Sieve: A Learned Data-Skipping Index for Data Analytics

Yulai Tong
School of Computer Science and Technology, WNLO, HUST
yl_t@hust.edu.cn

Jiazhen Liu
School of Computer Science and Technology, WNLO, HUST
jz_l@hust.edu.cn

Hua Wang
School of Computer Science and Technology, WNLO, HUST
hwang@hust.edu.cn

Ke Zhou
School of Computer Science and Technology, WNLO, HUST
zhke@hust.edu.cn

Rongfeng He
Huawei Cloud
herongfeng@huawei.com

Qin Zhang
Huawei Cloud
kevin.zhangqin@huawei.com

Cheng Wang
Huawei Cloud
wangcheng131@huawei.com

ABSTRACT

Modern data analytics services are coupled with external data storage services, making I/O from remote cloud storage one of the dominant costs for query processing. Techniques such as columnar block-based data organization and compression have become standard practices for these services to save storage and processing cost. However, the problem of effectively skipping irrelevant blocks at low overhead is still open. Existing data-skipping efforts maintain lightweight summaries (e.g., min/max, histograms) for each block to filter irrelevant data. However, such techniques ignore patterns in real-world data, enabling ineffective use of the storage budget and may cause serious false positives.

This paper presents Sieve, a learning-enhanced index designed to efficiently filter out irrelevant blocks by capturing data patterns. Specifically, Sieve utilizes piece-wise linear functions to capture block distribution trends over the key space. Based on the captured trends, Sieve trades off storage consumption and false positives by grouping neighboring keys with similar block distributions into a single region. We have evaluated Sieve using Presto, and experiments on real-world datasets demonstrate that Sieve achieves up to 80% reduction in blocks accessed and 42% reduction in query times compared to its counterparts.

PVLDB Reference Format:
Yulai Tong, Jiazhen Liu, Hua Wang, Ke Zhou, Rongfeng He, Qin Zhang, and Cheng Wang. Sieve: A Learned Data-Skipping Index for Data Analytics. PVLDB, 16(11): 3214 - 3226, 2023.
doi:10.14778/3611479.3611520

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/UmasouTTT/Sieve.

1 INTRODUCTION

Modern cloud-based data analytics services adopt various techniques to minimize data movement and access, so as to handle the ever-increasing large volumes of data efficiently. One common practice involves employing columnar storage, which helps avoid unnecessary access to irrelevant columns during query execution. Typically, these services separate computing resources from cloud storage services [3, 4, 6], making I/O from remote storage one of the dominant costs for query processing. These systems organize data records into blocks, each with tens of thousands to millions of records, to maximize compression ratios. To optimize throughput and minimize I/O operations per second, the smallest I/O unit from remote storage is a block or a subset of columns from a block.

To enhance query efficiency, cloud-based data analytics services usually maintain lightweight Small Materialized Aggregates (SMAs) [33] per block to avoid unnecessary access to irrelevant data blocks (i.e., skip data) during query processing. The most common form of SMAs is ZoneMap, which stores the minimum and maximum values for each column within a data block. For instance, if a block's ZoneMap indicates that its records encompass dates ranging from April to May, and the query filters for records with dates in February, then this particular block can be skipped during the execution of the query, eliminating the need to read it from storage.

ZoneMaps [1, 8, 10] are cheap to maintain and incur low storage overhead, but their effectiveness is highly dependent on the data layout. ZoneMaps are most effective on ordered attributes. For unordered attributes, which are much more common, ZoneMaps compromise too much on query performance because the value ranges (minimum and maximum values) in blocks may cover most query predicates, and numerous irrelevant blocks have to be scanned.

B+ trees are promising indexing structures to address the effectiveness problem of ZoneMaps. Essentially, B+ trees are multi-level indexing structures with inner nodes guiding the key search and leaf nodes storing individual keys and pointers to blocks. Such a fine-grained structure allows B+ trees to offer efficient indexing for both sorted and unsorted attributes compared to ZoneMaps.

However, the storage overhead of B+ trees prohibits their deployment in modern cloud-based data analytics services with large volumes of data. In our evaluation, we find that a B+ tree created on the Lineitem table from the TPC-H benchmark with 200GB size can consume about 25GB storage space. Such additional storage overhead results in non-ignorable dollar costs and significantly degrades the indexing performance because numerous nodes might have to be read to locate a key.

Various compression schemes [11, 12, 19, 45] have been developed to mitigate the storage overhead of B+ trees. These techniques

This work is done while Yulai Tong is an intern at Huawei Cloud.
Hua Wang is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611520

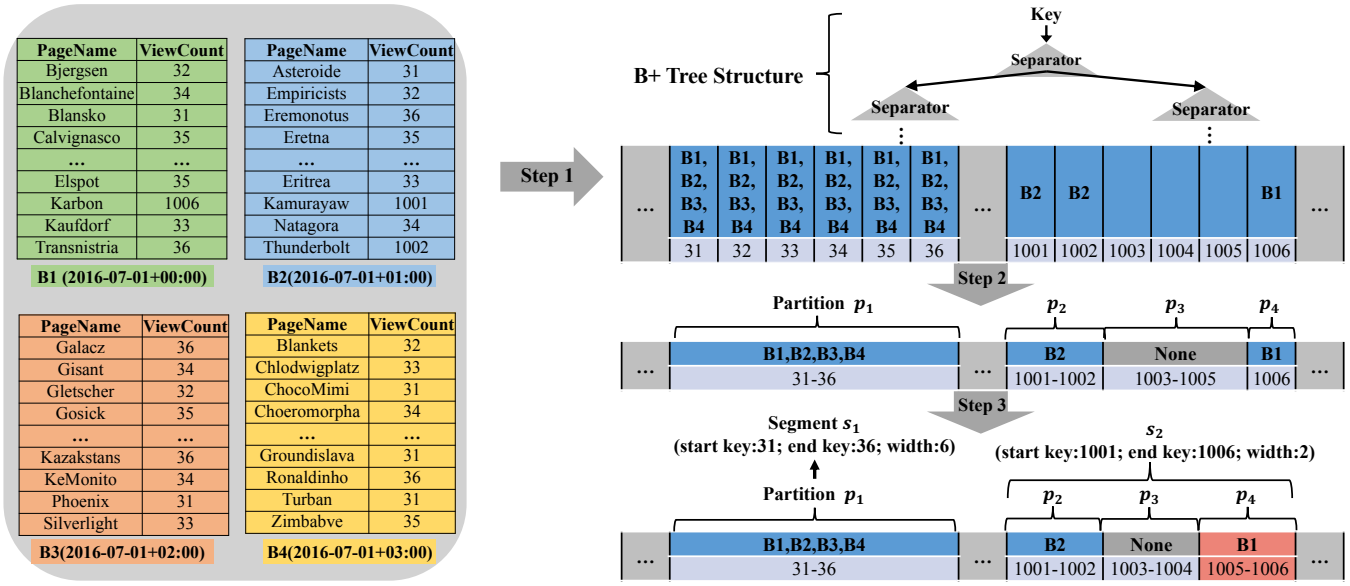


Figure 1: An intuition of how Sieve works based on the real-world Wikipedia dataset. The indexed attribute is ViewCount. Step 1 builds a sorted array of key->block pairs. Step 2 groups contiguous keys mapping to the same set of blocks to a partition. Step 3 further merges narrow-width partitions into a single segment by accepting minor false positives. After Step 3, partition p_4 highlighted in red color can cause false positions (i.e., Sieve answers the query `ViewCount=1005` with block B_1, but value 1005 actually does not exist in any blocks). Note that false positives do not affect the correctness since the query execution engine will ultimately filter the data at the row level.

aim to eliminate redundancy among keys and/or minimize the size of each key within an index node. For instance, the reduction of the overall tree size can be achieved by utilizing prefix and suffix truncation, which stores common parts of keys only once at each index node. Nevertheless, despite the benefits of this compression scheme in reducing individual index node sizes, the storage requirements for these indexes continue to scale linearly with the number of distinct keys to be indexed [18].

Recently, learned indexes have been proposed in the traditional database to overcome the mentioned problems of B+ trees. Learned indexes output the position of a given key in a sorted array by replacing the multi-level tree structure with piece-wise linear functions such that the models can not only reduce the index size but also improve the key lookup efficiency. Nevertheless, all the individual key-block pairs are still maintained in order to skip blocks.

This paper proposes Sieve, a novel indexing structure that trades off storage overhead and false positives by capturing patterns in data. The key observation of Sieve is that neighboring keys often exhibit similar trends in block distribution. That is, *a series of* consecutive keys are likely to belong to the same block set or be associated with different ones. Based on this observation, Sieve balances storage consumption and false positives by grouping neighboring keys into regions according to the identified trends. More specifically, Sieve clusters the key space that maps to the same block set into a broad region, effectively reducing storage costs. On the other hand, for the key space mapping to different blocks, Sieve divides the space into multiple narrow regions to mitigate false positives. By recording information about a region of the key space, instead of indexing individual keys, Sieve not only saves storage space but also reduces the searching time. At the core of Sieve lies piece-wise linear functions utilized to capture block distribution trends and determine the best-suited width for each region.

