HyperPebblesDB: An Enhanced Fragmented Log-Structured Merge Trees Key-Value Store

Zeyuan Hu
University of Texas at Austin

Wei Sun
University of Texas at Austin

Jianwei Chen
University of Texas at Austin

ABSTRACT

Log-structured Merge Tree (LSM) [17] is a data structure that is widely used in write-intensive storage system. However, it suffers from write amplifications, which can hinder the write throughput. Another data structure called Fragmented Log-Structured Merge Trees (FLSM) [19] is proposed recently to reduce write amplification while maintaining high write throughput. In this project, we build a key-value store called `HyperPebblesDB` based on `PebblesDB` [19]. We enhance `db_bench` by calculating the sstables distribution and implement guard-based parallel compaction strategy. We empirically investigate the impact of empty guards and the guard-based parallel compaction on the FLSM-based key-value store. In addition, we integrate Succinct Range Filter (SuRF) [23] into `HyperPebblesDB`. Our results show that empty guards have no impact on the Read throughput; in certain scenario, guard-based parallel compaction can improve system performance; SuRF can improve the range query performance but with limitations.

1 INTRODUCTION

the Log-structured merge tree (LSM) has become the central data structure for the modern key-value store implementation. At the same time, LSM-based Key-Value store has been a very hot research field [21, 20, 15, 13, 22, 9]. LSM-based storage engine originated from Google's LevelDB [2] exploit the sequential writes to improve the write performance. Specifically, the data are appended to log file and multiple log files are organized into levels. Later, the logs in multiple levels are merge sorted into one. This behavior brings the write amplification effect as we have to read the log from the local disk. Recently, another LSM-ish data structure called Fragmented Log-Structured Merge Trees (FLSM) has been invented [19] to reduce write amplification while improving the write performance. However, several features are not implemented in the FLSM-based key-value store `PebblesDB`. Whether those implementation decisions are based-on empirical evidence is left unkown to us. In addition, how those implementations affect the store performance under distributed database MongoDB is also uninvestigated. Thus, in this work, we enhance `PebblesDB` by implementing both guard-based parallel compaction and the new filter option: SuccinctRange Filter (SuRF) [23]. At the same time, our empirically investigation shows that whether to implement the guard deletion is uncritical.

Our paper is organized as follows: we first introduce LSM, FLSM, SuRF, MongoDB in 2 section. Then, in section 3, we examine one implementation decision made by `PebblesDB`: not implementing guard deletion and we show that not implementing guard decision is a good decision. In section 4, we enhance `PebblesDB` by implementing the guard-based parallel compaction and our experiment result shows that guard-based parallel compaction can improve performance when there are many guards in the being-compacted level. In section 5, we integrate SuRF, a replacement for bloom filter, and we show that SuRF can outpeform bloom filter for range query in `PebblesDB`. Lastly, we conduct experiment on MongoDB, a distributed database.

2 BACKGROUND

2.1 LSM-Tree and PebblesDB

Log-structured Merge Tree (LSM) [17] is a data structure that is used to provide good write performance by leverage log organizations. In details, write to LSM-based storage system is first written to in-memory log called `MemTable` by appending the corresponding key-value pair at the end of log. Doing the write through appending is the key differentiator from the B-tree-based storage system as we are doing the sequential write in-

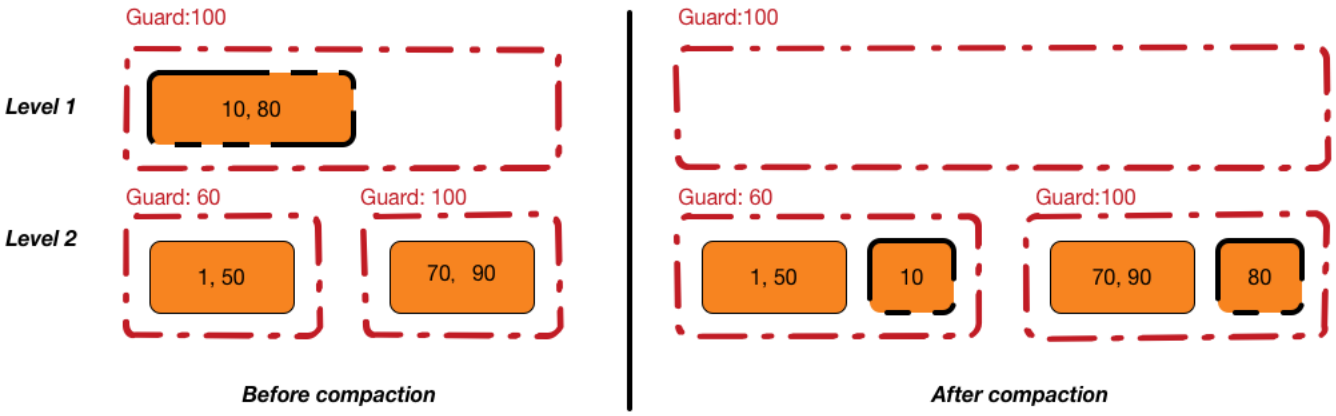

Figure 1: `PebblesDB` **Compaction Example: From Level 1 to Level 2**

Name	Description
fillseq	write N key-value pair in sequential key order
fillrandom	write N key-value pair in random key order
readseq	read N times sequentially
readrandom	read N times in random order
readreverse	read N times in reverse order
readmissing	read N missing keys in random order
readhot	read N times in random order from 1% section of DB
seekrandom	N random seeks
deleteseq	delete N keys in sequential order

Table 1: **Read/Write options in db_bench**

stead of the random update, which can reduce write amplification [6, 4] and improve the write performance.

