S3: A Scalable In-memory Skip-List Index for Key-Value Store

Jingtian Zhang[†], Sai Wu[†*], Zeyuan Tan[†], Gang Chen[†],
Zhushi Cheng[‡], Wei Cao[‡], Yusong Gao[‡], Xiaojie Feng[‡]
[†]Zhejiang University, Hangzhou, Zhejiang, China.
[‡]Alibaba Group, Hangzhou, Zhejiang, China.
{11421015, wusai, 3160103832, cg}@zju.edu.cn
{zhushi.chengzs, mingsong.cw, jianchuan.gys, xiaojie.fxj}@alibaba-inc.com

ABSTRACT

Many new memory indexing structures have been proposed and outperform current in-memory skip-list index adopted by LevelDB, RocksDB and other key-value systems. However, those new indexes cannot be easily intergrated with key-value systems, because most of them do not consider how the data can be efficiently flushed to disk. Some assumptions, such as fixed size key and value, are unrealistic for real applications. In this paper, we present S3, a scalable in-memory skip-list index for the customized version of RocksDB in Alibaba Cloud. S3 adopts a two-layer structure. In the top layer, a cache-sensitive structure is used to maintain a few guard entries to facilitate the search over the skip-list. In the bottom layer, a semi-ordered skip-list index is built to support highly concurrent insertions and fast lookup and range query. To further improve the performance, we train a neural model to select guard entries intelligently according to the data distribution and query distribution. Experiments on multiple datasets show that S3 achieves a comparable performance to other new memory indexing schemes, and can replace current in-memory skip-list of LevelDB and RocksDB to support huge volume of data.

PVLDB Reference Format:
Jingtian Zhang, Sai Wu, Zeyuan Tan, Gang Chen, Zhushi Cheng, Wei Cao, Yusong Gao, Xiaojie Feng . S3: A Scalable In-memory Skip-List Index for Key-Value Store. *PVLDB*, 12(12): 2183-2194, 2019.
DOI: https://doi.org/10.14778/3352063.3352134

1. INTRODUCTION

Many popular key-value stores, such as LevelDB [8] , RocksDB [13] and HBase [1], maintain an in-memory skip-list [32] to support efficient data insertion and lookup. The reason of adopting skip-list is two-fold. First, its maintenance cost is low, since it does not require complex adjustment operations in the tree-like index to keep balance. Second, when skip-list is full (4MB case in LevelDB), it can be efficiently flushed to disk and merged with other disk data structures (e.g., SSTable).

However, many new in-memory indexing structures [4, 20, 26, 7, 30, 37, 42] are proposed, showing superior performance than the classic skip-list index. Some core techniques, when designing those efficient new indexes, include cache-sensitive structure, SIMD speedup for multi-core architecture, specific encoding of the key, latch-free concurrent access and other optimization approaches. Those sophisticated designs allow the new indexes to achieve an order-of-magnitude better performance than skip-list[41]. The problem, on the other hand, is that they are difficult to be applied to existing disk-based key-value systems. Since many services on Alibaba Cloud are supported by our customized RocksDB (tailored for the cloud environment), we use the RocksDB as our example to illustrate the idea.

All hash-based in-memory indices [4, 20] cannot be directly integrated with the disk-part of LevelDB or RocksDB, because they do not maintain the keys in order. Consequently, they do not support range queries, or we need to transform the range queries into multiple lookup requests. Before flushing those data back to the disk as a SSTable, a sorting process is invoked, which is costly and may block write and read operations.

In RocksDB, the in-memory index is kept small (4MB-64MB) so that it can be efficiently flushed to disk and does not trigger high compaction overhead [1]. This strategy causes the in-memory index to be frequently rebuilt and flushed out. Many in-memory indices have not taken the case into consideration. To achieve a high in-memory performance, keys and values are normally formatted to be friendly to the cache. Trie-based indices, such as ART [26], Judy [7] and Masstree [30], split a key into multiple key slices to facilitate the search and cache usage. Each slice contains a few bytes of the original key. For example, each layer in the Masstree is indexed by a different 8-byte slice of key. This will significantly slow down the flushing process, as we need to assemble the keys back. Masstree also maintains the key

*Sai Wu is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352134

[1]Another reason is that skip-list does not perform well for large datasets. However, even if we replace skip-list with other indices, we still cannot maintain a too large in-memory index, since the compaction overhead will become the bottleneck and block the insertion and lookup operations.

and value separately and as a result, we need to merge them together before writing the data to disk.

Many other skip-list-based indices, such as CSSL [37] and PI [42, 43], will periodically restructure the index to optimize the performance. During the restructuring process, the read and write operations are blocked. Besides, both indices assume that keys are of the same size, so that a specified number of keys can be exactly processed by a SIMD vector. A single SIMD load instruction can load all those keys into a SIMD register. Such optimization may not be possible for real applications. On the other hand, FAST [25] addressed the problem by mapping variable length keys to 4-byte fixed size keys. But if we want to flush the memory data back to the disk, we must reverse the mapping.

In this paper, we present S3, a scalable in-memory skip-list index for the customized version of RocksDB in Alibaba Cloud. Usability is the first-class citizen in our design. S3 can be seamlessly integrated with the disk-part of LevelDB and RocksDB to replace their old skip-list index. S3 also supports high throughput insertion, efficient lookup and range search. The performance of S3 is approaching to the state-of-the-art in-memory index, ART [26], while it can be easily extended as a disk-based index.

S3 adopts a two-layer design. In the bottom layer, we build a semi-order skip-list, where keys inside one data entry are not required to be sorted. Data entries remain ordered. Namely, the maximal key of a data entry is smaller than the minimal key of its successor. This strategy allows us to further improve the insertion performance, but may slow down the search process. To address the problem, we intentionally create some guard entries as shortcuts. Instead of starting the search from the head of the skip-list, we jump to the closest shortcut and resume our search, which significantly reduces the search overhead. In the paper, we study the effect of guard entry selection on the performance theoretically.

But searching for the proper shortcut introduces additional overhead. So in the top layer, we adopt a cache-sensitive index for fast retrieval of guard entries. Currently, we use FAST [25] as our top layer index. Other cache-sensitive indices can be adopted as well. Note that since only a few guard entries are generated, the top layer index is quite compact, allowing to buffer most of the data in L2/L3 cache. The search cost of top layer is almost free, compared to the search in the bottom skip-list.

Finally, we provide the same API as the original skip-list in LevelDB and RocksDB to flush data back to disk. We also propose a series of optimization techniques to facilitate the guard entry selection and multi-threading access. The remaining of the paper is organized as follows. In Section 2 , we briefly review some recent work on the in-memory indexing. We show our general design in Section 3 and present our key ideas in Section 4. In Section 5, we discuss some optimization techniques. Experimental evaluation results are shown in Section 6 and we conclude the paper in Section 7.

2. RELATED WORK

Key-value stores are widely used for managing large-scaled data[13, 8, 14, 10, 5]. Some key-value stores, such as Redis[12], are designed as in-memory systems for fast data retrieval. But most others are targeting at supporting huge volume disk data on top of distributed file systems. Since accessing disk data generates more overhead[39], many key-value stores adopt the log-structured merge tree (LSM) architecture[31].

LevelDB is the most popular LSM key-value stores[8]. In LevelDB, a small in-memory index is employed to support fast data insertion and lookup. When the in-memory part, MemTable, is full, we will flush it back to the disk and create a new one. The disk files, SSTables, are also organized as multiple levels. If the number of SSTables at one level reaches the predefined threshold, we will pick one SSTable and merge it with the next level SSTables. This operation is called compaction in LevelDB and may incur extremely high processing overhead.

Original LevelDB adopts a simple concurrent process strategy. It buffers all insertions into a buffer and asks one thread to conduct all insertions sequentially. Hyper-LevelDB improves the strategy by allowing concurrent updates[6]. RocksDB, on the other hand, introduces a multi-threaded merging strategy for the disk components[13]. cLSM[21] replaces the global mutex lock with a global reader-writer lock and uses a concurrent memory component. Hence, operations can proceed in parallel, but need to be blocked at the start and end of each concurrent compaction. Write amplification is also an important performance issue for LSM-like systems. The LSM-trie data structure uses tries to organize keys to reduce write amplification. However, it cannot support range queries[40]. RocksDB's universal compaction reduces write amplification by sacrificing read and range query performance[13]. PebblesDB proposes a scheme to balance the costs of write amplification and range query[33]. WiscKey directly separates values from keys, significantly reducing write amplification regardless of the key distribution in the workload[29]. Most of the above works are orthogonal to our improvements on the memory component.

