SpongeKV: A memory efficient learned indexing technique for LSM key value stores

Arijit Pramanik
University of Wisconsin-Madison

Nithin Venkatesh
University of Wisconsin-Madison

ABSTRACT

Log structured merge trees have grown in importance as the back-end data structure for data management systems. LSM trees store data in disk-backed files which consist of key-value (KV) pairs in sorted order and with non-overlapping KV pairs except at the very first level on disk. The files consist of indexing structures for faster block-by-block data retrieval. In this work we study the various indexing techniques used in RocksDB, a widely used LSM based key value store and show how learning techniques can be used to decrease the memory footprint of the index without compromising latency.

1. INTRODUCTION

Key value stores are used for widely varying storage use cases in data centers and cloud environments, like storing application metadata which are required frequently for different cloud microservices. They have also been used as the back-end storage engine for databases. For example, InnoDB, MyRocks are used as the storage backend for MySQL. Today, we see a new class of stateless relation databases like TiDB which provide the abstraction for relational data but convert tuples to internal key-value pairs stored in the underlying key-value store.

RocksDB is an open source key-value store from Facebook developed as an extension to LevelDB initially developed by Google. RocksDB internally uses Log structured merge (LSM) trees which differ from other indexing structures like B-Trees. The basic idea of LSM trees is to have an in-memory data structure for absorbing the incoming writes which are batched and later written to persistent storage in the form of a structure called SSTables. SSTables contain entries i.e. key-value pairs in sorted order which would help in faster reads and scans, using binary search or even faster using hash indices. Absorbing write bursts in memory and periodically migrating them to persistent storage helps these key-value stores provide very high write throughput and low write latency.

.

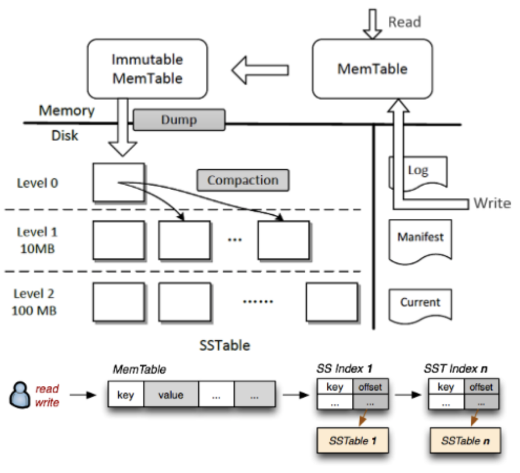

Figure 1: Write and Read paths in an LSM tree based key value store

1.1 Working of an LSM-tree key value store

As shown in Figure 1, the main data structures that are used in an LSM tree based key value store are the Memtable and SSTable (Sorted sequence table). **Memtable** is typically a *B-Tree, skip list* or a *hash table* data structure that is used to store the incoming writes in memory after the writes are synced to an append-only log for durability and crash consistency. **SSTables** are the data structures that are used to store data on the disk. They consist of key value pairs organized into data blocks with *delta encoding* for efficient storage. This implies that a common key prefix across a group of keys is stored with only subsequent differing portions stored for remaining keys. The block-partitioned format is common for block-based storage media, while a contiguous plain-table format works well for in-memory databases. After the Memtable reaches a certain preset size threshold, it is converted to an immutable SSTable file and flushed to disk.

LSM key value stores organize data into multiple logical levels. For example, RocksDB and LevelDB have 7 levels by default. The maximum amount of data that can be stored at a level is given by a predefined threshold ($10^{(level)} MB$ by default for RocksDB) except for level 0 which can store a maximum of a specified number of SSTables on disk.

Write path : When the `put` or the `write` call for a key-value pair is made on the database, the KV pair is appended to a *write-ahead log* and then inserted into the memtable data structure in memory. The writes continue till the memtable reaches a particular size of 4 MB. Once the size is reached, a new in-memory memtable is created to absorb the writes and the previous memtable is converted to an immutable version before being flushed as an SSTable to level 0 of the on-disk LSM tree.

Compaction process : Based on whether the number of files in level 0 is above a compaction threshold or if the data at a particular level exceeds the specified value (typically $10^{(level)}MB$), a *background compaction* process is triggered. The compaction process is used to remove the *duplicate* and *deleted* entries corresponding to a key by keeping only the most recent version for a given key. The compaction process chooses a particular level to carry out the compaction process. For each file in the chosen level, it looks for files in level + 1 which consist of *overlapping* keys. If such a file is found, then these files are combined using a technique similar to **merge-sort** and the newly formed file is stored at level+1 and the file at compaction level is deleted.

