recent value of k. *Scan(k1, k2)* returns a range of key-value tuples with the keys ranging from *k1* to *k2*. First, C_m is queried for keys in the *k1–k2* range. Then, SSTables in C_{disk} that may contain the *k1–k2* range are read, going down the levels, until all the keys are found. Client operations are enqueued and served in FIFO order by a dedicated thread.

- **Internal operations** : LSM KVs implement flushing and compaction as background processes. Flushing dumps C_m as is onto L_0. Because flushing speed affects the rate at which new memory components can be created, memory components are written to disk without additional processing. As a result, L_0 allows overlapping key-ranges between files. Compaction is the operation that cleans up/compacts the LSM tree by removing duplicates/deleted values. It merges SSTables in level L_i of C_{disk} into SSTables with overlapping key ranges in L_{i+1}, discarding older values in the process. When the size of L_i exceeds its threshold, an SSTable F in L_i is picked according to some priority or in a round-robin fashion and merged into the SSTables in L_{i+1} that have overlapping key-ranges with F, in a way similar to a merge sort. Compaction introduces large I/O overhead by reading the SSTables and writing new ones to disk. When a new internal work request is enqueued, it is placed at the end of the queue. In LevelDB, a background flush and compaction are enqueued when C_m fills up. An internal worker thread serves the requests in the internal work queue. An internal operation is enqueued whenever the system deems it necessary in order to maintain the structure of the LSM tree (e.g., when the maximum size or maximum number of files is reached on a level).

2.3 Important LSM KV parameters

There are several important tuning knobs in LevelDB which can lead to improvement in performance. We enlist some DB initialization or key value store startup time parameters which can be passed to the binary:

- `write_buffer_size` : This controls the upper limit of memtable size after which it is converted to an immutable version and flushed to disk. LevelDB uses a default value of 4 MB.

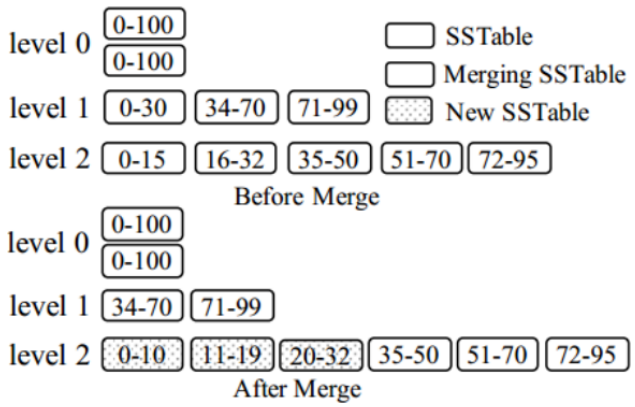

Figure 2: Levelling merge policy

- `max_file_size` : SSTable sizes can go upto this limit. As the compaction outputs are written, it is broken into chunks of the above size, where each such chunk is an SSTable. This is same across all levels. LevelDB's default value is 2 MB.
- `sync` : This boolean flag when set to true, ensures that any update is persisted to the commit log before being written to the memtable. When false, these updates are persisted lazily onto disk. This is similar to the "force" vs "no-force" policy used by buffer managers. LevelDB uses synchronous logging.

Now we enlist some interesting internal parameters modifying which needs a complete recompilation for running the binary:

- `kL0_CompactionTrigger` : L_0 to L_1 compaction is started when we hit this many SSTables at L_0. Default is 4 SSTables in LevelDB.
- `kL0_SlowdownWritesTrigger` : To prevent a huge delay as SSTables accumulate at L_0, writes are gradually delayed as L_0 files build up. Each individual write is delayed by 1 ms when L_0 files are more than this parameter value, whose default value is 8 in LevelDB.
- `kL0_StopWritesTrigger` : When the number of L_0 files is greater than this value, incoming writes are stalled on a conditional variable unless L_0 to L_1 compactions complete, reducing the number of L_0 files. The default is 12 SSTables in LevelDB.
- `kNumLevels` : Total number of levels in the LSM tree. All compactions push files to lower levels of the tree. Any compactions happening at the last level of the tree push new files onto the last level itself. LevelDB has 7 levels by default.

2.3.1 Bayesian Optimization

The impact different (input) parameters (e.g. buffer size, worker thread count, etc) can have on the (output) performance of a system for a given workload (input) can be modeled as a multidimensional function - whose equation we don't know apriori, but are instead trying to learn through careful sampling of the input space and experimentation (test/benchmark runs) to gather output points. *Bayesian optimization* is one technique for efficiently selecting the samples in the input space to learn the approximate shape of that function and find its optimum, i.e., the parameters that lead to the best performance.

Bayesian Optimization is a global optimization strategy, and can find the global optimum of a mathematical function that's not necessarily convex, without requiring information like gradients. Finding the global optimum of a general non-convex function is NP-hard, which makes it impossible to provide effective convergence guarantees for any global optimization strategy, including Bayesian Optimization. However, it has been found to be quite effective in the past.

3. MOTIVATION

We would like to design a system that is self-tuning based on the workload. For this work, we concentrate on workloads that have write bursts with intervening idle intervals. In order to absorb the write bursts, the system must be able to schedule the background work during these idle periods without affecting the foreground writes. If they overlap or interfere with the foreground writes, this might saturate all available CPU cores and disk bandwidth limits in cloud computing environments with constrained resources.

So, we aim at taking the first step towards building a system that has explicit control over the background work thereby leading to better performance in terms of metrics including throughput, latency with reduced read, write and space amplification. Any amplification implies extra overhead and should be minimized. E.g. Write amplification is the ratio of the amount of physical data written to disk to the amount of logical or actual data intended to be written. LSM trees use compactions to organize data in the form of sorted files to optimize lookup performance. We try to study the working of the compaction process in LevelDB and suggest ways to improve its scheduling and measure its impact on all the metrics discussed above.

Typical to any database system, LevelDB comes with several parameters that have to be tuned to obtain the best performance for any given workload characteristics. We explore the DB initialization time and compile-time parameters provided by LevelDB and their impact on the performance metrics. We jointly tune a combination of parameters and optimize for performance in terms of latency and throughput while also experimenting with reliability metrics like batch synchronization.