Some works have been done on the memory component of LSM key-value stores. RocksDB offers two hash-based memory components as options[13]. HashSkipList organizes data in a hash table with each hash bucket as a skip-list. HashLinkList also maintains data in a hash table with each hash bucket as a sorted single linked list. The biggest limitation of the hash-based memory component is that doing scan across multiple prefixes requires copy and sort, which is very slow and memory costly. FloDB adds a small in-memory buffer layer on top of the memory component[17]. New entries are inserted into a small hash table, and then shuffled to the bottom skip-list as a background process. Our proposed in-memory skip-list index, S3, can be used to replace the skip-list in FloDB and other skip-list based index, such as the ones used in LevelDB[8] or RocksDB[13].

Besides the LSM-like index, other in-memory indices are also very popular. There have been many works trying to improve the performance of the in-memory B+-tree. CSS-tree is a cache-sensitive search tree, where nodes are stored in a contiguous memory area in order to eliminate the use of child pointers in each node[34]. CSB+-tree applies a similar idea to B+-trees and achieves cache consciousness and efficient update[35]. The effect of node size is studied in CSB+-tree and B+-tree[22, 18]. The results have shown that using node size larger than a cache line results in better search performance. Masstree[30] is a trie variant of B+-tree, which can efficiently handle keys of arbitrary length with all the optimizations presented in [34]. It is capable of providing a high query throughput[30].

To avoid the overhead of latches, Bwtree uses CAS operations and can perform substantially better than traditional B+-trees[27]. PALM adopts a bulk synchronous parallel (BSP) model to process queries in batches[36]. FAST uses a similar searching method, together with Single Instruction Multiple Data (SIMD) processing to boost the key comparison during index traversal[25]. As a result, FAST achieves twice the query throughput of PALM while be quite efficient in scan and reverse scan.

The Adaptive Radix Tree (ART) also enables SIMD processing via a customized memory layout for the internal tree nodes and achieves a comparable performance to FAST[26]. However, ART does not necessarily cover full keys or values and is primarily used as a noncovering index. It cannot be used to replace the skip-list in the LSM-based system. Nor can it be used as our top layer index, since ART does not perform well in the scan and reverse scan process[16](we must support range queries for applications).

Skip-list is originally presented as a probabilistic data structure similar to B-trees [32]. Compared with B+-tree, a skip-list shows a similar search performance, but requires much less effort to maintain. However, native linked list based implementation of skip-list has poor cache utilization due to the nature of linked lists. CSSL [37] and PI [42, 43] solve this problem by separating the index layer from the storage layer and assume that a specified number of keys can exactly occupy a whole SIMD vector. Thus, the layout of the index layer can be optimized for cache utilization at the cost of fixed key length. However, both of the structure periodically restructure the index to optimize the performance. During the restructuring process, the read and write operations are blocked.

3. OVERVIEW OF THE FRAMEWORK

To resolve the confliction between the high performance and low compaction cost to disk in existing LSM-based systems, we proposed a two-layer in-memory indexing structure. In particular, the disk part of RocksDB remains the same and we mainly change its in-memory part. Figure 1 shows the overview of our indexing structure. The top layer can be any cache-sensitive index. For efficiency and simplicity, we adopt FAST(Fast Architecture Sensitive Tree) [25] in our current implementation. The bottom layer is a variant of skip-list [32]. The intuition of using a two-layer structure is two-fold:

- The performance of skip-list is not comparable to other recently proposed in-memory indexes [30, 27].
- Most in-memory indexes do not consider the problem of how to efficiently flush memory data into disk like what we do in LevelDB [8] and RocksDB [13].

The variant of skip-list, compared to the vanilla one, maintains two types of index entries:

- data entry (denoted as de_i), maintaining serval user-entered keys and values, same as the original definition.
- guard entry (denoted as ge_i), indicating a routing key for speeding up the search process of the skip-list.

One typical optimization for lookup and insertion operations on skip-list is to buffer the historical routing paths and reuse them as much as possible. When processing future queries, we check the routing paths and reuse the index entries that are closer to the destinations.

This motivates the idea of guard entries. We materialize some popular entries in the routing paths as shortcuts,

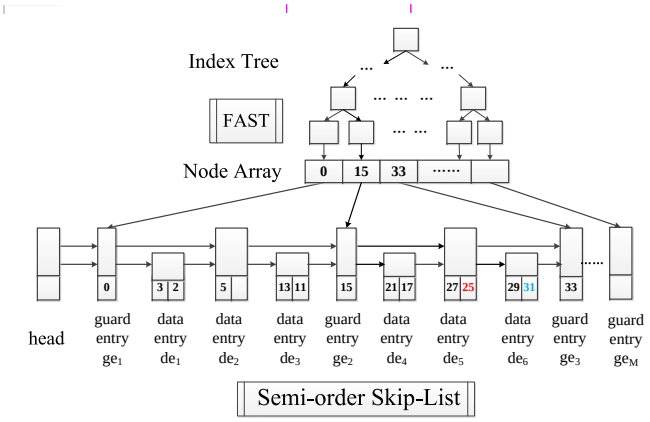


Figure 1: Two Layer Structure(Each data entry have two keys, the value is omitted)

which can potentially reduce the routing cost from $O(logN)$ to $O(log(\frac{N}{M}))$ (we will show this theoretically in the following section). However, remember that our indexing structure will be periodically flushed to disk and rebuilt (namely, freezed as a immutable MemTable and created a new one from scratch). It is challenging to decide which entry should be promoted as the guard entry during runtime. In this paper, we adopt a neural model to predict the importance of an index entry based on the data distribution and query distribution of the last index. Although it may not reflect the distributions of current index, the model is a good approximation since both data distribution and query distribution change smoothly over time. Moreover, after the current index is freezed for flushing, we will train and update the neural model using the new data. Therefore, the model will be up-to-date.

To lookup a key key_i, we need to find the guard entry that contains the largest key less than or equal to key_i. If too many guard entries are maintained, the search overhead of guard entries may compromise the performance gain. Hence, in the top layer, we use a cache-sensitive index, FAST, to support the search among guard entries. Because only few entries are used as guard entries, most FAST data can be maintained in cache. The search over FAST is almost free compared to the search over skip-list.

Another advantage of adopting the two-layer indexing structure is that it makes concurrent control over multiple threads much simpler. We limit the updates of skip-list within the data entries. So the pointer adjustments are always performed between guard entries and do not effect the data entries between adjacent guard entries. Thus, by assigning each thread to a series of guard entries, the query processing becomes latch-free.

One more optimization of our index is to employ partial order strategy to reduce the write overhead. As LSM-structures are designed to support write-intensive applications, we consider the priority of write operation over the read one. Our skip-list is organized as semi-ordered between two guard entries to further improve writing throughput. Specifically, the data entries and guard entries are kept in order, while the keys inside the data entries can be unordered. We will drill down the details of semi-order skip-list in the Section 5.1.

4. THE KEY IDEA

In this section, we use a series of examples to show how our two-layer indexing structure works. We briefly show the effect of the guard entries and our entry selection strategy. Finally, we discuss the implementation details of flushing the memory index data back as disk components, namely, SSTables in LevelDB [8] and RocksDB [13].

4.1 Basic Operators

We use the example structure in Figure 1 to demonstrate how operators, such as lookup, range query and insertion, are supported in our index. The search process is almost identical to the original skip-list, except for the processing of guard entry and semi-ordered data entry. Given a lookup request for $key_1 = 25$, we check our up-layer cache sensitive index (FAST[25] in current implementation) to find the guard entry with its key less than or equal to key_1. In Figure 1, this is the second guard entry denoted as ge_2, because the next guard entry ge_3 has a key 33, larger than key_1. We follow the links of ge_2 to route the query. Its top link points to data entry de_5 with two keys 27 and 25. We are lucky to find the exact key. Since we use semi-order skip-list, the data entry does not sort its keys. Hence, we may need to scan all keys to retrieve our requested key-value pair. This is fast, as all keys inside a data entry can be obtained via a few cachelines. If we cannot find the proper data entry in current link, we need to follow the other links of ge_2 until we find a data entry with its maximal key less than key_1 and route the query to that data entry. The search continues until we reach key_1 or report a miss after checking all possible links.

