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

Jingtian Zhang et al. VLDB'19

2024.01.24

Presentation by Boseung Kim, Yeongyu Choi

bskim1102@dankook.ac.kr, dusrb1418@naver.com

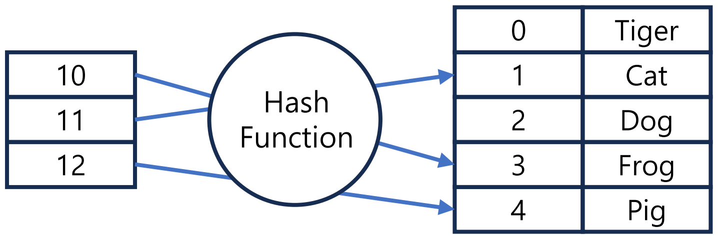
DANKOOK UNIVERSITY

Dankook University
System Software Laboratory

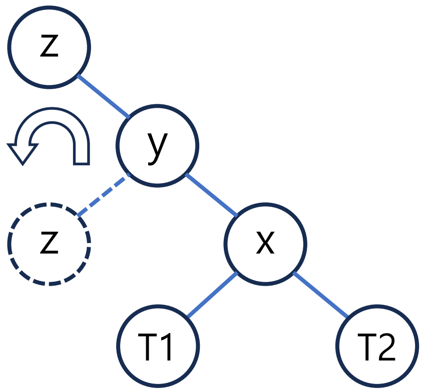
Contents

Introduction

- Existing in-memory indexing structures
  - Hash based:
    - HashSkipList, HashLinkList
    - Do not maintain the keys in order
    - Do not support range query



  - Tree based:
    - B+Tree, MassTree
    - Require complex operations to keep balance



Introduction

- Existing in-memory indexing structures
  - Skip-list based:
    - CSSL, PI
    - Restructure the index to optimize the performance
    - During the restructure process, R/W operations are blocked

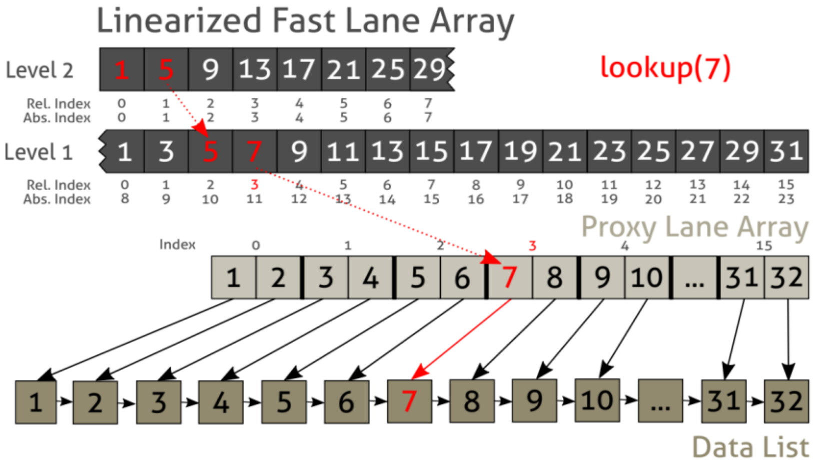


Fig. 2. A Cache-Sensitive Skip List that manages 32 keys with two fast lanes ($p = 1/2$).

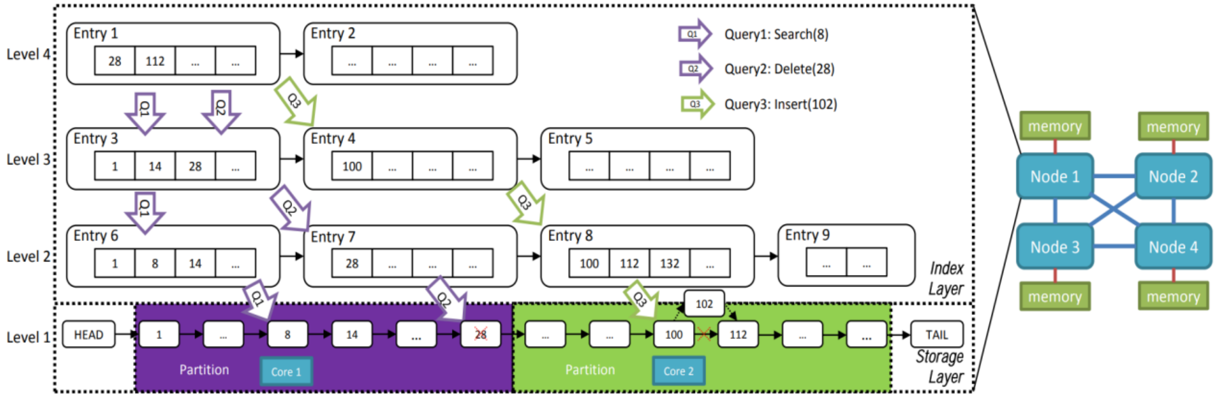


Figure 1: An instance of PI

Introduction

RocksDB

LEVELDB

APACHE HBASE

Using Skip-List for in-memory indexing structure. Because …

- Its Maintenance cost is low
- It can be efficiently flushed to disk

Introduction

RocksDB

LEVELDB

APACHE HBASE

Using Skip-List for in-memory indexing structure. Because …

- Its Maintenance cost is low
- It can be efficiently flushed to disk

But…

- Skip-list performance is worse than other recently proposed in-memory indexes
- Most in-memory indexes do not consider how to efficiently flush data into disk

Architecture of S3

- S3
  - Top Layer
    - Cache-sensitive index
      - FAST(Fast Architecture Sensitive Tree)
  - Bottom Layer
    - Semi-order skip-list
      - Guard Entry
      - Data Entry