We have incorporated Sieve with Presto [37] and performed evaluations on both real and synthetic datasets. Experimental results demonstrate that (a) on range queries, Sieve reduces data accesses by up to 80% and achieves query speedup by up to 1.7x compared to best counterparts at a comparable storage cost. (b) on point queries, Sieve occupies up to two orders of magnitude less storage space than a recently announced set member filter while still achieving comparable performance.

2 OVERVIEW OF SIEVE

At a high level, data-skipping indexes can be represented by a function that maps a key to blocks containing it. The key observation of Sieve is that neighboring keys often exhibit similar trends in block distribution. In other words, *a series of* consecutive keys are likely to belong to the same block set or be associated with different ones. We find such trends are common in cloud datasets.

As an example, real-time applications like monitoring sensors [20] and Facebook [13] have a constant ingest of timestamped data, which causes close timestamps placed in the same block. As another example, consider the real-world Wikipedia dataset shown in Figure 1. Each block is a log file that records page view statistics, including how many people (the ViewCount column) have visited an article (the PageName column) in a given period. The indexed attribute is ViewCount.

The small ViewCounts (key ranges 31-36 in Figure 1) are distributed in almost all the periodic blocks since a majority of the pages receive a low number of visits. Conversely, larger ViewCounts (key ranges 1001-1006 in Figure 1) are typically confined to different periodic blocks due to emerging hot topics.

Leveraging this observation, Sieve clusters the key space that maps to the same block set into a broad region, effectively reducing storage costs. On the other hand, for the key space mapping to different blocks, Sieve divides the space into multiple narrow regions to mitigate false positives.

Below, we give an intuition of how Sieve works step by step based on the real-world Wikipedia dataset.

Step 1: Sorting. Similar to traditional B+ tree-based indexing structures, Sieve needs to sort the indexed attribute. As shown in Figure 1, the individual keys on the leaf nodes (called "indirection layer") are kept in sorted order. Each key is associated with a pointer to the blocks containing it.

Step 2: Partitioning. For a range of contiguous keys mapping to the same set of blocks, Sieve groups them into a single partition. For example, keys 31-36 exist in the same block set $\{B_1, B_2, B_3, B_4\}$, so they are grouped into partition p_1. By storing information about a region of the key space on the leaf nodes instead of indexing individual keys, Sieve saves the storage space.

Step 3: Segmenting. To further minimize the storage requirements, Sieve needs to merge the narrow partitions that arise from distinct block sets (i.e., p_2, p_3, p_4 in Figure 1). To achieve this, Sieve organizes multiple narrow-width partitions that are adjacent to each other into a single segment by tolerating minor false positives.

For example, in Figure 1, Sieve normalizes the key ranges of partition p_2, p_3, p_4 to be equal size and groups them into a single segment s_2. In this case, Sieve causes false positives when answering query `ViewCount=1005` (i.e., Sieve answers query `ViewCount=1005` with block B_1, but value 1005 actually does not exist in any blocks). Therefore, instead of storing all partitions in the leaf nodes, Sieve stores only (1) the start and end key of a segment and (2) the normalized key ranges inside a segment in order to quickly compute the partition a key belongs to, and (3) corresponding block IDs for each partition (partition metadata). By doing so, Sieve not only saves storage space but also reduces the searching time.

To capture the block distribution, Sieve leverages a CDF that models the total number of times the block set mapped by contiguous keys has changed, as shown in Figure 2. Based on the CDF, Sieve divides the underlying key-block pairs into a series of variable-sized segments that approximate the block distribution.

The approximated linear functions represent the change period of the mapped block set between keys (i.e., for how many contiguous keys, a block set change occurs). According to this approximated frequency, Sieve divides a segment into partitions of equal width.

3 SEGMENTATION AND PARTITIONING

In this section, we first show how Sieve models and captures the block distribution trends. Then, we describe how Sieve organizes the key space of indexed attributes into segments and partitions based on captured trends. After this process, each segment is inserted into a B+ tree to enable efficient lookup and insert operations (Section 4). Table 1 summarizes the notations used in this paper.

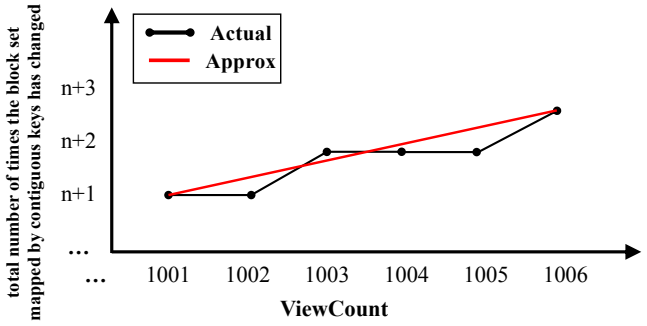


Figure 2: The cumulative distribution function that captures the block distribution. The Y-axis represents the total number of times the block set mapped by contiguous keys has changed. Points are approximated with a linear function shown in red color. The multiplicative inverse of the slope indicates the change period of the mapped block set between keys (i.e., for how many contiguous keys, a block set change occurs) and is utilized to decide the optimal width for partitions inside a segment.

Table 1: Notations

Notation	Explanation
bc_k	Whether mapped block(s) has changed at key k
tbc_k	Cumulative value of bc at key k.
$pred\_tbc$	Predicted tbc value by approximated linears.
$true\_tbc$	Actual tbc value by the CDF model.
$s.width$	Number of keys managed by segment s.
$s.p_{width}$	Number of keys managed by a partition in segment s based on the approximated linear function and storage budget.
S_ε	Number of segments for an error threshold ε.
p_{num}	Number of partitions of a segment or dataset.
B_k	The block set containing key k.
$block_{num}$	Number of blocks of a segment or dataset.
ε	Maximum error when constructing the segments

3.1 Trends of Block Distribution as CDF Models

A segment is essentially a key space region with similar trends of block distribution. To formally capture the difference of block distribution between keys, Sieve uses Block Change (bc_k) to represent whether the mapped blocks between key k and $k-1$ are identical:

Definition 1 (Block Change and Total Block Change.) *Block Change (bc) indicates whether the mapped block(s) has changed between contiguous keys. More specifically, if k and $k-1$ point to the same set of blocks, the value of bc_k is 0; otherwise, it is 1. Total Block Change (tbc) is defined as the cumulative value of bc that represents the total number of occurrences of block changes before a key.*

The cumulative distribution function (CDF) of bc models block change frequency between keys. With such CDF models, Sieve divides the key space into several linear segments that are able to

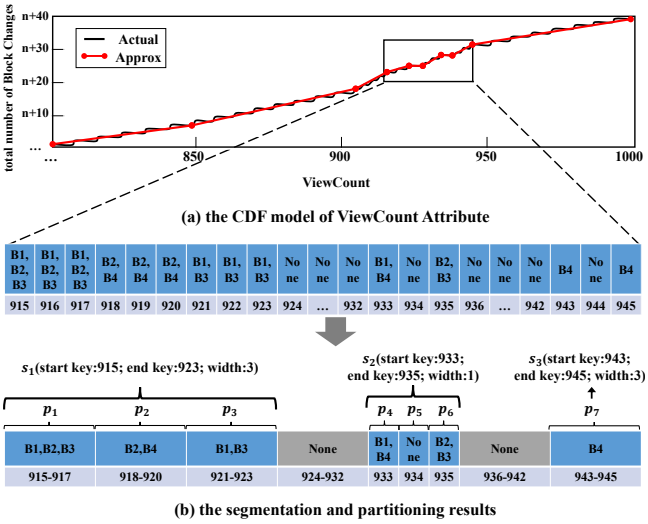


Figure 3: (a) Approximating cumulative distribution function of Block Change on ViewCount. The actual CDF model is in black, and the approximate linear functions are in red. (b) The segmentation and partitioning results from the approximated trends. The key space is organized into segments, and further divided into equal-width partitions.

accurately reflect the trends of block distribution between keys. For example, Figure 2 shows the CDF model of Block Change for keys $1001 - 1006$ from Figure 1, which is approximated by a linear function (in red color). Since key 1002 points to block set B_2 while key 1003 points to *None* (i.e., a missing key), *tbc* increases by 1. Similarly, *tbc* increases at key 1006 since key 1006 maps to different sets of blocks from key 1005.

The resulting piece-wise linear approximation, however, exhibits imprecision. Therefore, we define the *error* associated with our approximation as the difference between any key's actual and predicted total number of block changes. This is illustrated below, where $pred\_tbc(k)$ and $true\_tbc(k)$ return the predicted and actual number of block changes of an element k, respectively.

$$error = |pred\_tbc(k) - true\_tbc(k)| \tag{1}$$

Intuitively, the approximated linear functions represent the change period of the mapped block set between keys (i.e., for how many contiguous keys, a block set change occurs).