To insert $key_2 = 31$ into the index, we find the last guard entry whose key is less than or equal to key_2 (ge_2 in Figure 1). Then, we follow the top-level link of ge_2 in the skip-list to find the rightmost entry de_5 whose key is less than key_2. Then we move on to de_5 and repeats the search process in the next level. We find de_5 itself is the entry less than key_2 at the level. The search terminates, since we reach the bottom-level link. de_5's next neighbor, de_6, is our result. Since data entry does not necessary maintain its keys in order, we just append $key_2 = 31$ to the end of key list in de_6. For each data entry, we maintain its maximal key value explicitly. So if the data insertion violates the current maximal key, we also need to update it correspondingly. Currently, one data entry is allowed to maintain nL bytes key-value pairs, where L is the size of cacheline and n is a tunable parameter. Previous work assumes that both keys and values are of fixed sizes. So that specific optimization techniques can be applied to improve the cache hit ratio. In our case, real applications may pose arbitrary keys and values. Such optimizations are not valid. We just maintain keys continuously to improve the scan efficiency. Our intuition is to improve the insertion performance, while slightly sacrificing the lookup performance. This is because many of our applications are write-intensive.

For range query $[key_3 = 5, key_4 = 18]$, we retrieve the guard entries with the largest keys less than key_3 respectively via the top layer cache-sensitive index, which are ge_1 in Figure 1. Then, we traverse the semi-order skip-list to find the last data entry de_i whose $maxkey$ is less than key_3, which is de_1 in our example. To process the query, we scan from the neighbors of de_1, de_2, to de_4, whose $maxkey$ is greater or equal than key_4, to retrieve all qualified key-value pairs within key_3 and key_4. In our example, we get the keys $5, 11, 13, 15, 17$ and their corresponding values. $key = 21$ in the de_4 is not included as $21 > 18$.

4.2 Selection of Guard Entry

Apparently, the selection of guard entries plays an important role for the query performance. As said in Section 3, we suggest that both data distribution and query distribution change smoothly over time. Thus, we can select the guard entries based on the data distribution and query distribution of the last index.

In this section, we give a detailed theoretic analysis based on the assumption that both data and queries share the same distribution. Namely, queries are evenly distributed over the keys. The simple assumption surprisingly works well for most real applications. In next section, we will show our optimization technique to handle the case where data and queries are of different distributions.

We focus on two operations, lookup and insertion. In fact, the insertion operation is performed by a lookup and an update operation. For the insertion operation, when performing the lookup, we need to record the predecessor of each level in the skip-list to enable the update. After reaching the destination entry, if the entry is not full, we directly insert the key. Otherwise, we may need to create a new entry and update previous routing links for the recorded predecessors. However, note that this cost is bounded by L, the maximal level of skip-list. As a result, in the following discussion, we mainly focus on the lookup cost.

As a normal skip-list, for a newly generated entry, if it has created routing link for level i, then it will create a link for level $i + 1$ with the probability P. In Figure 1, we have $P = \frac{1}{2}$ and hence, the skip-list is actually a variant of binary search tree. In our implementation, we adopt the semi-order skip-list, where all entries share the same probability P and can only maintain at most num key-value pairs.

In our skip-list, the data has been split into θM equal-size partitions, where M is the number of guard entries and θ is a parameter for tuning the granularity. In other words, each guard entry ge_i is responsible for a few partitions denoted as $f(ge_i)$. We try to find the optimal f, so that the total lookup cost and insertion cost are minimized.

Let $(S_i, S_{i+1}]$ be the key range of the ith partition p_i. The data distribution in p_i is denoted by P_{d_i}, while the query distribution in p_i is represented as P_{q_i}. Suppose we have N data entries in total. The number of entries(N_k) maintained by guard entry ge_k can be estimated as

$$N_k = N \cdot \sum_{\forall p_i \in f(ge_k)} P_{d_i}$$

The query distribution in p_k can be estimated similarly as

$$Q_k = Q \sum_{\forall p_i \in f(ge_k)} P_{q_i}$$

where Q is the total number of queries.

If the data and query roughly share the same distribution (namely, queries are evenly distributed over data), we have $\sum_{\forall p_i \in f(ge_k)} P_{d_i} \propto \sum_{\forall p_i \in f(ge_k)} P_{q_i}$. To simplify, we use x_k and αx_k to denote $\sum_{\forall p_i \in f(ge_k)} P_{d_i}$ and $\sum_{\forall p_i \in f(ge_k)} P_{q_i}$ respectively. Thus, N_k and Q_k are represented as $N \cdot x_k$ and $\alpha N \cdot x_k$.

Based on the characteristics of skip-list, the maximal level of data entries in p_k, denoted as $h(p_k)$, is estimated as $\log_{\frac{1}{P}}(N_k)$. The level of the whole skip-list H can be computed similarly as $\max_{1 \le k \le M} h(p_k)$. Because in our case, the

guard entry always maintains the routing tables for all levels within its range, we perform at most H level switch for processing a lookup request. Assume the overhead of each switch is o_h, the level switch cost of lookup is $c_l = H \cdot o_h$.

Besides the overhead of level switch, we also need to route the request along each level within the partition p_k. Luckily, the original skip-list paper[32] gives an approximate estimation for the cost of such routing, which is roughly $l(p_k) = (\frac{1}{P} - 1) \cdot h(p_k)$. Assume the overhead of each hop along the linked list is fixed to o_l, the total overhead of routing within the partition p_k can be computed by $c_t(p_k) = l(p_k) \cdot o_l$. Take the query distribution of each partition p_k into consideration, we have the average routing overhead for our semi-order skip-list:

$$c_t = \sum_{k=1}^{M} \{(\sum_{\forall p_i \in f(ge_k)} P_{q_i}) c_t(p_k)\}$$

In other words, $c_t = \sum_{k=1}^{M} \{\alpha x_k \cdot c_t(p_k)\}$.

As we have $\sum_{k=1}^{M} x_k = 1$, we can obtain the following theorem.

THEOREM 1. *Assume the query and data follow the same distribution. Both the costs of level switch (c_l) and routing (c_t) are optimal when $x_1 = x_2 \cdot \cdot = x_M$.*

PROOF. First, we will show that c_l is optimal when $x_1 = x_2 \cdot \cdot = x_M$.

We have

$$\begin{aligned} c_l &= H \cdot o_h \\ &= \max_{1 \le k \le M} h(p_k) \cdot o_h \\ &= \max_{1 \le k \le M} \log_{\frac{1}{P}}(N_k) \cdot o_h \\ &= \max_{1 \le k \le M} \log_{\frac{1}{P}}(N \cdot x_k) \cdot o_h \end{aligned}$$

.

Obviously, c_l is minimized when $\max_{1 \le k \le M} x_k$ is minimized, and the only solution is that $x_1 = \cdot \cdot = x_M$.

Next, we show that c_t is also minimized when $x_1 = \cdot \cdot = x_M$. We have

$$\begin{aligned} c_t &= \alpha \sum_{k=1}^{M} \{x_k \cdot c_t(p_k)\} \\ &= \alpha \sum_{k=1}^{M} \{x_k \cdot l(p_k) \cdot o_l\} \\ &= \alpha \sum_{k=1}^{M} \{x_k \cdot (\frac{1}{P} - 1) \cdot h(p_k) \cdot o_l\} \\ &= \alpha \sum_{k=1}^{M} \{x_k \cdot (\frac{1}{P} - 1) \cdot \log_{\frac{1}{P}}(N \cdot x_k) \cdot o_l\} \end{aligned}$$

.

