Data Structures To Maintain Hot Data For RocksDB

Tanuj Gupta [1]

[1]*Graduate Student, Department of Computer Engineering, UCSC*

----------***----------

Abstract-

Rocks DB is an embedded, high-performance, persistent key-value storage engine developed at Facebook. It achieves an increase in transaction throughput and a significant decrease in write amplification, yet increases average read latencies by a marginal amount. Rocks DB achieves this by favoring recent writers against recent reads. The keys that have not been written/updated for a while will keep getting pushed to lower levels. This causes read scan latency to increase for important keys. Definitely caching helps address this problem to some extent, but the cache is small, and data can be very large.

In this paper, we explore the limitations of current approaches taken by RocksDB like caching to address the issue. We finally give a probabilistic constant time solution to solve this problem at the expense of extra CPU operations. We then explore how to make sure these extra CPU tasks do not interfere with other operations. The idea is to select the data that has a tendency of being read a lot, the "HOT DATA", and push them to higher levels.

Key Words: RocksDb, Embedded Databases, Read latency, Read amplification, Write amplification, LSM Tree, Bloom filters…

1. INTRODUCTION

Rocks DB is configured to give priority to write amplification, write throughput, and resource efficiency with a trade-off with read latency and amplification, as long as the latter remain acceptable. In particular, Rocks DB optimizes space/write efficiency while ensuring read latencies meet service-level requirements for the intended workloads. Rocks DB uses log-structured merge trees(LSM) to organize the data. An LSM-tree is typically far more space-efficient than a B-tree. LSM tree also provides way better write throughput in comparison to B-Tree-based databases. Rocks DB further uses compaction, compression to better optimize space and read amplification. The downside of LSM structures is that they have poor read latencies when compared to B+ Trees. Although Rocks DB has taken various measures to improve the read latencies and provide good average-case performance, tail latencies can be worse. This problem can exacerbate if data is huge and workload is read-intensive. For all the reads at lower levels, we incur slow response. The paper presents a solution to address this very problem in LSM based level order databases, in particular, the RocksDB. Before we

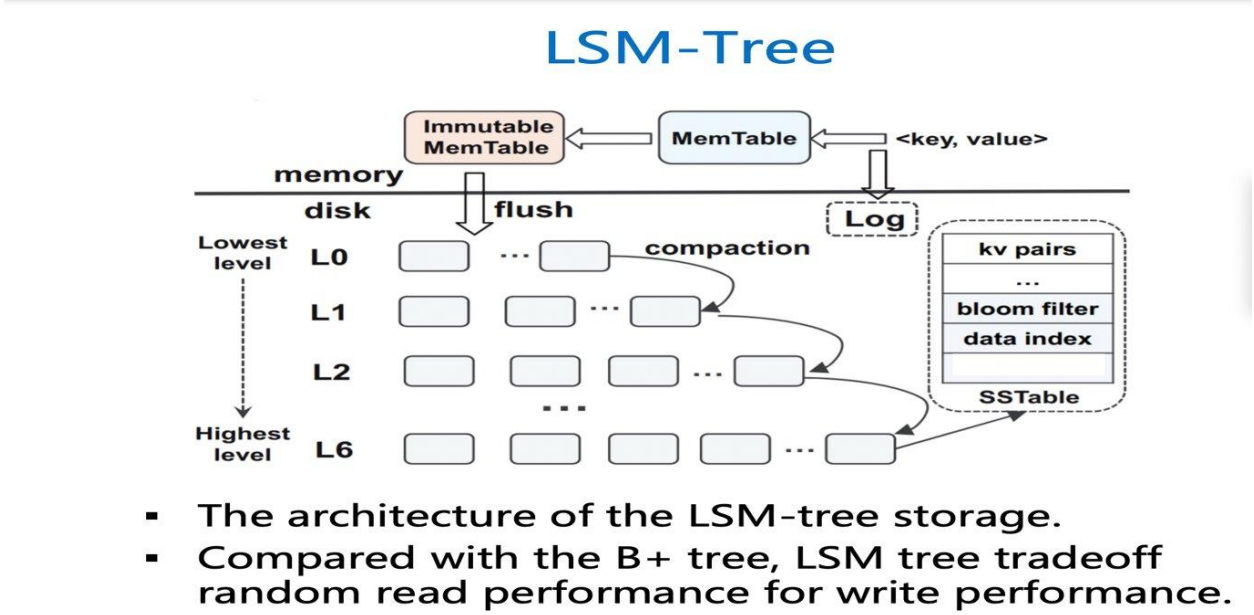
deep dive into our solution, we need to understand the basics of Rocks DB, and steps taken by Rocks-DB to address the problem of low read latencies.

2. BACKGROUND -> ROCKSDB

2.1 HIGH-LEVEL ARCHITECTURE

RocksDB is an embedded, high-performance, persistent key-value storage engine developed at Facebook. It is a storage engine library of key-value store interfaces where keys and values are arbitrary byte streams. Inspired by level Db, RocksDB uses log-structured merge trees(LSM) to organize the data. Data is organized in levels. In each level, entries are organized into SST (Sorted String Table) files. SST file is a big sorted array of key-value pairs. In each level except 0, SST files are sorted by keys.

The common operations provided by RocksDB are Get(key), NewIterator(), Put(key, Val), Delete(key), and SingleDelete(key).



- The architecture of the LSM-tree storage.
- Compared with the B+ tree, LSM tree tradeoff random read performance for write performance.

2.2 WRITE AMPLIFICATION, THROUGHPUT, AND SPACE AMPLIFICATION

In RocksDb, all writes are append-only. Every write is assigned a globally increasing sequence number to distinguish it. All the entries are appended to a Memtable first and also written in WAL (Write Ahead Log) for fault tolerance. When the Memtable is large enough, it will be flushed to a new SST in Level-0. Append only characteristic

significantly increasing transaction throughput.