4. RELATED WORK

LSM tree based key value stores provide various opportunities and tradeoffs between various metrics including write/read latencies and throughputs, write/read amplification and space usage of the database. There have been a significant number of works that have looked at these tradeoffs over the years. These works can be classified into looking at the core design of the LSM tree, algorithms considered for the compaction process, tradeoff between memory usage and user perceived metrics and using hardware trends for better and efficient design of LSM trees. The following are some of the works relevant to LSM key value stores.

SILK by Balmau et al. [2] talks about the high tail latencies observed in LSM KV stores and attributes this to the interference between client writes, flushes and compactions.

They introduce the notion of an I/O scheduler for LSM KV stores to reduce the said interferences. They mainly propose three techniques to reduce tail latencies. They opportunistically allocate more write bandwidth to the flushes and compactions during periods of low foreground load. This is achieved by rate limiting writes from the background jobs scheduled by RocksDB. They prioritize flushes and compactions at the lower levels (near level-0) of the tree. This is implemented by using high and low priority thread pools for different background work. The compactions corresponding to the lower levels (level-0 to 1) are scheduled using the high priority threads as opposed to the other compactions being scheduled using low priority threads. Whenever the flushes are stalled due to the background compactions, the compactions are pre-empted to provide higher priority to the flushes of immutable memtables to level-0 in the LSM tree. They show that they are able to achieve an order of magnitude reduction in 99^{th} percentile write latencies on production workload with write bursts at Nutanix using the proposed techniques.

MatrixKV by Yao et al. [30] proposes techniques to reduce write amplification and write stalls in LSM KV stores using matrix containers in Non Volatile Memory (NVM). The main observations that they make are that write stalls stem from large amounts of data involved in L_0 to L_1 compactions. They present techniques to do smaller and cheaper L_0 to L_1 compactions and they mainly concentrate on DRAM (used for Memtable)-NVM (used to store the L_0 level files)-SSD (other levels in the LSM tree) architecture. They devised a technique called column compaction where they divided the files in L_0 into key ranges and compacted based on key ranges rather than considering complete files in L_0. This technique helps in reducing the amount of data read and written as part of each compaction. They observe that the write amplification increases as the number of levels in an LSM tree increases. In order to address these issues, they propose increasing the width of the lower (closer to level-0) levels thereby decreasing the depth of the LSM tree and reducing write amplification. They show that using these techniques, they are able to obtain orders of magnitude improvement in tail latencies and random write throughput.

LSM based storage techniques: a survey by Luo et al. [17] presents a survey of the recent work on LSM trees. They provide a taxonomy to classify LSM trees and also discuss the strengths and tradeoffs of the works presented. The paper also presents a discussion about open source LSM-based NoSQL key value stores. They classify related works into the following categories.

- Reducing write amplification: Triad by Balmau et al. [1] uses the techniques of leveraging data skewness in the memory component to avoid frequent I/O operation. At the storage level, Triad uses the technique of batching multiple I/Os for better performance.
  PebblesDB by Raju et al. [26] presents a novel data structure inspired by skip lists which they term as fragmented LSM. They introduce the concept of guards in each level of the LSM tree. The guards are probabilistically selected and they separate the key range into multiple partitions and use them to avoid rewriting data at the same level reducing write amplification.
- Hardware trends: Wisckey by Lu et al. [16] presents a persistent LSM KV where they separate the key from the values to minimize I/O amplification, making use of the features provided by SSDs. They show that Wisckey is 2.5-111X faster as compared to LevelDB in loading the database and 1.6-14X faster for random lookups. Storing the values as a log gives better write and I/O performance compromising range queries.

The paper also discusses other papers in the category of auto-tuning LSM trees like Monkey and Dostoevsky, LSM trees for special workloads like LSM Trie and SlimDB and works focusing on merge operations including bLSM and FloDB.

Monkey by Dayan et al. [7] presents that key value stores backed by LSM trees have a tradeoff between lookup cost, update cost and main memory cost. They show that by allocating a higher number of bloom bits to the lower (near to level-0) one can achieve reduction in the lookup cost but with a given memory budget and false positive rate.
Dostoevsky by Dayan et al. [8] tries to show that equally expensive merge operations done across all levels of LSM tree are not actually required. They propose *lazy leveling* that removes merge/compaction operations from all levels except the last level improving the worst case update cost while maintaining the same bounds on point lookup costs and long range lookup costs. They introduce Fluid LSM, a generalization of the LSM tree design space that can be parameterized to assume existing LSM designs based on the application workload and underlying hardware.

Characterizing, Modeling,and Benchmarking RocksDB Key-Value Workloads at Facebook by Cao et al. [4] presents a characterization of workload from RocksDB production use case at Facebook. They show that the distribution of the keys and values are highly related to the use case and they show that access to the key values have good locality and follow certain patterns.

Bourbon by Dai et al. [6] explores how to reduce the read latency by introducing point wise linear regression for constant time lookup of the key in an SSTable of the LSM tree. They also develop a cost-benefit analysis model to decide when it is advantageous to build the linear model for an SSTable based on the average lifetime of SSTables.

5. NEW SCHEDULER DESIGN

We make the following changes to decouple the foreground activity (writes) from background activity (compactions). LevelDB is typically used as a *single-threaded* application. Foreground writes are blocked when the immutable memtable is being flushed to level-0 of the LSM tree. Background compaction process is scheduled on a different thread from the foreground write thread, but performs flushing of immutable memtable (if any) before that on the same thread running compaction. Any subsequent compactions triggered will run on a separate thread.

We have devised a way to exclude the foreground writes from overlapping with memtable flushes and background compactions. As shown in the Fig 3, writes are appended

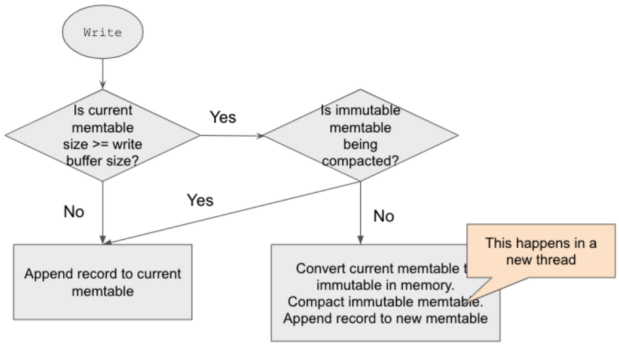

Figure 3: New write path designed for improved performance

to the current memtable which is in-memory after writing to the log. Once the size of the memtable exceeds the current `write_buffer_size` value, we check if the previous memtable is still being flushed, and if that is the case we use the current memtable to absorb pending writes. If there is no previous memtable being flushed, we convert the current memtable into its immutable version, and create a new, empty memtable to take the incoming writes while triggering the flushing of this immutable memtable using a new thread, which flushes it to level-0 of the LSM tree.