Read path : If a `read` or `get` request is issued for a key, first the in-memory memtable is searched to see if the key exists. If so, then the most recent version of the key and the corresponding value is returned. Else, the LSM tree levels are searched starting from level 0. At level 0, each SSTable is searched one by one for the key and the most recent version is returned if that key is found in level 0. Else, the key is searched in other levels. Now, since the SSTables in the other levels are sorted and are disjoint, the search can be performed efficiently. **Bloom filters**, *binary search* and *regression* techniques are used to efficiently carry out this search without excessive disk reads. An additional hash index can also aid in quick search at some extra space overhead.

Once the SSTable likely to contain the key is determined, the corresponding index block which is typically stored in memory is consulted to find the data block inside the SSTable that contains the key and only that particular data block is read to avoid excess reads. Once the data block is read, we do a binary search over all its entries to find the required key. A data block can also have an additional hash index which allows constant time lookup of its constituent keys. We will go into more details of this process in Sec 2.

For efficient lookup, the index blocks corresponding to each SSTable file in an LSM key value store are cached in memory. RocksDB partitions main memory into `BlockCache` for storing these data blocks and `TableCache` to cache the index and filter blocks, but memory can be highly contended for large database sizes.

2. BACKGROUND AND MOTIVATION

As seen in 1.1, the memory footprint of index blocks can be significant, affecting the performance of the key-value store. The index blocks might compete with the data blocks for DRAM thereby leading to *poor cache hit rates* and hence poor performance. With decreasing DRAM to SSD ratios, this problem is exacerbated. At higher cost per byte for DRAMs, reducing the memory footprint without mitigating performance can be a huge benefit for today's cloud data centers. In this work, we look at the memory footprint of the indexing techniques used in LSM based key value stores and explore ways to reduce the memory footprint of the index blocks.

2.1 A case study of TitaniumDB

Key value stores are increasingly being used in cloud and on campus cluster environments. An example of a distributed cloud-based relational database is TiDB, built on top of LSM-based key value store RocksDB as the storage engine. Our analysis and methods are not limited to this specific use case but the application serves as a good example to illustrate the problem of memory overhead for index blocks.

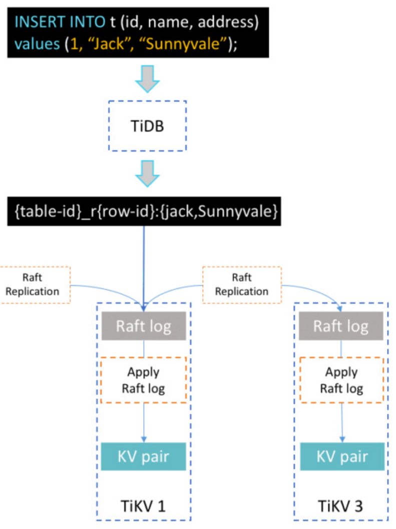

Figure 2: Working of TiDB, and relational database on LSM based key value store RocksDB

TiDB [2] is an open source NewSQL database that supports HTAP workloads and provides horizontal scalability, strong consistency and high availability using back-end key value stores, thereby providing the abstraction of a stateless relational database on top of key value stores. The high level working of such a system is as shown in Fig 2. Users submit SQL queries, to TiDB, the corresponding tuples are converted to key value pairs, which are replicated using a consensus algorithm like Raft, and then stored on each TiKV node running the RocksDB key value store.

The keys are encoded in Memcomparable[1] and then the hexadecimal strings are sent to the RocksDB key value store in the back-end. The encoding for table data into corresponding key value pairs is shown in Table 1. The size of keys thus formed can vary from a few bytes to hundreds of bytes. This can have an adverse effect on the index data since, the index entries store the block offsets corresponding to the last or the first key of each data block in a SSTable file. Larger keys can lead to the index data occupying more of DRAM. The fact that data block sizes can be variable

Data type	Key	Value
Table row	t{tableID}_r{rowID}	$[col_1, \ldots, col_n]$
Primary key/ unique index	t{tableID}_i{indexID}_ indexedColumn(s)Value	rowID
Secondary key/ non-unique index	t{tableID}_i{IndexID}_ indexedColumn(s)Value_ {rowID}	NULL

Table 1: Format of key value pairs stored in TiDB

incorporates additional metadata overhead of storing their individual sizes. Note that data blocks are individually compressed and their sizes are required for efficiently iterating over the data contents of the data block.