Assume $g(x_k) = x_k \cdot (\frac{1}{P} - 1) \cdot \log_{\frac{1}{P}}(N \cdot x_k)$, we have $c_t = \alpha \sum_{k=1}^{M} g(x_k) \cdot o_l$ and

$$\begin{aligned} g(x_k) &= x_k \cdot (\frac{1}{P} - 1) \log_{\frac{1}{P}}(N \cdot x_k) \\ &= x_k \cdot (\frac{1}{P} - 1)(\log_{\frac{1}{P}}(N) + \log_{\frac{1}{P}}(x_k)) \\ &= (\frac{1}{P} - 1) \log_{\frac{1}{P}}(N) \cdot x_k + x_k \cdot (\frac{1}{P} - 1) \log_{\frac{1}{P}}(x_k) \\ &= (\frac{1}{P} - 1) \log_{\frac{1}{P}}(N) \cdot x_k + x_k \cdot (\frac{1}{P} - 1)(\frac{ln(x_k)}{ln\frac{1}{P}}) \\ &= (\frac{1}{P} - 1) \log_{\frac{1}{P}}(N) \cdot x_k + \frac{(\frac{1}{P} - 1)}{ln\frac{1}{P}} x_k \cdot ln(x_k) \end{aligned}$$

.

Hence, we have $g'(x_k) = (\frac{1}{P} - 1) \log_{\frac{1}{P}}(N) + \frac{(\frac{1}{P} - 1)}{ln\frac{1}{P}} \cdot (1 + ln(x_k))$ and $g''(x_k) = \frac{(\frac{1}{P} - 1)}{ln\frac{1}{P}} \cdot \frac{1}{x_k}$. Since P is always less than 1 and $x_k > 0$, we have $f''(x) > 0$. According to Jensen inequality, $\sum_{k=1}^{M} g(x_k) \cdot o_l \ge M \cdot f(\frac{\sum_{k=1}^{M} x_k}{M}) \cdot o_l$. The equal sign only holds in the case that $x_1 = \cdots = x_M = \frac{1}{M}$. So that, c_t is optimal when $x_1 = x_2 = \cdot \cdot = x_M$. □

Based on Theorem 1, the optimal selection of guard entries, namely, the selection of function f, are equal-size partitions under the assumption that the data and queries follow the same distribution. This is a strong assumption. However, it works good for real applications. When data and queries significantly show different distributions, we will apply a neural model to predict how to select guard entries. Details of the neural model will be discussed in the next section.

4.3 Implementation Details

One naive approach is to increase the number of guard entries to improve the search efficiency. In this way, the search cost of top layer cache-sensitive index also increases, compromising the performance benefit and introducing other maintainance overhead. Our current implementation maintains a very small top layer index. The configuration depends on the hardware. Since our current servers adopts a Xeon CPU with 4MB L2 cache, we set our top layer index to be 2MB at most (50% of the L2 cache). The intuition is to guarantee that almost all top layer index can be maintained in the cache.

Our two-layer indexing structure not only speeds up the query process, but also makes it simple and elegant for concurrent processing. To process an insertion, we need to update the routing links of skip-list. Since we set the guard entries to have a full set of links for all levels, the pointer adjustments are limited within guard entries and do not effect the data entries in adjacent guard entries. Suppose we have n threads and m guard entries. The ith guard entry and data entries between the ith and $i+1$th guard entries are all handled by the $i\%n$th thread. A processing thread, on the other hand, maintains $\frac{m}{n}$ queues, one for each guard entry. All requests within the ith and $i+1$th guard entries will be maintained in its corresponding queue and processed one by one. In this case, the queries are naturally distributed and latch-free. One problem is the load imbalancing. We will discuss our optimization techniques in the next section.

Finally, similar to LevelDB and RocksDB, when the in-memory index is large enough (64MB in our case), we will freeze it as a immutable structure and recreate a new one. The immutable structure is written back to the disk as a SSTable following the same interface of RocksDB. The new index is initialized with all M guard entries and the corresponding top layer cache-sensitive index. This helps us address the cold-start problem. We have a model to predict how the guard entries should be created based on previous data and query distributions. We discuss it in our optimization section.

5. OPTIMIZATIONS

In this section, we introduce the optimization techniques adopted in the S3. In Section 5.1, we show the structure of semi-order skip-list. The neural model for guard entry

selection is discussed in Section 5.2. Finally, we show how multiple skip-lists can be supported in Section 5.3.

5.1 Semi-order Skip-List

The traditional skip-list adopts a dynamic memory allocation strategy, where new nodes are allocated in an ad-hoc way. In this strategy, the adjacent nodes of skip-list may not reside with a continuous memory area and hence resulting in a poor cache line utilization. In this section, we introduce our semi-order skip-list to address the problem, which achieves high write efficiency with slightly sacrificing the read performance.

As mentioned before, semi-order skip-list is a multiple-layer linked list consists of two types of entries: data entry and guard entry. The data entries are used to maintain the user data (keys and values) while the guard entries are used to speed up the search and insertion operations. The two types of entries are linked in the same list, and we use a boolean variable to indicate that the entry is a data entry or a guard entry.

Each guard entry only maintains an indicative key which is used to routing queries. The *entry area* of a guard entry ge_i is defined as the part of semi-order skip-list from ge_i to the next guard entry ge_{i+1} (including ge_i but not ge_{i+1}).

The data entry has two more attributes, compared to the original skip-list:

- (1)num, which indicates the current number of keys and values in the entry.
- (2)$maxkey$, which is the maximal key in the entry and can be used in the search process of semi-order skip-list. We maintain a general order for data entries. So the $maxkey$ of a predecessor should be smaller than the minimal key of its successor.

In this way, our data entry maintains a block of key-value pairs. To further improve the performance of write, we do not sort the keys inside each data entry. So the insertion is conducted as an "append" operation, which can significantly improve the throughput. Each data entry can hold at most M key-value pairs. M is a tunable parameter to balance the insertion and lookup performance. Current setting of M is 8. Moreover, since each block is stored in a continuous memory area, the scan of data entry is extremely fast via a few cacheline reads.

The penalty of semi-order skip-list is two-fold. First, we need to sort keys in a data entry before flushing it back to the disk. Fortunately, we still maintain a general order for data entries and only M key-value pairs are maintained together. Suppose there are totally N key-value pairs. It adds $N \log M$ sorting cost for the compaction process. This is an acceptable cost. Second, the semi-order skip-list may slightly slow down the search process. We can skip a data entry by comparing the $maxkey$, but if the query overlaps with the data entry, we need to read out the data to check. The details of search and insertion process are shown below (It should be noted that deletion is performed by insertion and compaction in the LSM-based structure and we omit details of deletion here.).

Before performing search queries and insertion queries, we find guard entry ge_i that has the largest key less than or equal to the user-specified key_i using the top layer index[25]. ge_i is then used as the start point of routing our queries, instead of the head node.

To search key key_i, we move along the top level of skip-list until we find the rightmost data entry whose $maxkey$ is less than key_i. If the $maxkey$ of its next entry is equal to key_i, we scan the next entry to find the requested key and values directly. Otherwise, we move to the next level of skip-list and repeat our search process progressively, until we reach the bottom level of the semi-order skip-list. We then follow the bottom level link to scan data entries one by one to find the requested key and value.

To process an insertion query for key_i, we need to perform a search first. We move along all levels of skip-list until we find the rightmost data entry whose $maxkey$ is less than key_i. Note that we need to trace the rightmost data entries in our search at every level of skip-list to support possible pointer adjustment process incurred by the insertion. We never insert user data into a guard entry, since the guard entry is only used to speed up the search and insertion operations.

Algorithm 1 Insertion($key\_and\_value\ kv$)

```
key = kv.getkey()
ge_i = Find_guard_entry(key)
x, prev = Find_less_than(key, ge_i)
next = x.getnext(0)
if next is a guard entry then
  if x is not full and x ≠ ge_i then
    insert kv into x
    adjust maxkey in x if necessary
    return
else if next is not full then
  insert kv into next
  return
generate new data entry y
adjust pointers using prev
if next is not a guard entry then
  redistribute keys and values in y and next
return
```

After find the rightmost data entries, we insert the data into the semi-order skip-list as follows.