A separate thread is created when opening the DB that continuously monitors the client write throughput and determines whether any background compaction is needed. This thread itself performs background compaction of SSTables from one level to another. For a bursty workload comprising mostly writes, we choose simple heuristics to schedule background compactions whenever the write throughput is below a certain threshold, or after the last write of a write burst in the optimized system.

Through this, we show how one can have explicit or learned rules based on the workload, that can be used to control when the background compaction is supposed to happen. Since the background compaction process is both CPU and disk (I/O) intensive, this can lead to significant improvements in terms of throughput and latency for foreground writes as observed by the application or client.

6. EVALUATION

6.1 Workload

We concentrate on designing a workload with write bursts, wherein a process writes data for short intervals of time followed by a sleep duration which the process keeps repeating for the entire workload duration. Taking inspiration from Fig 4, we design the workload and implement it as a part of the micro benchmarks provided by LevelDB. We refer to the experimental workload as *"write random bursts in time"*, although we can choose to perform a certain number of writes before going to sleep, referred to as *"write random bursts in count"*. We can run the workload by specifying the total duration of experiment, write burst interval and the sleep duration or the idle time between write bursts. We can simulate various workload durations, where the writer process writes for bursts of intervals specified in an array followed by corresponding sleep intervals.

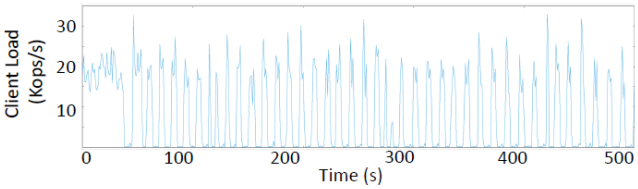

Figure 4: Workload trace from Nutanix production environment

6.2 System Setup

We use Cloudlab bare-metal c220g1 machines to run the experiments with Intel Xeon CPU E5-2630 v3 2.40GHz x86 architecture. We experiment using both HDD and SSD as the underlying storage medium. We present the results for SSD as the storage medium as we observe the same performance improvement across both media. Linux `cgroups` is a kernel utility that can be used for limiting the resource usage for a process or a group of processes. We use this utility to limit the number of cores to the bare minimum required for the experiments. We use of `cgroup v2` to limit the disk write bandwidth to the bare minimum 100 MB/s that can satisfy the client load in the default setting of LevelDB and to also observe the behaviour of the scheduling policy coupled with different device bandwidth constraints.

6.3 Results

6.3.1 *Approach 1: Tuning parameter knobs*

We use a remote optimizer, Microsoft's MLOS [20] leveraging Bayesian Optimization whose architecture is shown in Fig 5. It runs on a separate thread that probes latency and throughput values (output performance metrics) at every interval and tries to suggest new values for various parameters of LevelDB for each run. It then again monitors these output metrics for subsequent runs with new parameters to check for potential improvements and provides further suggestions based on that. The optimization proceeds in iterations where it explores new values while staying close to the most recent optimal value. We configure the exploration and exploitation ratio as 1:10 using random forests to find optimal parameters for our given workload within as few iterations/runs as possible.

Here, we use one of the default workloads of LevelDB, `"fillrandom"`, that randomly inserts keys in the range of 1 to 1 million sampled from a uniform distribution. We try to maximize write throughput as a function of `write_buffer_size` (memtable size), `max_file_size` (SSTable size) and internal parameters in 2.3 that we export as runtime parameters like `kL0_StopWritesTrigger` and `kL0_SlowdownWritesTrigger` for this experiment. We perform optimization for parameters one-by-one as well as demonstrate the joint optimization of parameters.

Logging : We also run the same `"fillrandom"` workload with both synchronous and asynchronous logging as described in 2.3. Synchronous logging persists updates in batches of size `entries_per_batch` on disk. Default synchronous logging persists updates one-by-one. Turning that off gives a much higher throughput, but doesn't provide any guarantee of inorder on-disk persistence. Using larger batches gives performance comparable to asynchronous log-

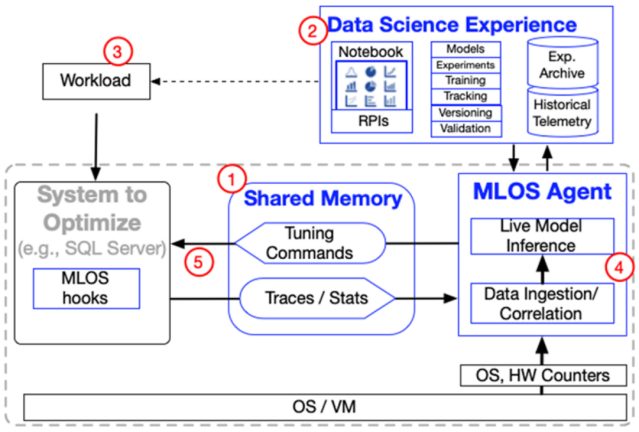

Figure 5: MLOS Architecture

Mode	Batch size	Throughput	Latency
Async	-	21.8 MB/s	5.87 μs/op
Sync	1	0.4 MB/s	389.85 μs/op
Sync	1000	24.1 MB/s	5.4 μs/op

Table 1: Results for synchronous vs asynchronous logging

ging but an entire batch of updates may get lost. The results are provided in Table 1.