To improve the read performance and make the system scalable to the large datasets, LSM-based storage system organizes their logs into levels. The first layer (numbered as 0), which contains `MemTable` is stored in memory. All the logs (`sstables`) in the rest levels are written to the persistent storage devices [3, 2]. Acceptable read performance is maintained by lowering down the number of logs. This is done by `compactions`, which merges the logs in the upper level into the lower level (i.e., merging sstables from level 1 to level 2). However, there is problem with the merging process as we need to read both logs from the upper level and lower level into memory and perform the merge. This mechanism naturally introduces the write amplifcation effect, which decreases the write performance. To combat this issue, Fragmented Log-Structured Merge Trees (FLSM) and its implementation `PebblesDB` are proposed [19]. The key idea is shown in Figure 1. FLSM uses `guard`, which can be thought of as the key for a collection of logs. the keys in the collection of logs have to be in the range between the current guard's key and the previous guard's key. During the compaction, the the log in the upper level is split across the guard keys in the lower level. In the example shown in Figure 1, 10 and 80 from level 1 is split to 10 in guard 60 and 80 to guard 100 in level 2. As one can see, this split and append avoids reading the data from lower level (e.g., level 2), which reduces the write amplification and improve the write performance. Read performance stays still thanks to the guard as we can first search guard key and then logs within the guard to locate the corresponding value for the given key.

2.2 Bloom Filters and Succinct Range Filter

Creating an LSM Tree involves the trade-off between read performance and some additional storage space for faster random inserts[18]. Sstables are distributed across different levels in disk. Querying a key have to cost many disk I/Os to sstables. Therefore, LSM based storage engiens create a Bloom filter[8] for each sstable to predict whether a given key is in the sstable. A Bloom filter is a probabilistic data structure which cost space efficiently. It allows LSM to avoid reading unnecessary sstables and reduce the read overhead. However, it cannot make a difference to range query tasks since it does not keep knowl-

edge of adjacent keys. When dealing with range query in PebblesDB, it must keep on seeking for the next key in a set of candidate sstables until the end of the given range, which takes significant memory cost and search complexity.

HyperPebbleDB manages to integrate start-of-art index structure Succinct Range Filter (SuRF)[23] as an alternative of the Bloom filter. The high level idea is to utilize a succinct trie to keep an approximate membership structure as well as the capability to traverse keys approximately. It requires to insert keys in order, which fit the property of sstables well. HyperPebbleDB can build up SuRF on sstable-level and achieve better query range performance.

2.3 MongoDB

MongoDB [16] is a NoSQL database, which can be used in both distributed and single instance settings. It can scale horizontally and achieve the high availability. In distributed setting, MongoDB is organized into clusters. There are two major components within a cluster: replica sets and sharding sets. Replica sets provide redundancy and high availability along with automatic failover. Sharding is used to partition the data across multiple machines. In this project, we use HyperPebblesDB as the storage engine for MongoDB to evaluate the HyperPebblesDB's performance in the distributed database setting.

3 IMPACT OF EMPTY GUARDS

Several features described in FLSM are not implemented in PebblesDB. One of them is guard deletion. Whether we implement the guard deletion feature in HyperPebblesDB largely depends on the impact of empty guards. That is, if the system performance (e.g., read throughput) is negatively correlated with the number of empty guards, then it is necessary to have the guard deletion feature implemented. Authors of PebblesDB carries out an empirical study of empty guard performance impact and claim that there is no significant performance drop given the existence of empty guards. However, there are several issues that are not fully addressed in their discussion:

- Only the read throughput for sequential write and read is reported. However, how the empty guards impact the sequential write, random write, and many other read/write patterns are not discussed.
- Read throughput is varied between 70 and 90 KOps/s in the PebblesDB. Whether this throughput is significant or not is un-addressed.
- The distribution of empty guards (e.g., number of empty guards, fraction of empty guards among total guards) in the experiment is not disclosed.

3.1 Experiment Setup

3.1.1 Experiment 1

We replicate PebblesDB's experiment setup in our empty guards investigation. The experiment procedure is following: for a given read/write pattern, we repeatedly execute the pattern for 20 iterations. At the end of each iteration, Similar to YCSB benchmark experiment setup on HyperDex [1, 9], we sleep 5 minutes to quiesce and we calculate the desired statistics. The list of read-/write patterns are shown in Table 2 and the description of each read/write pattern is detailed in Table 1.

For all the experiments in this section, unless specific noted, we have value size 512B, and data size: 504MB by default. We turn off snappy compression for fair comparison with PebblesDB. Between each iteration, we increment the base key by 10M. For example, given Pattern A, we do `fillseq,readseq,seekrandom,deleteseq`: we insert 10M key-value pairs (0 - 10M) in sequential key order, perform 5M sequential reads operations, 5M random seeks, and delete all keys. For the next iteration, we use the same pattern but with keys ranged from 10M to 20M. This is the same experiment as PebblesDB's. The motivation behind the experiment setup as stated in PebblesDB is to maximize the number of empty guards: in each iteration, empty guards are expected to accumulate due to the deletion of all the inserted keys.

3.1.2 Experiment 2

Experiment 2 follows the exact experimental procedure as the Experiment 1 with the same environmental setup. The only thing changed is the datasize, which scales from 504MB to 5GB. The motivation for this experiment is that datasize may have huge impact on the system performance [12, 7, 13]. In our scenario, 504MB may well be cached in memory and the cache hit ratio may be high. Thus, we scale up our datasize that is inporportional to the size of memory. With this setup, we want to see the impact of empty guards on the "big data" setting.

3.2 Implementation

We use db_bench, a native benchmark shipped with LevelDB family, to perform our experiment. However, to address the issues we pose ealier, we extend the db_bench as follows:

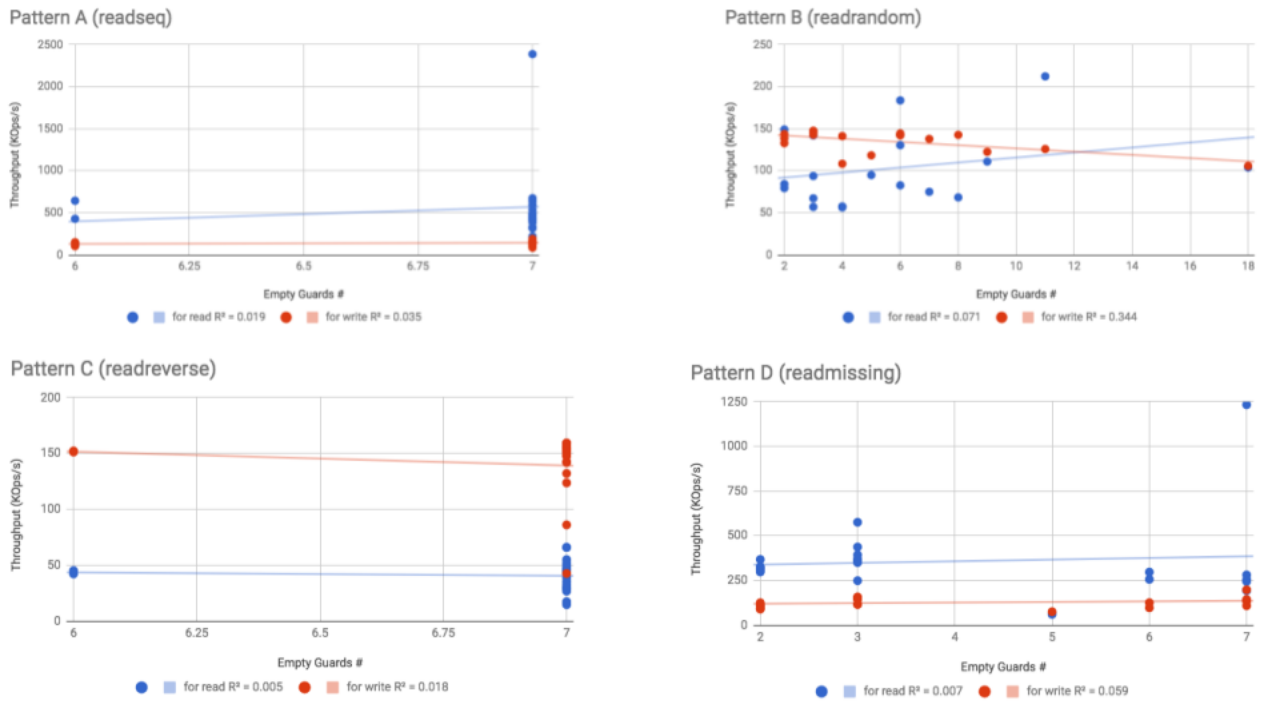

Figure 2: **Scatter plots for Patterns A,B,C,D in Experiment 1**

Read/Write Patterns	Description
A	fillseq,readseq,seekrandom,deleteseq
B	fillseq,readrandom,seekrandom,deleteseq
C	fillseq,readreverse,seekrandom,deleteseq
D	fillseq,readmissing,seekrandom,deleteseq
E	fillseq,readhot,seekrandom,deleteseq
F	fillseq,readseq,readrandom,readreverse,readmissing,readhot,seekrandom,deleteseq
G	fillrandom,readseq,readrandom,readreverse,readmissing,readhot,seekrandom,deleteseq

Table 2: **Read/Write Patterns for Empty Guards Experiment**

- `db_bench` shipped in `PebblesDB` does not allow the sequential write and random write on the existing database, which is crucial for our experiment as the motivation behind the experiment design is to accumulate as many guards as we can. Thus, we modify the benchmark to meet our goal.

- `db_bench` does not have a clear summary of the sstables and guards distribution. Specifically, the benchmark does not give the sstables per guard across all levels in an intuitive way. In addition, there is no built-in statistics calculation on the number of guards, average number of sstables per guards, number of empty guards, the fraction of empty guards, the variance and standard deviation of sstables per guard. Thus, we enhance `db_bench` to support those statistics calculation by introducing one extra knob `emptyGuards`.

3.3 Experiment Results & Analysis

We use Ubuntu 16.04 Server VM, 1GB RAM, 2 cores of i7 8700K with 50GB SSD to perform our experiments. The experiment result shown in Table 3. We list out the number of guards, number of empty guards, fraction of empty guards, read throughput, and write throughput in the first iteration, the last iteration, and on average. There are several observations we can make given the table:

Many iterations do not mean many empty guards. For example, for Pattern A, we have 7 empty guards at the very first iteration but the number stays the same at the end of 20th iteration. In other words, the empty guards does not accumulate linearly with the number of iteration we run with the experiment. This scenario is

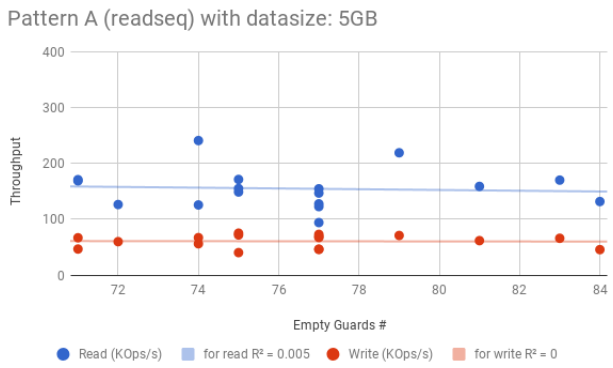

Figure 3: **Scatter plot for Pattern A in Experiment 2**

possible as the deletion is lazily-handled. When a key is deleted, a deletion tomb flag is appended to memtable. At this moment, the key is not actually deleted (i.e., occupied memory is freed); it only becomes inaccessible during the read. The actual deletion happens during the compaction. Thus, the number of empty guards depends on how aggressive we perform compaction. In addition, we observe the empty guards reuse during the experiment. In other words, even some empty guards appeared in the previous iteration, some of them may be reused by covering future inserted keys, which may even out the number of newly-created empty guards.

Empty guards impact is negligible to the Read-/Write performance. For example, in Pattern B, we have three empty guards in the first iteration and the 20th iteration. However, the Read throughput attains 143 KOps/s in the first iteration and degrades to 75 KOps/s in the last iteration. Given the relative stable experiment environment, the only change for this Pattern is the number of guards, which increases from 7 to 40. Thus, the number of guards that `HyperPebblesDB` has to search through impacts the read performance more significantly when compared with number of empty guards. To further verify this, we have Pattern F, which we have the same number of guards but the number of empty guards drops between two iterations (2 vs. 6). In this scenario, if we take a look at read throughput for *readrandom* ("random" in the table) and the sequential write throughput, we can see that the both write and read performance are actually dropped. Since the experiment is performed on SSD, we think that the performance degration might be due to the underlying SSD characteristics: good performance for instanteous workload but the performance may be dropped in the long-term (i.e., unsustainable) [10]. How can we separate the SSD impact from the `HyperPebblesDB`'s performance measurement is left for the future study.