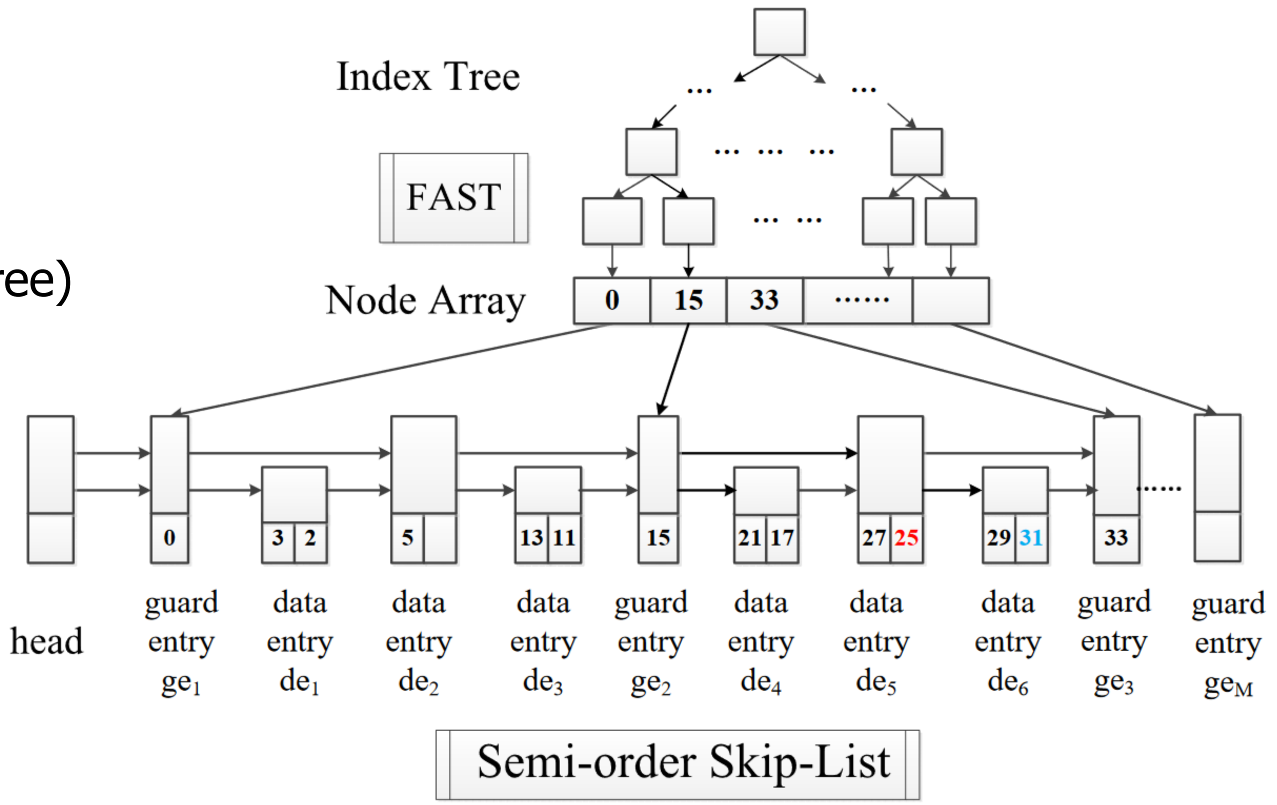


Figure 1: Two Layer Structure(Each data entry have two keys, the value is omitted)

Top Layer of S3

- FAST
  - Architecture sensitive layout of the index tree
  - Using SIMD to boost the key comparison during index traversal
  - Small enough to always be in cache

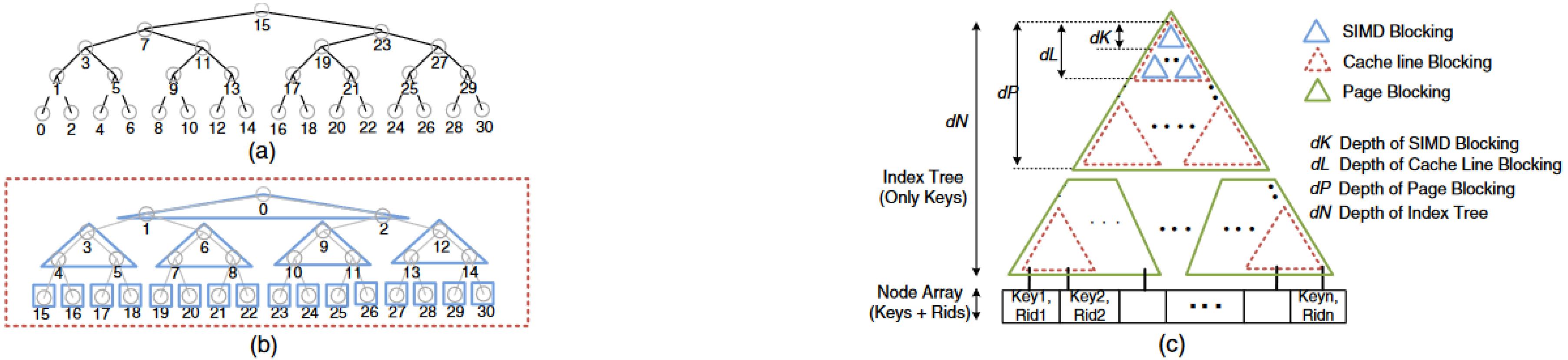
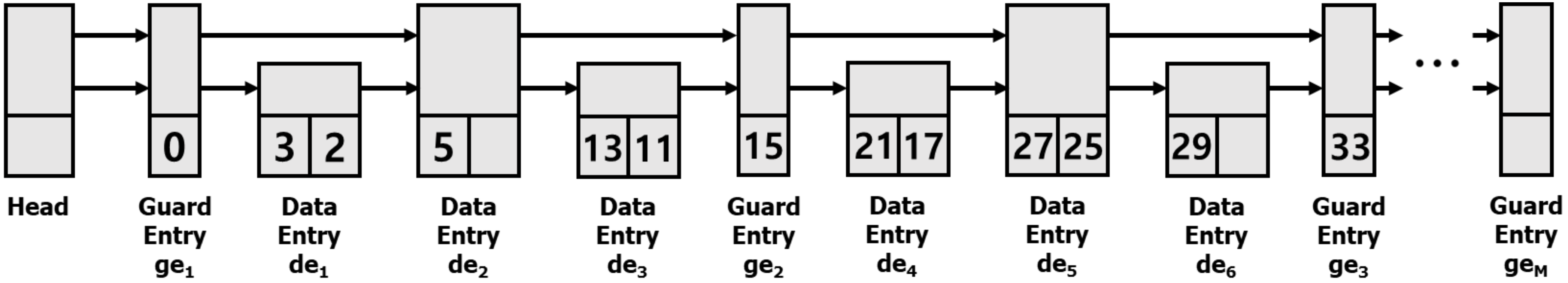


Figure 1: (a) Node indices (=memory locations) of the binary tree (b) Rearranged nodes with SIMD blocking (c) Index tree blocked in three-level hierarchy – first-level page blocking, second-level cache line blocking, third-level SIMD blocking.

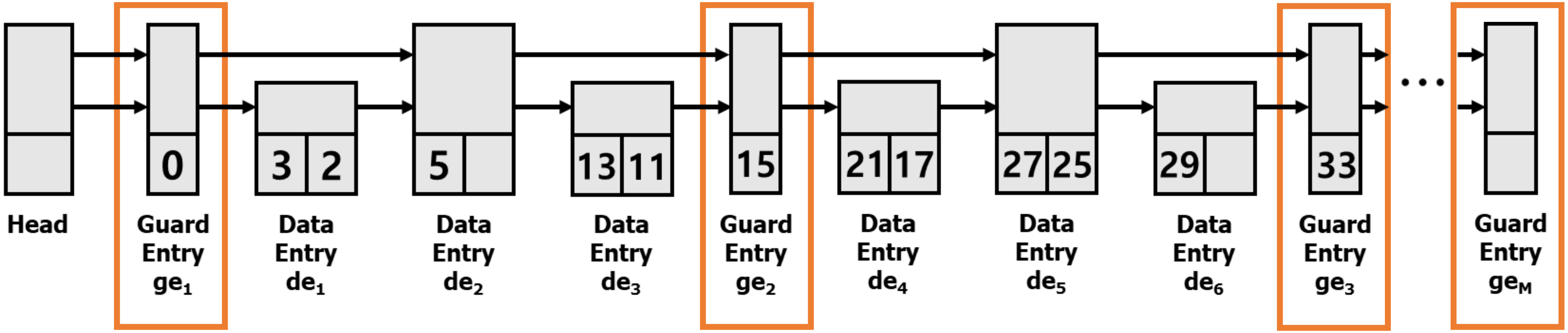
Bottom Layer of S3

- Semi-order Skip-List
  - Two types of entries: Guard Entry, Data Entry
  - Using guard entries as short-cut
  - Maintaining general order for data entries



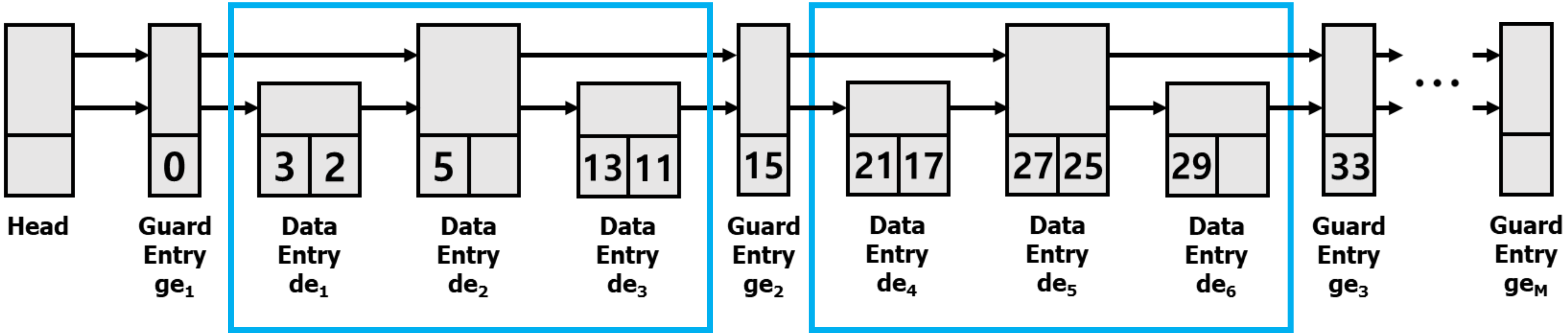
Architecture of S3

- Semi-order Skip-List
  - Guard Entry
    - Indicating a routing key for speeding up the search processing
    - Guard entries are created during the initialization of the index structure