The need for reducing the memory footprint of LSM based key value stores as expressed by database administrators : *"It should be several percent of the data size. Assume index and bloom filter blocks are all cached, and we have memory being 5 percent of data size, then we have little to no memory to cache the data. And we are thinking push the data density further to like 1-2 percent of data size (e.g. 64GB memory per TiKV node with 4TB data each). And it would help a lot if we can minimize the index size."*

2.2 Indexing techniques in RocksDB

Some general space optimizations for constructing indices involve shortening the key being stored. Suppose last key of i^{th} data block is a and first key of $(i+1)^{th}$ data block is b, then we seek a key $[a, b)$ with the shortest length and store the same as the index entry. E.g. "12" can act as an index entry for i^{th} data block ending in "1000" while the next data block begins with "1348". Similar savings can be achieved for the last key of the last data block in an SSTable. Index entries can also be delta-encoded. Suppose, we have the following keys as index entries : "100101", "100152", "100223" and "100289". We can simply store "100101" and note that the common prefix across these 4 keys in "100" (i.e. first 3 bytes), and subsequently store "152", "223" and "289" for the remaining 3 keys.

The index blocks are generally not compressed. Also, if we increase the size of each data block, the number of index entries decrease, but the trade-off here would be fetching large data blocks into memory for single key (point) lookups. Since fewer such blocks can be stored in memory, this may lead to thrashing (continual eviction and re-fetching) in case of widely ranging random key lookups. This also amplifies the problem of memory fragmentation.

- **Single level index** : For each data block, we store the *last key* as the corresponding index entry along with the *size* and *offset* of the data block. As seen in Fig 3, we have keys ranging from 3 to 5 bytes and value being 1 byte, KV-pair offsets range from 4 to 6 bytes. For the 3rd data block containing "01111", "012" and "020", its corresponding index entry is "020" stored alongside its starting offset of 25 within the SSTable file and its size 4+4+6 = 14 bytes. Note that the keys are strings sorted lexicographically.

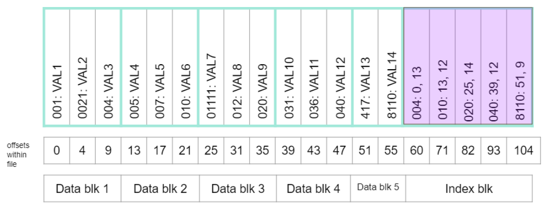

Figure 3: A single level index structure in RocksDB

- **Partitioned index** : The index block above is stored as a single block and needs to be fetched *entirely* for reading the corresponding SSTable file. Since SSTables can be as large as ≈ 100 MBs, index blocks can easily be ≈ 100 KBs. So, we partition the index block into fixed-size blocks and build another index on top. For every read, we fetch the top-level index corresponding to each SSTable and accordingly fetch the required index block for the data block lookup to reduce the footprint of cached index blocks. As shown in Fig 4, we have 2 2^{nd} level index blocks with top-level index having 2 entries corresponding to the first index entry for each of those blocks. We also store the offset and size of the index blocks, each being a 4 bytes long integer.

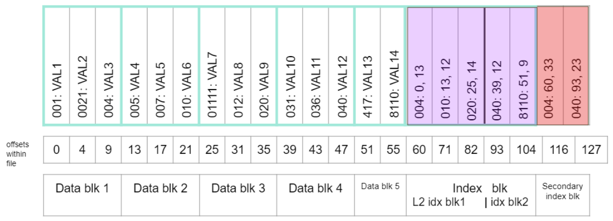

Figure 4: Partitioned / two-level index in RocksDB

As can be seen above, the single-level indexing strategy has higher memory footprint for random key lookups spanning multiple SSTables, while the partitioned indexing strategy might end up with increased latency for some reads due to an additional indirection for demand fetching of the second level index blocks. If we start caching the 2^{nd} level index blocks, then our memory footprint might exceed that of previous strategy since the cumulative size of index blocks across both levels can be much larger. In order to **reduce the memory footprint of indexing while not affecting the read latency**, we turn to learned techniques for indexing.

2.3 Learned indexes

B^+ trees are widely used in today's databases for fetching records. The index is traversed by searching for keys at each level and finally reaching the leaf node which contains a pointer to the desired record. This incurs an $O(logn)$ time complexity and can be slow if there are too many index entries for today's petabyte-scale databases.

A learned model can be used to enhance/replace traditional indexes. Since CPU-SIMD/GPUs are common today, training such ML models is computationally cheap. Predicting the position given a key inside a sorted array implies effectively approximating the cumulative distribution function

(CDF). Hence, indexing literally requires learning a data distribution and if appropriately estimated, reduces storage requirements to the *weights of the model*. Such learned approaches can also be used to replace Bloom filters. We can also organize these models in a tree-like structure, termed **Recursive Model Index** so that each can focus on certain subsets of the data.