RocksDB cannot keep all the entries forever. It needs to compact and drop old entries when necessary, and SST files will be compacted into different levels later. An LSM-tree provides much-optimized space and write amplification in comparison to B-Tree. RocksDB further uses compaction, compression to better optimize space and read amplification.

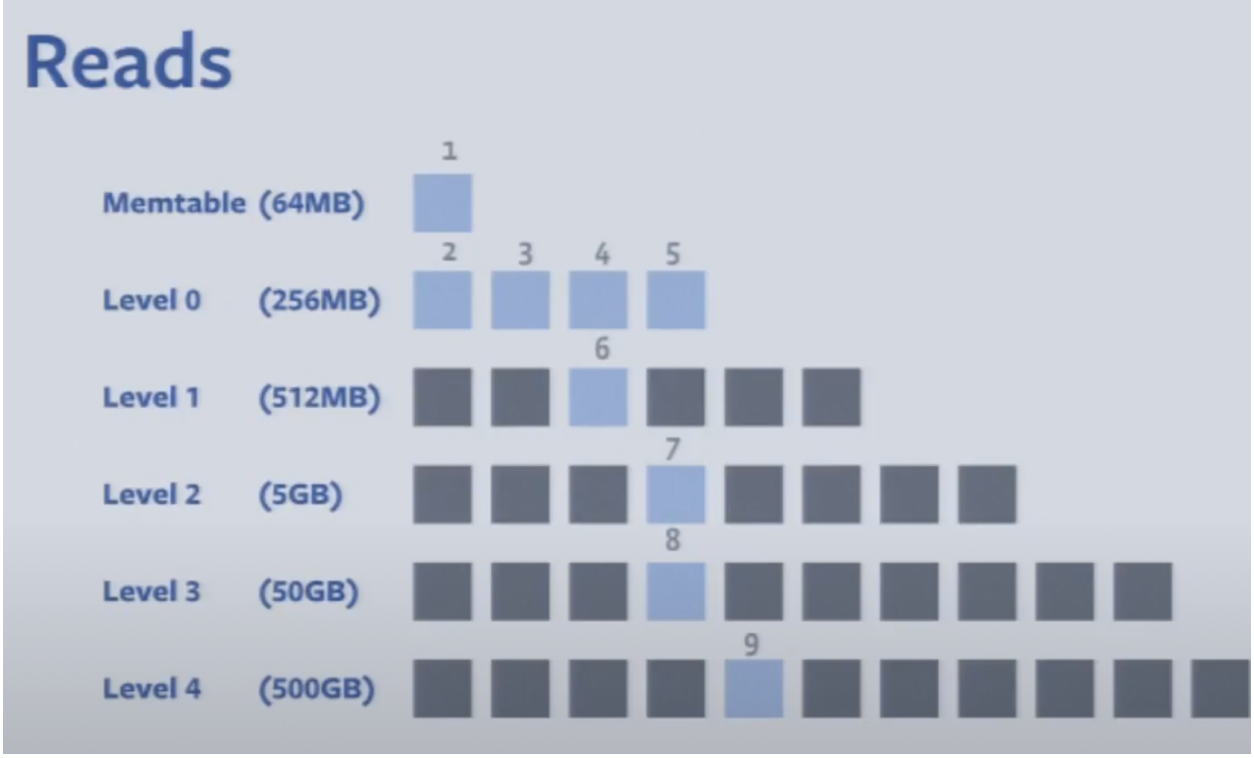
2.3 READ/GET OPERATION IN ROCKSDB

To address the problem of low read latency in RocksDB, we need to understand how does a read work in RocksDB? What happens when a read query is run in RocksDB?

As mentioned earlier, a key in RocksDB is written to memtable, which is flushed to level 0, and compacted to lower levels later. The database can have multiple copies of a key at various levels at any time. But, the latest write/update of the key lies at the highest level. So to find this latest copy of a key, RocksDB makes a linear scan from memtable to level 0 to lower levels, until we find a key. This first hit is the latest copy of the key. In the worst case, this will need a full scan. For databases with as small as 1GB of data, a single read will take a lifetime. Obviously, it is not as bad as it sounds.

To begin with, SST files in all the levels are sorted by the keys. So, one can find relevant SST files per level in O(log(n)) time where n is the number of SST files in that level. Level 0 is not sorted, and thus needs O(n) time to find the relevant SST files. But linear scan at level 0 is adjustable because it generally has very little data that is compacted to lower levels periodically. Also, the keys in every SST are divided into several blocks. Each SST has an index block to help search for the relevant block using binary search. Although this is way better than a linear scan of all data, we still incur a lot of disk I/O at every level. This is extremely costly if we have a read-heavy workload.

RocksDB addresses these issues using caches, bloom filters, and data block hash index. A brief description and the limitations for these approaches are discussed in the next section.