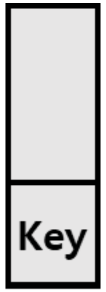
Architecture of S3

- Semi-order Skip-List
  - Data Entry
    - Maintaining the user data
    - Keys in a data entry are not sorted, append keys to the end of list
    - Continuous memory area



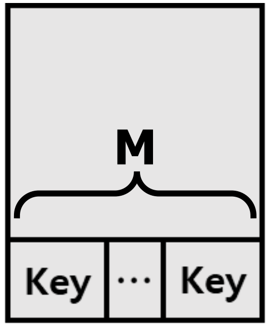
Architecture of S3

- Semi-order Skip-List
  - Guard Entry
  - Data Entry



Guard Entry
ge_k

- Boolean variable
- Indicative Key
- Pointer

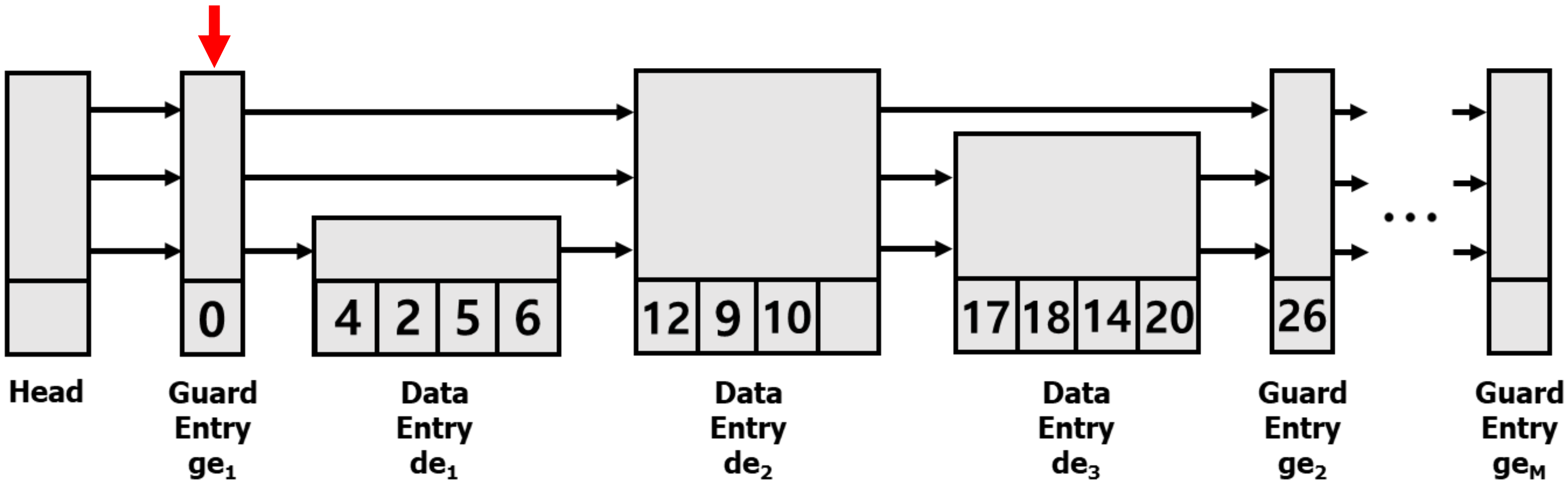


Data Entry
de_k

- Boolean variable
- Indicative Keys
- **Num**
- **Max_key**
- Pointer
- Value

Architecture of S3

- Penalty of Semi-order Skip-List
  - Keys in a data entry are not sorted
    - Slightly slow down the search process
    - Sorting the keys before flushing



Architecture of S3

- Selection of Guard Entry

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* $x_1 = x_2 \cdots = x_M$.

- The lookup cost is optimal when the sum of the data distributions of the partitions that each guard entry is responsible for are all equal

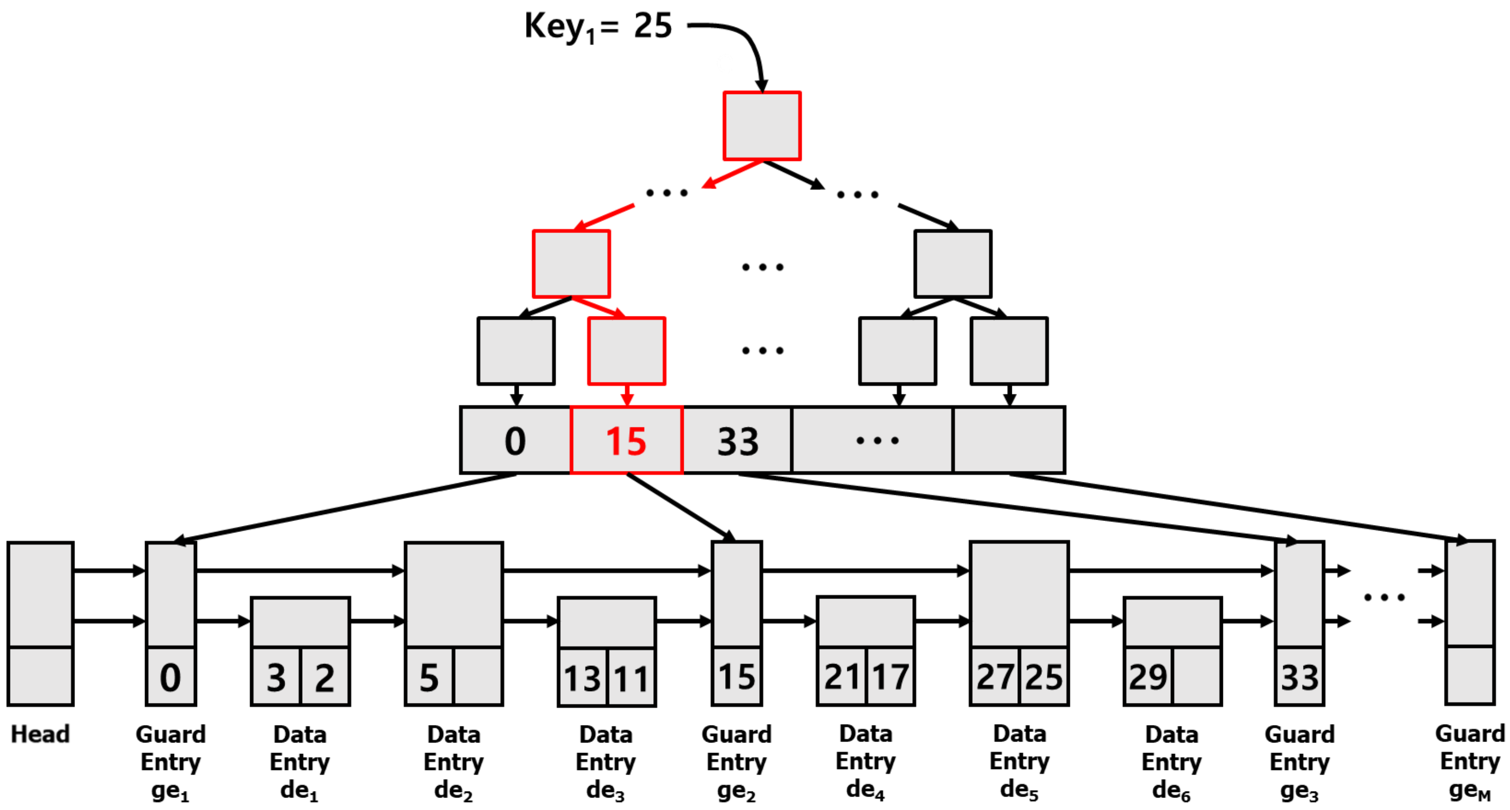
Implementation Details

- Size of Top layer Index
  - Keep the top layer index(FAST) small enough to always be in cache
- Concurrent Processing
  - Threads are assigned to manage specific guard entries and data entries
- Cold-Start Problem
  - Predict which guard entries should be created based on the previous index