- For the rightmost data entries x at the bottom level, we check if its next entry x_{next} at the bottom level is a guard entry.
  - If x_{next} is a guard entry, then we check if x is not full. If the answer is yes, we insert the data into x and update the $maxkey$ of x if necessary. As the x_{next} is a guard entry, insert the data into x will not cause the data redistribution between the *entry area* of x and x_{next}, which is used to maintain the semi-order characteristic of the skip-list. Thus, the data insertion completes.
  - If x_{next} is not a guard entry, then we check if x_{next} is not full. If the answer is yes, we insert the data into x_{next} and finish the insertion. As the $maxkey$ in the x_{next} is larger than or equal to key_i, inserting the data into x_{next} will not cause any data redistribution. Thus, the data insertion completes.
- In all cases if the data entry is full, we build a new entry y between x and x_{next}. The height of y is randomly generated with a probability P and the pointers are adjusted as in the traditional skip-list. Besides, if x_{next} is not a guard entry, the data in y and x_{next} need to be redistributed in order to maintain the semi-order characteristic.

Algorithm 1 shows the pseudo-code of our insertion process.

S3 also supports range queries. The difference is that range query $[key_i, key_j]$ may overlap with multiple entry areas of guard entries. First, we traverse the top-layer indexing structure to find the guard entries with the largest keys

less than key_i . Then, we traverse the semi-order skip-list to find the last data entry de_{l_i} whose $maxkey$ is less than key_i. We traverse from the next entry of de_{l_i}, denoted as de_i, until we find the first data entry de_j, whose $maxkey$ is larger than or equal to key_j, in order to find the data whose key are between key_i and key_j. The data in the de_i and de_j need to be checked to answer range query $[key_i, key_j]$. To improve the performance, we can split a range query into some sub-queries. Let $\mathcal{G} = \{ge_0, ge_1, ..., ge_{k-1}\}$ be all the guard entries between key_i and key_j. We start up k threads, one for each guard entry, and forward the query to those threads, which will invoke our range query processing algorithm concurrently. This is very effective to reduce the response time of long range search.

5.2 Neural Model for Guard Selection

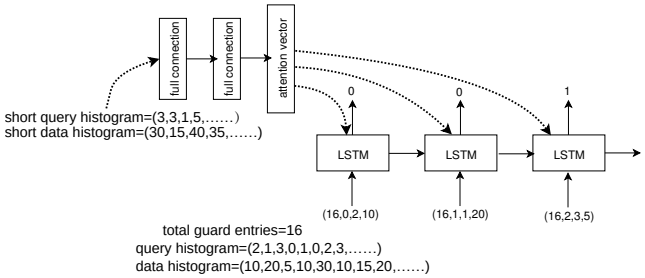


Figure 2: Neural Model for Guard Entry Selection

Our theoretic analysis for guard entry selection is based on the assumption that the data and queries follow the same distribution, which may not be true for real applications. Even if the data and queries share the same distribution, it is costly to use the mathematic equations to generate the optimal guard selection strategy for a complex distribution. Therefore, we proposed a neural model to produce an approximate result, since neural model is proved to be able to catch the data/query distribution effectively. One philosophy when designing the neural model is that the model should be small and efficient. So we do not include complex computation here.

Figure 2 shows the structure of our neural network. In particular, we adopt a simple seq2seq model [38], which is widely used for processing streaming and sequential data. The idea is to partition the whole key space into equal-size small ranges, e.g., $R = \{r_0, r_1, ..., r_n\}$, and we try to predict that whether a guard entry should be created for one small range r_i. If so, we generate a guard entry with a specific key $\frac{r_i.low + r_i.up}{2}$, where $r_i.low$ and $r_i.up$ are the lower and upper bounds of r_i respectively. To precisely catch the data and query patterns, the partitioning is conducted in a very fine granularity. In current setting, n is set as 250,000.

Since the impact of guard entries on query performance is mainly affected by the query and data distribution, our model accepts the corresponding two histograms as its inputs. Specifically, before a MemTable is flushed to the disk, we build a query and data histogram for the key range, $H_q = \{q_0, q_1, ..., q_n\}$ and $H_d = \{d_0, d_1, ..., d_n\}$. q_i and d_i denote the number of queries and keys in r_i during the lifetime of the MemTable. Then, we train our neural network to predict the optimal guard entries for the MemTable.

We consider H_q and H_d as a sequence, and each time, we will employ an LSTM [24, 23] network to do a binary classification. The result (0 or 1) indicates whether we should build a guard entry or not. One of the simple LSTM model is that when predicting a label, it does not have the knowledge about the whole data and query distribution. To address the problem, we add an attention module.

In the attention module, we cannot use H_q and H_d directly, as they are very large vectors and costly to compute in a neural network. Instead, we summarize them into short query and data histograms with only m ($m << n$, e.g., m=2000) dimensions. We get two coarse histograms, $\bar{H}_q$ and $\bar{H}_d$, which are used as inputs for the attention module. The attention module consists of two fully connected layers and one softmax layer. The output is a m-dimension vector $V = \{v_0, v_1, ..., v_m\}$. v_i is a float between 0 and 1, and we have $\sum_{i=0}^{m} v_i = 1$. V actually estimates the PDF (probability density function) of the data.

The output of attention module is linked to the last layer of the LSTM network with a weighted matrix. So when predicting the labels, the LSTM is tuned to be aware about the general data/query distribution. To better leverage the attention module, we transform the input sequence as a tuple as (c, i, q_i, d_i), where c is the total number of guard entries that we plan to generate, i is the id of the sub-range, q_i and d_i are as defined before. Note that except c, all data need to be normalized as floats between 0 and 1 to achieve a better prediction performance.

In our running scenario, we generate H_q and H_d for the last MemTable and use it to predict the guard entries for the new MemTable. In other words, the guard entries will be set up during the initial process of a new MemTable. And, we assume that the data and query distribution remain unchanged or change slowly. This is true for many real applications. Another problem is that the neural model may generate δ more or fewer guard entries than required. If more guard entries are predicted, we simply randomly delete some ones. If fewer guard entries are created, we run the model again to do the prediction by setting c as δ.

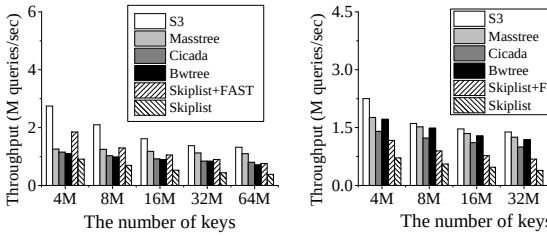
One possible optimization is to split the prediction into multiple phases during the lifetime of a MemTable, e.g., 25% full of MemTable and 50% full of MemTable. This allows us to build guard entries gradually to better catch the dynamic data and query distributions. However, due to its complex learning process, we do not implement the strategy in our running system.

To train the neural model, we use the log data to repeat the MemTable creation and flushing process. Each time a MemTable is full, we use the mathematic equations to generate an approximate result [2] of the optimal guard entry selection as our groundtruth result. We collect 50,000 training records, among which 45,000 are used as training data and 5,000 are used for testing.

5.3 Multiple Semi-order Skip-Lists

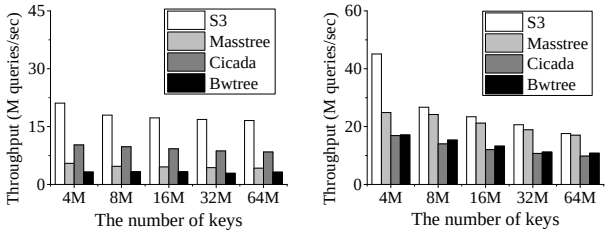
In RocksDB, multiple immutable MemTables can be created to delay the compaction and improve the memory utility. In that case, to search for a specific key, we need to check all corresponding skip-lists, before we switch to the disk search. This does not add additional overhead to RocksDB. But for the S3, since we maintain a top-layer cache-sensitive index for fast guard entry search, the search becomes complicated. The top-layer index may become the bottleneck.

[2]The problem of finding optimal selection for an arbitrary distribution is an NP-complete problem. We discard the details here.