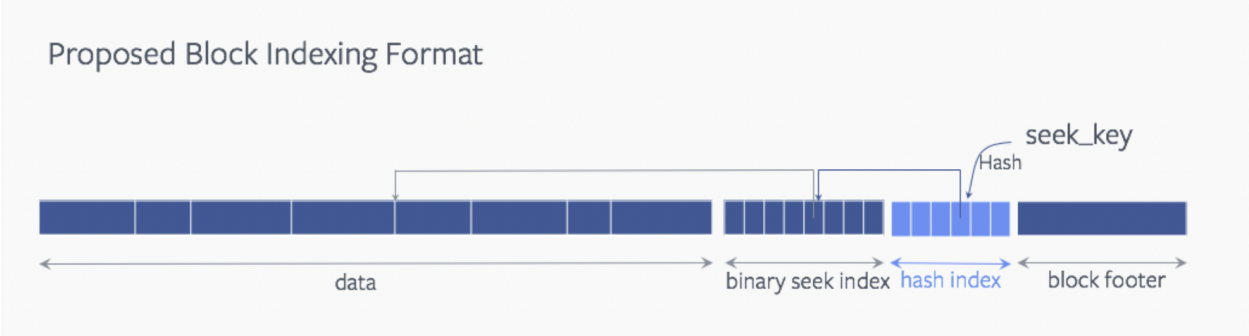


2.4 ROCKSDB APPROACH TO ADDRESS SLOW READS

2.4.1 Data Block Hash Index

Index block help optimize the searching for the data block in an SST. To find the right location where the key may reside using binary search, multiple key parsing and comparison are needed. Each binary search branching triggers CPU cache miss, causing higher CPU utilization. RocksDB has designed and implemented a data block hash index that has the benefit of both reducing the CPU utilization and increasing the throughput for point lookup queries with a reasonable and tunable space overhead.

Specifically, RocksDB appends a compact hash table in every SST for efficient indexing. It is backward compatible with databases created without this feature. After turning on the hash index feature, existing data will be gradually converted to the hash index format. Obviously, some keys will have collisions, and hence the hash index cannot confirm the presence of all the keys in SST. If the key is not found using the hash index block, we will use the binary search over the index block as usual.



2.4.2 RocksDB Bloom Filters

In RocksDB, every newly created SST file will contain a Bloom filter, which is used to determine if the file may contain the key we're looking for. The filter is essentially a bit array. For any arbitrary set of keys, an algorithm may be applied to create a bit array called a Bloom filter. Given an arbitrary key, this bit array may be used to determine if the key may exist or definitely does not exist in the key set. Multiple hash functions are applied to the given key, each specifying a bit in the array that will be set to 1. At read time also the same hash functions are applied on the search key, the bits are checked, i.e., probe, and the key definitely does not exist if at least one of the probes return 0.

Bloom filter has 100% true negative and helps avoid reads of irrelevant SST files. This significantly reduces Disk I/O and improves performance.

2.4.3 RocksDB Caching

RocksDB uses two implementations of cache, namely LRUCache and ClockCache. Both types of cache are sharded to mitigate lock contention. Capacity is divided evenly to each shard and shards don't share capacity. By default, each cache will be sharded into at most 64 shards, with each shard having no less than 512k bytes of capacity.

By default, index block and bloom filter are cached outside of block cache, and users won't be able to control how much memory should be used to cache these blocks, other than setting max_open_files. Users can opt to cache index and filter blocks in block cache, which allows for better control of memory used by RocksDB. To cache index block and bloom filter dictionary blocks in block cache:

3 LIMITATIONS OF CURRENT APPROACHES

Even though, the bloom filter helps to avoid disk read for both data blocks and index blocks in SST that does not have the required key, we still have to read bloom filter block once per level. This is a non-trivial cost for keys that lie at lower levels. Every read for keys at lower levels incurs a lot of memory/disk I/O. Also, as the data grows, the

probability of a key to be found at lower level increases as well. Given that most of the data for any LSM based database lie at the lower levels, makes things even worse.

One solution to this problem is caching which significantly improves the read latency of important keys. But the cache is generally very small and only helps prioritize a small subset of these keys. On the contrary, the database can have a lot of keys at lower levels. Hierarchical caching can be a solution to address this problem. But again, it would not be sufficient if a lot of keys at lower levels are being read again and again. Also, the multiple hierarchies of caches would incur high cost and space.

3.1 UNDERLYING PROBLEM

To find a better solution, we first need to establish the underlying problem that the hash index, bloom filters, or caching could not solve.

Any key at a lower level is going to incur huge read latency in LSM tree-based data structure, whatever be the case. But, if the key has a tendency to be updated soon enough, this key will be written to a higher level with an updated value. These keys will incur high latency only for a short duration(when they are in lower levels) until they are updated and written to higher levels again. Also, if some key at a lower level is not read at all, we don't care about its low read latency anyway.

So, the data of concern,i.e data incurring high read latencies has two properties here:
1) The time between the updates of the key is very large. Because of this, the key stays at a lower level at adds to high read latency.
2) The key is read frequently. This makes the key valuable and important, something we can't afford to have a slow response on.
We call such keys **Hot Data** because this is the critical data we need to give priority to in some way.

One of the most natural solutions to address the high read latency of this hot data is to manually write/move it to higher levels. This is the very solution we present in the paper as well. But biggest challenges include determining:
a) When and how to mark a key as hot data,
b) What data structures to use to store hot data,
c) When to manually write/push the hot data on some higher level,
d) How to deal with the relationship of "hot data" and new values.
We will explore these questions in detail in the next sections.

4 WHEN AND HOW TO MARK A KEY AS HOT DATA

Any key/data which has a high probability of incurring high read latency is the hot data, the data of concern**.** In the previous section, we noted the following properties that define hot data. To reiterate, these properties are:

1) The time between updates of the key is very large. Because of this, the key stays at a lower level at incurs high read latency.
2) The key is read frequently. This makes the key valuable and important.

To be able to mark any data as hot, we need to capture these properties. To capture the second property, we need a way to count the number of reads for every key. To capture the first property, we need to count the average interval between updates of a key. This capturing of properties demands some extra operation both while writing and reading a key. We cannot afford to slow writes in RocksDB, because RocksDB promises very high write throughput to its customers. In other words, we don't want to slow down writes to achieve better read latency. So we need a better way to capture both the properties without affecting writes.

We can say, if the latest update of a key lies below a certain level, the key has not been updated for a while. This was the very essence that the 1st property captured. Basically, if some key lies at a level below some threshold level(Lt), we can say that the key has not been updated for a while. So if we only measure the number of reads for a key below this threshold level(Lt), we will capture both properties 1 and 2 in one go. Also, now we do not need to track counts for the keys at higher levels.