Basic Operators

- Search

  1. Find rightmost guard entry ge_i less than key_i

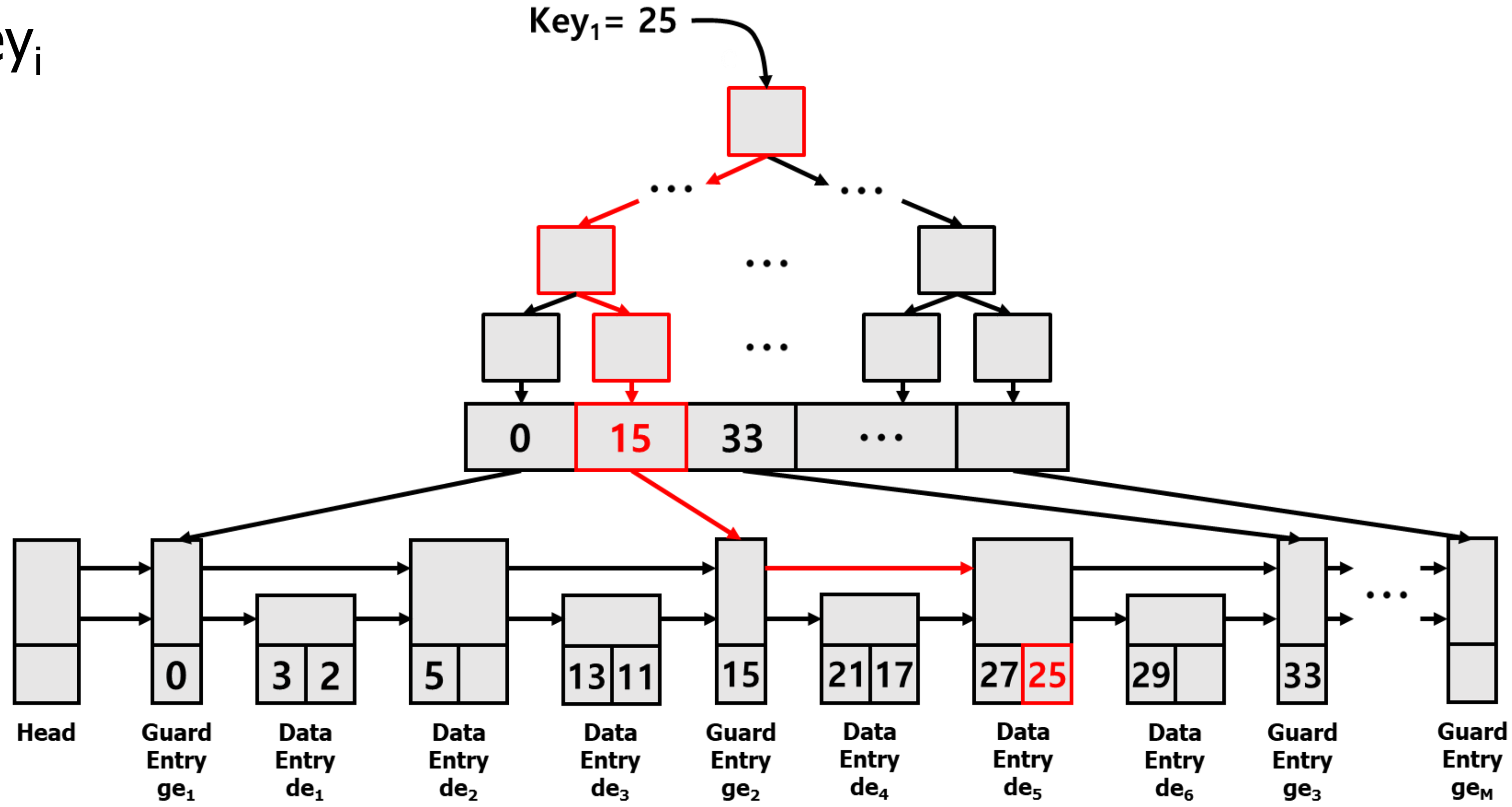


Semi-order Skip-List

- Search

  2. Find rightmost data entry which max_key is less than key_i

  3. Follow the link to find key_i



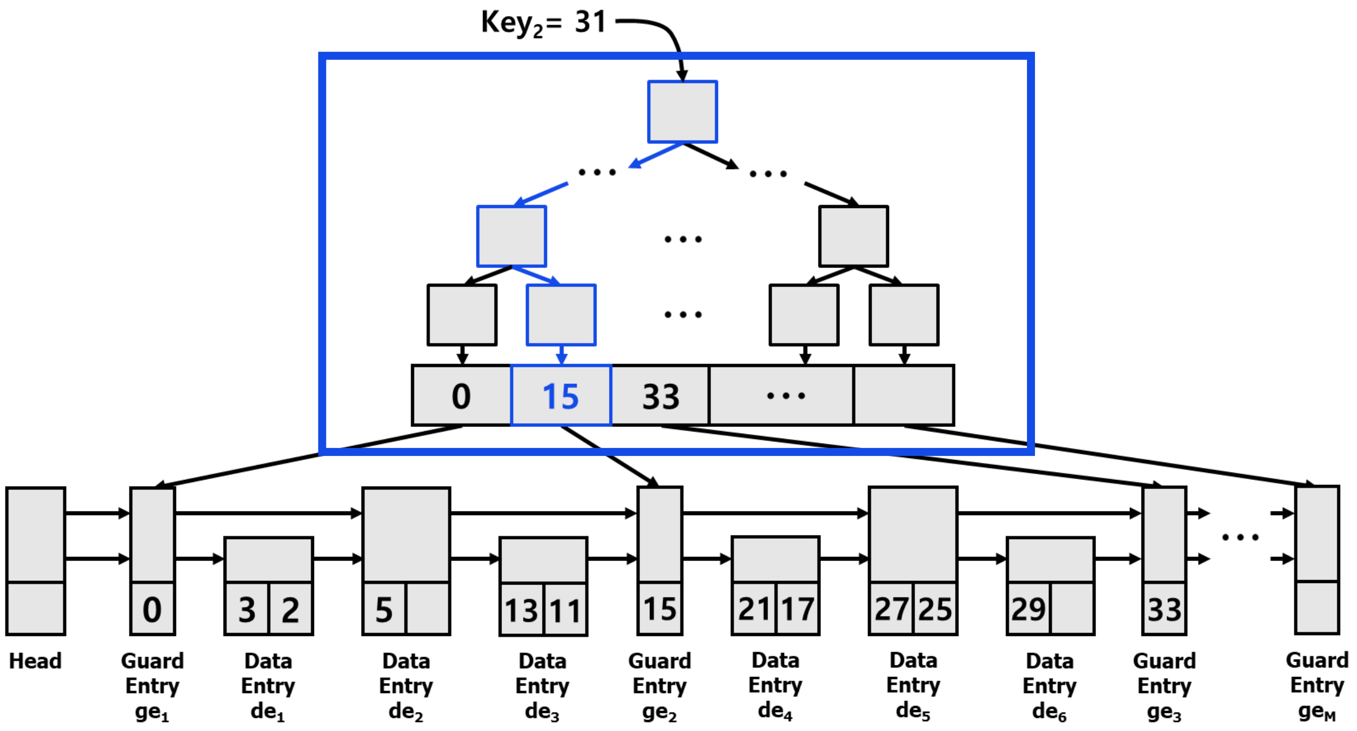
Basic Operators

- Insert

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

$\boxed{\begin{array}{l} key = kv.getkey() \\ ge_i = Find\_guard\_entry(key) \end{array}}$