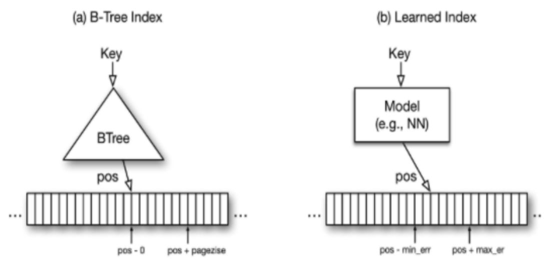

Figure 5: Working of learned indexes as a replacement to traditional indexes

Given the prediction (offset, err_{min}, err_{max}) $= model$(key), this reduces lookup time to $O(1)$ by searching in (offset - err_{min}, offset + err_{max}) interval as shown in Fig 5. Though this works well for reads, updates are hard to implement since the model needs to be re-learned, which can be a significant overhead if updates are frequent. Since in our case SSTables once built are *immutable* and serve only reads, the learned index fits our goal very well.

3. DESIGN FOR LEARNED INDEXES

We demonstrate some simple statistical models to approximate the distribution of <keys, offsets> as we learn a model to *predict the data block* inside which a given key occurs and do a (binary) search after fetching it to get the corresponding value. As SSTables are created during compactions in Sec 1.1, we learn the *starting keys of each block and corresponding block offsets for all data blocks* within the file. We term this as ***file-based learning*** and is triggered whenever compaction creates a new SSTable. The models are stored along with the SSTable in place of the index and gets deleted along with it during compaction. Since the lifetime of SSTables at higher levels on disk (L2 onwards) are pretty high, these models are relatively stable and server lookups faster.

Both our regression methods involve learning lines with slope a and intercept b such that it minimizes the least squares error $\sum_{i=1}^{n}(y_i - (ax_i + b))^2$ for $n < x_i, y_i >$ data points corresponding to <keys, offsets>.

3.1 Greedy Piecewise linear regression

The <keys, offsets> being sorted by key will be non-decreasing, but may not be linear. So we fit a series of line segments. We fix a specified *error* threshold for initiating a new line segment. We can control the number of data blocks that need to fetched and searched for a given key. E.g. an error value in (min KV-pair size, max block size] will lead to fetching 2 data blocks in the rare case that the search interval straddles across the boundary between 2 data blocks, even if as small as 24 bytes though typical block sizes are 4 KB. Algorithm 1 run in $O(n)$ time for n keys. A dynamic programming approach runs in $O(n^3)$ time, but requires specifying the number of segments before hand and makes several passes over the data to find the best boundary points. Since this runs for every compaction, this can potentially *stall writes* as memtable gets full and is unable to flush to disk due to all threads being involved in long-running compactions.

Algorithm 1: Greedy piecewise linear regression

```
Data: $< x_1, y_1 >, < x_2, y_2 > \ldots, < x_n, y_n >, err$
$a_1 = \frac{y_2 - y_1}{x_2 - x_1}$,   $b_1 = \frac{y_1 x_2 - x_2 y_1}{x_2 - x_1}$,   $last\_start = x_1$;
$segments = \{\}$,   $prev\_seg =< x_1, a_1, b_1 >$;
for $i = 3$ to $n$, $k = 0$ do
    if $|(prev\_seg.a_k * x_i + prev\_seg.b_k) - y_i| > err$
     then
        add $< last\_start, a_k, b_k >$ to $segments$;
        $last\_start = x_i$;
        $a_{k+1} = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$;
        $b_{k+1} = \frac{y_i x_{i+1} - x_i y_{i+1}}{x_{i+1} - x_i}$;
        $prev\_seg =< x_i, a_{k+1}, b_{k+1} >$;
        increment $k$ by 1;
    end
end
Result: $< s_1, a_1, b_1 >, \ldots, < s_k, a_k, b_k >$
```

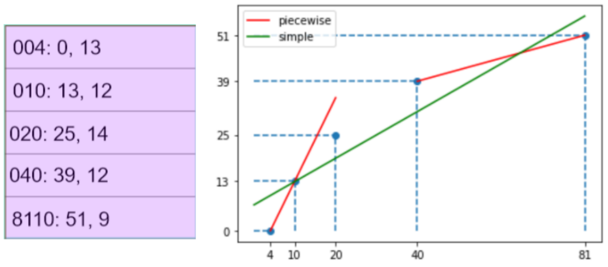

Figure 6: Fitting simple and piecewise linear regression to predict block offsets from the key's integer value. "004", "020" are encoded as 4, 20 respectively. Encode last key "8110" as 81 to save space.

In single-level and partitioned index, we stored sizes and offsets for each data block. But now we can estimate offset using our learned model. So, we only store the sizes for each data block. Following the same example in Fig 3 and 4, we illustrate the index block in Table 2. Previously, number of index entries was the same as number of data blocks = 5. But now using our regression techniques, we can approximate the distribution of offsets and store only the starting key, slope and intercept for each segment where number of segments = 3 is generally much lower than number of data blocks = 5.