Memtable size : With `"fillrandom"` workload, we measure throughput as a function of memtable size, starting from 1 MB to 128 MB in steps of 2X. We observe that the write throughput increases as we keep on increasing the size of the memtable as shown in Fig 8 until 16 MB. Larger memtables absorb more write bursts needing lesser flushes. The write throughput decreases after that since keys have to be inserted in their proper place in an already large sorted skip list that LevelDB uses. This is empirically verified by MLOS in Fig 6 which produces an optimal memtable size of 9.86 MB for a maximum write throughput of 36.3 MB/s in 7. However, the cost of lookups increases with memtable size. This experiment acts as an example to affirm that Bayesian optimization can be used for database parameter tuning.

SSTable size : We empirically evaluate the optimal SSTable size for maximum write throughput. We see that MLOS converges to a value of 26.87 MB in Fig 9, much larger than the default size of 2 MB. This decreases the frequency of background compactions, making it less likely to interfere with client writes. Fig 10 shows how write throughput evolves with MLOS' optimal value giving a max throughput of $\sim$ 31.77 MB/s. This is similar to the optimal SSTable size of 32 MB that we obtain in Fig 11 on running with various SSTable sizes from 32 KB to 64 MB in steps of 2X.

Joint optimization : Now, we maximize throughput as a function of both memtable size and SSTable size. We find an optimal memtable size of 8.47 MB and SSTable size of 12.85 MB from Fig 12 and 13 respectively. Jointly optimizing parameters taking into account their *correlation* gives us higher throughput ($\sim$ 44 MB/s) than found previously as in Fig 14.

Slow down writes trigger : We now try to manually find the optimal value of `kL0_SlowdownWritesTrigger` that maximizes throughput while keeping other parameters to their default values, except for `kL0_StopWritesTrigger`, which we keep at 512 as we vary the `kL0_SlowdownWrites Trigger` from 1 to 128 in steps of 2X in 17. A higher value will not slow down writes and hence increase client throughput at the expense of accumulation of more L_0 files which can lead to queuing up of too many L_0 to L_1 compactions. We also plot values found by MLOS and their corresponding throughput in 15 and 16 respectively. MLOS' optimal value of 58 differs slightly from 32 that we initially obtained.

Stop writes trigger : Now, we investigate the effect of only `kL0_StopWritesTrigger` on throughput, while keeping `kL0_SlowdownWritesTrigger` to 512 to prevent its influence on throughput. Here, we vary `kL0_StopWritesTrigger` from 4 to 128 in steps of 2X in 20. Since `kL0 _CompactionTrigger` is set to 4, this means that if the `kL0_StopWritesTrigger` is set to anything below that, the program will deadlock since it will be waiting on a lower number of L_0 files than what is required to trigger background compaction, which in turn signals the conditional variable on which writes wait. A higher value implies that lesser writes will be stalled significantly improving performance on bursty workloads while again building up too many L_0 files leading to L_0 to L_1 compactions that might interfere with future bursts. MLOS finds an optimal value of 82 as in Fig 18 and Fig 19, which is close to 64 as seen in Fig 18.

6.3.2 *Approach 2: Heuristic-based scheduling*

We use the `"write random bursts in time"` workload described above in 6.1 for a total duration of 120 seconds. We set the write bursts interval to 10 seconds, and idle interval to 20 seconds. So we observe continuous and random writes in a uniformly distributed key range from 1 to 1 million for 10 seconds followed by an idle time of 20 seconds. The key observation that we try to make here is how beneficial is it to do the background compaction work in the idle period compared to overlapping of the background compaction work with foreground writes as happens in LevelDB.

We use the `dstat` utility to measure the disk throughput and CPU utilization metrics. Using `pthread`s library functions, we bind the load generator thread, the memtable flushing thread and the background compaction thread onto different, independent CPUs and pin them for the entire workload duration. We opt for synchronous logging with a batch size of 100 in both cases.

Figure 21 shows how well the default compaction strategy works in LevelDB while Figures 22, 24 show 2 sets of observations to illustrate how our new heuristics affect throughput. The X-axis represents workload duration while the shaded blue intervals depict write bursts and white regions depict idle intervals. The five sub plots presented in each observation/graph are as follows:

- Subplot 1: LevelDB throughput indicates the throughput as observed by the client application doing the writes. Disk reads and disk writes indicate the throughput as observed by the disk measured using `dstat`.
- Subplot 2: This shows how the data in the in-memory memtable and at each level of the LSM tree changes over time. We can observe how the data is moved from higher levels to the lower levels (lower here being level-3 and higher being level-0). Flushes push data from memory onto level-0 while background compactions move the data to lower levels in the LSM tree.

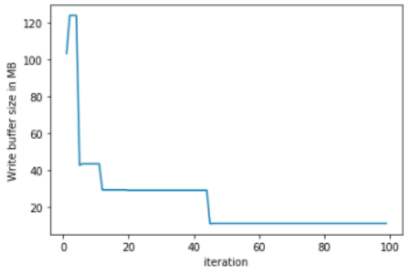

Figure 6: converging to optimal memtable (write buffer size) size vs Bayesian optimizer iteration

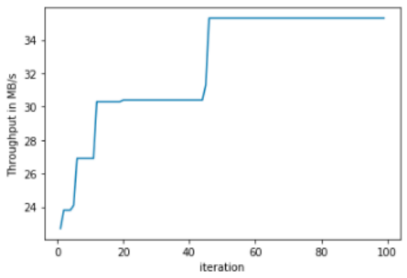

Figure 7: Peak throughput obtained when optimizer converges to the optimal Memtable size

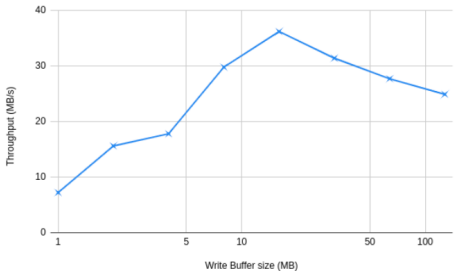

Figure 8: Throughput vs memtable size by manually tuning the memtable size for verification