To further examine the impact of empty guards, we also make scatter plots on both read and write throughputs against the number of empty guards for each iteration. In addition, we calculate the R-squared statistics. The results shown in Figure 7. As indicated by the R-squared statistic, the number of empty guards cannot explain fully about the variation in both read and write throughputs. However, for Pattern B (readrandom), the number of empty guards can seem explain the write performance decrease ($R^2 = 0.344$). However, we think that may be due to the effect of SSD, which needs further investigation. The observation from plots also hold for large data setting (see Figure 3). In fact, in large data setting, R-squared statistic becomes even lower, which attains 0.005 for Read and 0 for Write.

In conclusion, we think that the number of empty guards has limited impact on both read and write performance, which is consistent with `PebblesDB`'s result. Implementation-wise, since we can access the number of sstables covered under each guard (via. `GuardMetaData` structure), we can check whether the guard is a empty guard in constant time and no disk I/O is performed if the guard is empty. Thus, we think the overhead of existence of empty guards is small for both read and write. Furthermore, empty guard reuse amortizely decreases the space wasted by the empty guards. Therefore, we conclude that there is no reason to implement the guard deletion in `HyperPebblesDB` especially when the deletion may introduce additional overhead.

4 GUARD-BASED PARALLEL COMPACTION

According to the original `PebblesDB` paper, the author mentioned that the compaction of FLSM data structure is trivially optimizable. As indicated in the structure of the guard distribution, compaction of the sstable under a guard would only involves compacting the items of the same guard range at the next level. Therefore, compacting one guard never interferes with compacting another guard in the same level. However, the published code base of `PebblesDB` did not implement such feature because the author argue that the compaction time of the non-parallelized version is much better than its rivals like RocksDB.

In this project, we implemented this feature in our `PebblesDB` fork (`HyperPebblesDB`) to see how much performance increase it would bring to the system.

4.1 Implementation Detail

We forked `PebblesDB` to create a new version called `HyperPebblesDB` to implement the feature. The code base of the `PebblesDB` Implementation is complex with legacy contribution from `LevelDB` and `HyperlevelDB`.

Pattern		1st Iteration Guards #	Empty Guards #	Empty Guards %	Read	Write	20th Iteration Guards #	Empty Guards #	Empty Guards %	Read	Write	Overall Guards #	Empty Guards #	Read	Write
A		40	7	17.5%	**2387**	**190**	40	7	17.5%	673	150	40	6.85	543	141
B		7	3	43%	**143**	**187**	40	3	7.5%	75	138	38.35	5.35	103	138
C		40	7	17.5%	**2000**	**192**	40	7	17.5%	46	148	40	6.9	139	143
D		40	7	17.5%	**1233**	**198**	40	3	7.5%	370	152	40	4.5	359	126
E		40	7	17.5%	603	**180**	40	3	7.5%	**610**	153	40	5.2	555	138
F	seq				1642					**2907**				2755	
	random				**183**					50				58	
	reverse	30	6	20%	1434	**183**	30	2	6.7%	**2475**	124	30	2.2	2368	162
	missing				**739**					186				196	
	hot				78					**752**				706	
G	seq				**2488**					36				335	
	random				119					**159**				98	
	reverse	40	7	17.5%	**2584**	**187**	40	7	17.5%	47	143	49	6.95	267	156
	missing				**1748**					131				285	
	hot				**824**					775				767	

Table 3: **Empty Guards Experiment 1 Results. Read and Write are measured in KOps/s. The highlighted numbers are based on the comparison between 1st iteration and 20th iteration.**

Significant portion of our study was focused on understanding the structure of the codebase as most of the documentation are for API users rather than developers.

From what we found during the study, the major part of the code concerning compaction lies in **data_impl.cc** and **version_set.cc**. `PebblesDB` implements compaction in a separate thread that is started in the constructor of the class `DBImpl`. From the constructor, it invokes the 1 thread wrapper called `DBImpl::CompactMemTableWrapper` and multiple instances `DBImpl::CompactLevelWrapper`, The latter of which is of our concern in this project. All the wrapper does is just hold the mutex of the DB and initialize a file level bloom filter builder. During normal operation, the function then enter an infinite loop and checks whether the current version set of the database needs compaction. If not, the thread is set to wait for other event to wake it up.

The major compaction work lies in the method `DBImpl::BackgroundCompactionGuards`. This function takes in the file level bloom filter we just created then pick an level to compact by call the current version's `PickCompactionLevel`. Here the function will check whether the level to compile is the highest level to determine if it's a horizontal compaction. Once this is done, the function will use the function `PickCompactionForGuards` of current version set to create a `Compaction` object and fill in a list of complete guards that should be present after the compaction. Before the major work, the function also has to set the bit for the lock of the levels involved in the compaction. With the previous preparation done, the authors used a `CompactionState` object to describe the `Compaction` they just created and finally call the actual compaction algorithm `DoCompactionWorkGuards` with paramethers like the compaction state, the list of all guards to be used and the file level filter builder. One the compaction work is finished, this method logs the state and does some cleanup then return its upper caller with the status.

The `DoCompactionWorkGuards` is bit trickier to understand as it involves multiple modules of the system, and some of the modules are ill-documented. A blurry picture we have established by reading the code and the reference paper is as follow. At the beginning of the function, it does some sanity checks and initialize some variables. Then it call the current version set's `MakeInputIteratorForGuardsInALevel` to generate a iterator to go through all the data involving all the guards to compact in a sequential manner based on keys. In the main while loop, the method iterates through all the data entries, removing stale and deleted keys and checking the boundaries for the guards to move along. In the meanwhile, it also monitors the current output file size via the compaction states' builder variable and flush them to disk when the file size is appropriate. If the `FILE_LEVEL_FILTER` is turned on, the method also builds the file level bloom filter along the way.

4.2 Design Strategy

As mentioned above, the original implementation of `PebblesDB` is single threaded execution that does compaction whenever needed. Based on the principle of minimal alternation, we intend to keep this thread as the handler of compaction work and let it spawn multiple threads when encountered a compaction whose top level thread involves multiple guards. We created a parallel version of method `DBImpl::BackgroundCompactionGuards` that check such occasions and spawn one thread for each upper level guards. A more ideal design is to use a fixed-number thread pool and producer-consumer

model, since that would incur less overhead in race condition, thread creation and destruction. However, our empirical results shows that in a typical 7-level PebblesDB and reasonable size (10 - 20 million), the max number of guard of the compaction is 5. As most server-grade CPUs has 4 - 16 cores, Our current implementation would achieve good performance without the complexity. To reuse existing code, we still call the same DoCompactionWorkGuards under our DBImpl::BackgroundCompactionGuardsParallel and split the original guards and compaction state object in multiple by the upper level guards.