Although more complex functions (e.g., higher order polynomials) can be used to approximate the true functions, the benefit of using piece-wise linear function approximations is two-fold for Sieve. First, linear segments are able to accurately reflect various trends of block distribution among keys. The linearity of a segment enables Sieve to subdivide the segment into a series of equal-width partitions to balance false positives and storage space (Section 3.3). Second, compared to complex functions, piece-wise linear approximations are more cost-effective. This dramatically (1) reduces the initial index construction cost, and (2) improves lookup and insert latency (see Section 4).

Algorithm 1: Segmentation Process

Input: keys of indexed attribute r, segment error threshold ε, missing key ranges m.
Output: *segments*

```
Initialization: segments ← ∅, s ← [ ], true_tbc ← 0, i ← 0.
while i < r.size do
    if B_{r[i]} ≠ B_{r[i−1]} then
        true_tbc += 1
    end
    if B_{r[i]} ! = ∅ then
        s.cal_err(r[i], tbc)
        if s.err violate error threshold ε then
            segments.add(s)
            make new segment s from key r[i]
        end
        s.add(r, true_tbc)
    else
        missing key range l ← m_{r[i]} − r[i]
        if l violate error threshold ε then
            segments.add(s)
            make new segment s from key m_{r[i]}
        else
            assign keys from r[i] to m_{r[i]} to s
        end
    end
end
```

3.2 Segmentation Algorithm

Various optimal piece-wise linear approximation algorithms [16, 24, 30] have been proposed. However, these techniques either suffer from prohibitively high computational costs or fail to ensure a maximal error. To efficiently support construction and inserts while guaranteeing a maximal error, we seek a highly efficient one-pass error-bounded linear algorithm. In the following, we describe a proposed segmentation algorithm, similar to FSW [18, 31, 42], which is linear in runtime, has low constant memory usage, and guarantees a maximal error in each segment. Importantly, though, we (1) prove that we can limit false positives by binding the maximal error for these linear approximations (Section 5.3) and (2) address how to balance the extra storage consumption and false positives caused by gaps (e.g., keys $924 - 932$ in Figure 3) in the key space.

The core concept underlying the segmentation approach (Algorithm 1) is that a new key can be included in a segment only if it satisfies the error constraint and does not cause any existing keys within that segment to violate it.

More specifically, the algorithm defines a cone using three components: an origin point, a high slope, and a low slope. The combination of the starting point with the low slope determines the lower bound of the cone, while the combination with the high slope determines the upper bound. Intuitively, the cone represents the set of viable linear functions for a segment originating from the cone's origin point (the high and low slopes define the permissible range of slopes).

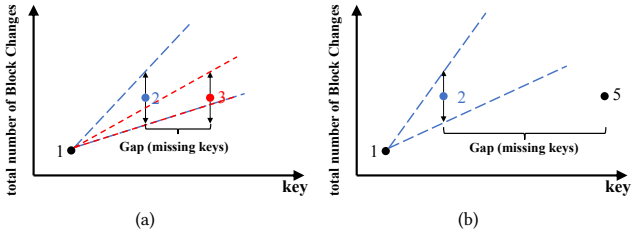


Figure 4: Given the error threshold, SIEVE decides whether an entire gap starts a new segment or is added to a segment. Point 1 is the origin of the cone. Point 2 (starting key of the gap) is then added, resulting in the dashed cone. In Figure 4(a), Point 3 (ending key of the gap) is added next, yielding the dotted cone. In Figure 4(b), Point 5 (ending key of the gap) is outside the dotted cone and therefore starts a new segment.

When the new key intended for inclusion in the segment falls within the cone, it does not violate the error threshold. In such cases, the cone is either narrowed or maintains its size based on the added new key. Specifically, the algorithm determines the lowest high-slope value and the highest low-slope value by comparing the newly computed slopes with the previous slopes. However, if the new key falls outside the cone, it indicates that at least one existing key within the segment violates the error constraint. Consequently, the new key cannot be included in the segment, and serves as the origin point for a new segment (lines 6-13).

SIEVE handles gaps by considering their boundaries. When a gap is added to a segment, the high and low slopes are calculated using the starting key of the gap and the key's Y-axis value plus error and minus error (respectively). A gap whose ending key is inside the cone is included in the segment (point 3 in Figure 4(a)). Then, the lowest high slope and the highest low slope values are updated according to the starting and ending key's cone (line 19).

On the contrary, a gap whose ending key is not inside the cone cannot be included in the segment (e.g., point 5 in Figure 4(b)), and becomes the origin point of a new segment (lines 14-17). After this process, the key space is divided into several linear segments that are able to accurately reflect the trends of block distribution between keys (red lines in Figure 3).

3.3 Partitioning Algorithm

To find the optimal key range of a partition inside a segment, SIEVE divides the length of the segment ($s.width$) by the number of partitions determined by the changes to mapped blocks of the segment.

$$s.period = \frac{s.width}{true\_tbc(s.end) - true\_tbc(s.start) + 1} \quad (2)$$

Intuitively, $s.period$ represents the change period of the mapped blocks between keys (i.e., for how many contiguous keys, a block set change occurs) in a linear segment. Ideally, there should be no false positives when grouping $s.period$ consecutive keys into a partition. In Figure 2, the best-suited partition width is $s.period = \lfloor \frac{6}{3} \rfloor = 2$.

However, in some settings, a system user may want to give the indexing structure a storage budget to use. In this scenario, we need to fit a limited amount of partition metadata (i.e., corresponding block IDs for a partition) to the storage. In other words, the goal becomes to limit the number of generated partitions to fit into the specified space budget while minimizing false positives.

Algorithm 2: Partitioning Process

Input: output from the segmentation processing *segments*, total number of partitions for the dataset $D.p_{num}$
Output: *segments*

1 $width\ threshold \leftarrow \frac{\sum_{s_i \in S} s_i.width}{D.p_{num}}$
2 available number of partitions $P \leftarrow D.p_{num}$
3 **foreach** $s \in Segments$ **do**
4 $\quad s.period = \frac{s.width}{true\_tbc(s.end) - true\_tbc(s.start) + 1}$
5 $\quad$ **if** $s.period >= width\ threshold$ **then**
6 $\quad\quad s.p_{width} = s.period$
7 $\quad\quad s.p_{num} = \frac{s.width}{s.p_{width}}$
8 $\quad\quad P \leftarrow P - s.p_{num}$
9 $\quad\quad Segments \leftarrow Segments - s$
10 $\quad$ **end**
11 **end**
12 **foreach** $s \in Segments$ **do**
13 $\quad$ **if** $s.period < width\ threshold$ **then**
14 $\quad\quad s.score = s.block_{num} \cdot s.width$
15 $\quad$ **end**
16 **end**
17 **foreach** $s \in Segments$ **do**
18 $\quad$ **if** $s.period < ideal\ width$ **then**
19 $\quad\quad s.p_{num} = P \cdot \frac{s.core}{\sum_{s_i \in S} s_i.score}$
20 $\quad$ **end**
21 **end**
22 **foreach** $s \in Segments$ **do**
23 $\quad start \leftarrow s.start$
24 $\quad$ **for** $i = 1$ *to* $s.p_{num}$ **do**
25 $\quad\quad end \leftarrow start + s.p_{width}$
26 $\quad\quad$ assign $s.p_i.blocks$ with blocks containing key in range $(start, end)$
27 $\quad\quad start \leftarrow end$
28 $\quad$ **end**
29 **end**

For a given storage budget S_{req}, assume the maximum number of partitions that can be generated to fit into S_{req} for a dataset is $D.p_{num}$ (See Section 5.1 for more details about the estimation of $D.p_{num}$). Now, the question follows: how to fit $D.p_{num}$ partitions into S_ε segments?

Based on the above consideration, we use a partition approach (Algorithm 2) to avoid false positives as much as possible under a given storage budget. First, we assume all the $D.p_{num}$ partitions are of equal width among the S_ε segments. With this assumption, the width of the partitions is:

$$width\ threshold = \frac{\sum_{s_i \in S} s_i.width}{D.p_{num}} \quad (3)$$

For a segment whose $s.period$ is larger than the *width threshold*, we can directly assign its desired number of partitions (e.g., s_1 in

Figure 3). Therefore, for this type of segments, their final width is $s.p_{width} = s.period$ (lines 3-11).

On the other hand, a segment whose $s.period$ is smaller than the *width threshold* means more partitions are required for this segment to satisfy its optimal width.

Limited by the storage budget, Sieve may not be able to provide enough partitions to satisfy *s.period*.

We introduce a heuristic score for each segment to decide how many partitions should be allocated to each segment. Based on the partitioning result, we can get the final $s.p_{width}$.

Heuristic #1: segment with more blocks and larger width leads to more false positives.

Intuitively, we should give more space to segments that may cause more false positives. The false positives of a segment depend on two variables: (1) the number of blocks the segment contains and (2) the width of the segment. As the number of blocks in a segment increases, the likelihood of having distinct blocks between keys also increases, leading to a higher potential for false positives. Given a certain number of partitions for a segment, increasing the segment's width leads to a larger corresponding $s.p_{width}$, which in turn causes grouping over a larger key space. Since $s.period$ is small, this results in more false positives.