So let's define a threshold level Lt. On every read, if the key is found below level Lt, we will increment its count. Now the question is, where to store these counts? What is the most optimized data structure we can use to achieve our goals? Now we will explore these questions. But before we answer these questions, let's understand Probabilistic data structures. We will be using bloom filters and cuckoo hashing as a part of our solution all along. So, before we deep dive into our solution and approach, let's understand what these structures are.

4.1 PROBABILISTIC DATA STRUCTURES

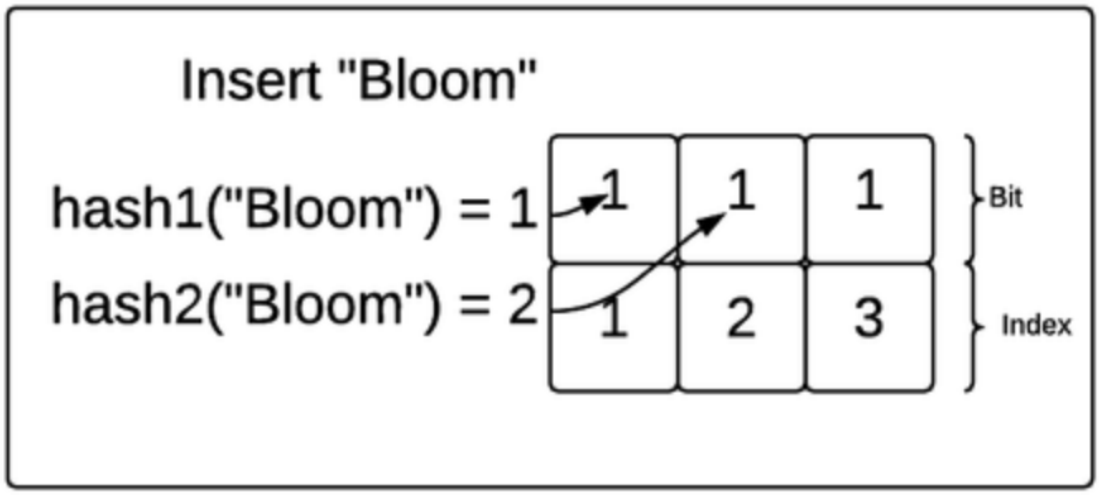
Probabilistic data structures are very useful, especially when processing large data sets. Most of the time, whilst working on the data side of things, one would want to do a simple “is the item not present” or “is the item already present” query whilst processing the real-time data. The typical approach to such queries would be to use either a HashMap or a HashTable, or store it in some external cache (like Redis), but the problem is with large datasets, these simple data structures can’t fit into memory. This is where probabilistic data structures come into play because of their space and time advantages.

4.1.1 Bloom Filter

A bloom filter is a probabilistic data structure that is based on hashing. It is extremely space-efficient and is typically used to add elements to a set and test if an element is in a set. Though, the elements themselves are not added to a set. Instead, a hash of the elements is added to the set.

When testing if an element is in the bloom filter, false positives are possible. It will either say that an element is *definitely not* in the set or that *it is possible* the element is in the set. A bloom filter is very much like a hash table in that it will use a hash function to map a key to a bucket. However, it will not store that key in that bucket, it will simply mark it as filled. So, many keys might map to the same filled bucket, creating false positives.