During our debugging process, we found the major problem is the handling of the mutex of DBImpl. The original design requires holding the lock during picking compaction level and gathering the guards and release the lock while actually running the compaction algorithm, then it reacquire the lock to actually install and write the output files. Such implementation assumes holding the mutex lock in the executing of DBImpl::BackgroundCompactionGuards and only releasing them in DoCompactionWorkGuards. To avoid deadlock while maintaining the original locking schema, we release the lock before spawning the threads and reacquire the lock after all threads are joined. We also added lock acquire and release lock in the lock in the individual thread wrapper that does the actual compaction work. In future version of the work, we intend to implement a thread-pool model to reach maximum compaction speed in a parallel setting.

4.3 Experiments and Evaluation

In this section, we measure the effect of introducing parallel guard-based compaction in two ways. In the coarse grained approach, we analyze how HyperPebblesDB perform with and without this feature under different workloads using db_bench. And in the fine grained approach, we logged each compaction time in the database log, and analyze how the compaction speed up with each compaction.

4.3.1 Coarse Grained Measurement

For hardware configuration, we used a desktop machine with Intel Core I7 8700K and 32GB RAM and ran the program on a 500GB SSD storage. We used Ubuntu 16.04 as our compiling and running program.

To measure the overall performance with the integration of parallel guards-based compaction, we used the db_bench utility that ships with HyperPebblesDB. As parallel guard based compaction can only be triggered on writes, we used two different write workloads to test it: fillseq and fillrandom. We ran each workload three times to compare the throughput as in Figure 5.

Number of Guards	Number of Compactions
0	361
1	22
2	5
3	3
5	3
6	6
7	1
8	1
9	2
10	1
15	1
61	1

Table 4: Compaction Guards Distribution

As we can see, this feature does not lead to performance improvement significantly as we supposed. The throughput on sequential write was actually smaller than non-parallel version while random writes won by a small margin. As we examine the result of the actual running process, we found that it is numbers fluctuate much and this does not lead to a convincing result that parallel compaction yields better performance. To further investigate this, we devised a fine grained experiment in the next section.

4.3.2 Fine Grained Measurement

To investigate how the parallel compaction did not lead to much higher increase in performance. We enabled detailed logging of each compaction process in the previous db_bench approach. Specifically, we logged each compaction's input files, upper level guards, compaction level, sum of input file sizes and actual running time.

Table 4 gives the counts of number of guards at the upper level in a total of 407 compaction. we shall see that 94% of compaction has 0 or 1 guards at the upper level, which means they are essentially still single-threaded executions. We expect to see more fraction of multiple guard compaction under bigger workloads. In the workload of 1 million keys, multi-threading does not show an significant improvement of performance because single threaded execution play a bigger role.

For each compaction job, we used the (sum of) size of input files processed divided by the compaction time to use as a notion of compaction speed. We then compared the average compaction speed grouped by the number of guards and scatter plot it in Figure 6. We shall see that although different data points represent significantly different number of compaction, the parallel version has apparently higher compaction speed. However, multiple-

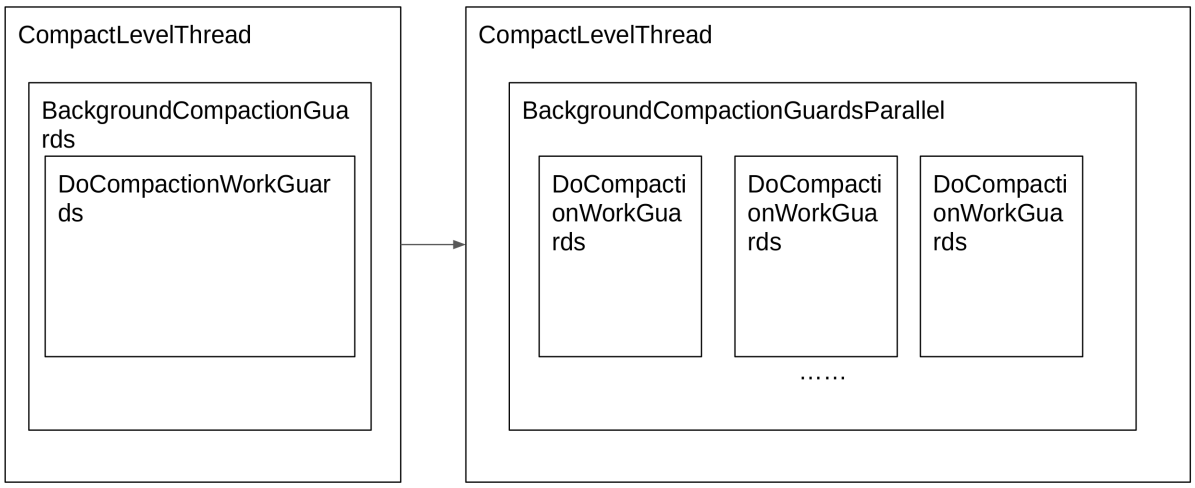

Figure 4: The transformation from single-threaded compaction to parallel compaction in function calls

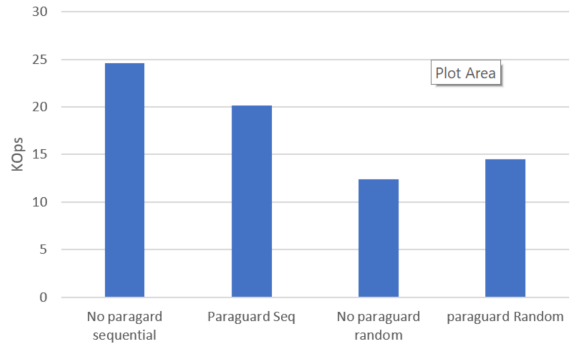

Figure 5: Comparison of Throughput with and without parallelism

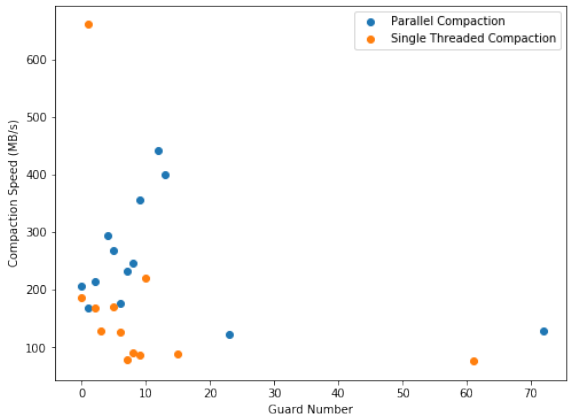

Figure 6: Compaction Speed with respect to number of guards

guard compaction instances are rare in general thus they do not contribute to a higher throughput in the previous experiment.