$x, prev = Find\_less\_than(key, ge_i)$
$next = x.getnext(0)$
if $next$ is a guard entry **then**
  if x is not full **and** $x \neq ge_i$ **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

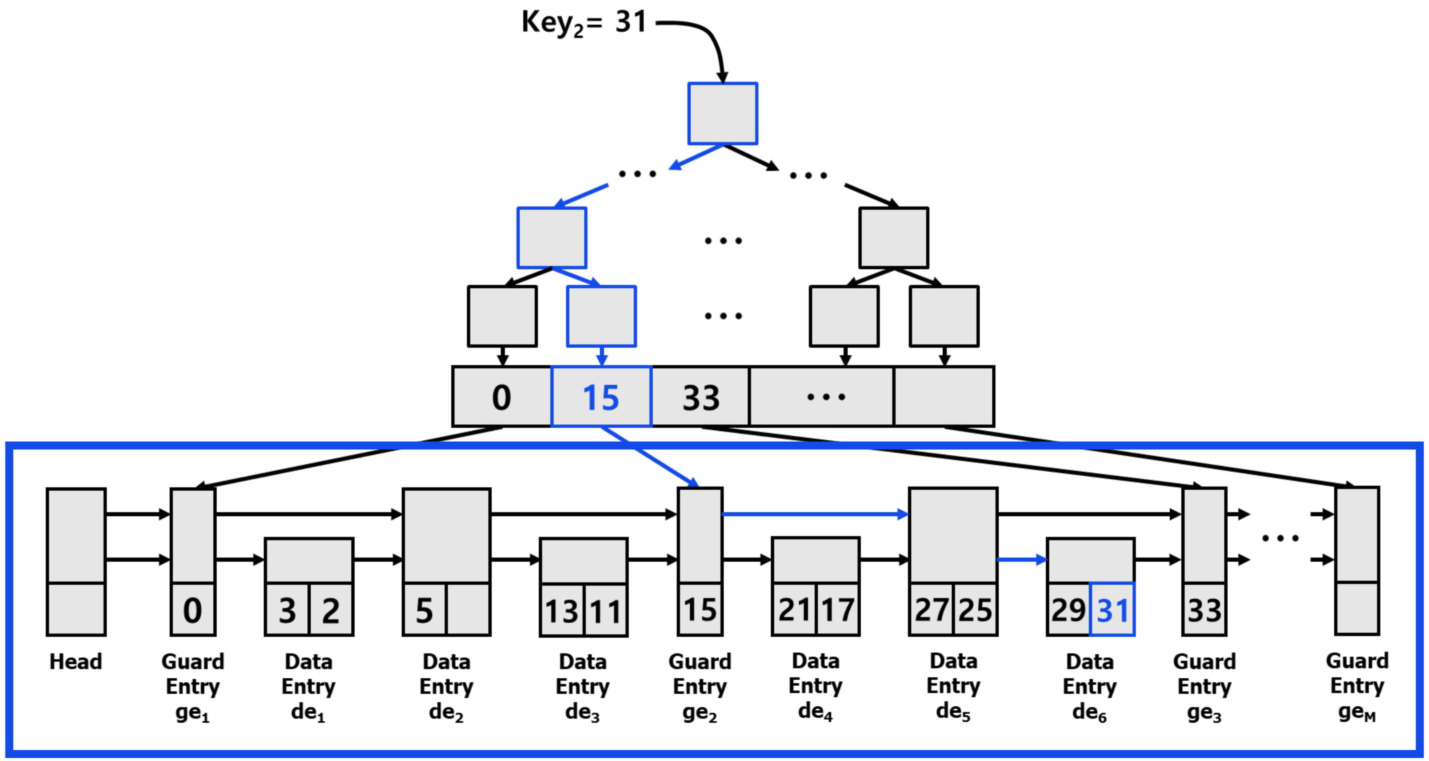


Basic Operators

- Insert

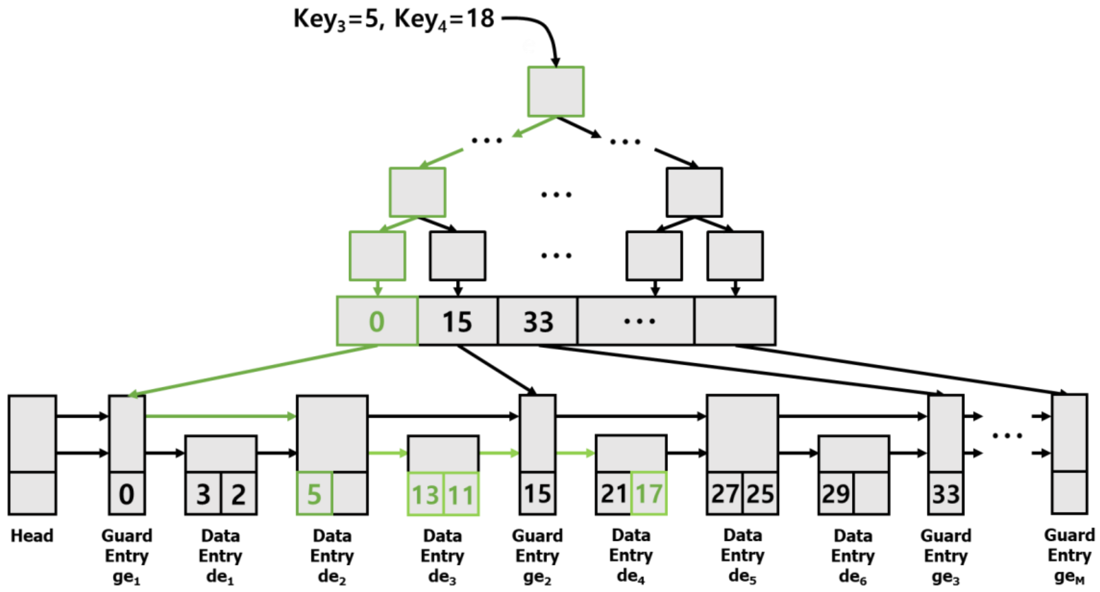
Algorithm 1 Insertion($key\_and\_value\ kv$)

$key = kv.getkey()$
$ge_i = Find\_guard\_entry(key)$
$x, prev = Find\_less\_than(key, ge_i)$
$next = x.getnext(0)$
if $next$ is a guard entry **then**
  if x is not full **and**$x \neq ge_i$ **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