To lookup a key, we simply do a binary search in the list of segments and using the corresponding segment, we get the predicted offset as (*slope*×*key* + *intercept*). Using the preset error bound, we now fetch all data blocks whose starting offset lies within (offset - error, offset + error) to lookup the KV pair.

Start key	Slope	Intercept
"004"	2.167	13
"040"	0.292	27.29
"9"	-1	-1

Data blk #	Size
1	13
2	12
3	14
4	12
5	9

Table 2: Index block for greedy PLR. "9" is a dummy key larger than all keys in SSTable to bound our binary search within the set of segments

Start key	Slope	Intercept
"004"	0.607	6.761
"9"	-1	-1

Data blk #	Size	Pred offset
1	13	4.104
2	12	9.796
3	14	13.438
4	12	25.578
5	9	31.648

Table 3: Index block for simple linear. "9" is a dummy key larger than all keys in SSTable to bound our binary search within the set of segments

3.2 Simple linear regression

If we use the above approach but approximate using a single line, then we need to execute the model for every key and remember the worst over and under prediction of an offset to calculate the min and max error bounds. This can be quite large and we might need to fetch too many data blocks in memory, significantly increasing the read latency. The algorithm 2 for fitting a single line segment to the set of <keys, offsets> is computationally quite simple and runs in $O(n)$ time.

Algorithm 2: Simple linear regression

```
Data: $< x_1, y_1 >, < x_2, y_2 > \dots, < x_n, y_n >$
$a = \dfrac{\sum_{i=1}^n y_i \sum_{i=1}^n x_i^2 - \sum_{i=1}^n x_i \sum_{i=1}^n x_i y_i}{n\sum_{i=1}^n x_i^2 - (\sum_{i=1}^n x_i)^2}$;
$b = \dfrac{n\sum_{i=1}^n x_i y_i - \sum_{i=1}^n x_i \sum_{i=1}^n y_i}{n\sum_{i=1}^n x_i^2 - (\sum_{i=1}^n x_i)^2}$;
$err = 0$, offset_preds = {};
for $i = 1$ to $n$ do
    $err = max(err, |(a * x_i + b) - y_i|)$;
    add $< x_i, (a * x_i + b) >$ to offset_preds;
end
Result: $< x_1, a, b >$, $err$, offset_preds
```

So, instead we store the ***offset predictions of the starting key of each data block*** and given any key for lookup, we compute its predicted offset as $a \times key + b$ and do a ***binary search over the list of offset predictions*** to find the exact data block in which the key resides. The offset predictions for data block starting keys are *strictly monotonic irrespective of the underlying key distribution.* This ensures that only a single data block is read for every key lookup. This is close to being *pareto-optimal*, since we achieve lower latency with significant space savings in terms of index footprint as evaluated in Sec 5. Hence, this outperforms traditional indexing strategies in terms of memory-latency tradeoff for LSM KV stores. To the best of our knowledge, this is a novel technique of storing the output of a simple linear regression model at regular intervals based on memory availability to narrow down the search space.

As seen for greedy piecewise linear regression in Table 2, number of segments can be large for small error bounds especially when data block sizes are small. For simple linear, we actually need to store only a single segment to predict the offsets for keys and then do a binary search within the list of offset predictions for each data block, which involves a larger search space than the list of segments previously, but the logarithmic increase is insignificant. Table 3 shows the index block. Say, we want to lookup "007". The model will predict an offset of $7 \times 0.607 + 6.761 = 11.01 \in [9.796, 13.438)$. If its offset is smaller than the start key of data block 3 while being larger than that of data block 2, then it has to be in data block 2 which can be verified from Fig 3.

4. IMPLEMENTATION

We implement greedy piece wise linear regression and simple linear indexing strategies as part of RocksDB. As explained in section 3, we learn an index model corresponding to a SSTable file on-the-fly during compaction. We temporarily store the keys being inserted to an SSTable during compaction in a container-like list or vector and use this for further processing. For simple linear regression model, we use the closed form equations to estimate the slope and intercept given a set of keys and their offsets. For the greedy PLR model, we start with a pair of keys, and greedily add keys until the predicted offset for the keys added to the current line segment exceeds the given error bound as demonstrated in Algorithm 1. The implementation for greedy PLR is the reference implementation as presented in the Bourbon [5] paper.

RocksDB uses two types of caches in memory apart from the OS-managed page cache. Block cache is used to cache the data blocks but can be configured to cache the index and filter blocks as well. Table cache is used to prefetch the index and filter blocks into memory when a SSTable file is opened. For the partitioned indexing strategy, only the top level index blocks of the SSTable files are prefetched into the table cache and the second level index blocks are loaded to the block cache on demand. For the other indexing strategies, single level indexing and the newly implemented learned techniques, we prefetch all the indexing data to the table cache whenever an SSTable file is opened.