As a rough estimate, most parallel multi-guard compactions has twice the speed than single threaded version, and such ratio does not vary with guard size. We think that thread creation and destruction in each compaction is causing large amount of overhead in the process. By the way, we also noticed that the compaction at upper level contains guard size way larger than other levels, spawning threads proportional to the number of guards might not be a good approach. In our future work, we intend to refactor our parallel compaction routine to a fixed-size thread-pool implementation, which can avoid these caveats.

5 IMPROVING RANGE QUERY BASED ON SURF FILTER

Guard based design results in range query performance degrade [19] in `PebblesDB`. `PebblesDB` can return all key-value pairs falling within a given range for a range query request based on seek() and next(). However, the set of associated sstables in a guard does not follow a total sequence. It brings extra complexity to seek the next key among sstables. `HyperPebblesDB` integrates the state-of-art Succinct Range Filter (SuRF) [23] to simplify range query complexity by the capability to query the prefix as well as decrease space consumption.

Workload	Description
Load A	100% writes
A	50% reads, 50% writes
B	95% reads, 5% writes
C	100% reads
D	95% reads (latest value), 5% writes
Load E	100% writes
E	95% Range queries, 5% writes
F	50% reads, 50% Read-modify-writes

Table 5: **YSCB Workloads. The running order of YCSB workloads follows the exact ordering of the table**

5.1 Drawback of FLSM

The main design of `PebbleDB` is Fragmented Log-Structured Merge Trees(FLSM) compared to traditional LSM storage engine. FLSM is a trade-off between performance and write-IO. It weakens the sequence assumption in the traditional LSM and decreases IO effort to maintain the sequence. It may increase the complexity a bit to read due to the loose of sequence. Meanwhile, it also decreases the complexity of sorting keys among sstables. Thus, performance on some read tasks improves in `PebblesDB`. However, the performance on range query degrades obviously compared to LevelDB, RocksDB, etc[19].

Given a range (key1, key2), the range query returns all key-value pairs from key1 to key2. It is implemented by calling a iterator to do a seek() to key1 and do next() calls until key2. In traditional LSM based engine, keys have a strict sequence between sstables. Thus, it just requires to seek in order to find the next key for one sstable to a next sstable. FLSM weakens the sequence restrict as it guarantees strict sequence only between guards, which consists a set of associated sstables. When the iterator tries to seek the next key of key1, it not only have to seek in the local sstable of key1 but also have to seek other sstables in the same guard. It also have to sort these canditate keys and maintain these keys. Although `Pebbles` put a lot of effort to optimize range query, such as seek-based compaction and parallel seeks, there still 30% overhead of small range queries and 11% overhead of large range queries[19].

5.2 Bloom Filter and its limitation

It is challenging for LSM structure to query fast because keys can reside in SStables from all levels. Querying a key directly from a sstable may incur multiple disk I/Os. Bloom filter is a approximate membership data structure which is small enough to reside in memory. It has one-sided errors. If the key exists, the bloom filter return true and query in the sstable; If the key is absent, the bloom filter will return false in a high possibility. Bloom filter trades off between space efficiency and false positive rate. It requires only 10 bits for each key along with around 1% false positive rate commonly[23].

However, bloom filter can not provide any assist to the range query in LSM based storage engines. Given a range query request from key1 to key2, the engine must read related sstables in all levels and implement merge sort on these sstables. It become even worse in FLSM since FLSM stores have to read more sstables on per level. bloom filter can just predict whether the key is in or not, it cannot predict the next key's approximate value. There are some extensions of bloom filter such as prefix bloom filter which can optimize certain fixed-prefix queries, despite they are inflexible for more general range queries[19].

5.3 Succinct Range Filter

`HyperPebblesDB` integrates the state-of-art uncompressed index structue Succinct Range Filter (SuRF) []. SuRF also guarantees one-sides errors for point query and range query basically. Besides, SuRF builds upon an advance trie and supports level cursor to move sequentially in itself for range queries. Therefore, SuRF assists range queries between sstables since it maintains additional key sequence knowledge.

SuRF builds up on a space-efficient data structure called the Fast succinct Trie(FST), which is an extension of classical Level-Ordered Unary Degree Sequence(LOUDS)[11]. FST consumes few bits per node which is close to information-theoretic lower bound. FST designs prefix level of a key as a dense LOUDS. It records the prefix precisely. Meanwhile, FST designs the suffix of a key as sparse LOUDS. It maps this part into a simple data bits. Therefore, if a key is inserted into FST, its prefix is stored in LOUDS-Dense and its suffix is mapped in LOUDS-Sparse. A false positive case can only happens when prefix is equal and suffix maps to a same value in a low possibility.

Intuitively, FST keeps the property of trie and ordinal tree to traverse keys. It encodes keys in a sstable efficiently and uses the rank & select primitives. When the storage tries to a range query by seeking the next key, it can utilize a cursor in FST to move to next key efficiently. The engine can search and sort keys in SuRFs for sstables from different level and choose much fewer candidate keys as well as fewer disk I/Os.

The improvement can be more significant for FLSM after it integrates with SuRF as an alternative of Bloom filter. FLSM will involve much more sstables from all levels and its weakened key sequence causes it harder to merge sort among these associated sstables. With the

assist of SuRF of each sstables, the storage engine can preprocess on filters and degrade search space in sstables.