False-positive probability: $(1-e^{-kn/m})^k$.
False-negative probability: 0

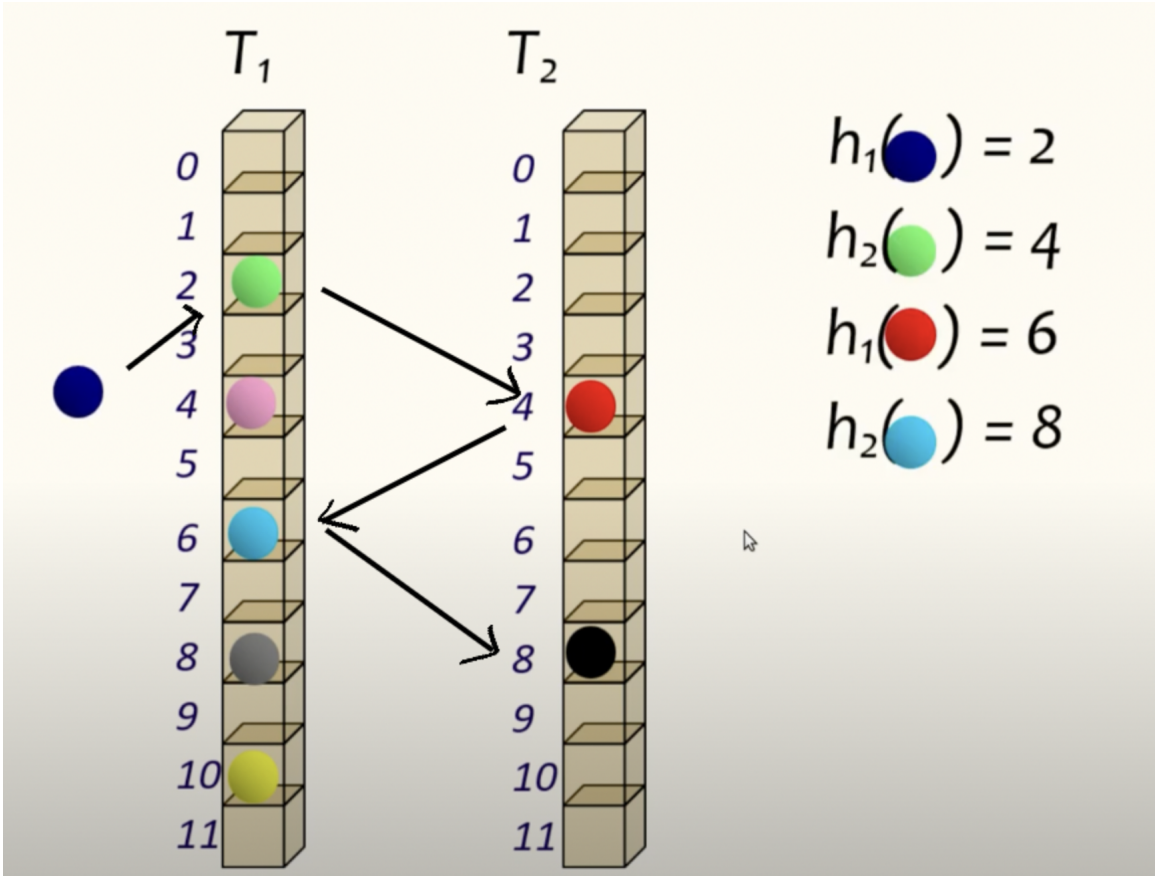


Bloom Filter

4.1.2 Cuckoo filters

Cuckoo filters improve upon the design of the bloom filter by offering deletion, limited counting, and a bounded false positive probability, while still maintaining a similar space complexity. They use cuckoo hashing to resolve collisions and are essentially a compact cuckoo hash table.

Cuckoo and bloom filters are both useful for set membership testing when the size of the original data is large. They both only use 7 bits per entry. They are also useful when an expensive operation can be avoided prior to execution by a set membership test. For example, before querying a database, a set membership test can be done to see if the desired object is even in the database.



4.2 DATA STRUCTURE TO TRACK READ COUNT

To start with, we need some data structure that fulfills three goals in an optimized way. The first goal is to store the count for all the hotkeys, the second is to get these counts against a key, and the third is to update the count. These get and update will be triggered every time we read some key at level>Lt. Since the read operation is being impacted here, we definitely need a data structure that can help us get and update the count in the minimum time possible. Since we are tracking only counts, and no sort of order is required, a hash map is the most natural choice here. The hash map has constant time update and lookup which fits our requirements perfectly.

The second thing to note is that all the keys below a threshold level(Lt) are the possible candidates for hotkeys. So, we need to store the read count for all these keys. We know RocksDB has almost all the data at lower levels. So tracking the count of all the keys below this threshold level(Lt) would cost us huge space. In a smaller hashmap, this means “a large number of collisions”. This is where probabilistic data structures and probabilistic hashmaps come into the picture, namely bloom filters and cuckoo hashing.

We will use these structures to say with some probability that this can be a possible hotkey. We here trade off a small chance of the wrong prediction with huge space. In other words, we trade-off *exactness* for efficiency. So we will use the counting bloom

filter for tracking the count of keys at level>Lt.

4.2.1 Wrong prediction: False positive in Bloom Filters

We know that the bloom filter has 0 false negative probability. So it is mainly used for membership checks in **constant space** and **time**. Bloom filter can tell with exact certainty if some element is not present.

But in our case, we plan to use it for the opposite scenario. We want to get the read count for the specific element read. Unfortunately, the bloom filter does have a false positive rate given by the formula: $(1-e^{-kn/m})^k$. Here m is the size of the bloom filter, and k is the number of hash functions used. Definitely, if we increase the number of hash functions used k, and the size of bloom filter m, we can reduce the chances of false positive. But our goal is to increase speed and save space. So there is a limit to maximum m and k we can use.

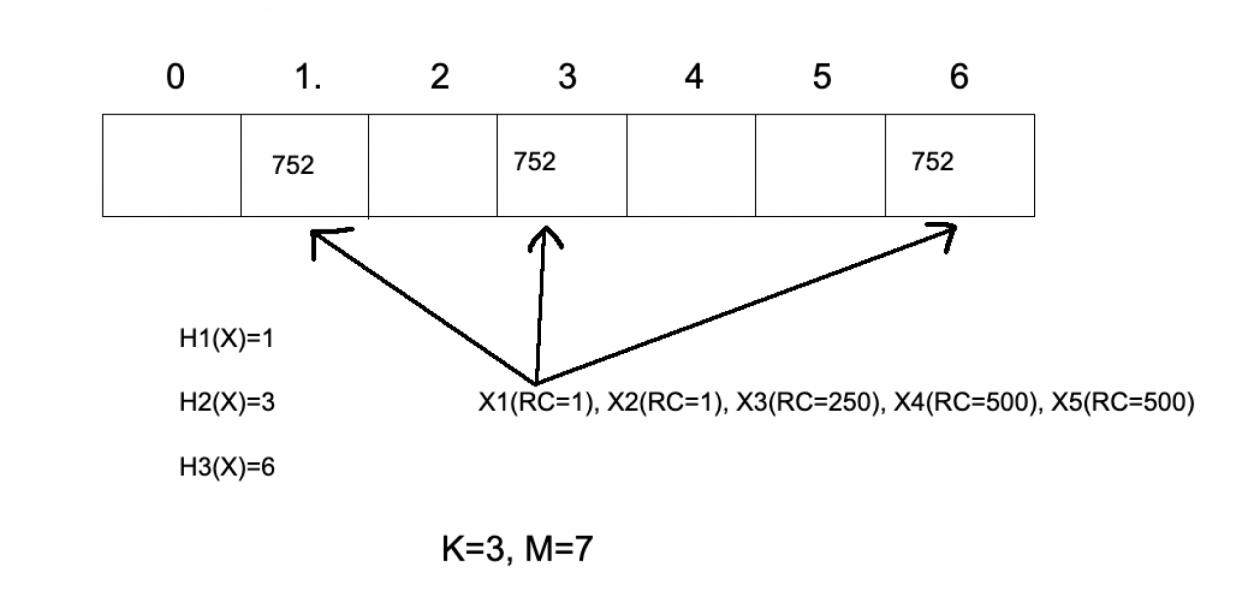
As a result of this is that we may end up selecting the wrong candidate for hot data. This has huge negative implications. We will end up using a lot of CPU bandwidth and space for manually re-inserting not-so-important keys at higher levels. Let's take an example to understand this.

