Cabin: a Compressed Adaptive Binned Scan Index

YIYUAN CHEN and SHIMIN CHEN[*], SKLP and Center for Advanced Computer Systems, Institute of Computing Technology, CAS, China and University of Chinese Academy of Sciences, China

Scan is a fundamental operation widely used in main-memory analytical database systems. To accelerate scans, previous studies build either record-order or sort-order structures known as *scan indices*. While achieving good performance, scan indices often incur significant space overhead, limiting their use in main-memory databases. For example, the most recent and best performing scan index, BinDex, consists of a sort-order position array, which is an array of rowIDs in the value order, and a set of record-order bit vectors, representing records in pre-defined value intervals. The structures can be much larger than the base data column size.

In this paper, we propose a novel scan index, Cabin, that exploits the following three techniques for better time-space tradeoff. 1) *filter sketches* that represent every $2^w - 2$ value intervals with a w-bit sketched vector, thereby exponentially reducing the space for the bit vectors; 2) *selective position array* that removes the rowID array for a fraction of intervals in order to lower the space overhead for the position array; and 3) *data-aware intervals* that judiciously select interval boundaries based on the data characteristics to better support popular values in skewed data distributions or categorical attributes. Experimental results show that compared with state-of-the-art scan solutions, Cabin achieves better time-space tradeoff, and attains 1.70 − 4.48x improvement for average scan performance given the same space budget.

CCS Concepts: • **Information systems** → **Data access methods**; **Data scans**.

Additional Key Words and Phrases: scan index; time-space tradeoff; filter sketches; data awareness

ACM Reference Format:
Yiyuan Chen and Shimin Chen. 2024. Cabin: a Compressed Adaptive Binned Scan Index. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 57 (February 2024), 26 pages. https://doi.org/10.1145/3639312

1 INTRODUCTION

Scan is a fundamental operation widely used in analytical data processing [9, 13, 16–18, 23, 25, 26, 30, 31, 34, 35, 37]. Given a base data column in a table and a filter predicate, a scan operation returns a result bit vector or a rowID list that indicates which records satisfy the predicate. A plain scan reads the data column sequentially and evaluates the predicate for each data value. There are two main ways to improve the plain scan in the literature. The first approach is to optimize the layout of the base data column so that the scan can effectively exploit SIMD to speed up computation, and reduce the amount of data accessed with early pruning techniques [13, 26]. The second approach does not change the base data. Instead, it builds auxiliary data structures (a.k.a. *scan indices*) to accelerate scans [11, 16, 25, 27, 29]. Compared to the first approach, scan indices have been shown to achieve higher performance. However, scan indices often incur significant space overhead, limiting their use in main-memory databases. In this paper, we aim to improve the time-space tradeoff for scan indices.

[*]Shimin Chen is the corresponding author.

Authors' address: Yiyuan Chen, chenyiyuan20s@ict.ac.cn; Shimin Chen, chensm@ict.ac.cn, SKLP and Center for Advanced Computer Systems, Institute of Computing Technology, CAS, No. 6 Ke Xue Yuan South Rd, Haidian District, Beijing, China and University of Chinese Academy of Sciences, No.1 Yanqihu East Rd, Huairou District, Beijing, China.

ACM 2836-6573/2024/2-ART57
https://doi.org/10.1145/3639312

We consider the design principles for scan indices. A scan deals with two data orders: the record order and the sort order. Both the base data and the scan result are in the record order. On the other hand, for a predicate (e.g., $<$, $>$, $\leq$, $\geq$, $=$, $\neq$, or BETWEEN) that compares the data value with constant(s), the evaluation can be better supported if the data is sorted. The two orders are the same if values are sorted in the base data column. In this case, a search on the data column can efficiently compute the result bit vector or rowID list. However, the more common and challenging case is where values are not sorted. This is the focus of previous scan index studies, as well as the focus of our work.

To improve scans, scan indices build either record-order structures or sort-order structures or both. Record-order structures can reduce the amount of access to the base data, while sort-order structures can accelerate the computation of the predicate.