5.4 Evaluation

Zhang[23] has present various micro benchmark on false positive rate, performance and space of SuRF.In this paper, we focus more on the performance and space in HyperPebblesDB comparing to PebblesDB. The experiment is set up on the server with Inter Core i7-8700k and 32 GB RAM. We mainly use benchmark in db_bench and include the performance on sequential write and a several different read task. We insert 10 million key-value pairs and execute 4 million reads. Due to we does not apply an isolated environment, there is always a bit variance for each patch of experiment. Therefore, we run experiments more than 10 rounds and take the mean value of them after discarding outliers. In all, HyperPebblesDB achievement a % improvement for range query and with a bit trade off on some other tasks.

Table 6 presents the comparison of micro benchmark on range query, read and write of HyperPebblesDB and PebblesDB.Range query which is corresponding to random scan improves 40% compared to PebblesDB while sequential read degrades 8.8%. It indicates that the gain of range query does not come from potential improvement on read performance. It benefits more from the in-memory search of SuRF. Besides, sequential write of HyperPebblesDB downgrade a bit compared to original PebblesDB with Bloom filter, which indicts that constructing a SuRF costs a bit more complexity.

However, HyperPebblesDB does not have good performance on space usage. It consumes comparable and even more memory than PebblesDB. There should still be some implementation bugs for HyperPebblesDB and SuRF. We have even fixed serious memory leak of SuRF before. There may be some other implicit errors inside SuRF and between the interference of SuRF and HyperPebblesDB. In the future work, we will fix all the errors and make the performance more stable.

6 EVALUATION

In previous sections, we implement parallel guard-based compaction to improve read-throughput and used SuRF to improve the range query performance in HyperPebblesDB. In this section, we used the industry standard Yahoo! Cloud Serving Benchmark to evaluate the performance of HyperPebblesDB. We used HyperPebblesDB as the engine of MongoDB to used as a driver of YCSB. As we didn't touch the entire architecture of Pebbles in our experiment, mongo-pebbles works out of box for our HyperPebblesDB development. We ran the benchmark along with other well-known KV stores such as RocksDB, WiredTiger as well as our prototype HyperPebblesDB.

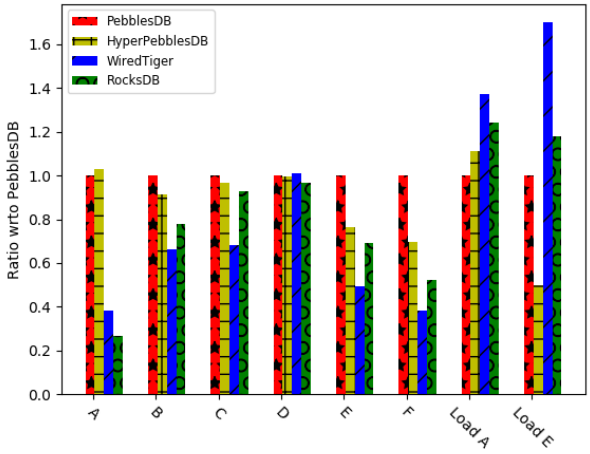

Figure 7: **YCSB Benchmark Performance of MongoDB when using different key-value stores as the storage engine. For convenience, the throughtput is presented in a relative value to** PebblesDB

6.1 Experiment Setup

We used a Game Laptop with 256GB SSD to run our experiment. It comes with Intel Core i7-6700HQ and 8 GB RAM. We chose such configurations because smaller RAM would incur more disk read/write in the same workload, we can thus mimic the huge industry workload with fewer running time and small amount of disk space. We utilize MongoDB[16] to integrate HyperPebblesDB as the storage engine. MongoDB is a widely-used NoSQL store. It supports a single instance as well as distributed instances. However, we have not considered experiments on distributed instances for two issues. First, there are some unexpected difficulties on configure MongoDB with non-default storage engine, which is time consuming for out limited project cycle. Second, we find that network latency and scheduling is the dominate factor for the throughput of distributed nodes according to empirical knowledge from our prior project on distributed key-value store, which is not the emphasis of our modification and enhancement on HyperPebblesDB. From the view of this paper, The single instance can bring enough information of latest design on HyperPebblesDB. Finally, we experiment on MongoDB with four storage engines: HyperPebblesDB, PebblesDB, the default WiredTiger[5] and RocksDB. WE set up the record count as 5M and operations count as 1M for each workload. Besides, there will be a 300

Task	HyperPebblesDB(KOps/s)	PebblesDB(KOps/s)	Improvement
scan random	108.12	77.17	40.1%
fill seq	20.98	22.4	-6.3%
read seq	1658.37	1818.18	-8.8%
read reverse	1600	1686.34	-5.1%
read random	206.87	168.66	22.66%

Table 6: **Range Query/Read/Write comparision on HyperPebblesDB and PebblesDB**

seconds sleep for each run.

6.2 Experiment Result

Figure 7 shows different throughput of workload A-F. `HyperPebblesDB` and `PebblesDB` has similar performance in most tasks although there are some weird throughput such as Load A and Load E. In a general view, they both outperform `WiredTiger` and `RocksDB` except full write task. However, `HyperPebblesDB` does not perform better than `PebblesDB`, especially on workload E which is all range query tasks. This is not so convincing that `PebblesDB` even performs much better than `RocksDB` on range query, which is opposite to the conclusion in prior work[19]. We investigated why these suspicious throughput result in our set. It mainly results from both hardware side and software side. The laptop is stale and the SSD in it has degraded during these years. This hardware environment may cause fluctuations for each run. From the software side, there are a considerable meta data and files generated which results a negative effect on the next run. The total SSD space has even been consumed up in some rounds, which reflects that there are some implicit errors in the storage engine. In all, most of the throughput results are reasonable. We can develop a more stable version as well as provide a better isolated hardware platform for better benchmark of these storage engines.

7 FUTURE WORK

Impact of Empty Guards. Even though we have performed careful empirical study on the impact of empty guards. There are still some work left to be done in the future. For example, `db_bench` can be further improved to incorporate the detailed statistics on read amplification, write amplification, and space amplification, which is similar to RocksDB. In addition, for Pattern A, C, D, we observe high read performance in the first iteration and low read performance after, which might be coupled with the instantaneous performance and long-term performance of SSD [10] and the interaction between key-value store and underlying file system [14]. How we can improve `HyperPebblesDB` to have more steady long-term performance on SSDs is important for further investigation. Furthermore, our experiment is still not fully-controllable since we cannot predetermine how many empty guards we want to have at the end of each iteration. We think `manual_compaction` might be helpful for this, which we are left out for future implementation. All those investigations are tightly linked with the accurate measurement of overall system performance. In our experiment 2, we have not measured the cache hit ratio and other system statistics and thus we need to perform more detailed experiment environment measurement and possible statistic study on the relationship between read-/write performance and different variables in our future study.