Example

Suppose we use a bloom filter of size 7,i.e., m=7. The number of hash functions used is 3. Now we will have multiple elements in our filter each mapping to 3 indexes. Assume, 5 of these elements X1, X2, X3, X4, and X5 map to the same 3 indexes 1,3, and 6. So they will all contribute to the total count of reads for those indexes.

Let's assume, the actual read count of x1 and x2 is 1, x4 and x5 is 500, and x3 is 250. So the read count stored at indexes 1,3, and 6 will be the sum of all the key reads, which is 752.

Now if we read key X1 or X3, its actual read count is 1, but the counting bloom filter will give their count as 750. Even though the read count for the keys x1, x2 is extremely low, we will end up selecting them as hotkeys.



To address this problem, we use an extra layer of counting bloom filter called possible_hot_data.

4.2.2 Reducing selection probability of the wrong candidate

If we straight away select a key as hot data based on the read count given by counting the bloom filter, we may end up selecting the wrong key. This will end up wasting a lot of space and CPU cycles. To optimize, we need to understand 2 things.

First, even if we don't end up selecting the best candidate, we need to make sure that we don't end up selecting the worst candidate. Selecting some good candidates and not the best ones may work as well. For instance, in the previous example, even if we select X3 as the candidate with a read count of 250 for hot data, it's far better than selecting X1 or X3 with a read count of 1. Second, we are not exactly interested in read count here. We want to capture the tendency of a key to be read when it's at a low level.

Our solution uses a separate layer of possible_hot_data counting bloom filter. This will also keep track of the count of keys, but only for the keys that are selected as the possible candidate from the first counting bloom filter. Because far fewer elements will be competing for the 2nd counting bloom filter possible_hot_data, it will have far fewer collisions, and hence fewer chances of the wrong prediction. To reiterate, counting bloom filter1 would give us possible candidates for hot data. These candidates can be wrong or right candidates. To make sure that we don't end up choosing the wrong/worst candidate, we put the candidate in our possible_hot_data structure. Only when the count of keys in possible_hot_data reaches above a certain threshold number N2, we will finally mark the element as hot.

Once the element is chosen as hot data, we can set the read count for the element as 0 in both the counting filters. This is because we have pushed the selected key up and

given the key a chance in some way. If the key after being pushed up still has a tendency to be read a lot and is also not being updated, the counting bloom filters will select the key again. If we don't set the count to 0 in the bloom filter, the filter will keep on selecting the key again and again, even if the key is not at lower levels anymore.

4.2.3 Algorithm for selecting hot data

Data structures used for selecting hot data

1. **Read_Counting_bloom_filter->** Store count of the number of times the key is read at level > L. Used to get a probabilistic idea if some key is read a lot when it reaches level >L
2. **Possible_Hotdata_Counting_bloom_filter->** Stores Possible candidates for Hotkeys. Also, store count of the number of times the key is read after being inserted. Used to confirm the probability of a key being hot data.
3. **Hotdata_Memtable**-> Used for storing sorted keys that are marked hot.

Algorithm

1. Any data below level Lt, will not be considered as hot data for obvious reasons

2. For data above level Lt:-
    - Create a counting bloom filter to store the number of possible reads for the key.

3. On read, if data lies at level > Lt and

    A. Update/Increment the count of the key against all the hashes in Read_Counting_bloom_filter. (O(1) time)

    B. If the count of all hashes of the key in bloom filter is greater than Threshold Number N1 (O(1) time):

        If the key is in Possible_Hotdata_Counting_bloom_filter**:**
            Update/Increment the count of the key against all the hashes in Possible_Hotdata_Counting_bloom_filter (O(1) time)**.**

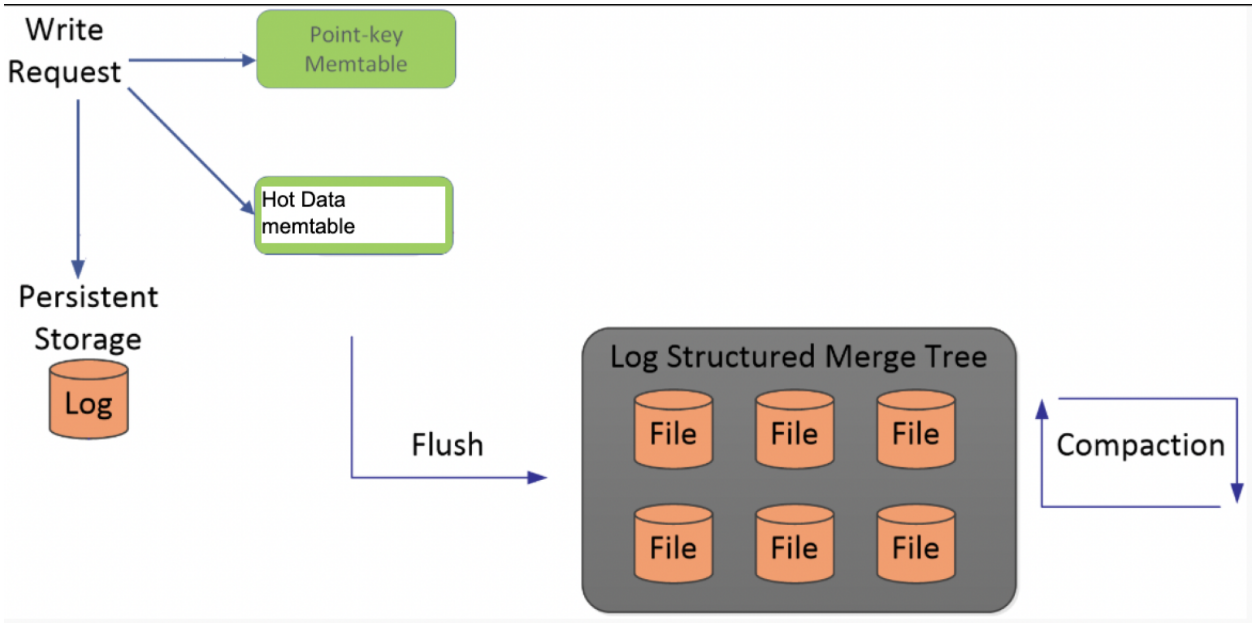
            If the count in Possible_Hotdata_Counting_bloom_filter > N2:
                Push the key to Hotdata_Memtable (O(1) time)