- B+-Tree [11] is a sort-order scan index. A scan performs a search, then follows the linked list of leaf nodes to retrieve all rowIDs that satisfy the predicate. This process is efficient if the predicate selectivity is low. However, for medium to high selectivities, it incurs many expensive random memory accesses.
- Bitmap Index [29] maintains record-order bit vectors for each distinct values. It can retrieve a stored bit vector for a scan with an equality predicate. However, for other predicate types, it has to merge multiple bit vectors, and thus becomes less efficient if more distinct values satisfy the scan predicate.
- Column Sketches [16] create a sketched column in the record order. Value ranges are mapped to small (e.g., 8-bit) code words. Then the sketched column stores a code per record. A scan evaluates the predicate by checking the per-record code in the sketched column. This is sufficient for most records. For a small fraction (e.g., $\frac{1}{256}$) of records, the check is uncertain and the scan reads the base data values. In this way, Column Sketches significantly reduce the amount of data to read.
- BinDex [25], the most recent and best performing solution, combines both sort-order and record-order structures to speed up scans. The former is an array of rowIDs (a.k.a. position array) in the sort order of values. The latter is a set of record-order bit vectors, each representing the records in a pre-defined value range. For a scan, the predicate specifies a target value range. BinDex finds the closest pre-defined range to the target range, and copies the associated bit vector as the draft result. Then, it searches the position array for records in the intersection of the target and the pre-defined range, and corrects the corresponding bits in the draft result to obtain the scan result.

However, the improvement of scan performance comes at a price: scan indices introduce significant space cost. Both record-order and sort-order structures store an element per record. The element size can be comparable or even larger than the value size in the base data. In a B+-Tree [11], there is a (value, rowID) entry per record. Suppose the rowID takes 4B. Then the B+-Tree incurs at least 5x, 3x, 2x, and 1.5x space overhead for 8-bit, 16-bit, 32-bit, and 64-bit values, respectively. In a Bitmap Index [29], every bit vector stores 1 bit per record. The space cost increases linearly as the number of distinct values. For example, the bit vectors for 64 distinct values lead to 1–8x space overhead for 64-bit – 8-bit values. Column Sketches [16] store a (e.g., 8-bit) code per record. Therefore, the space overhead is 0.125–1x, which is low among scan indices. In a BinDex [25], the position array contains a (e.g., 4B) rowID per record. Each bit vector costs 1 bit per record. The number of stored bit vectors is determined by the number of pre-defined value ranges. For example, if there are 64 pre-defined ranges, then the BinDex sees 1.5–12x space overhead for 64-bit – 8-bit values. As the number of pre-defined ranges increases, BinDex performs better because the intersection between a target range and its closet pre-defined range tends to become smaller.

Therefore, for achieving higher scan performance, it can be necessary to store more bit vectors, incurring even higher space overhead.

In this paper, we propose a novel scan index, Cabin (compressed adaptive binned scan index). Cabin exploits the following three techniques for better time-space tradeoff:

- *Filter Sketches*: BinDex stores a bit vector for each value range, which consumes a lot of space. To reduce space, we replace every $2^w - 2$ bit vectors with a w-bit sketched vector (e.g., $w \leq 9$). A w-bit code can represent $2^w - 2$ value intervals (and two reserved virtual intervals for all values less than or greater than the set of intervals). We call the resulting record-order structure *filter sketches*. Cabin divides all the value intervals into groups of $2^w - 2$ intervals each, and build w-bit filter sketches per interval group. In this way, Cabin reduces the space cost of bit vectors by a factor of $\frac{2^w-2}{w}$. In addition, we describe a novel encoding and computation scheme, MLO, that minimizes the number of SIMD operations for generating a draft result from filter sketches given a pre-defined value range.
- *Selective Position Array*: The position array in BinDex incurs a fixed space overhead. If space budget is tight, we propose to selectively store the position array, i.e., storing rowIDs for a fraction of intervals. If the predicate of a scan hits a value interval whose rowIDs are not stored, Cabin relies on the filter sketches for evaluating the predicate. Note that our design of filter sketches covers the entire value range $(-\infty, +\infty)$ for every interval group (with the two virtual intervals). Consequently, we can use filter sketches in a similar fashion to Column Sketches to support scans.
- *Data-aware intervals*: Bindex divides the full value range into even-sized intervals for obtaining the pre-defined value ranges. This works well when the number of duplicates of each distinct value is small. However, real-world data may contain a lot of duplicates because of skewed data distribution (e.g., power-law graphs) or low unique value counts (e.g., categorical attributes). In such cases, Cabin judiciously makes data-aware selection of intervals to optimize for popular values.