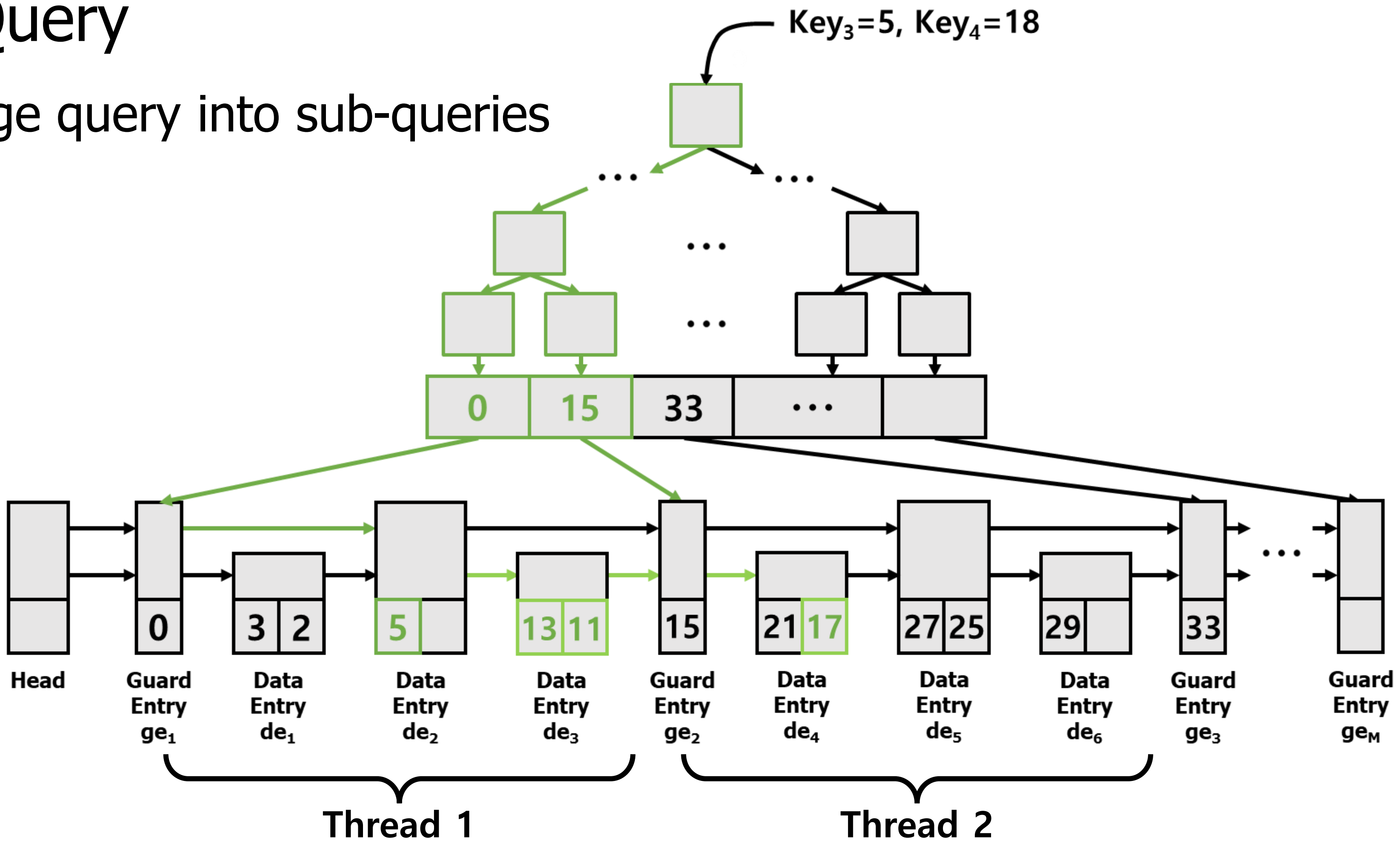
Basic Operators

- Range Query
  1. Find rightmost guard entry ge_i larger than or equal to key_i
  2. Find the last data entry de_i which max_key is less than key_i
  3. From the de_i, find the first entry de_j which max_key is larger than or equal to key_j



Basic Operators

- Range Query
  - Split range query into sub-queries



Optimizations

- Neural Model for Guard Selection
  - Simple seq2seq model
  - Predict the guard entries for the new MemTable

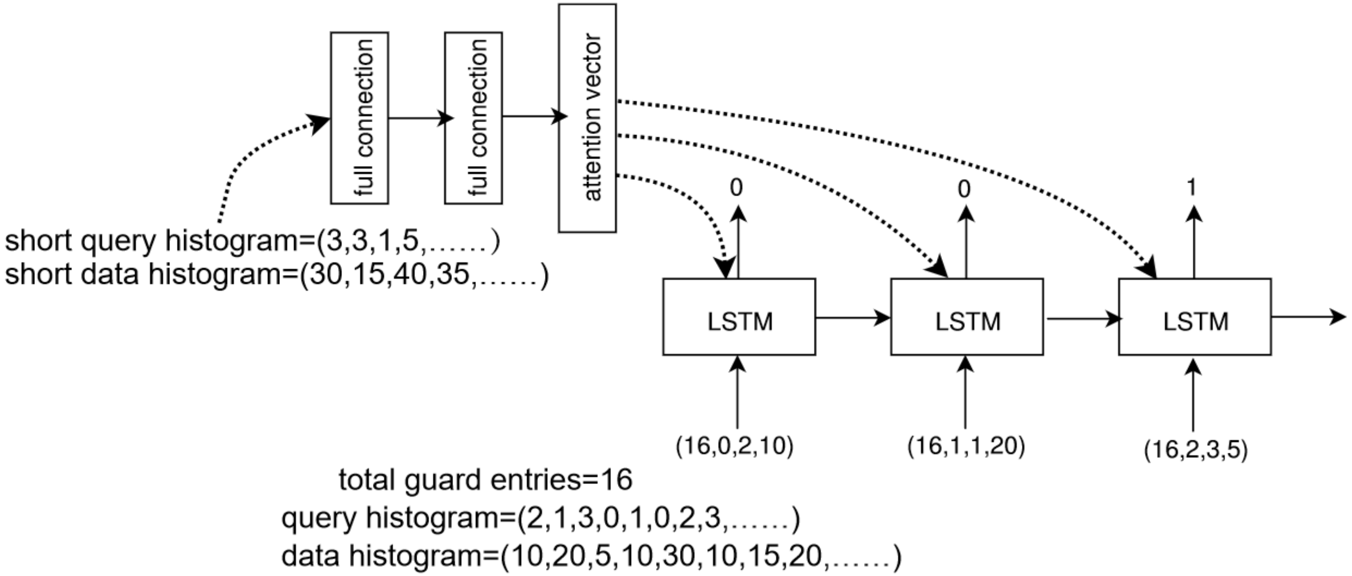


Figure 2: Neural Model for Guard Entry Selection

Optimizations

- Multiple Semi-order Skip-list
  - Global top layer index for all skip-lists
  - Even if we have multiple skip-lists, the top-layer index can be maintained in cache