Both block cache and table cache use a generic LRU cache mechanism to promote and evict blocks and files respectively. By default the table cache is configured to support a specified number of files and when the number of files exceed the said threshold, the least recently used file is evicted. We modify this strategy to evict the files based on a preset capacity threshold (in bytes) on the table cache. So, for a read, when an SSTable is opened and read to the table cache, the size of its index and filter blocks is added to the table cache size, if the table cache size exceeds the set capacity threshold, then the least recently used file is evicted.

The learning strategies are implemented in a separate module and other learning based techniques can be easily integrated with slight modifications. These are currently implemented for the *block-based format* for SSTables.

5. EVALUATION

We use the Intel (R) Xeon (R) CPU D-1548 @ 2.00 GHz, x86_64 with 16 cores and 64 GB RAM on Cloudlab corresponding to the type m510 for our experiments. For all the experiments shown in the figures below, we load 10M key value pair with varying key sizes and a value size of 100 bytes. We measure the memory requirements and average read latency as perceived by the client for the traditional and the learned indexing strategies. For simplicity and to show the effectiveness of the learning strategies in reducing the memory footprint of indexing, we disable the page cache, block cache and filter blocks for the experiments. Note: Block cache is enabled for partitioned or two level indexing as it only uses the block cache for storing the second level indexes. For the other indexing strategies, the index data is stored as part of the table cache.

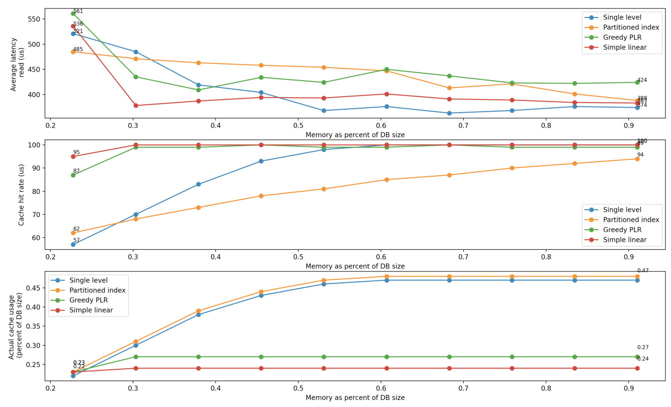

Figure 7: Memory reduction using learned techniques as compared to the traditional indexing strategy for key size 8B and value size 100B

Figure 7 shows that the cache hit rate for both the greedy PLR and simple linear indexes reach to 100 percent with the cache size being 0.3 percent of the DB size whereas single level reaches a 100 percent hit rate when the cache size is 0.54 and since the partitioned index uses block cache and not table cache for caching, and the index blocks compete with data block for cache space, the cache hit rate for the index blocks does not reach 100 percent even when the cache size provided is more than the space that is required to fit all index blocks. The cache memory requirement for single level indexing is 0.47 percent of the DB size whereas it is around half at 0.27 and 0.24 for greedy PLR and simple linear indexing techniques. We also observe that, we are able to reduce the memory requirement for indexing using the learned techniques without compromising on the average latency as shown the first graph in figure 7.

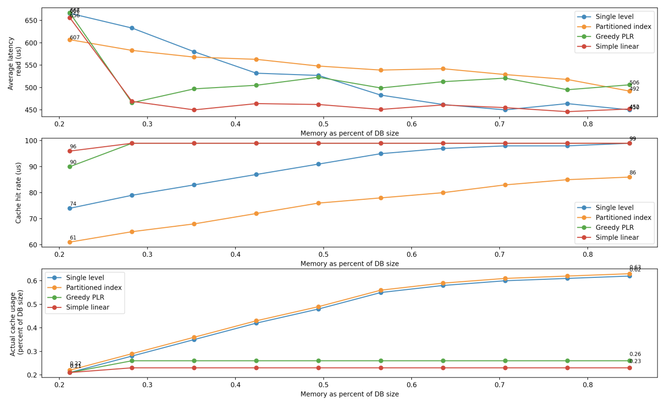

Figure 8: Memory reduction using learned techniques as compared to the traditional indexing strategy for key size 16B and value size 100B

In order to see how the memory requirements for the indexing strategies change as the key size is increased, we increase the key size and run the experiments. Figure 8 shows the average read latency, cache hit rate and the cache memory requirements for the indexing strategies. We observe that the memory requirement as a percentage of the DB size for greedy PLR and simple linear techniques decreases as compared to key size of 8B while the ratio increases for the single level and partitioned indexing strategies. The cache memory requirement with key size of 16B and value size of 100B for greedy PLR, simple linear, single level and partitioned indexing techniques are 0.26, 0.23, 0.62 and 0.63 percent of the DB size respectively. We also observe that simple linear regression based indexing is able to achieve less average read latency as compared to the other indexing strategies due to lesser evictions when the memory is scarce. When sufficient memory is available for all the indexing strategies, the simple linear regression based technique utilizes the minimum amount of DRAM while providing latency similar to other techniques strongly suggesting the use of learned techniques, especially simple linear regression based indexing for LSM key value stores.