Based on this observation, we can obtain the score for each segment using the following equation (lines 12-16):

$$s.score = s.block_{num} \cdot s.width \tag{4}$$

Given the score for a segment, we can determine the number of partitions allocated to this segment (e.g., s_2 and s_3 in Figure 3) according to its weight over all the segments (lines 17-21).

After this process, segments are partitioned based on their assigned width, and the corresponding block set will be recorded for each partition (lines 22-29).

Algorithm 3: Lookup Algorithm

```
1  Function SearchTree(key, node):
2      Binary search of the root node and read the appropriate
         child node;
3      Recursive search for internal nodes until a leaf is
         reached;
4      Read min_key and max_key from the leaf node.
5      if key ∈ [min_key, max_key] then
6          return leaf
7      else
8          key is a missing key
9          return
10     end
11
12 Function SearchSegment(seg, key):
13     pos ← (key − seg.start) / seg.p_width
14     return seg.partitions[pos]
15
16 Function LookUp(key, tree):
17     seg ← SearchTree(key, tree.root)
18     p ← SearchSegment(key, seg)
19     returnp.blocks
```

4 OPERATIONS

Lookup and Insert operations make up the majority of index operations. Here, we describe the process used by Sieve to perform point queries, followed by an explanation of how this approach can be expanded to cover range queries. Then, We present a highly efficient insert strategy designed to reduce the insert overhead.

4.1 Look Up

The process of searching a Sieve for block(s) containing a single key consists of two steps: (1) searching the tree to locate the segment that the key belongs to, and (2) finding the partition where the key is located. These steps are outlined in Algorithm 3.

Tree Search As described in Section 3, each segment is stored in a B+ tree, and we must first search the B+ tree to locate the segment containing the key. Standard tree traversal algorithms can be used to traverse the B+ tree from the root to the leaf (as outlined in the *SearchTree* function of Algorithm 3). This process stops when reaching a leaf node pointing to the segment where the key is located. The runtime for searching for the segment that a key belongs to is $O(log_a(S))$, where a represents the fanout of the B+ tree, and S is the number of segments.

Segment Search Once Sieve locates the segment for a key, it then finds the partition to which the key belongs (*SearchSegment* function). Recall partitions inside a segment are of equal size. So we can directly locate the partition containing the key using the following equation.

$$p = \lfloor \frac{key - s.start}{s.p_{width}} \rfloor \tag{5}$$

As shown in Equation 5, we subtract the starting key of the segment ($s.start$) from the searched key (key) and then divide the result by the segment's p_{width} width.

For range queries, Sieve utilizes Algorithm 3 to efficiently locate the start partition and end partition. It then examines the partitions intersecting the specified range to obtain blocks that need to be accessed. This search process is fast because the number of partitions is significantly smaller compared to the entire key space.

4.2 Insert

Unlike typical B+ trees, insert operations in Sieve require additional consideration (Algorithm 4) since the newly added key might introduce distinct blocks to the partition it belongs to, increasing false positives of the corresponding partition.

Note that insert operations only increase false positives of the partition the key belongs to. However, from the perspective of the segment, the increased false positives from a single partition might be negligible. So, in order to measure the influence of the insert operation on the whole segment, Sieve calculates the average number of newly inserted blocks per partition in the segment, as shown below in the following equation. Obviously, the more newly inserted blocks, the more false positives are likely to be.

$$segment\ insert\ block\ density: \frac{\sum_{p \in s} |p_{new\ blocks}|}{s.p_{num}} \tag{6}$$

Algorithm 4: Insert Algorithm

```
1  Function InsertKey(tree, key, block):
2      seg ← SearchTree(key, tree.root)
3      p ← SearchSegment(seg, key)
4      update p.blocks with block
5      if seg.insert_fp_density > rebuild threshold then
6          sort key-block pairs from seg.blocks
7          segs ← SEGMENTATION(sorted array)
8          segs ← PARTITION(segs)
9          foreach s ∈ segs do
10             tree.insert(s)
11         end
12         tree.remove(seg)
13     end
14     return
```

To decide whether the segment should be re-built, Sieve leverages a parameter called *segment insert fp density*. A segment's insert false positive density is defined as the ratio of *segment insert block density* to the total number of blocks in the dataset.

$$segment\ insert\ fp\ density : \frac{segment\ insert\ block\ density}{total\ block\ num\ in\ the\ dataset} \quad (7)$$

Once a segment's *insert fp density* reaches the rebuild threshold set by the system user, Sieve builds a sorted array of key-block pairs from the segment's blocks. Then, the sorted array will be re-divided using the previously described segmentation (Algorithm 1) and partition (Algorithm 2) algorithm to create a series of valid segments that satisfy the rebuild threshold (lines 6-8). Note that depending on the data, the number of segments after this process can be one (i.e., the data inserted into the origin segment does not violate the rebuild threshold) or several. Finally, each of the new segments generated from the process is inserted into the tree, and the old segment is removed (lines 9-12).

The overall runtime for inserting a new element into Sieve is the time required to locate the partition and union the mapped blocks of the element with those of the partition. With S segments stored in a Sieve, and a fanout of a (i.e., the number of keys in each internal separator node), inserting a new key into a Sieve has the following runtime:

$$insert\ runtime : O(log_a(S)) \quad (8)$$

Note that when the *insert ratio* is violated and the segment needs to be re-segmented, the runtime has an additional cost of $O(d + sort\ time(s.width))$ due to sorting, segmenting, and partitioning, where d is the total number of records in the segment's blocks and $s.width$ represents the length of the segment.

5 COST MODEL

This section deduces cost models for Sieve. First, we will show how to estimate the index size of Sieve. Then, we give the estimation of the cost of searching a Sieve index. Finally, we estimate the false positive rate caused by a Sieve index.

5.1 Space Estimation

We can estimate the size of Sieve for a given error threshold of ε using the following equation, where S_ε is the number of segments that are created for an error threshold of ε, a is the fanout of the tree, and $D.p_{num}$ is the number of partitions that are created for an error threshold of ε for a dataset.

$$SIZE(e) = \underbrace{S_\varepsilon \cdot log_a(S_\varepsilon) \cdot 16B}_{Tree} + \underbrace{S_\varepsilon \cdot 24B}_{Segment} + \underbrace{D.p_{num} \cdot n \cdot bit}_{Partition} \quad (9)$$

The first term is a pessimistic bound on the storage cost of the tree (leaf + internal nodes using 8-byte keys/pointers). The second term is additional metadata about each segment (i.e., each segment has a start key, an end key, and a partition width). The third term represents the additional metadata about each partition. This metadata is essentially a bit array used for storing the blocks for each partition. n represents the total number of blocks in the dataset.

5.2 Search time complexity

As discussed, lookups require finding the segment and then locating the partition for the relevant blocks. Since the number of segments created is influenced by the error threshold (i.e., a smaller error threshold results in more segments), we use a function that estimates the number of segments created for a given dataset and error threshold. This function can either be learned for a specific dataset or a general function can be used [18]. Let S_ε denote the number of resulting segments for a given dataset when the error threshold is ε. Therefore, the estimated lookup latency for an error threshold of ε can be represented by the following expression, where a is the tree's fanout.

$$LATENCY(\varepsilon) = O(log_a(S\varepsilon)) \quad (10)$$

Since partitions inside a segment are normalized to have equal width, Sieve can simply determine the partition the searched key belongs to in $O(1)$ by subtracting the starting key of the segment from the searched key and dividing by the normalized width.

5.3 False positive rate

We first quantify the relationship between error and false positives. Ideally, there should be no block set changes (thus no false positives) inside a partition from the perspective of the linear segment. The real total number of block set changes in a partition p is:

$$real\ changes = true\_tbc(p.end) - true\_tbc(p.start) \quad (11)$$

In the worst case, the difference between the predicted and true tbc value is still bounded by the error ε:

$$\begin{aligned} &\Rightarrow (pred\_tbc(p.end) + \varepsilon) - (pred\_tbc(p.start) - \varepsilon) \\ &= 2\varepsilon \end{aligned} \quad (12)$$

The false positive rate (fpr) of a partition (p) depends on the real total number of block set changes, and we can obtain the rate via the following equation:

$$p_{fpr} = 1 - \frac{1}{2\varepsilon} \quad (13)$$

Given a storage budget S_{req}, Sieve needs to adjust the width of partitions inside some segments to enable fewer number of partitions so as to fit a limited amount of partition metadata (i.e., corresponding block IDs for a partition) to the storage. Along with

the error threshold, this adjustment can also cause false positives because it eagerly groups more neighboring keys into a partition.

Assume we have a total of $D.p_{num}$ partitions for a given S_{req} in the dataset (See Section 5.1 for more details about the estimation of $D.p_{num}$), and $1-m$ percent of the $D.p_{num}$ partitions are not adjusted. For these $1-m$ percent of partitions, their p_{fpr} are only bounded by ε. On the contrary, the rest m percent of partitions violate ε, so their p_{fpr} is 1 in the worst case.