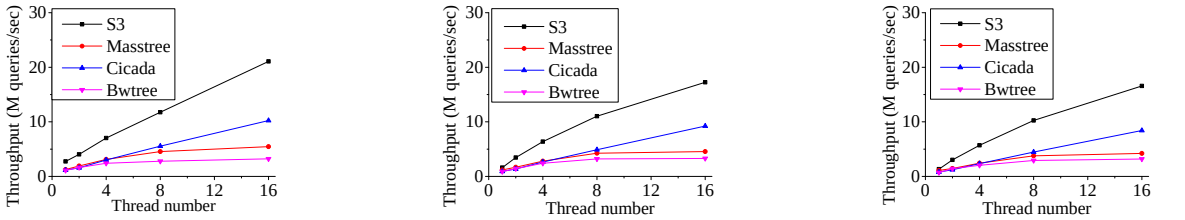
(a) Insertion (b) Lookup

Figure 3: Query Throughput(uniform workload,single thread)



(a) Insertion (b) Lookup

Figure 4: Query Throughput(uniform workload)



(a) The number of keys=4M (b) The number of keys=16M (c) The number of keys=64M

Figure 5: Query Throughput(Insertion,uniform workload)

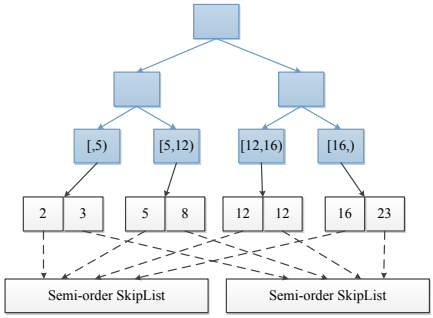


Figure 6: Top-Layer Index for Multiple Skip-Lists

A simple solution is to maintain one top-layer index for each semi-order skip-list. In this way, our assumption about the top-layer index (almost all top-layer index can be maintained in L3 cache) does not hold any more. When searching different skip-lists, we will occasionally swap in/out the data of different top-layer indices, resulting in low cache utility. Therefore, instead of maintaining an individual top-layer index for each skip-list, we build a global one for all skip-lists. Figure 6 illustrates the idea by using the FAST as our index structure.

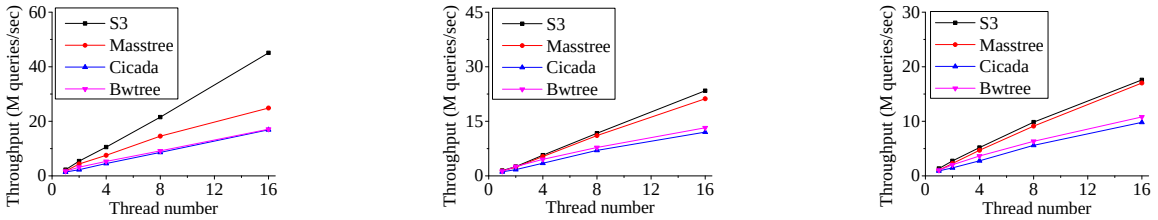
The inner nodes of the FAST remain unchanged. The leaf nodes are used to maintain keys within a key range. Those keys refer to guard entries of different skip-lists. To search a key k_i, we first locate the leaf node responsible for the overlapped key range. Then, we scan the corresponding leaf node to find the guard entries with maximal keys smaller than k_i. Normally, we will obtain one guard entry for each skip-list. The search can continue in two strategies. One is that we first jump to the newest skip-list to start our search. If no result is found, we switch to the next newest one, until all skip-lists have been searched. This one reduces the total search cost. The second strategy is to issue the query for all skip-lists, which can leverage the concurrent search capability to improve the performance. Our current implementation adopts the second strategy.

In Section 5.2, we show that our guard entries are created during the initialization of a skip-list. So during the lifetime of a skip-list, the guard entries are static. Moreover, we ask the user to predefine the maximal number M of MemTables (skip-lists) that can be maintained concurrently (default value is 1). So we have all information about guard entries of the M skip-lists. We sort them altogether and build our top-layer index in a batch way from the bottom to the top. The only update happens, when one or more MemTables are flushed to the disk. In that case, the top-layer index will be rebuilt once the new skip-list completes its initialization.

6. EXPERIMENTS

We conduct extensive experiments to evaluate the performance of our approach. First, we show the lookup and write throughputs with different number of keys. Then, we vary the number of threads to investigate the concurrent throughput. Besides, we use a multi-center zipfian distribution workload to study the effectiveness of our neural model for guard entry selection. We also show the performance of our index for processing range queries. Afterwards, we test the performance of minor compaction, namely the cost of flushing the S3 index to the disk as a SSTable. Finally, we show the overall performance of the whole RocksDB embedded with our in-memory skip-list index. All experiments are conducted on a server equipped with Intel Xeon Processor E5 2660 v2(25M Cache, 2.20 GHz).

For in-memory comparison, we show the results of Cicada[28], Masstree[30] and Bwtree[27] under the same experiment settings. Both Cicada and Masstree codes are retrieved from their authors' open sourced implementations[3][9] . BwTree[2] does not provide an official implementation. Therefore, we retrieve an implementation from the code of Peloton[11]. Since ART does not necessarily cover full keys or values and is primarily used as a non-covering index, we did not include it in our current evaluation. We did not include FAST in our evaluation too, as it is optimized for lookup queries only. The key length is four bytes for all in-memory indexes. The query workloads are generated using a C-implementation[15] of Yahoo's YCSB benchmark[19]. The generated queries are persisted in the disk and read into memory for experiments in order to ensure the same query workload for all in-memory tests. To evaluate the



(a) The number of keys=4M (b) The number of keys=16M (c) The number of keys=64M

Figure 7: Query Throughput(Lookup,uniform workload)

improvement of RocksDB with our in-memory skip-list index, we compare the overall throughput equipped with and without our in-memory skip-list index using the benchmark provided by RocksDB.

6.1 Effect of Data Size

In this experiment, we show the effect of the numbers of keys. We limit the search to memory for a fair comparison, so the maximal number of keys is 64 million. The keys in the workload follow a uniform distribution.

First of all, we use single thread for processing search and insertion. The results are shown in Figure 3. In the following, we use "Skiplist+FAST" to represent the index structure which uses the traditional skip-list as the bottom layer and FAST for the upper layer. Obviously, our two-layer in-memory indexing structure S3 performs better than the Masstree, Cicada and Bwtree. Besides, S3 achieves almost 2X better than traditional skip-list and 1X better than the "Skiplist+FAST". Due to the relatively poor performance of traditional skip-list and the "Skiplist+FAST", we omit them in the following experiments.

By default, we start 16 threads for processing search and insertion in the following experiments.

As shown in Figure 4, our two-layer in-memory indexing structure S3 performs better than the Masstree and Bwtree, especially for insertions. For memory-only processing, all indices show a better performance for the lookup operations than the insertion operations. In skip-list based indices, insertion is performed by a lookup and routing link restructure process. In the tree-like index, tree structure also needs occasionally restructured. The advantage of skip-list approaches for write-intensive workload is when data need to be flushed to the disk, the LSM(log structured merge tree) can delay the high I/O operations into the compaction process.

Finally, both of the insertion and lookup throughput decrease as the number of keys increases from 4 millions to 64 millions. The reason is that as the number of keys increases, less index entries can be maintained in the cache, resulting in low throughput. This also explains why the throughput gap shrinks as the number of keys increases.

6.2 Concurrent Test

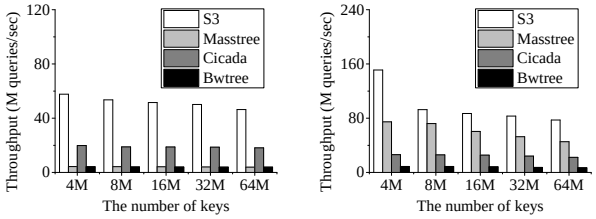
In this section, we examine the query throughput with varies number of processing threads. We test three cases, where the number of keys is 4 millions, 16 millions and 64 millions respectively. The results are shown in the Figure 5 and 7. The keys in the workload follow a uniform distribution.

We can see that both lookup and insertion throughput increase as the number of threads increases, but among all four approaches, S3 achieves the best speedup, showing its support for highly concurrent processing.

For the read-only query, almost all the index achieves a linear scale-up for small number of keys. As the number of threads and keys increase, the read-only query throughputs of Bwtree and Cicada gradually converge. Even if the number of keys reaches 64 millions, which is quite large for the in-memory structure of LSM-based systems, the read-only query throughput of S3 is still better than the other three.

For the insertion operation, the gap between S3 and the other two approaches becomes larger than the lookup case. This is due to the usage of semi-order skip-list, which further reduces the cost of insertion. Another reason is that the structure of S3 is latch free, so it can support high concurrent processing.