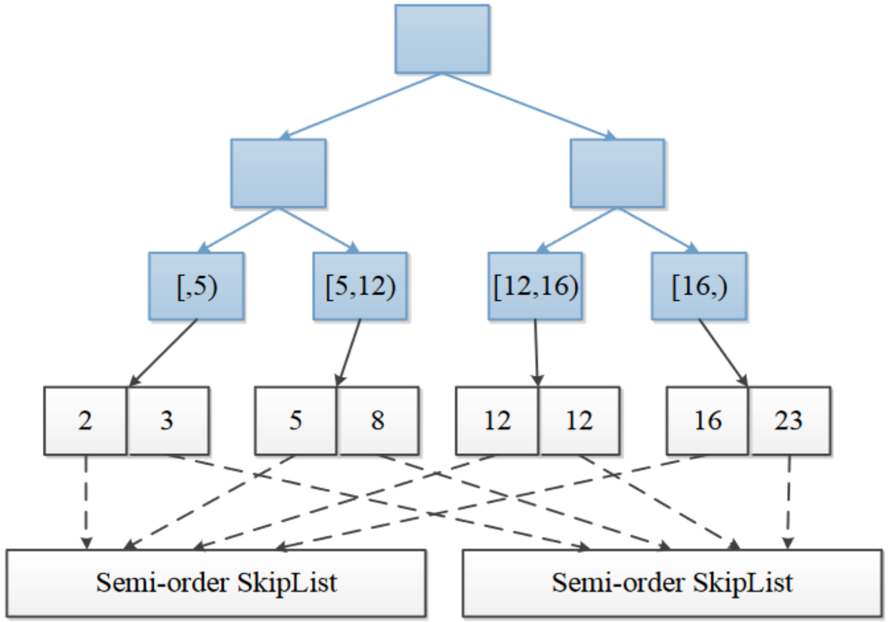


Figure 6: Top-Layer Index for Multiple Skip-Lists

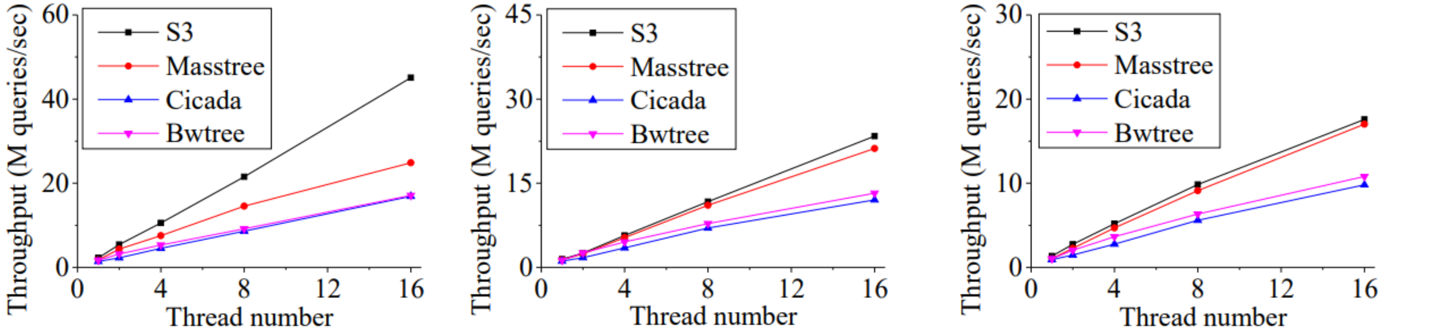
Evaluation

- CPU : Intel Xeon Processor E5 2660 v2 (25M Cache, 2.20 Ghz)
- In-Memory Index Key Length : 4bytes
- Comparison Target
  - Cicada, Masstree: Open Sourced
  - Bwtree: Retrieve from code of Peleton
- Query Workloads: YCSB C-Implementation (for generating)

Evaluation

Concurrent Test (Uniform Workload)

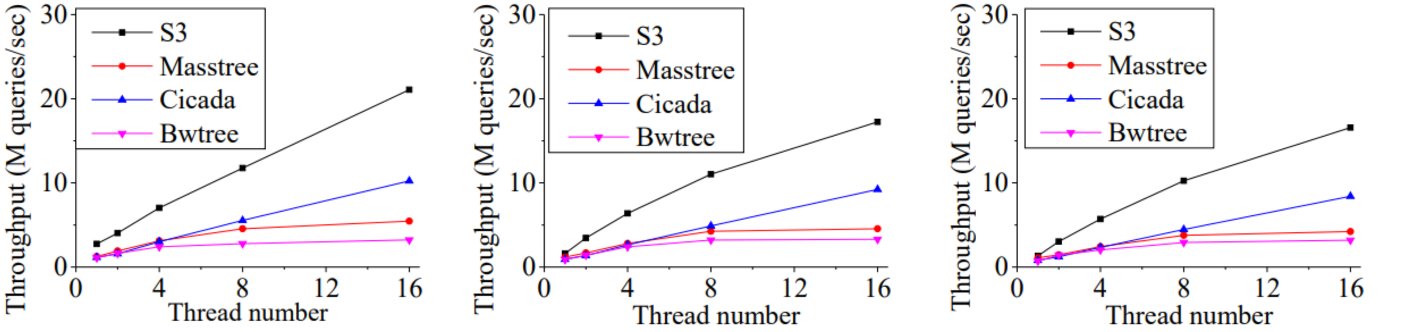
- **Which test are we do?**
  - Performance comparison test
    by number of keys and threads



(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)

- **What did we know?**
  - **As increased number of threads**
    - Insertion and lookup throughput increased
  - **As increased number of keys**
    - Insertion and lookup decreased
    - Shrink the throughput gap



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

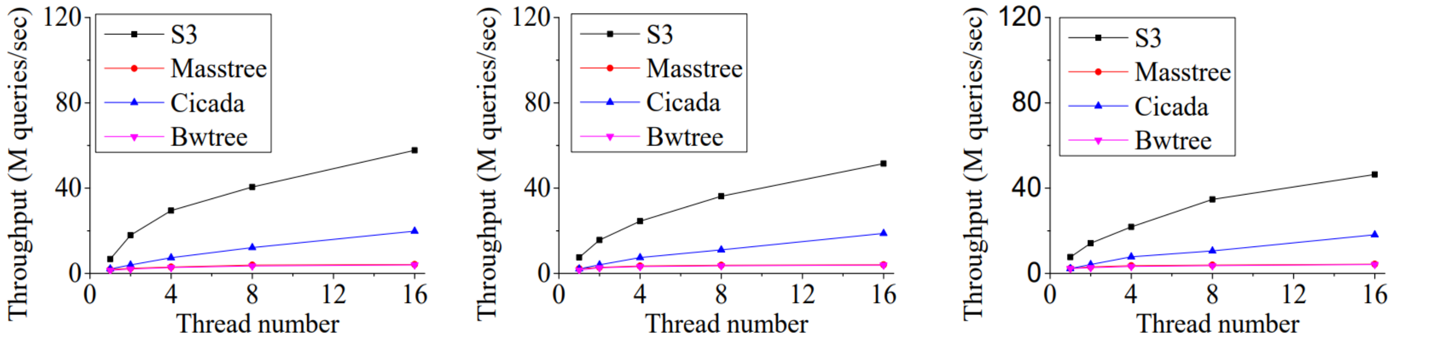
Figure 5: Query Throughput(Insertion,uniform workload)

- **S3 is better than others**

Evaluation

Concurrent Test (Zipfian Workload)

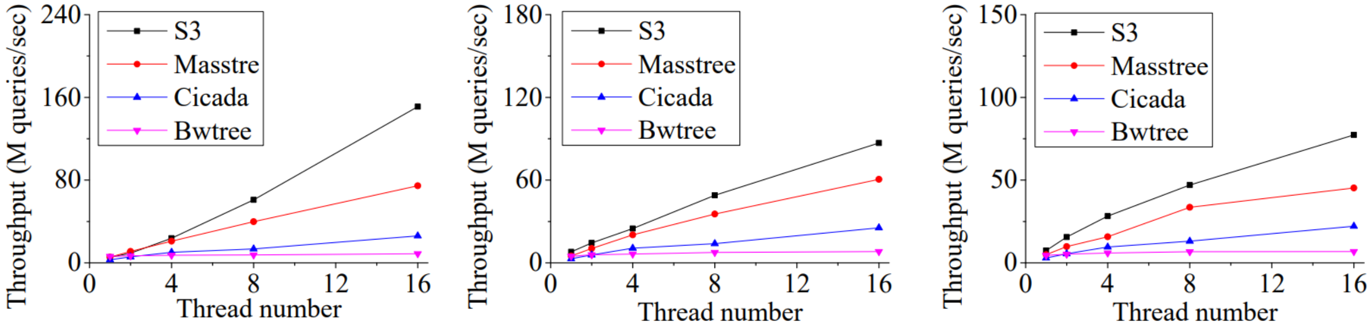
- **Which test are we do?**
  - Performance comparison test by number of keys and threads



(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)

- **What did we know?**
  - **Same results like uniform distribution**
    - But, scalability drops for the complex distribution
  - **On complex workload, speedup of lookup is better than insertion**
    - Cause, insertion incurs high processing overhead



(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)

- **S3 is better than others**

Evaluation

Query Throughput Test (Guard Entry Selection)

- **Which test are we do?**
  - To examine effect of the number of guard entries
- **How are we do test?**
  - First, insert 64M keys
  - Second, collect the total insertion time of each module
  - Third, accumulative time of each module for processing all 64M insertions and lookup