Assume there are $D.block_{num}$ blocks in the dataset, so the number of false positive blocks (fb) of a partition is estimated as follows:

$$p_{fb} = (m \cdot 1 + (1-m)(1 - \frac{1}{2\varepsilon})) \cdot \frac{D.block_{num}}{D.p_{num}} \tag{14}$$

For range queries, the query's selectivity (SF) (i.e., the number of tuples that satisfy the predicate) also affects the false positive rate. Consider the example in Figure 1 in which the {1005->None} key-block pair is merged with {1006->B_1} into a partition. In this example, Sieve answers the query `key=1005` with block B_1, resulting in false positives. However, for range query `key=1005 and key=1006`, Sieve also answers with block B_1, incurring no false positives. Since the false positives incur only at the starting and ending partition of a range query, the final false positive rate is:

$$false\ positive\ rate = \frac{2 \cdot p_{fb}}{SF \cdot D.block_{num}} = \frac{2 \cdot (1 - \frac{1-m}{2\varepsilon})}{SF \cdot D.p_{num}} \tag{15}$$

In conclusion, the false positive rate is determined by (1) error threshold (ε) and (2) storage budget (S_{req}), and (3) query selectivity factor (SF).

Given Equation 15, we observe the following: (1) For a certain ε and S_{req}, the higher the value of SF, the fewer false positives there exist. (2) when S_{req} and SF are fixed, the higher ε there is, the more false positives there are. (3) when SF and ε are fixed, the smaller S_{req} there is, the more false positives there exist.

When working on sparse data with a fixed storage budget, Sieve has to either (1) group the missing keys with the neighboring existing keys to the same region, or (2) record those missing key ranges as individual regions. The former approach requires a larger ε, which imposes more false positives according to Equation 15. The latter method requires more storage budget for generating segments. In this case, the value of $D.p_{num}$ gets lower (Equation 9), leading to more false positives.

6 LIMITATIONS OF SIEVE

Sparse Data. In general, for a given storage budget, Sieve performs better on dense datasets than on sparse datasets. Sparse data would introduce gaps between keys. In this case, Sieve has to either group the missing keys with the neighboring existing keys to the same region, or record those missing key ranges as individual regions. The former approach stretches the CDF over a larger key range, making slopes less steep and partitions wider, which can lead to increased false positives. The latter approach pays the cost of larger storage space.

To balance the storage consumption and false positive rate, Sieve utilizes a greedy streaming algorithm that, given the starting point of a segment, attempts to maximize the length of a segment while satisfying a given error threshold. As shown in Figure 4, a gap can be added to a segment if and only if the entire gap does not violate the error constraint of any previous key in the segment.

In Section 5.3, we provide a theoretical analysis of the false positive rate of Sieve on sparse data. In Section 7, we conduct an extensive performance evaluation of Sieve on sparse data.

Correlation Between the Indexed Attribute and the Underlying Block Distribution. Sieve is effective when applied to datasets characterized by consistent block distribution patterns among neighboring keys. In other words, *a series of* consecutive keys are likely to belong to the same block set or be associated with different ones.

For instance, real-time applications such as sensor monitoring [20] and Facebook [13] continuously receive timestamped data, resulting in close timestamps being stored in the same block. Another example is the analysis of page view statistics for Wikipedia pages, which provides insights into the number of visitors a page receives within a specific time frame. The ViewCount attribute is indexed to record the cumulative number of page visits per hour. ViewCounts with small values (key ranges 31-36 in Figure 1) are spread across nearly all the periodic blocks, reflecting the fact that most pages attract a relatively low number of visits. Conversely, larger ViewCounts (key ranges 1001-1006 in Figure 1) tend to be allocated to distinct periodic blocks, primarily due to the presence of emerging hot topics.

7 EVALUATION

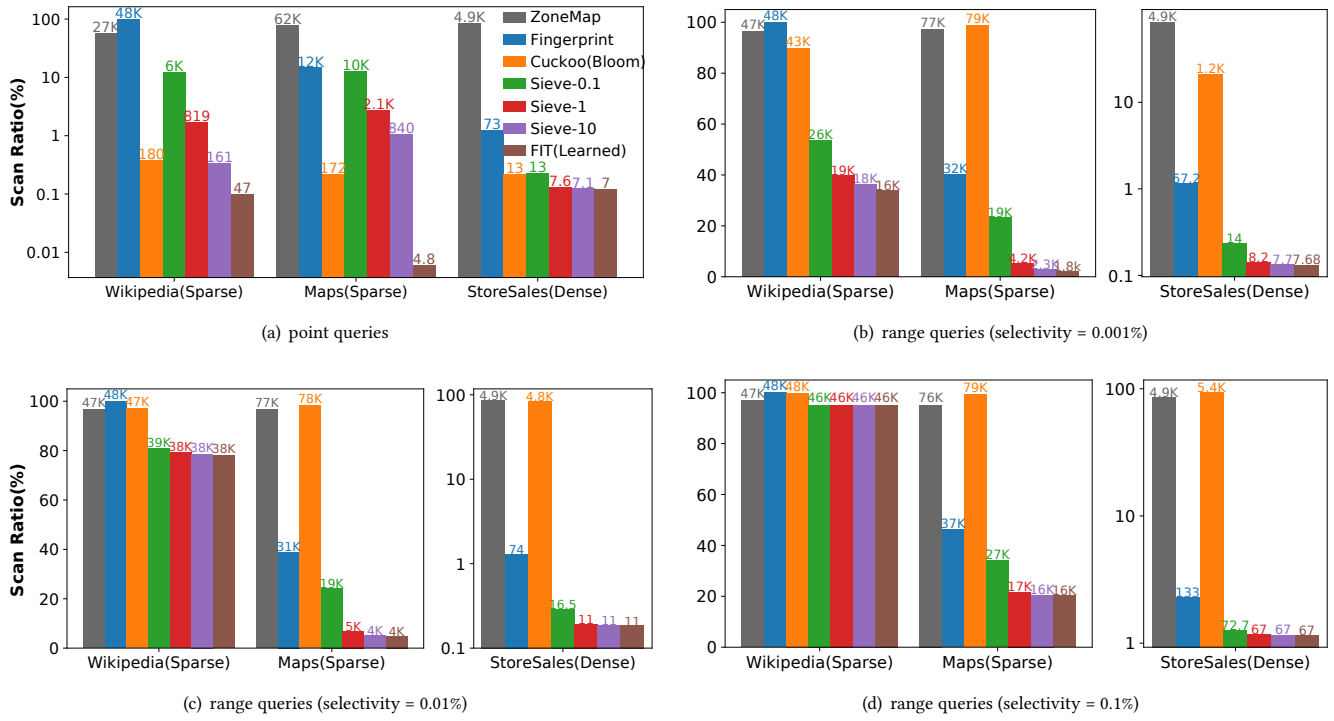
In Section 7.2, we compare the overall performance of Sieve with existing data-skipping techniques using both sparse and dense datasets. In Section 7.3, we measure the construction cost of Sieve. Section 7.4 measures Sieve's maintenance performance when inserting various amounts of data. In Section 7.5, we show the impact of block size on the filtering performance of Sieve. Finally, Section 7.6 shows how Sieve performs for adversarial synthetically generated datasets (i.e., worst-case data distribution).

7.1 Setup

Infrastructure: All experiments were conducted using Presto 3.7.0 on an eight-node cluster, each with Linux 4.15, 2.20 GHz dual-socket hex-core Intel Xeon with 20 hyperthreading cores, 192GB memory, 1TB HDD, and 480GB SSD. All data is stored as Parquet files with a block size of $50K$ records reference to production practice [44] (we also show the influence of block size in Section 7.5).

Compared Schemes: We experimentally compare Sieve with the following schemas: (1) Fingerprint (FP) [28]: a recently announced data-skipping method that uses heuristic-based histograms to represent the data distribution within each block. (2) Cuckoo index[25]: a recently announced set membership filter that extended Cuckoo filters with variable-sized fingerprints to avoid key shadowing. (3) ZoneMap: a widely used index that maintains the minimum and the maximum value of each block. (4) FIT [18]: a learned index optimized for a full B+ tree. Theoretically, FIT should have the optimal filtering performance because it maintains all the key-block pairs.

Workloads: Cloud-based data analytics services typically push predicate evaluation down to the data source for early data filtering. Thus, we focus on scan-intensive queries in our analysis. Following previous work [34], we use queries of the template below to avoid



(a) point queries

(b) range queries (selectivity = 0.001%)

(c) range queries (selectivity = 0.01%)

(d) range queries (selectivity = 0.1%)

Figure 5: Scan ratio (i.e., the fraction of blocks accessed out of the total number of blocks) at different selectivity factors.