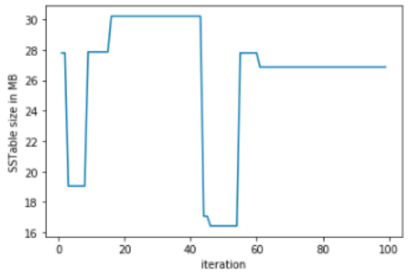

Figure 9: Optimal value of SSTable size vs iteration

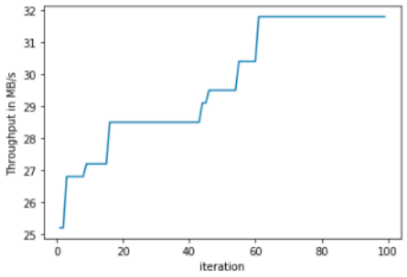

Figure 10: Peak throughput value vs iteration

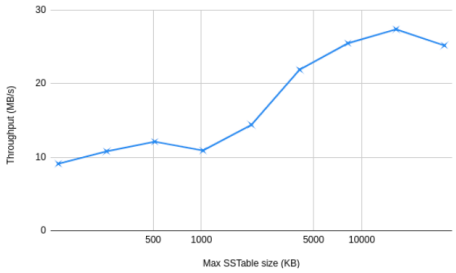

Figure 11: Throughput vs SSTable size with manual SSTable size tuning

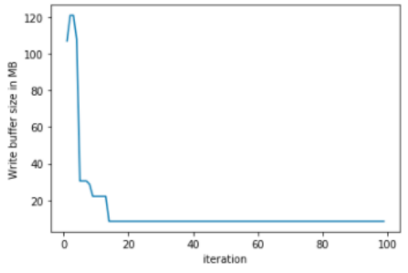

Figure 12: Optimal memtable size vs Bayesian optimizer iteration

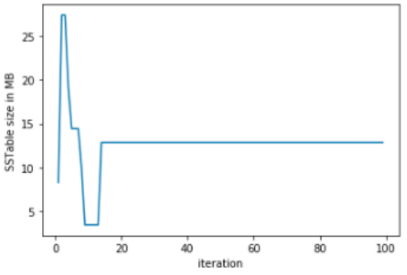

Figure 13: Optimal SSTable size vs Bayesian optimizer iteration

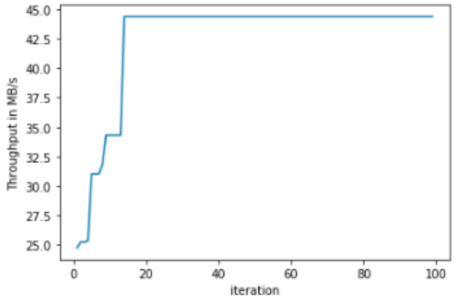

Figure 14: Throughput increase as the optimizer converges

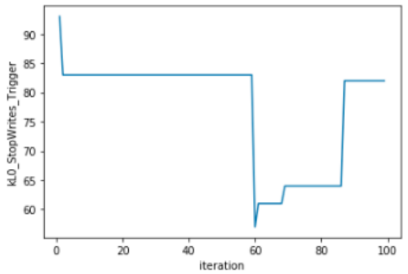

Figure 15: Optimal Slowdown Writes Trigger vs iteration

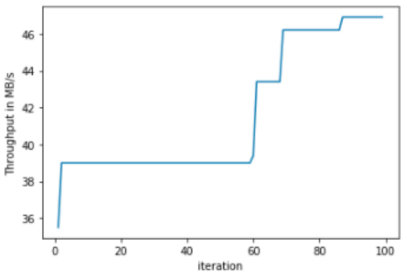

Figure 16: Peak throughput value vs iteration

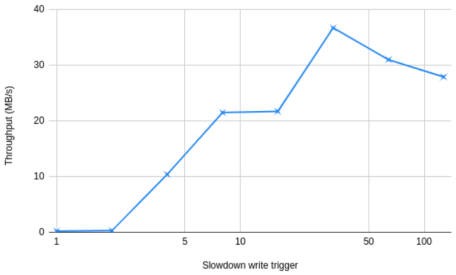

Figure 17: Throughput vs Slowdown Writes Trigger

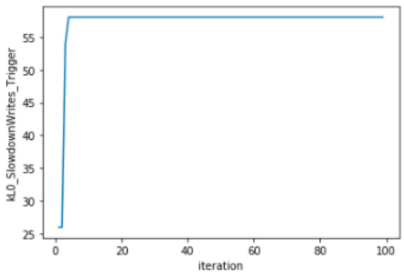

Figure 18: Optimal Stop Writes Trigger vs iteration

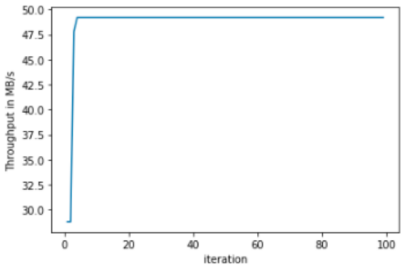

Figure 19: Peak throughput value vs iteration

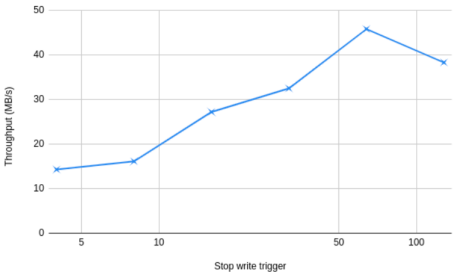

Figure 20: Throughput vs Stop Writes Trigger

- Subplot 3: This shows the CPU utilization during the course of the workload. We have limited the number of cores to 2 for default LevelDB configuration and 3 for the other experiments for ease of observation.
- Subplot 4: This plot shows data written due to the background compaction process. It also indicates the output level of the LSM tree to which the compaction outputs are written.
- Subplot 5: This plot shows the data written to level-0 of the LSM tree due to memtable flushes.

In Figure 21, we observe both L0 and L1 compactions happen during the write burst interval as the disk bandwidth and CPU cores remain largely unutilized during idle intervals. We see frequent flushes happening before each L0 to L1 compaction from subplots 4, 5. These severely decrease overall write throughput and increase latency as resources are over-utilized during write bursts and under-utilized during idle periods in between.