6.3 Result on Skewed Distribution



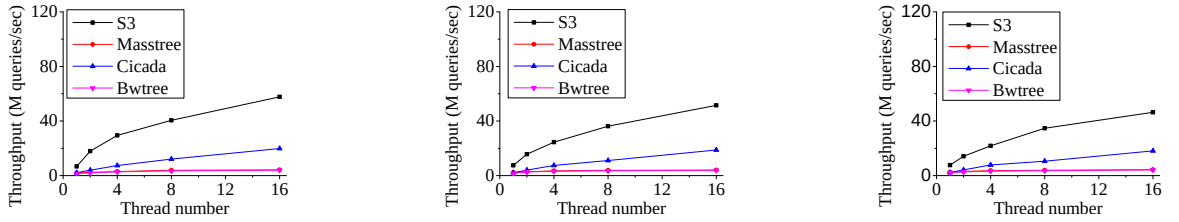
(a) Insertion (b) Lookup

Figure 8: Query Throughput(complex workload)

In this section, we examine the query throughput in a skewed distribution, where both of the data distribution and query distribution follow a multi-center zipfian distribution with the probability parameter $\theta = 0.99$. In particular, we random select some keys as our centers and generate a normal zipfian distribution for each center. We then merge those distributions into one complex distribution. The data and query distribution are generated separately with different center sets.

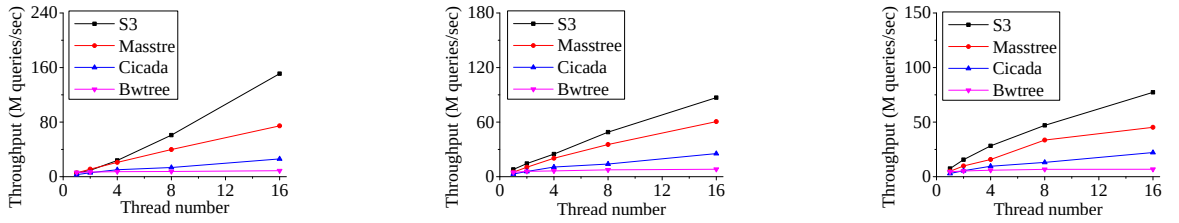
The experiment results are shown in the Figure 8. We get the same observation as in the uniform distribution case. S3 still has a much better performance than the Masstree, Cicada and Bwtree. Since the query and data follow different distributions, the guard entries in S3 are selected by the neural model. We will show the effect of different guard entry selection approaches in the Section 6.5.

The concurrent results are shown in the Figure 9 and Figure 10 respectively. We can see that given a complex workload, the speedup of lookup operation is better than the insertion operation. This is because insertion incurs high processing overhead and may trigger the routing link updates for the skip-list. Compared to the uniform case (Figure 5 and Figure 7), the scalability drops for the complex distribution. This is because the selection of guard entries is more difficult and different guard entries may be responsible for varied numbers of keys, resulting in imbalanced workloads between threads.



(a) The number of keys=4M (b) The number of keys=16M (c) The number of keys=64M

Figure 9: Query Throughput(Insertion,complex workload)



(a) The number of keys=4M (b) The number of keys=16M (c) The number of keys=64M

Figure 10: Query Throughput(Lookup,complex workload)

6.4 Mixed Workload

In this section, we examine the query throughput in the mixed workloads. Four datasets are used during the experiment and the number of threads is 16 . The ratio of write operations to the whole query set is set as 20%, 40%, 60%, 80%, respectively. The number of insertion is fixed to 64 millions. The keys in the workload follow a uniform distribution.

As shown in Figure 11, with the increase of write ratio in the query workload, the performance of S3 drops slightly, demonstrating S3's capability in processing uniform query workload. Masstree, Cicada and BwTree also experience a similar variance trend in the query throughput.

6.5 Guard Entry Selection

In this section, we study the effect of the number of guard entries first. We set the total number of keys to 64 millions, and the number of guard entries varies from 2^{19} to 2^{23}. The keys in the workload follow a uniform distribution. 16 threads are started up for processing during the experiment. In the experiment, we first insert 64 million keys and collect the total insertion time of each module. Then, we perform 64 million times random lookup to test the lookup time of each module. The result is shown in Figure 12. We show the accumulative time of each module for processing all 64 million insertions and lookups.

We can see that as the number of guard entries increases, the search cost of top-layer index increases, while the search cost of bottom-layer index decreases. The total search cost is optimal, when guard entry number is set to 2^{22}.

In the next test, we use three workloads(uniform workload, Gaussian distribution with $\sigma = 10^7$, and zipfian distribution with $\theta = 0.5$) to examine our guard entry selection strategy. To examine the effectiveness of our neural model for guard entry selection, we compare with the other two strategies. One is to randomly select a key to create a guard entry. The other one is to uniformly generate guard entries among all keys. The result is shown in Figure 13. For the uniform distribution, three guard selection approaches do not differ much. This is consistent with our expectation. But for Gaussian distribution and zipfian distribution, the neural model based guard entry selection shows significantly improvement over the other two approaches.

6.6 Range Query Performance

In this test, we study the performance of range query. The number of keys varies from 4M to 64M. The query range varies from 0.0001% to 0.1% key range. The keys in the workload follow a uniform distribution. And we totally issue 4 million random range queries. 16 threads are used during the processing.

We split the cost into two parts, lookup cost and scan cost. The former denotes the time of locating the first key, while the latter is the time of scanning the index to retrieve all key-value pairs in the range. Although we do not maintain a strict key order in the skip-list, the general order between data entries are still valid. So the range search is performed as a normally skip-list.

We first fix the number of keys to 4 millions and the key range varies from 0.0001% to 0.1%. The result is shown in Figure 15. Since the key number remains the same, the lookup cost does not change much. But the scan time increases gradually and plays a more important role in the total time.

In the second test, we fix the key range to 0.0005% and vary the number of keys from 4 millions to 64 millions. The result is shown in Figure 16. Both the lookup time and the scan time increases as the number of keys increases. It seems that the lookup cost is more sensitive to the number of keys. This is because the search cost of skip-list is $O(N \log N)$, while the scan cost is always linear to the number of keys.

6.7 Cost of Flushing Data as SSTables

In this section, we examine the efficiency of flushing the data from S3 to the disk part of RocksDB. We compare the performance of S3 with the original skip-list using the interface of RocksDB, together with the best case that sorted entries are kept in the contiguous memory space(which is abbreviated as "Full order"). The number of keys varies from 4 millions to 64 millions, and the keys in the workload follow a uniform distribution. The result is shown in Figure 17. Note that in the test, we only use a single thread for data flushing. If multiple threads are employed, the performance can be further improved.

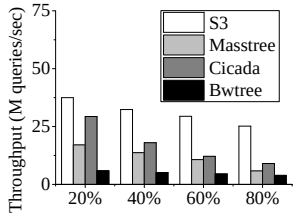


Figure 11: Query Throughput

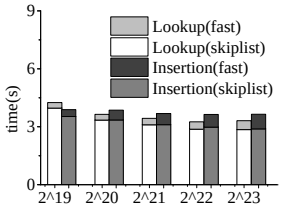


Figure 12: Query Throughput(64M Queries)

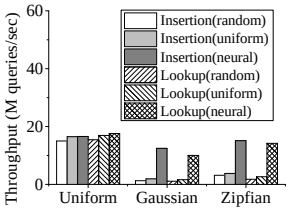


Figure 13: Query Throughput(64M Queries)