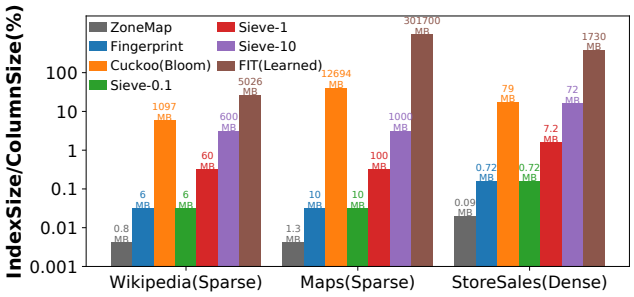


Figure 6: Index size on different datasets.

large overhead in the output of the resulting tuples, which may affect the measurement time ($OP : <, >, =$):

```
SELECT agg(A), agg(B), ..., agg(N) FROM R
WHERE A OP X (AND A OP Y)
```

Datasets: Since the performance of Sieve depends on the distribution of blocks in a given dataset, we evaluate its performance on three real-world datasets with different distributions, including two sparse datasets and one dense dataset. Similar to previous works [26, 32, 36], we use *Sparsity Degree* to describe the sparsity of a keyset. Formally, we denote a key by k and its key universe as $\mathcal{K}$, where $|\mathcal{K}| = m$. The set of all keys of an index is denoted as $K \subseteq \mathcal{K}$. The set of keys K has size n and contains no multiplicities. The *Sparsity Degree* of a keyset is defined as $1 - |K|/|\mathcal{K}| = 1 - n/m$. The Wikipedia dataset [2], with a high *Sparsity Degree* of 0.99, records how many people have visited an article per hour for the period from December 2007 to July 2016. The indexed attribute is ViewCount. The Maps dataset[5] contains the longitude of 2B features (e.g., museums, coffee shops) across the world. The longitude of locations in this dataset shows a *Sparsity Degree* of 0.7. The StoreSales dataset from the decision support benchmark TPC-DS[9] is dense with *Sparsity Degree* equal to 0. The indexed attribute is TicketNumber in StoreSales.

7.2 Exp.1: Overall Performance

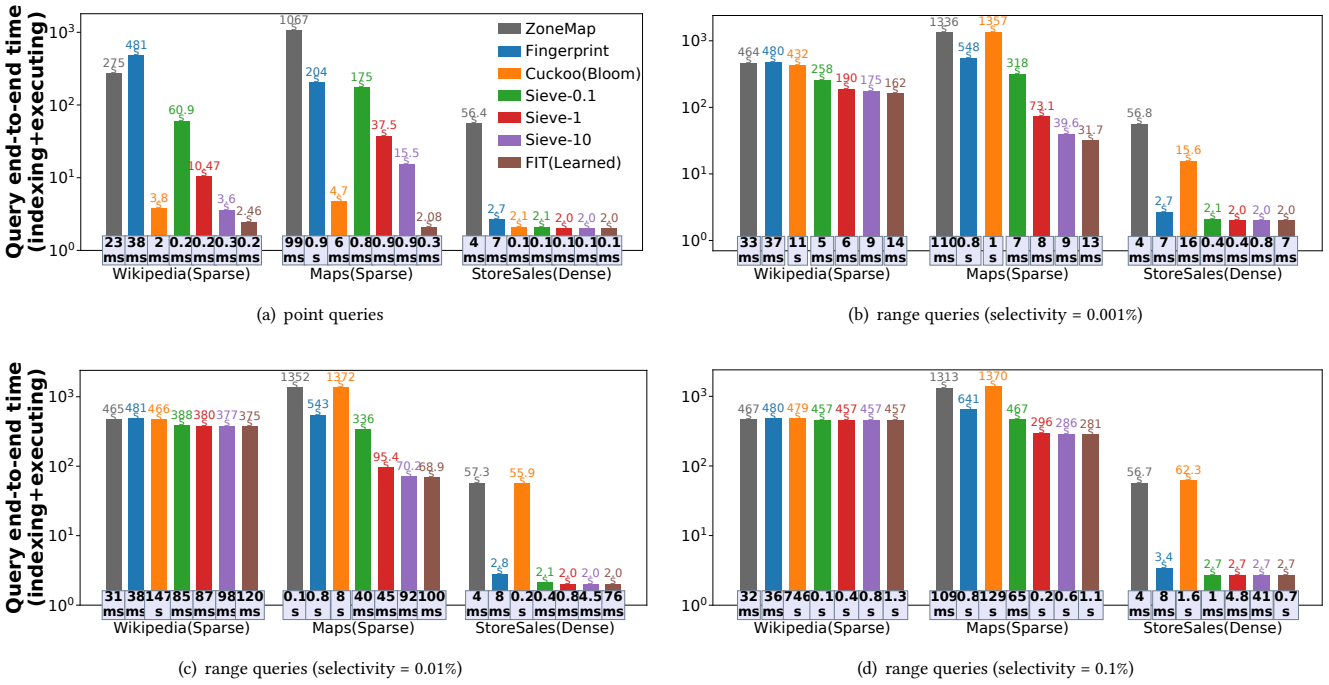
This section studies the overall performance of Fingerprint (FP), Cuckoo index, ZoneMap, and Sieve (with different storage budgets). Sieve-0.1 means the index size of Sieve is limited to 0.1% of the indexed column.

7.2.1 *Performance on Dense Data.*

Index size on dense data: ZoneMap shows the lowest storage overhead since it only maintains the min/max values for each block. Fingerprint also achieves low storage cost since it tends to maintain lightweight summaries, and Sieve incurs a similar space when the index budget is limited to 0.1% (Sieve-0.1). Moreover, the Cuckoo index gives a much higher storage budget (two orders of magnitude larger than Sieve-0.1).

Accessed blocks with different selectivity factors: Figure 5 shows the scan ratio (i.e., the fraction of blocks accessed out of the total number of blocks) on the dense dataset StoreSales.

FIT achieves optimal performance at the cost of maintaining all the key-block pairs. Sieve is also able to achieve closer to optimal



(a) point queries

(b) range queries (selectivity = 0.001%)

(c) range queries (selectivity = 0.01%)

(d) range queries (selectivity = 0.1%)

Figure 7: Breakdown of end-to-end query response time. The Y-axis is split into two portions. The bottom portion represents the index processing time (The index processing time of Cuckoo is orders of magnitude higher than Sieve on range queries.), while the top portion denotes the time spent on query execution.

performance across all selectivities, even using the smallest storage budget (Sieve-0.1). In addition, we can also see that the performance of Sieve improves as the query selectivity and storage budget size increase, which confirms our cost model in Section 5.3.

For point query, we can see Sieve-0.1 achieves comparable performance with Cuckoo index and achieves 82% reduction in block accesses compared to Fingerprint. The reason is StoreSales shows similar block distribution trends for neighboring keys and is almost unaffected by missing keys. For range queries, we can see Sieve-0.1 achieves 45%-80% reduction in block accesses compared to the best alternative method.

Response time with different selectivity factors: Figure 7 presents the breakdown of query response time. For point queries, Cuckoo index, FIT, and Sieve have similar end-to-end time because they are all able to filter most of the blocks, and thus the query execution dominates the end-to-end time.

As described in Cuckoo's paper and source code, Cuckoo Index is limited to equality predicates and does not support range predicates. To make Cuckoo support range queries, we have to do hash checking for every key (including both missing keys and existing keys) in the specified range. As depicted in Figure 7(b), Figure 7(c), and Figure 7(d), Cuckoo Index imposes a much higher overhead in indexing time than other techniques for range queries.

For range queries, we can see the reduction in query execution time (20% to 25%) is not as pronounced as for access blocks, when compared to the best counterpart. The reason is the large number of filtered data makes the I/O bottleneck less evident.

7.2.2 Performance on Sparse Data.

Index size on sparse data: As shown in Figure 6, Fingerprint and Sieve-0.1 still show low storage overhead on both sparse datasets, Wikipedia and Maps. Cuckoo index occupies 0.82 to 11.6 times more storage space compared to Sieve-10 on these datasets.

Accessed blocks with different selectivity factors: Figure 5 shows the scan ratio on Sparse datasets Wikipedia and Maps.

As expected, Sieve achieves lower performance on sparse data. The reason is that sparse data causes gaps (i.e., missing keys) in the key space, forcing Sieve to either allocate more space to capture the gaps or tolerate more false positives by grouping the gaps with neighboring keys (Section 6).

It is interesting to note that the distribution of gaps over the key space has a great impact on Sieve's performance. More specifically, although Wikipedia has a much higher *sparsity degree* than Maps, Sieve achieves closer to optimal performance in terms of scan ratio in the Wikipedia dataset. This is due to the fact that the missing keys of Wikipedia dataset show a clustered pattern over the key space, primarily because of the rare occurrence of large ViewCounts. In this case, Sieve is able to capture these gaps with a small number of segments. On the contrary, the missing keys of Maps have a dispersed pattern, since the longitude of locations is scattered across the world. Therefore, Sieve needs to pay more false positives given the same storage space on the Maps dataset.

Cuckoo index is essentially an enhanced bloom filter that enables data skipping for secondary columns by associating a key for the column to the multiple blocks containing it. Since Cuckoo index is

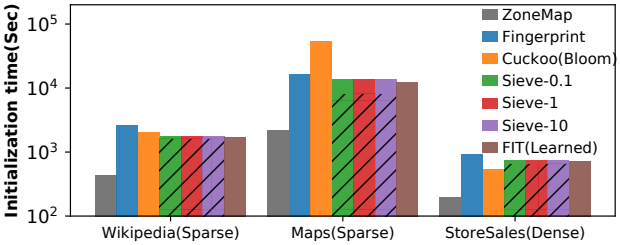


Figure 8: Index initialization time (logarithmic scale) on different datasets. The shaded parts in Sieve represent the time to build a sorted array of key->block pairs.

designed for equality predicates, it exhibits poor performance on range queries.