Contributions. The main contributions of this paper are threefold. First, we propose a scan index, Cabin[1], with three novel techniques, i.e., filter sketches, selective position array, and data-aware intervals, to improve the time-space tradeoff for scans. Second, we model the time and space cost of Cabin, and design an algorithm to compute the optimal design parameters for Cabin. Finally, we perform extensive experimental evaluation of Cabin under various data workloads. Our experimental results show that compared with state-of-the-art scan solutions, Cabin achieves better time-space tradeoff, and attains 1.70 − 4.48x improvement for average scan performance given the same space budget. In addition, we evaluate Cabin in a full fledged main memory analytical database system, MonetDB, using a subset of queries in TPC-H and SSB benchmarks. Our results show that Cabin can improve the query performance of MonetDB by a factor of 1.1−49.9x.

Organization. The rest of the paper is organized as follows. Section 2 provides the background on existing scan solutions. Section 3 presents the Cabin design. Then, Section 4 analyzes the space and time cost of Cabin for optimal design selection. After that, Section 5 performs the experimental evaluation. Section 6 discusses a number of practical issues. Finally, Section 7 concludes the paper.

2 BACKGROUND AND RELATED WORK

We describe background on data scans in Section 2.1, then discuss related work on scan indices in Section 2.2.

[1]The source code is available at https://github.com/schencoding/Cabin

2.1 Data Scan

Problem Definition. In this paper, we focus on single-column data scan operations. We consider the same problem definition as in previous studies [13, 16, 25, 26].

- *Base data*: A column of fixed sized values, which is the common storage layout in main-memory analytical databases [6, 8, 12, 22, 32, 38]. Note that variable sized fields are often encoded into fixed sized codes with dictionary encoding [7, 12].
- *Query predicate*: A filter predicate (e.g., <, >, ≤, ≥, =, ≠, or BETWEEN) that specifies a value range of the input data.
- *Output*: A result bit vector, whose i-th bit is set to 1 if the i-th record satisfies the predicate. Compared to other return formats (e.g., rowID list), bit vectors can be efficiently merged with bitwise operations to compute complex predicates [10].

Two Ways to Optimize Scans. A plain scan reads the base data sequentially and checks if each value satisfies the filter predicate. There are two main ways to improve plain scans in the literature.

The first approach is to optimize the layout of the base data. The design goals are to exploit SIMD and/or to reduce the amount of data accessed by the scan. BitWeaving [26] proposes bit-level layouts. In comparison, ByteSlice [13] proposes a byte-level columnar layout, where the i-th byte of all values are stored contiguously. Predicate evaluation employs SIMD to compare multiple values from the most to the least significant byte. It stops early if the higher-order bytes already determine the predicate outcome. This early pruning technique can significantly reduce the amount of data to read.

The second approach does not change the base data. Instead, it builds auxiliary data structures to improve scan performance. Examples include B+-Tree [11], Bitmap Index [29], Zone Maps [27], Column Sketches [16], and BinDex [25]. We call such auxiliary data structure *scan index*. Compared to the first approach, scan indices can perform better, but often incur significant space overhead. We aim to design a scan index with better time-space tradeoff.

2.2 Scan Index

Let us consider the principles of designing a scan index. From the problem definition, we see that there are two orders: the record order and the sort order. Both the base data and the output bit vector are in the record order, while predicate evaluation can be more efficient if the values are sorted. As a result, it is natural to consider three types of structures in scan indices:

- *Record-order structure*: It contains one item for every value in the base data. The order of the items is in the record order of the base data. An example is the bit vector in Bitmap Index.
- *Sort-order structure*: It contains one item for every value in the base data. The order of the items is in the sort order of the values. An example is the B+-Tree leaf nodes.
- *Summary structure*: The structure is small. It contains aggregates of the base data, or metadata of the other structures.