We allow memtable flushes, which are less CPU and I/O intensive to be scheduled automatically whenever memtable fills up. We schedule background compactions if need be whenever the write throughput is below 0.5 MB/s as measured by the monitoring thread. Client throughput is probed at intervals of 500 ms. Figure 22 shows an $\sim$ 2.2X increase in write throughput over default setting. From subplot 1, we observe no competition for disk bandwidth in Figure 21. As evident from subplots 2, 3, and 4, compactions happen only in idle intervals between bursts. We observe a single chunk of L_0 to L_1 compaction, followed by comparatively less intensive L_1 to L_2 compaction. If the probing is too frequent, this may add some overhead and a very small interval ($\sim$ 1μs) can lead to no write in such short intervals even within a write burst, leading to compactions creeping in. However, tuning this threshold is important since a higher threshold may lead to some compactions happening within the write burst itself as shown in Figure 23. This is a characteristic of the workload itself, hence we explore other heuristics which are completely independent of the pattern of write bursts.

Figure 24 demonstrates scheduling of background work 500 ms after the last write in a write burst. In this case, the probing thread checks very frequently to accurately record the last write within each burst. This probing adds some CPU overhead as seen in subplot 3. Importantly, we see that here also, all compactions are scheduled in the idle intervals with a major L_0 to L_1 compaction followed by minor L_1 to L_2 compactions. Flushes with lightweight CPU and memory footprint happen as memtable becomes full during the write bursts itself. As before, background compactions consume CPU and disk bandwidth only during idle intervals leading to a 2.34X increase in write throughput and marginal decrease in write latency.

Approach	Data written	Throughput	Latency
Default	741.7 MB	6.1 MB/s	18.060 μs/op
1	1659.3 MB	13.7 MB/s	8.075 μs/op
2	1737.7 MB	14.3 MB/s	7.711 μs/op

Table 2: Comparison of different background scheduling approaches

A comparison among these approaches is provided in Table 2, showing the benefits of our approach for bursty write workloads that dominate today's datacenters.

7. CONCLUSION

In this project, we presented a way to make the internal maintenance in LSM KV stores aware of the client workload for better scheduling of internal background tasks to fully utilize resources without hampering client performance. Specifically, we show that by decoupling the foreground writes from memtable flushes and background compactions, we can have explicit control over scheduling the background compactions. We show that this can be beneficial for industrial workloads with write bursts and idle intervals, since these idle time periods can be used for the background compaction process. This has showed $\sim$ 2X increase in client write throughput and decrease in write latency without any need for tuning parameters controlling the LSM tree structure for each workload. We show scenarios where we can obtain throughput improvement, one where client issues batch synchronous writes in bursts as well as asynchronous writes in bursts. Although these are specific cases, we believe that the technique is general enough and machine learning techniques can be used to learn the best compaction scheduling policy and corresponding mechanisms for LSM trees.

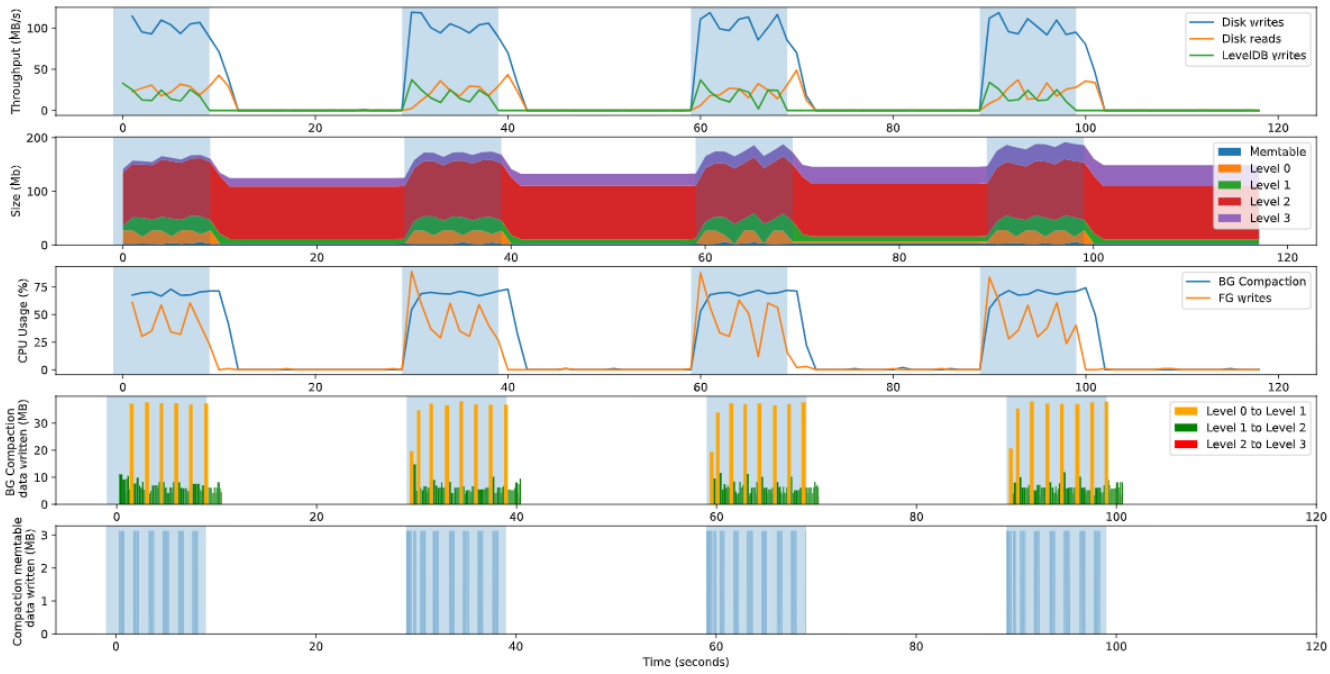

Figure 21: Default compaction scheduling policy in LevelDB: Write throughput: 6.1 MB/s