For point queries, we observe that compared with Maps and StoreSales, Cuckoo index pays less storage space to achieve a similar scan ratio in the Wikipedia dataset because low ViewCounts in Wikipedia are common and distributed in a majority of blocks. Furthermore, Sieve exhibits a storage space reduction of at least 45% compared to Cuckoo index, while maintaining comparable performance on sparse datasets. This is due to the fact that Cuckoo index focuses on indexing individual keys, while Sieve leverages block distribution patterns over the entire key space.

Response time with different selectivity factors: As shown, on point queries, Sieve-10 achieves similar performance to Cuckoo index on Wikipedia but causes 2.3x more time on the Maps dataset (although still significantly less than ZoneMap and Fingerprint). On range queries, Sieve-0.1 achieves up to 42% reduction in query execution time when compared to the best counterparts.

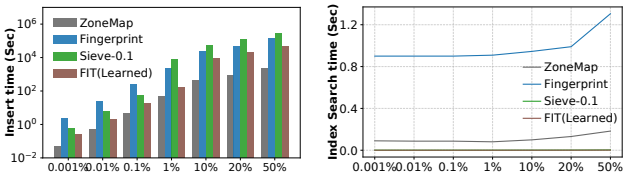
7.3 Exp.2: Initialization overhead

In Figure 8, we quantify the cost of index construction. Same as typical B+ trees, Sieve's initialization process starts with building a sorted array of key->block pairs. Based on the sorted array, Sieve uses a one-pass algorithm to generate segments and partitions that record the information about a region of the key-block pairs.

As shown in the shaded parts of Figure 8, the major cost in Sieve's initialization is to scan the records and build the sorted array. FIT and Sieve achieve comparable performance because FIT does similar operations in initialization. Same as Sieve, Cuckoo index also needs to examine every element in the data. However, Cuckoo index has a relatively higher initialization overhead than Sieve due to its heavy use of hash computation. As expected, ZoneMap has the lowest initialization overhead because it simply records summarized statistics. On average, the initialization in Sieve takes about 3x longer than ZoneMap.

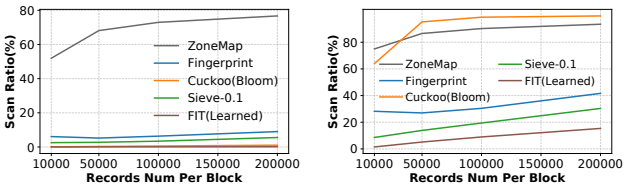
7.4 Exp.3: Impact of data insertion

This section investigates the impact of data insertion. This experiment uses a fair setting which counts the insert time and search time after randomly inserting a certain amount (0.001%, 0.01%, 0.1%, 1%, 10%, 20%, 50%) of records. Due to space limitation, we only show the performance on the sparse dataset Maps because Sieve exhibits the poorest performance on this dataset. As described in



(a) Insertion time (logarithmic scale). (b) Search speed.

Figure 9: Impact of data insert at different insertion percentages on Maps(Sparse) dataset.



(a) Maps(Sparse): point queries (b) Maps(Sparse): range queries

Figure 10: Impact of block size on filtering performance.

Cuckoo's paper [25] and source code [7], it is designed for a read-only (immutable) setting and does not support inserts, so its insert performance is not listed in Figure 9.

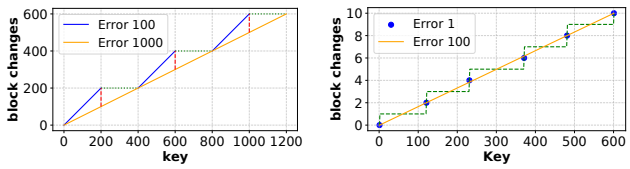
Insert Time. When inserting less than 1% of data, Sieve outperforms Fingerprint by up to 3.3x because the segment rebuild threshold is not violated, whereas Fingerprint needs to update the histograms of the affected block for each insertion. Sieve is 12x slower than ZoneMap because Sieve needs to search the tree for the corresponding segment and partition of the newly inserted key.

In Figure 9(a), when 1% of data is inserted, Sieve takes 2.2x longer than Fingerprint due to re-segments. Sieve is slower than FIT because FIT may trigger re-segment only when new keys are inserted, while Sieve considers block set changes from any partition. Note that when more than 1% of data is inserted, the amount of extra overhead imposed by Sieve is correlated with the number of re-segments, which grows linearly with the volume of data inserted.

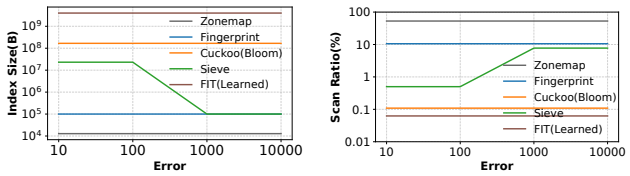
Search Time. As shown in Figure 9(b), Sieve scales better than other indexes in terms of lookup latency. This is because the search time of ZoneMap and Fingerprint grows linearly with the total number of blocks, but Sieve's search time grows logarithmically with the number of segments.

7.5 Exp.4: Block Size Scalability

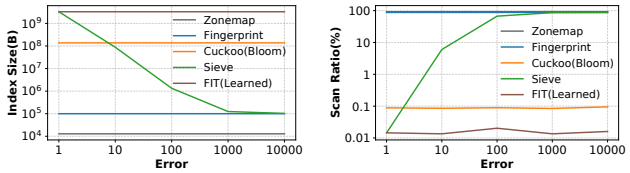
As shown in Figure 10, the performance of all existing indexes is significantly affected by the block size (i.e., the number of tuples in each block). As expected, a smaller block size yields better filtering performance. Sieve's scan ratio increases by 3% for point queries and 21% for range queries when the block size grows from 10,000 to 200,000. Even though, Sieve still achieves the best performance on range queries across all block sizes, and only causes 4% more scan ratio than Cuckoo index on point queries.



(a) Block distribution on worst-case dense dataset. (b) Block distribution on worst-case sparse dataset.



(c) Index size on worst-case dense dataset. (d) Scan ratio on worst-case dense dataset.



(e) Index size on worst-case sparse dataset. (f) Scan ratio on worst-case sparse dataset.

Figure 11: Worst Case Analysis.

7.6 Exp.5: Worst Case Data

Since the data distribution influences the performance of Sieve, we synthetically generated data to illustrate how our index performs with both sparse and dense data representing a worst-case.

Worst-case for dense data For dense data, we define the worst-case as a dataset which maximizes the number of segments given a specific error, and every two consecutive keys map to different blocks. To do this, we generate data using a step function with a fixed step size of 200, as shown in Figure 11(a) (blue line).

As shown in Figure 11(c), for error thresholds less than 100, the size of Sieve is larger than Fingerprint but still smaller than Cuckoo index and FIT. This is due to the fact that for error thresholds less than 100, Sieve creates segments of width 200, resulting in a large number of nodes in the tree. For an error threshold that is larger than 100, Sieve is able to represent the dataset with only a single segment, dramatically reducing the index's size while achieving comparable performance with Fingerprint.

Worst-case for sparse data For sparse data, we define the worst-case as a dataset in which every two consecutive existing keys are separated with a gap. As shown in Figure 11(b), we set the sparsity degree of synthetic data to 0.99.

As shown in Figure 11(e) and Figure 11(f), when the error threshold is set to 1, Sieve causes a similar storage cost to the FIT index and achieves no false positives since no key is grouped at this threshold. On the other hand, as the error threshold increases, the index size of Sieve decreases, and Sieve achieves comparable filtering performance with Fingerprint when the error threshold is set to 1000.

8 RELATED WORK

8.1 Data Skipping

While ZoneMap[33, 35, 40] is widely used to filter data, its performance is impacted by gaps in the blocks. Column Imprints [38] and Column Sketches [22] speed up scans by maintaining lossy index structures, but would resemble Zonemap when used on larger data blocks than cache lines [25]. GRT [14], Hippo [43], and Fingerprints [28] try to capture gaps within blocks with specific structures (e.g., histograms) but may incur severe false positives. Set-membership filters[25, 29] achieve a low false positive ratio but unsuitable for practical distributed storage systems due to the relatively sizable required space [23]. (Tree-based) index[41] eliminates all false positives but causes unacceptable index overhead. (Compressed) bitmap indexes[21, 39] can reduce storage cost but mainly suit low cardinality attributes which are quite rare. Unlike existing methods, Sieve groups keys into key ranges based on captured block distribution trends, thus achieving a balance between false positives and space.

8.2 Learned Index

In cloud-based OLAP systems, a basic fine-grained index stores all key->block pairs in a sorted array. Although learned index [15, 17, 18, 27] can quickly output the position of a given key in the sorted array by replacing the traditional index structure with machine learning models, it still needs to maintain all the key-block pairs. Different from the learned index, Sieve is designed to optimize the storage space of the sorted array by storing information about a region of the key space with similar block distributions, instead of indexing individual keys.