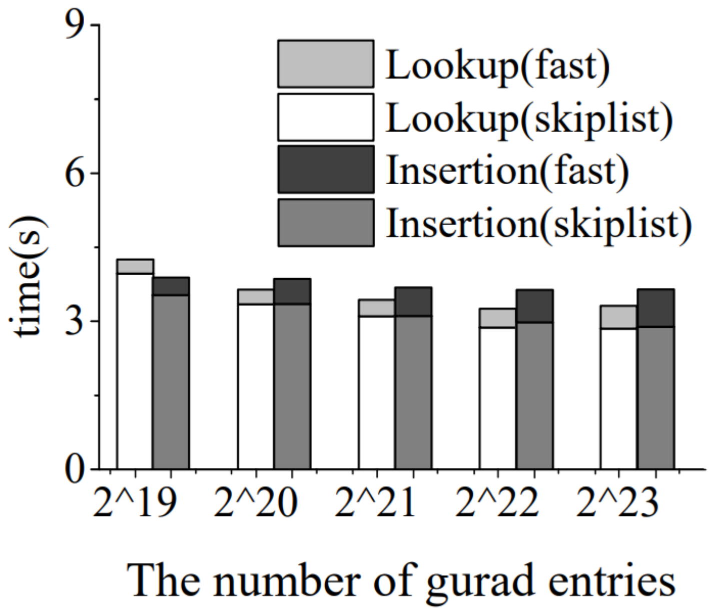


Figure 12: Query Through-put(64M Queries)

- **What did we know?**
  - **As increase the number of guard entries**
    - The search cost of top-layer index increases, while the search cost of bottom-layer index decr eases
- **Total search cost is optimal, when guard entries number 2^22**

Evaluation

Query Throughput Test (Guard Entry Selection)

- **Which test are we do?**
  - To examine which workloads are effectiveness of neural model for guard entry selection
- **How are we do test?**
  - Random :Randomly select a key to create a guard entry
  - Uniform: To uniformly generate guard entries among all keys
  - Neural: Neural model based guard entry selection
- **What did we know?**
  - **Uniform distribution do not differ much**
    But for Gaussian and Zipfian distribution, the neural model shows improvement over the others

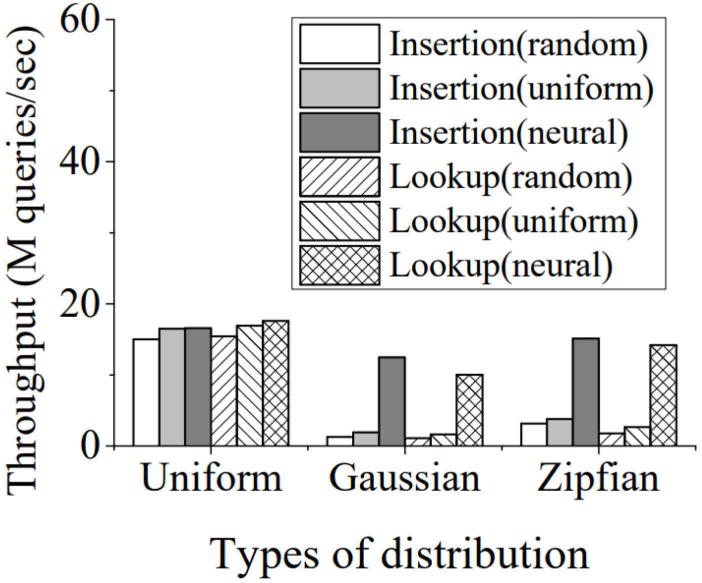


Figure 13: Query Through-put(64M Queries)

- **The neural model based on guard entry selection is best performance on Gaussian and Zipfian distribution**

Evaluation

Concurrent Test
(Range Query Performance)

- **Which test are we do?**
  - To examine the performance of range query
- **How are we do test?**
  - The number of keys varies from 4M to 64M
  - The query range varies from 0.0001% to 0.1% key range
- **What did we know?**
  - As increase the range size, decrease the range query throughput

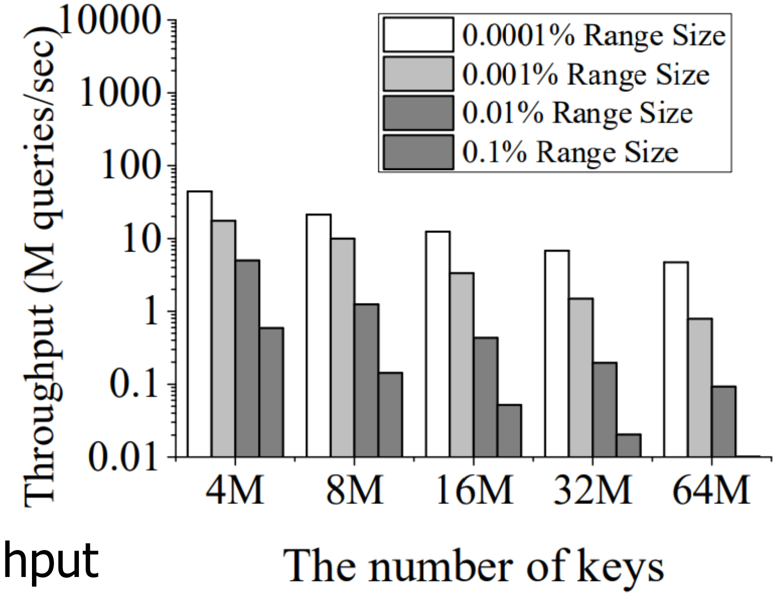


Figure 14: Range Query Throughput

- **Unsorted date decreases the throughput of range query**

Evaluation

Cost of Flushing Data as SSTables

- **Which test are we do?**
  - To examine the efficiency of flushing the data from S3 to disk part of RocksDB
- **What is Full order?**
  - The best case that sorted entries are kept in the contiguous memory space
- **What did we know?**
  - The flushing efficiency of S3 is better than skip-list

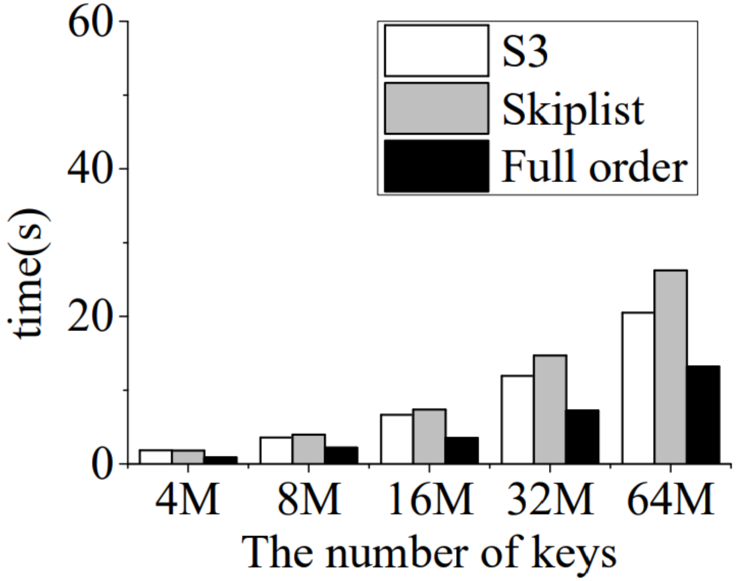


Figure 17: Cost of Writing SSTables

- **S3 needs to sort the data before flushing them back to the disk, however the cost is acceptable compared to the benefit of using continuous storage**

Conclusion

- S3 is an in-memory skip-list index for disk-based key-value stores
  - Using guard entries to enable fast search operations
- S3 is designed with two layers
  - Top layer: Cache Sensitive Index(FAST)
  - Bottom layer: Semi-order skip-list
    - Achieve high write performance while slightly sacrificing the read performance
- The performance of RocksDB can be improved by the equipment with S3

Thank you

2024.01.24

Presentation by Boseung Kim, Yeongyu Choi

bskim1102@dankook.ac.kr, dusrb1418@naver.com