        Else
            Push the key to Possible_Hotdata_Counting_bloom_filter (O(1) time)

Next question is, where to store hot data? How, when, and where to flush hot data?

5 WHEN, WHERE, AND HOW TO PUSH HOT DATA

In RocksDB, data is written to memtables which is then flushed to Level L0. Since we aim to finally flush down the data we mark as hot data, the most natural choice is to keep hot data in a separate set of memtables called “Hot Data Memtables”. When the memtable fills up or crosses a certain size we can flush it to level 0. We can launch a separate set of threads that opportunistically flush these hot data memtables to the database.



The biggest challenge is how to deal with the new values being written. Basically, simply flushing hot data memtable to the database can overwrite the latest update of the key. So we have to maintain time ordering between the actual write/update and our own writes. There are 2 possible solutions to address this problem.

Every write in RocksDB is appended in memtable. The data in memtable is time ordered. So the most natural choice to maintain time ordering between actual writes and hot data writes is to append hot data in normal memtables. But this doesn’t solve the problem. Suppose, the read that selects a key k as hot was initiated at time t1. Assume the value of this key k at t1 was v1. Also assume, the key k was selected as hot data at time t2. Obviously here t1 < t2. Now let's say, the key k was updated to a new value V2 between time t1 and t2 say t3. Here t1 < t3 < t2. In this scenario, we will end up overwriting the latest updated value of the key even if we time-order new data and hot

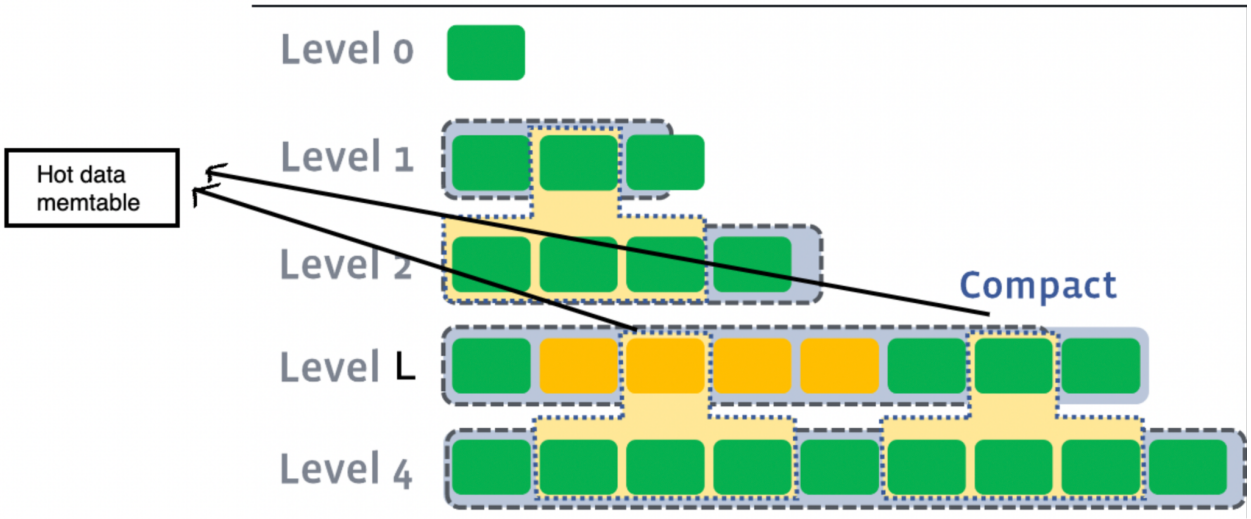
data.

5.1 HOW TO DEAL WITH NEW VALUES BEING WRITTEN

To solve this problem, we first need to note 2 things. First, any write/update in RocksDB takes time to push down to lower levels depending on how fast the compactions are. Second, the time interval between t1 and t2 will be very less in general. So any update that happened between t1 and t2 say t3, will still be at higher levels at time t2.

Let's take a level L which is maybe less than 3. In our solution, we assume that any update or write that happened between t1 and t2 will still be at level L or higher. If this is the case, then we just need to make sure the time ordering of HOT data memtabe writes and all the compactions that push SST from level L to L+1.

So basically, whenever some SST is compacted from level L to L+1, we will also include HOT DATA memtable in compaction. And for any key that is present both in level L SST and HOT DATA memtable, we will use the SST version only.



5.2 SELECTING VALUE FOR LEVEL L AND TRADE OFFS

Now the challenge is to decide level L, the level to which we hotdata memtable will be flushed. To select the best possible value for L, let's discuss the pros and cons of selecting higher and lower values.