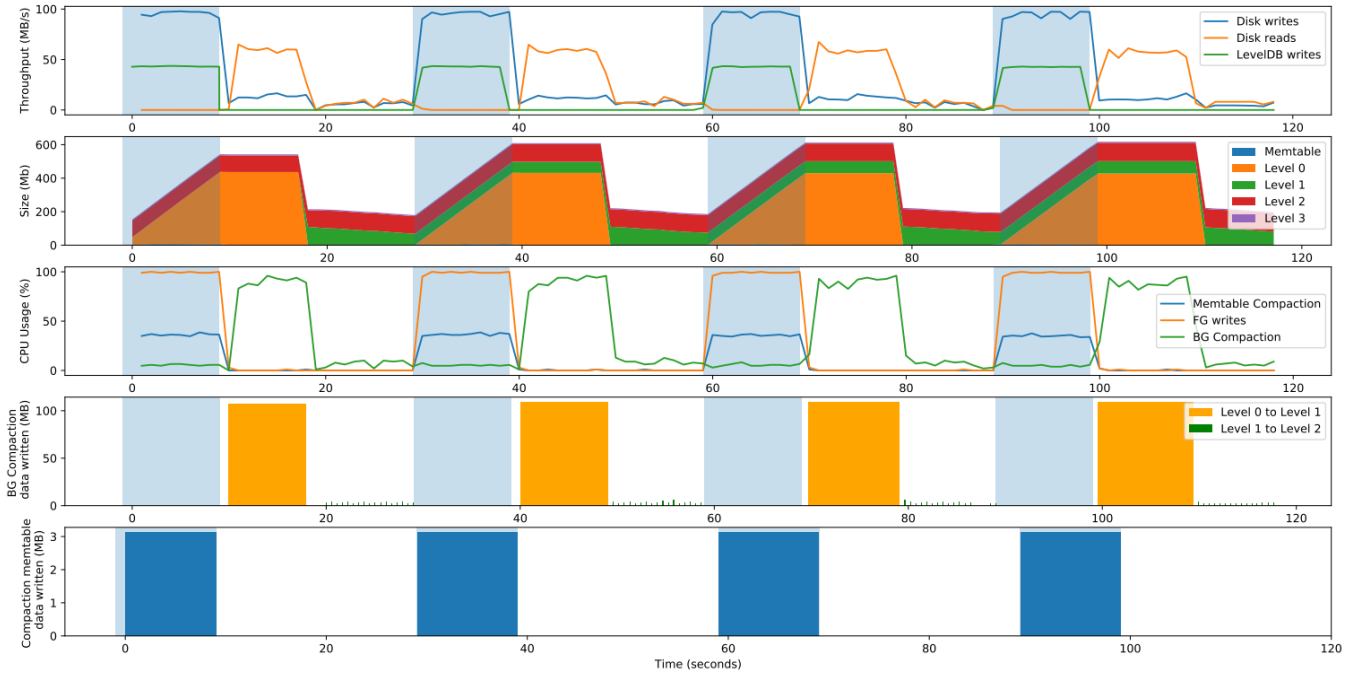

Figure 22: Scheduling compactions for write throughput below a certain threshold: Write throughput: 13.7 MB/s

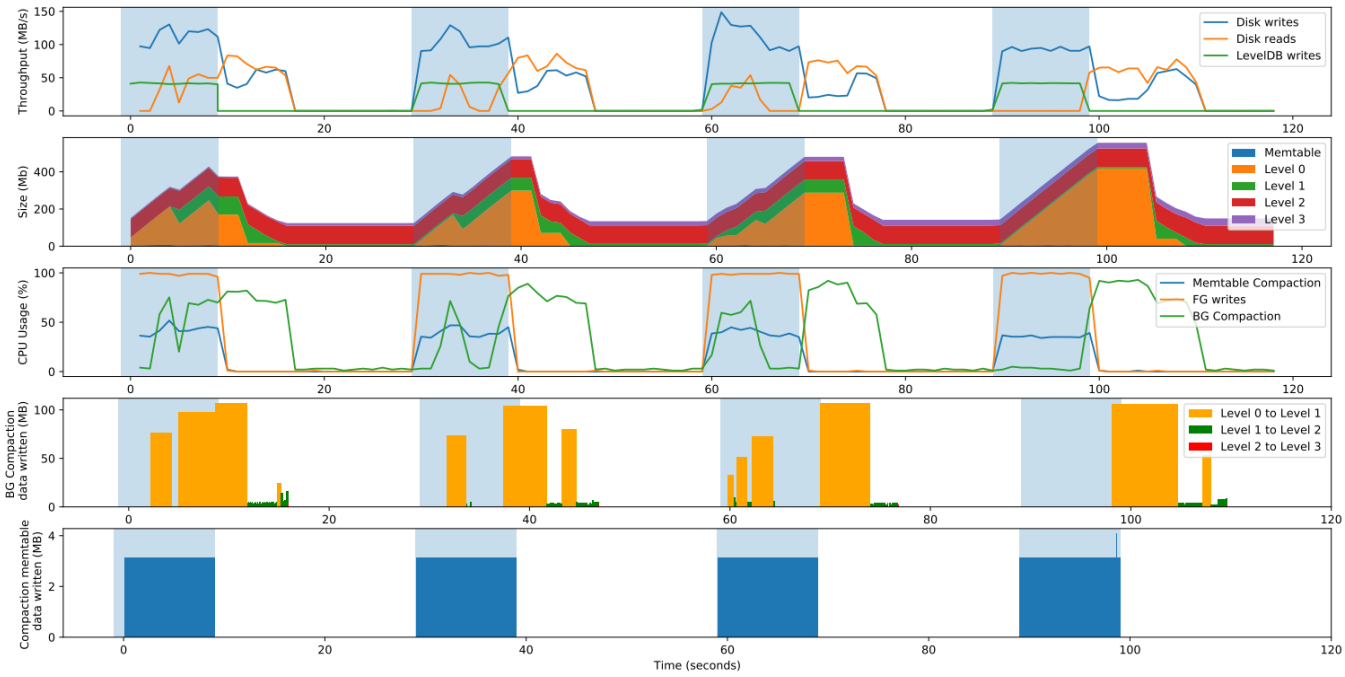

Figure 23: Scheduling compactions without a low enough threshold for write throughput: Write throughput: 11.9 MB/s