Although both learned index and Sieve use piece-wise linear functions to approximate the CDF model, CDF models are built over different variables, and the approximate linear functions are also used in different scenarios: (1) learned index uses CDF functions to model the physical position of a key while Sieve employs CDF functions to model the block distribution differences between keys and (2) unlike learned index, which replaces the B+ tree structure with approximated linear functions to quickly locate the position of a key, Sieve employs linear functions to find the optimal key ranges to group to balance false positives and storage space.

9 CONCLUSION

Modern data analytics need to handle large amounts of block-based data stored in remote storage, making I/O a bottleneck. To effectively balance false positives and storage overhead, we present Sieve, a learning-enhanced index that exploits piece-wise linear functions to approximate the block distribution trends. Based on captured trends, Sieve groups individual keys into regions to reduce index size. Our evaluation of Sieve using real-world datasets demonstrates that it can effectively eliminate the I/O bottleneck.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (Grant No.62172180, No.62232007, No.61821003). We sincerely thank the anonymous reviewers for their insightful feedback and suggestions. We also thank CloudLab for contributing free computing resources.

REFERENCES

[1] 2016. Amazon Redshift Engineering's Advanced Table Design Playbook: Compound and Interleaved Sort Keys. Retrieved October 7, 2022 from https://aws.amazon.com/cn/blogs/big-data/amazon-redshift-engineerings-advanced-table-design-playbook-compound-and-interleaved-sort-keys/

[2] 2016. Page view statistics for wikimedia projects. Retrieved June 5, 2022 from https://dumps.wikimedia.org/other/pagecounts-raw/

[3] 2017. Azure Data Lake Analytics. Retrieved October 5, 2022 from https://azure.microsoft.com/en-us/services/data-lake-analytics/

[4] 2018. Amazon S3. Retrieved October 7, 2022 from http://aws.amazon.com/s3/

[5] 2018. OpenStreetMap database. Retrieved June 5, 2022 from https://aws.amazon.com/public-datasets/osm

[6] 2019. Windows Azure Storage BLOB. Retrieved September 11, 2022 from https://azure.microsoft.com/en-us/services/storage/blobs/

[7] 2020. Source code of Cuckoo Index. Retrieved May 11, 2023 from https://github.com/google/cuckoo-index/

[8] 2021. Database Data Warehousing Guide: Using Zone Maps. Retrieved July 5, 2022 from https://docs.oracle.com/en/database/oracle/oracle-database/19/dwhsg/using-zone-maps.html

[9] 2021. TPC-DS database. Retrieved June 5, 2022 from https://trino.io/docs/current/connector/tpcds.html

[10] 2022. Columnstore indexes - Query performance. Retrieved June 20, 2022 from https://learn.microsoft.com/en-us/sql/relational-databases/indexes/columnstore-indexes-query-performance

[11] Manos Athanassoulis and Anastasia Ailamaki. 2014. BF-tree: approximate tree indexing. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1881–1892.

[12] Rudolf Bayer and Karl Unterauer. 1977. Prefix B-trees. *ACM Transactions on Database Systems (TODS)* 2, 1 (1977), 11–26.

[13] Dhruba Borthakur. 2013. Petabyte scale databases and storage systems at facebook. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1267–1268.

[14] Jeremy Chen, Reza Sherkat, Mihnea Andrei, and Heiko Gerwens. 2018. Global Range Encoding for Efficient Partition Elimination. In *International Conference on Extending Database Technology (EDBT)*. 453–456.

[15] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. 2020. ALEX: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 969–984.

[16] Hazem Elmeleegy, Ahmed Elmagarmid, Emmanuel Cecchet, Walid G Aref, and Willy Zwaenepoel. 2009. Online piece-wise linear approximation of numerical streams with precision guarantees. *Proceedings of the VLDB Endowment* 2, 1 (2009), 145–156.

[17] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1162–1175.

[18] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 International Conference on Management of Data*. 1189–1206.

[19] Goetz Graefe and P-A Larson. 2001. B-tree Indexes and CPU Caches. In *Proceedings 17th International Conference on Data Engineering*. IEEE, 349–358.

[20] IS Group et al. 2012. Managing big data for smart grids and smart meters. *IBM Corporation, whitepaper (May 2012)* (2012).

[21] Gheorghi Guzun, Guadalupe Canahuate, David Chiu, and Jason Sawin. 2014. A tunable compression framework for bitmap indices. In *2014 IEEE 30th international conference on data engineering*. IEEE, 484–495.

[22] Brian Hentschel, Michael S Kester, and Stratos Idreos. 2018. Column sketches: A scan accelerator for rapid and robust predicate evaluation. In *Proceedings of the 2018 International Conference on Management of Data*. 857–872.

[23] Srikanth Kandula, Laurel Orr, and Surajit Chaudhuri. 2019. Pushing data-induced predicates through joins in big-data clusters. *Proceedings of the VLDB Endowment* 13, 3 (2019), 252–265.

[24] Eamonn Keogh, Selina Chu, David Hart, and Michael Pazzani. 2001. An online algorithm for segmenting time series. In *Proceedings 2001 IEEE international conference on data mining*. IEEE, 289–296.

[25] Andreas Kipf, Damian Chromejko, Alexander Hall, Peter Boncz, and David G Andersen. 2020. Cuckoo index: a lightweight secondary index structure. *Proceedings of the VLDB Endowment* 13, 13 (2020), 3559–3572.

[26] Evgenios M Kornaropoulos, Silei Ren, and Roberto Tamassia. 2022. The price of tailoring the index to your data: Poisoning attacks on learned index structures. In *Proceedings of the 2022 International Conference on Management of Data*. 1331–1344.

[27] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*. 489–504.

[28] Carmen Kwan. 2019. Fingerprints for Compressed Columnar Data Search. In *Proceedings of the 2019 International Conference on Management of Data*. 1835–1837.

[29] Harald Lang, Thomas Neumann, Alfons Kemper, and Peter Boncz. 2019. Performance-optimal filtering: Bloom overtakes cuckoo at high throughput. *Proceedings of the VLDB Endowment* 12, 5 (2019), 502–515.

[30] Domine Leenaerts and Wim MG Van Bokhoven. 2013. *Piecewise linear modeling and analysis*. Springer Science & Business Media.

[31] Xiaoyan Liu, Zhenjiang Lin, and Huaiqing Wang. 2008. Novel online methods for time series segmentation. *IEEE Transactions on Knowledge and Data Engineering* 20, 12 (2008), 1616–1626.

[32] Masoud Makrehchi and Mohamed S Kamel. 2008. Automatic extraction of domain-specific stopwords from labeled documents. In *Advances in Information Retrieval: 30th European Conference on IR Research, ECIR 2008, Glasgow, UK, March 30-April 3, 2008. Proceedings 30*. Springer, 222–233.

[33] Guido Moerkotte. 1998. Small materialized aggregates: A light weight index structure for data warehousing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 476–487.

[34] Matthaios Olma, Manos Karpathiotakis, Ioannis Alagiannis, Manos Athanassoulis, and Anastasia Ailamaki. 2017. Slalom: Coasting through raw data via adaptive partitioning and indexing. *Proceedings of the VLDB Endowment* 10, 10 (2017), 1106–1117.

[35] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. 2013. DB2 with BLU acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1080–1091.

[36] Hassan Saif, Miriam Fernandez, Yulan He, and Harith Alani. 2014. On stopwords, filtering and data sparsity for sentiment analysis of twitter. (2014).

[37] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. 2019. Presto: SQL on everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1802–1813.

[38] Lefteris Sidirourgos and Martin Kersten. 2013. Column imprints: a secondary index structure. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 893–904.

[39] Kurt Stockinger and Kesheng Wu. 2007. Bitmap indices for data warehouses. In *Data Warehouses and OLAP: Concepts, Architectures and Solutions*. IGI Global, 157–178.

[40] Liwen Sun, Michael J Franklin, Jiannan Wang, and Eugene Wu. 2016. Skipping-oriented partitioning for columnar layouts. *Proceedings of the VLDB Endowment* 10, 4 (2016), 421–432.

[41] Grisha Weintraub, Ehud Gudes, and Shlomi Dolev. 2021. Needle in a haystack queries in cloud data lakes.. In *EDBT/ICDT Workshops*.

[42] Zhenghua Xu, Rui Zhang, Ramamohanarao Kotagiri, and Udaya Parampalli. 2012. An adaptive algorithm for online time series segmentation with error bound guarantee. In *Proceedings of the 15th International Conference on Extending Database Technology*. 192–203.

[43] Jia Yu and Mohamed Sarwat. 2016. Two birds, one stone: a fast, yet lightweight, indexing scheme for modern database systems. *Proceedings of the VLDB Endowment* 10, 4 (2016), 385–396.

[44] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, et al. 2019. Analyticdb: Real-time olap database system at alibaba cloud. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2059–2070.

[45] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. 2006. Super-scalar RAM-CPU cache compression. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 59–59.