5.2.1 Higher value for L

If we select a higher value for L, we are basically inserting our manual hot data writes at a lower level. But our idea was to write the keys at a higher level to achieve better read latencies for them. So we can definitely say that the value of L cannot be very high.

Also, the value L should definitely be lesser than the threshold level Lt that we use to select our keys as hot data. So the maximum possible value for L is Lt-1, or else we would end up selecting a key at sitting at a higher level and re-writing it to at a lower level.

5.2.2 Low value for L

If we select a low value for L, there are 2 drawbacks:
1) Our assumption that any update or write that happened between t1 and t2(say t3) will still be at level L or higher, will not hold value if L is very less say 0,1, etc. Because the latest update might get pushed down to level 1 or 2 between time t3 and t3.
2) Flushing extra hot table mem table at higher levels will stall writes at level 0.

So, we suggest the value of this level L be somewhere in the middle, that captures best of both the worlds. Something around 3 or 4 seems reasonable. Also, we can conclude our approach would work best where data is large, and the lowest level is greater than or equal to 10.

5.3 LEVELED VS UNIVERSAL COMPACTION

Our approach would work much faster with universal compaction in comparison to leveled compaction. This is because our approach incurs the extra cost of reading hot data memtable for every compaction from Level L to Level L+1. The time taken in our approach is thus proportional to the number of compactions from Level L to Level L+1. The number of these compactions from Level L to Level L+1 is far lower in Universal styled Compaction in comparison to Leveled Compaction.

The key difference between the two strategies is that leveled compaction tends to aggressively merge a smaller sorted run into a larger one, while "tiered" waits for several sorted runs with similar size and merges them together.

6 CONFIGURABLE PARAMETERS

1. **Lt**-> Minimum level to be considered for Hot Data. (Recommended L> 6)
2. **L**-> Level where Hotdata memtable will be flushed. (Recommended L < Lt)
3. **M1,K1**-> Size and hash functions used for Read_Counting_bloom_filter
4. **M2,K3**-> Size and hash functions used for Possible_Hotdata_Counting_bloom_filter
5. **Threshold Number N**-> To decide when to push candidate key to Possible_Hotdata_Counting_bloom_filter.
6. **Threshold Number N2**-> To decide when to push candidate key to Possible_Hotdata_Counting_bloom_filter.

Typically, Size of **Read_Counting_bloom_filter >> Possible_Hotdata_Counting_bloom_filter >> Hotdata_Memtable**

7 DRAWBACKS

This paper only presents a design perspective. No implementation or testing is done on the real workloads. There are some obvious drawbacks to this approach. We will note down these drawbacks here. But all the drawbacks are comprehensible only after proper implementation and aggressive testing.

The first drawback of the approach presented is that it targets very specific workloads. RocksDB is built for write-heavy workloads. Only when the workload has a high number of reads, this approach would give a performance boost. Secondly, the solution manually writes a set of hotkeys at level L. This will create a lot of extra contention at level L. The result of this may lead to slow compactions and write stalls. Third, to achieve better read performance, the solution adds extra CPU cycles to maintain and update all the data structures used. It also slows down all reads for keys at level>Lt. The biggest drawback for the approach is the assumption “any update or write that happened between t1 and t2(say t3) will still be at level L or higher” that the solution is based on.

7. CONCLUSION

In this paper we presented a new design to address the low read latency issue in RocksDB. Our solution aims to move frequently read data higher in the LSM Tree. We present the challenges in the current approaches taken by RocksDB to address low latency reads. We then explore the major challenges in our approach and provide a possible solution for them. Mainly we answer major questions like how to determine which data to include in a "hot data" look-aside level, and how to deal with the relationship of that "hot data" and new values. Although it is just a design perspective, and no implementation or testing was done, we discuss the possible scenarios where our approach may not perform well and scenarios where it will boost up the performance.

7.1 ACKNOWLEDGMENTS

We would like to thank Prof. **Sheldon (Shel) Finkelstein,** Lecturer at UC Santa Cruz. This work would not have been possible without his guidance. Also, this paper is written as a part of a course project for CSE 215 taken at UCSC in winter 2021. The course content helped lay a strong foundation to be able to come up with the system design presented in the paper.

8 FUTURE WORK

A. Implementation

This paper only presents a design perspective to address low read latency in RocksDB. The work presented needs to be implemented for a start. We need to explore if the design suggested is actually implementable or not.

B. Benchmarking and testing

The design suggested in the paper targets a specific workload. The changes may not work produce results with all possible workloads. So aggressive testing is needed with different workloads, write-heavy, read-heavy, and mixed.

Also, the design has various configurable parameters. Testing is needed to choose the best values.

C. Deal with the new values being written

The solution mentioned in the paper is based on assumption that “any update or write that happened between t1 and t2(say t3) will still be at level L or higher”. We need a better and certain solution to address this issue. Because, if not solved, this may lead to corruption of data.

8 REFERENCES

[1] https://github.com/facebook/rocksdb/wiki

[2] https://brilliant.org/wiki/bloom-filter/

[3] https://medium.com/techlog/cuckoo-filter-vs-bloom-filter-from-a-gophers-perspective-94d5e6c53299

[4] https://www.cs.cmu.edu/~dga/papers/cuckoo-conext2014.pdf

[5] https://medium.com/system-design-blog/bloom-filter-a-probabilistic-data-structure-12e4e5cf0638

[6] Siying Dong1 , Mark Callaghan1 , Leonidas Galanis1 , Dhruba Borthakur1 , Tony Savor1 and Michael Stumm 2 “*Optimizing Space Amplification in RocksDB”*

[6] Oana Balmau, Florin Dinu, and Willy Zwaenepoel, University of Sydney; Karan Gupta and Ravishankar Chandhiramoorthi, Nutanix Inc.; Diego Didona, IBM Research–Zurich “*SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores”*