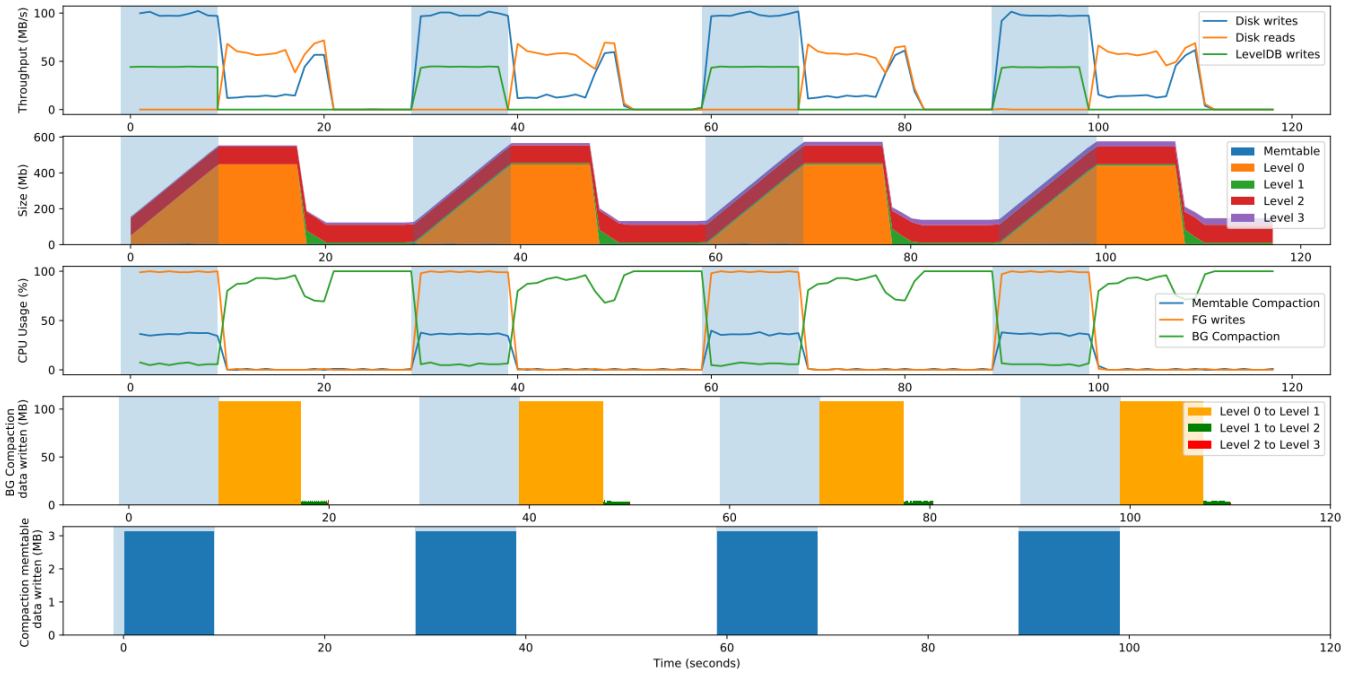

Figure 24: Scheduling compactions just after last write in a write burst: Write throughput: 14.3 MB/s

8. INDIVIDUAL CONTRIBUTIONS

Since, we are two people working on the project, both of us were involved almost equally in all of the stages.

- Both of us contributed equally in experiments of using MLOS for tuning the Memtable size, SSTable size and other initialization and compile parameters for better performance
- Equally involved in decoupling the background compactions and flushes from foreground writes
- Contributed equally to writing all sections of the report

9. REFERENCES

[1] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka. Triad: Creating synergies between memory, disk and log in log structured key-value stores. *USENIX Annual Technical Conference* , 2017.

[2] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona. Silk: Preventing latency spikes in log-structured merge key-value stores. *USENIX Annual Technical Conference*, 2019.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[4] Z. Cao, S. Dong, S. Vemuri, and D. H. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. *USENIXFAST*, Feb. 2020.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[6] Y. Dai, Y. Xu, A. Ganesan, R. A. andBrian Kroth, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. From wisckey to bourbon: A learned index for log-structured merge trees. *USENIX OSDI*, Nov. 2020.

[7] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal navigable key-value store. *SIGMOD*, May 2017.

[8] N. Dayan and S. Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. *SIGMOD*, 2018.

[9] DynamoDB. https://aws.amazon.com/dynamodb/.

[10] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar. Scaling concurrent log-structured data stores. In *Proceedings of the Tentrh European Conference on Computer Systems*, pages 1–14, 2015.

[11] Y. Hua, B. Xiao, B. Veeravalli, and D. Feng. Locality-sensitive bloom filter for approximate membership query. *IEEE Transactions on Computers*, 61(6):817–830, 2011.

[12] Jing Han, Haihong E, Guan Le, and Jian Du. Survey on nosql database. In *2011 6th International Conference on Pervasive Computing and Applications*, pages 363–366, 2011.

[13] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Redesigning lsms for nonvolatile memory with novelsm. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 993–1005, 2018.

[14] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[15] LevelDB. https://github.com/google/leveldb.

[16] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *USENIX File and Storage Technologies*, Feb. 2016.

[17] C. Luo and M. J. Carey. Lsm-based storage techniques: A survey. *VLDB Journal*, 2019.

[18] MariaDB. https://mariadb.org/.

[19] F. Mei, Q. Cao, H. Jiang, and L. Tian. Lsm-tree managed storage for large-scale key-value store. *IEEE Transactions on Parallel and Distributed Systems*, 30(2):400–414, 2018.

[20] MLOS. https://microsoft.github.io/mlos/.

[21] MySQL. https://www.mysql.com/.

[22] J. Ousterhout and F. Douglis. Beating the i/o bottleneck: A case for log-structured file systems. *ACM SIGOPS Operating Systems Review*, 23(1):11–28, 1989.

[23] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[24] A. Papagiannis, G. Saloustros, P. González-Férez, and A. Bilas. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*, pages 537–550, 2016.

[25] PostgeqSQL. https://www.postgresql.org.

[26] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. *SOSP*, Oct. 2017.

[27] RocksDB. https://rocksdb.org/.

[28] N. J. Ruta. *CuttleTree: Adaptive Tuning for Optimized Log-Structured Merge Trees*. PhD thesis, 2017.

[29] M. Weise. On the efficient design of lsm stores. *arXiv preprint arXiv:2004.01833*, 2020.

[30] T. Yao, Y. Zhang, J. Wan, Q. Cui, L. Tang, PingCAP, H. Jiang, C. Xie, and X. He. Matrixkv: Reducing write stalls and write amplification in lsm-tree based kv stores with matrix container in nvm. *USENIX Annual Technical Conference*, 2020.