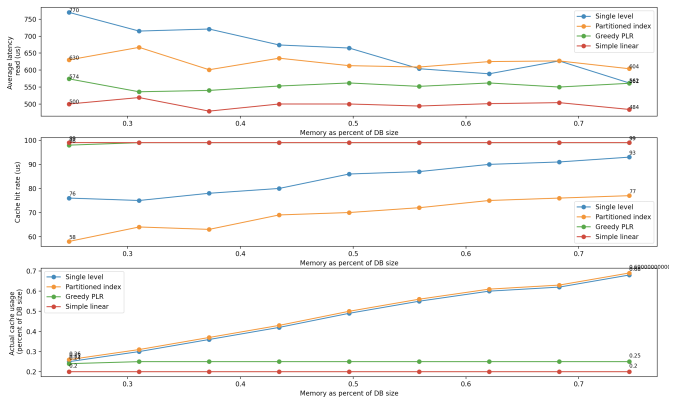

Figure 9: Memory reduction using learned techniques as compared to the traditional indexing strategy for key size 32B and value size 100B

We observe similar trends increasing the key size further.

Figure 9 shows the average read latency, cache hit rate and cache memory requirement for keys size 32B and value size 100B. The cache memory requirement to DB size ratio further decreases for the learned indexes at 0.25 and 0.2 percent for the greedy PLR and simple linear regression models as compared to 0.68 and 0.69 for greedy PLR and simple linear regression techniques. Also, simple linear and greedy PLR techniques achieve less latency compared to the traditional techniques.

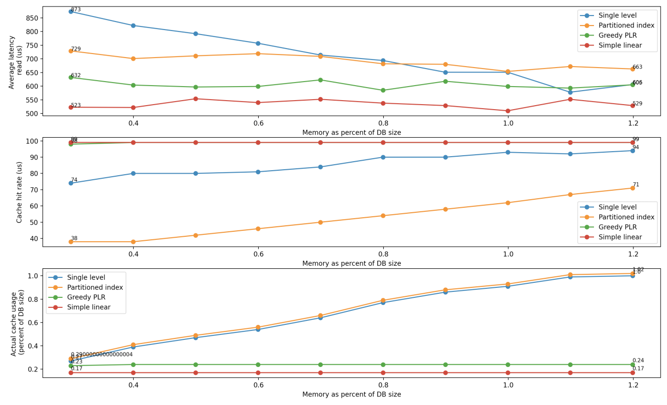

Figure 10: Memory reduction using learned techniques as compared to the traditional indexing strategy for key size 64B and value size 100B

For key size 64B and value size 100B, difference in the cache memory requirement for learned strategies and traditional strategies further increase. Greedy PLR and simple linear methods require 0.24 and 0.17 percent of the DB size simple and partitioned indexing strategies require 1 and 1.02 percent of the DB size respectively.

5.1 Greedy PLR index memory footprint with varying error bounds

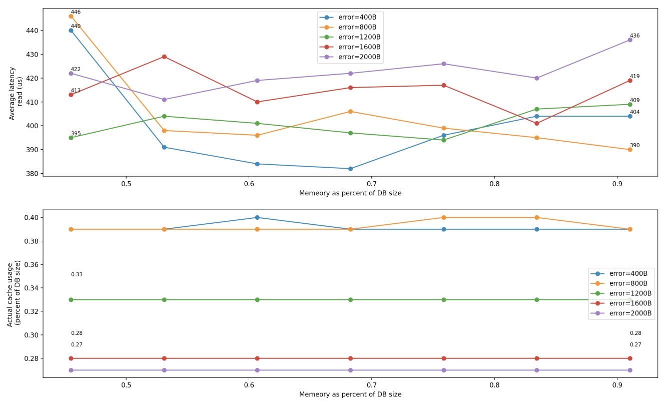

Figure 11: Average latency and memory requirement with varying error bounds for greedy PLR indexing technique

Figure 11 shows how the how the memory requirement for the greedy PLR indexing strategy decreases as the error bound is increased. The size of a data block is 4KB, and we vary the error bound from 400B to 2KB and record the observations. An error bound of 400B means, during the process of greedy PLR training, if the offset predicted varies from the correct offset by more than 400B, then a new line segment is created, which leads to more segments and hence more memory requirement. We observe that, cache memory requirement is around 0.39 percent of the DB size at 400B error bound and this reduces to 0.27 percent for 2000B as the error bound. Correspondingly, the average latency increases as the error bound is increased since more than one block (two blocks in this case) have to be read for a higher percentage of reads. This indicates, that one has to carefully choose the error parameter in case of greedy PLR based learned indexing strategy. Tuning this parameter might be hard without proper analysis. Simple linear regression, on the other hand has no parameter to tune and provides a simple, both memory and performance efficient alternative to the traditional indexing strategies in memory constrained LSM key-value stores.