Parallel Guard-based Compaction In our current implementation, we have successfully added the feature of parallel guard-based compaction, however it is not without limitations. During our experiment and evaluation, we found that the performance improvement is not consistent and apparent under all write workloads, and we further attribute it to the small fractions of multiple-guard compaction. It remains to test it against much bigger workload to demonstrate the performance improvement of this feature as bigger workloads would trigger more multiple guard compaction.

In our fine grained analysis, we did found our implementation boosted the compaction speed in the cases of multiple guard compaction. We found the parallel efficiency is not great (averages to 2 on a 6 core machine) due to overhead of spawning threads and thread destruction. Therefore we also intend to refactor the code to use a fixed size thread pool to do all compaction jobs.

Guard Level SuRF Filter `HyperPebblesDB` implements SuRF for each sstable. We consider to integrate SuRF for each guard. One main issue is that SuRF construction requires providing a sorted key list. It is possible to extend SuRF to accept out-of-order keys, which will bring huge complexity in construction. Since sstables are sorted when they are built up, it is naturally to construct SuRFs at the same time. However, sstables residing in a guards are not in a totally order and they will dynamic changes for each compaction. Therefore building a SuRF for each guard requires a bit extra effort. In another side, it can benefit range query by a better in-

dex to seek the next key as well. The sequence between guards are strictly guaranteed. We can open a set of guard SuRFs and operate merge sort among them. There are fewer SuRFs as well as fewer keys since duplicate keys among sstables are unique in guard SuRFs. Then we can search sstables of the candidate guards following the hierarchy structure efficiently. We can explore the potential of guard level SuRF in the future.

8 CONCLUSION

We build a key-value store called HyperPebblesDB based on PebblesDB [19]. We empirically investigate the impact of empty guards and show that guard deletion is uncritical to the HyperPebblesDB's performance. We also implement the guard-based parallel compaction and we attain better performance compared to PebblesDB when there are a large amount of guards in the compaction level. Furthermore, by integrating the SuRF into HyperPebblesDB, we have a better performance in range query but with the cost of increased memory usage and a slight write degration. We test out our implementation on MongoDB and we performance improvement in workload A of the YCSB benchmark.

9 ACKNOWLEDGEMENTS

We thank Vijay Chidambaram for his helpful discussion on the potential improvements to PebblesDB and the preliminary experiment results of HyperPebblesDB.

10 AVAILABILITY

We have made the source code for HyperPebblesDB available at the following URL: https://github.com/xxks-kkk/HyperPebblesDB.

References

[1] Hyperdex benchmark setup. http://hyperdex.org/performance/setup/, 2018.

[2] LevelDB. https://github.com/google/leveldb, 2018.

[3] RocksDB. https://github.com/facebook/rocksdb, 2018.

[4] RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide, 2018.

[5] Wiredtiger. http://www.wiredtiger.com//, 2018.

[6] ATHANASSOULIS, M., KESTER, M. S., MAAS, L. M., STOICA, R., IDREOS, S., AILAMAKI, A., AND CALLAGHAN, M. Designing access methods: The rum conjecture. In *EDBT* (2016), vol. 2016, pp. 461–466.

[7] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review* (2012), vol. 40, ACM, pp. 53–64.

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

[9] ESCRIVA, R., WONG, B., AND SIRER, E. G. Hyperdex: A distributed, searchable key-value store. *SIGCOMM Comput. Commun. Rev. 42*, 4 (Aug. 2012), 25–36.

[10] HE, J., KANNAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. The unwritten contract of solid state drives. In *Proceedings of the Twelfth European Conference on Computer Systems* (2017), ACM, pp. 127–144.

[11] JACOBSON, G. Space-efficient static trees and graphs. In *Foundations of Computer Science, 1989., 30th Annual Symposium on* (1989), IEEE, pp. 549–554.

[12] JIN, X., LI, X., ZHANG, H., SOULÉ, R., LEE, J., FOSTER, N., KIM, C., AND STOICA, I. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), ACM, pp. 121–136.

[13] LU, L., PILLAI, T. S., GOPALAKRISHNAN, H., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS) 13*, 1 (2017), 5.

[14] MEI, F., CAO, Q., JIANG, H., AND TINTRI, L. T. Lsm-tree managed storage for large-scale key-value store. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), ACM, pp. 142–156.

[15] MERRITT, A., GAVRILOVSKA, A., CHEN, Y., AND MILOJICIC, D. Concurrent log-structured memory for many-core key-value stores. *Proceedings of the VLDB Endowment 11*, 4 (2017), 458–471.

[16] MONGODB. Mongodb. https://www.mongodb.com/, 2018.

[17] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The log-structured merge-tree (lsm-tree). *Acta Inf. 33*, 4 (June 1996), 351–385.

[18] ONEIL, P., CHENG, E., GAWLICK, D., AND ONEIL, E. The log-structured merge-tree (lsm-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[19] RAJU, P., KADEKODI, R., CHIDAMBARAM, V., AND ABRAHAM, I. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)* (Shanghai, China, October 2017).

[20] REN, K., ZHENG, Q., ARULRAJ, J., AND GIBSON, G. Slimdb: a space-efficient key-value storage engine for semi-sorted data. *Proceedings of the VLDB Endowment 10*, 13 (2017), 2037–2048.

[21] SEARS, R., AND RAMAKRISHNAN, R. blsm: a general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), ACM, pp. 217–228.

[22] TAN, W., TATA, S., TANG, Y., AND FONG, L. L. Diff-index: Differentiated index in distributed log-structured data stores. In *EDBT* (2014), pp. 700–711.

[23] ZHANG, H., LIM, H., LEIS, V., ANDERSEN, D. G., KAMINSKY, M., KEETON, K., AND PAVLO, A. Surf: Practical range query filtering with fast succinct tries. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data* (2018).