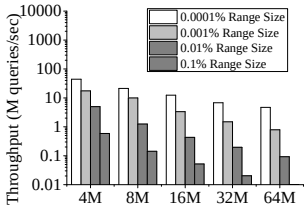


Figure 14: Range Query Throughput

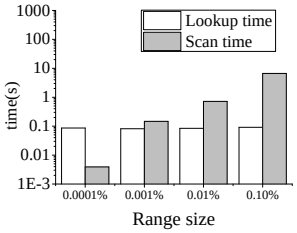


Figure 15: Range Query Time for 4M Queries

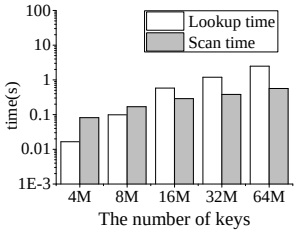


Figure 16: Range Query Time for 4M Queries

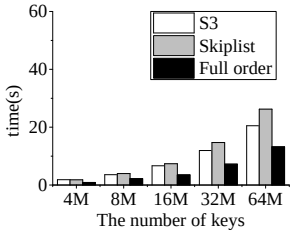


Figure 17: Cost of Writing SSTables

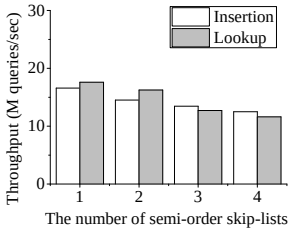


Figure 18: Query Throughput

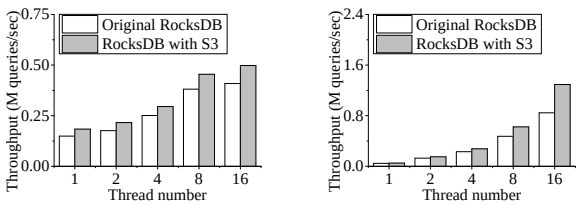
We can see that the flushing efficiency of S3 is even a little bit better than the original skip-list, despite of the additional overhead of re-sorting the data in the data entries. The reason is as follows. In a traditional skip list, each data entry maintains one key-value pair. Data entries are created on the fly and hence, keys do not reside within a contiguous memory area. Non-contiguous memory access causes high cache misses. In contrast, S3 maintains several keys in a data entry which are grouped in one continuous memory area and can be obtained via a few cachelines, making the access more efficient. S3 needs to sort the data before flushing them back to the disk. However, since the data are actually partially ordered, the cost is acceptable compared to the benefit of using continuous storage.

6.8 Multiple Semi-order Skip-Lists

In this section, we study the performance when multiple semi-order skip-lists are created during the processing, which is described in the Section 5.3. Each semi-order skip-list contains 64 million keys. The number of semi-order skip-list varies from 1 to 4 and we show the results in Figure 18.

If we try to maintain more semi-order skip-lists in memory, the in-memory query throughout decreases, especially for the lookup workload. This is no surprise. But remember that when more data are maintained in memory, we actually reduce the processing cost of disk parts of RocksDB. So the total cost of the whole system is still improved.

6.9 Overall Performance



(a) Insertion (b) Lookup

Figure 19: Overall performance

Lastly, we show the effect of our in-memory skip-list index integrated with RocksDB. We first insert 100 million entries. Then, we perform 100 million times random lookup. For the sake of fairness, we mainly use the default setting of "db_bench" benchmark provided by RocksDB, where each key-value entry has a key size of 16 bytes and a value size of 100 bytes(50 bytes after compression). The maximum number of concurrent background compactions that can occur in parallel is set to 16. The maximum number of concurrent background flushes that can occur in parallel is set to 4. The results are shown in the Figure 19. Note that RocksDB is a complex system and its performance relies on many different modules. Optimizing one module may not result in a huge improvement. However, we can see that both of the RocksDB's insertion and lookup performance are improved by the equipment with our in-memory skip-list index, due to the enhancement of in-memory throughput.

7. CONCLUSIONS

In this paper, we propose S3, a scalable in-memory skip-list index for disk-based key-value store. S3 is built as an in-memory part of the LevelDB and RocksDB, which can be seamlessly integrated into such systems. The intuition of S3 is to create some guard entries as the shortcuts in the skip-list to speed up the search process. S3 is designed as a two-layer index. The top layer is a cache-sensitive index used for fast retrieval of guard entries. The bottom layer is a semi-order skip-list, where keys inside a data entry are not required to be sorted. The semi-order skip-list enables our bottom layer to achieve high write performance while slightly sacrificing the read performance. To process a request, the top layer will return a guard entry, and we start our search in the bottom layer from the guard entry, instead of the head of the skip-list. We built a mathematical model to analyze the selection of guard entries when queries and data follow a similar distribution. For more complex scenarios, a neural model is proposed to generate an approximate strategy. We conduct performance study on a series of workloads. The results show that our proposed index, S3, is more efficient than Masstree, Cicada and Bwtree. It can also flush the memory data as a SSTable efficiently. As a result, the performance of RocskDB can be improved by the equipment with our in-memory skip-list index.

8. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable suggestions and opinions. Besides, this work was supported by the Fundamental Research Funds for Alibaba Group through Alibaba Innovative Research (AIR) Program, the Central Universities (2018FZA5015), NSFC (61661146001, 61872315) and ZJNSF(LY18F020005).

9. REFERENCES

[1] Apache hbase. http://hbase.apache.org/.
[2] Bw-tree. https://github.com/wangziqi2013/bwtree.
[3] Cicada. https://github.com/efficient/cicada-engine.
[4] Clht. https://github.com/lpd-epfl/clht.
[5] Cockroachdb. https://github.com/cockroachdb/cockroach.
[6] Hyperleveldb. https://github.com/rescrv/hyperleveldb.
[7] Judy arrays. http://judy.sourceforge.net/.
[8] Leveldb. http://ccrma.stanford.edu/ jos/bayes/bayes.html.
[9] Masstree. https://github.com/kohler/masstree-beta.
[10] Mongodb. https://www.mongodb.com.
[11] Peloton. https://github.com/cmu-db/peloton.
[12] Redis. https://redis.io/.
[13] Rocksdb. http://rocksdb.org.
[14] Search results apache flink: Scalable stream and batch data processing. https://flink.apache.org.
[15] Yahoo! cloud serving benchmark in c++. https://github.com/basicthinker/ycsb-c.
[16] V. Alvarez, S. Richter, X. Chen, and J. Dittrich. A comparison of adaptive radix trees and hash tables. In *ICDE*, pages 1227–1238, 2015.
[17] O. Balmau, R. Guerraoui, V. Trigonakis, and I. Zablotchi. Flodb: Unlocking memory in persistent key-value stores. In *EuroSys*, pages 80–94, 2017.
[18] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *SIGMOD*, pages 235–246, 2001.
[19] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.
[20] T. David, R. Guerraoui, and V. Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *ASPLOS*, pages 631–644, 2015.
[21] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar. Scaling concurrent log-structured data stores. In *EuroSys*, pages 32:1–32:14, 2015.
[22] R. A. Hankins and J. M. Patel. Effect of node size on the performance of cache-conscious b^+-trees. In *SIGMETRICS*, pages 283–294, 2003.
[23] S. Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(2):107–116, 1998.
[24] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.
[25] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: fast architecture sensitive tree search on modern cpus and gpus. In *SIGMOD*, pages 339–350, 2010.
[26] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *ICDE*, pages 38–49, 2013.
[27] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The bw-tree: A b-tree for new hardware platforms. In *ICDE*, pages 302–313, 2013.
[28] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *SIGMOD*, pages 21–35, 2017.
[29] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *TOS*, 13(1):5:1–5:28, 2017.
[30] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, pages 183–196, 2012.
[31] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996.
[32] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.
[33] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *SOSP*, pages 497–514, 2017.
[34] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *VLDB*, pages 78–89, 1999.
[35] J. Rao and K. A. Ross. Making b+-trees cache conscious in main memory. In *SIGMOD*, pages 475–486, 2000.
[36] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. PALM: parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. *PVLDB*, 4(11):795–806, 2011.
[37] S. Sprenger, S. Zeuch, and U. Leser. Cache-sensitive skip list: Efficient range queries on modern cpus. In *IMDM*, pages 1–17, 2016.
[38] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.
[39] G. Wu, X. He, and B. Eckart. An adaptive write buffer management scheme for flash-based ssds. *TOS*, 8(1):1:1–1:24, 2012.
[40] 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 *USENIX*, pages 71–82, 2015.
[41] Z. Xie, Q. Cai, G. Chen, R. Mao, and M. Zhang. A comprehensive performance evaluation of modern in-memory indices. In *ICDE*, pages 641–652, 2018.
[42] Z. Xie, Q. Cai, H. V. Jagadish, B. C. Ooi, and W. Wong. PI : a parallel in-memory skip list based index. *CoRR*, abs/1601.00159, 2016.
[43] Z. Xie, Q. Cai, H. V. Jagadish, B. C. Ooi, and W. Wong. Parallelizing skip lists for in-memory multi-core database systems. In *ICDE*, pages 119–122, 2017.