6. RELATED WORK

Characterizing, Modeling,and Benchmarking RocksDB Key-Value Workloads at Facebook by Cao et al. [4] presents a characterization of workloads 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. [5] based on improving Wisckey [8] explores how to reduce the read latency by introducing piece wise linear regression for constant time lookup of the key in an SSTable of the LSM tree. They also develop a cost-benefit analysis to decide when it is advantageous to build the linear model for an SSTable based on the average lifetime of SSTables. However they only support fixed size integer keys and show the benefits of using learned techniques when most of the data can fit in memory. In this work, we have showed that learned indexes can achieve comparable read latency even when the data is stored on disk while incurring significantly less memory cost.

Reducing DRAM footprint with NVM in Facebook [6] presents one the first NVM based key value stored deployed in production. They aim to reduce the total cost of ownership of their MySQL cluster based on MyRocks by reducing the DRAM footprint for caching while maintaining comparable mean latency, 99th percentile latency and Queries per second. They build MyNVM as a replacement to MyRocks using the main idea of using higher NVM capacity to counter reduced DRAM memory. They show that they are able to obtain similar performance using 16GB DRAM and 140GB NVM in place of 96GB DRAM. They also show how to counter the NVM read bandwidth bottleneck using the partitioned indexing strategy discussed earlier.

Learned indexes for a Google scale disk based database [3] presents a study emphasizing the practicality of learned indexes structures in real systems like Bigtable. The motivation here is to reduce the index memory footprint for Bigtable. They use simple linear regression to decide the block number for a given key value pair, which can lead to non-uniform block sizes. We use a complementary approach of storing the output of simple linear regression model for block boundaries and ensure that the block sizes remain uniform. Non uniformity in block sizes can lead to more than one block being read for read operation since they do not align on page boundaries.

LSM trie [9] observes that small key value pairs are common in production key value stores. For example, in a Facebook key value store, 90 percent of the key value pairs are less than 500B. They observe that large index set leads to read degradation with index spilling out of memory. They propose a novel compaction design to reduce write amplification and eliminate index using LSM trie, compromising range query support. They cluster on disk bloom filters for efficiency.

In the case for learned index structure by Kraska et. al. [7], they present that most of the traditional index structures can be replaced by other models including deep learning models which they term as learned indexes. They show that, by using neural networks, they are able to outperform cache optimized B-Tree index by up to 70 percent in speed thereby saving an order of magnitude in memory over several real world datasets. We draw inspiration from this in looking at how learned indexing can be used to reduce memory footprint for LSM tree based key value stores.

7. CONCLUSION

As modern cloud infrastructure is moving towards serverless computing and key-value stores are rapidly being used to store application-critical metadata across thousands of microservices, it is of critical importance to reduce the storage overhead of these backend KV stores. Newer technologies like NVM are seeing lesser and lesser DRAM being used inside cloud datacenters which calls for stricter bounds on storage for the index and metadata structures on DRAM associated with key-value stores. Even if sufficient memory is available, administrators want to utilize it for caching data blocks instead of indexing metadata. With the economies of scale at play, simple ML models like linear, polynomial regression can alleviate much of the storage requirements of indexes in modern key-value stores. With modest computational overheads for training, these can be integrated into today's key-value backed databases without compromising service-level agreements while providing a much lighter memory footprint.

8. REFERENCES

[1] https://github.com/facebook/mysql-5.6/wiki/myrocks-record-format.

[2] https://github.com/pingcap/tidb.

[3] H. Abu-Libdeh, D. Altınbüken, A. Beutel, E. H. Chi, L. P. Doshi, T. K. Kraska, X. S. Li, A. Ly, and C. Olston, editors. *Learned Indexes for a Google-scale Disk-based Database*, 2020.

[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] Y. Dai, Y. Xu, A. Ganesan, R. Alagappan, B. Kroth, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. From wisckey to bourbon: A learned index for log-structured merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 155–171. USENIX Association, Nov. 2020.

[6] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. Hazelwood, C. Petersen, A. Cidon, and S. Katti. Reducing dram footprint with nvm in facebook. *EuroSys*, 2018.

[7] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. *CoRR*, abs/1712.01208, 2017.

[8] 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.

[9] X. Wu, Y. Xu, Z. Shao, and S. Jiang. Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 71–82, Santa Clara, CA, July 2015. USENIX Association.