We examine the structures of each scan index, as shown in Table 1. Figure 1 depicts a high-level picture of the time-space trade-off of scan solutions. (Please see the space-time analysis under different workloads in our experiments in Section 5.)

B+-Trees [11]. In a B+-Tree, the leaf level stores the list of (value, rowID)'s in the sort order. This consumes more space than the base data. A scan performs a search to reach the leaf level, then follows the linked list of leaf nodes to retrieve all rowIDs that satisfy the predicate and sets the relevant bits in the result bit vector. This is efficient when the predicate selectivity is low, as shown in Figure 1. However, it can be costly with medium to high selectivities (not shown) because of pointer chasing and random memory accesses.

Table 1. Comparing scan index structures.

Scan Index	Record-Order Structure	Sort-Order Structure	Summary Structure
B+-Tree	-	tree nodes	-
Bitmap Index	bit vectors	-	key-to-vector map
Zone Maps	-	-	bucket aggregates
Column Sketches	sketched column	-	compression map
BinDex	filter bit vectors	position array	area map
Cabin (our solution)	filter sketches	selective position array	data-aware interval table

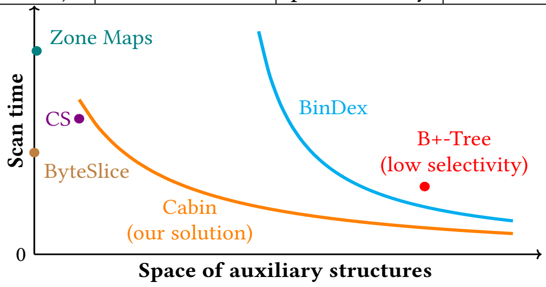


Fig. 1. Comparison of optimized scan solutions. (CS stands for Column Sketches. The performance of B+-Tree at low selectivity (e.g., 0.1%) is shown.)

Bitmap Index [29]. Bitmap index stores a bit vector for each distinct value, and a summary structure that maps keys to vectors. For an equality predicate, it returns the relevant bit vector, which is very fast. However, there are two main problems. First, for common data columns with many distinct values, Bitmap index can be much larger than other scan indices. Second, predicates that specify value ranges often require merging multiple bit vectors, leading to degraded performance. Hence, we do not further discuss or evaluate Bitmap index in the paper. Note that the above problems can be addressed by more sophisticated use of bitmaps, such as range-based bit vectors in BinDex and filter sketches in our solution.

Zone Maps [27]. It divides the input data into buckets, and maintains per-bucket aggregates (e.g., min, max). The summary structure incurs little space overhead, as shown in Figure 1. For a scan, Zone Maps skip a bucket if the bucket aggregates show that the entire bucket satisfies or dissatisfies the predicate. However, bucket skipping is effective only when values are mostly sorted or the selectivity is very low or very high. In other cases, Zone Maps essentially perform a plain scan, making it the slowest scan index.

Column Sketches [16]. As shown in Figure 2(a), Column Sketches consist of a sketched column and a compression map. The latter maps value ranges to codes. The former contains a code per value in the record order. In the example, x_0=5 $\in (-\infty, 8]$, thus its code is 000. x_1=23 $\in (22, 28]$, so its code is 100. The space cost is $\frac{N \cdot w}{8}$ bytes, where N is the number of records and w is the code width. Since w (e.g., 8 bits) is often smaller than value size (e.g., 64 bits), Column Sketches consume smaller space than the base data.

A scan first maps the predicate value to the (predicate) code. Then, it compares each code in the sketched column with the predicate code. In most cases, code ≠ predicate code, then the comparison suffices. When code = predicate code, the scan has to check the value in the base data. Therefore, the scan reads the sketched column and $\frac{1}{2^w}$ (e.g., $\frac{1}{256}$ for w=8) of the base data. Consequently, Column Sketches significantly reduce the amount of data to read.

BinDex [25]. BinDex combines features of B+-Tree and Bitmap Index, as shown in Figure 2(b). First, the position array and the virtual sorted values are similar to the sorted (value, rowID)'s in the B+-Tree's leaf level. To reduce space cost, BinDex stores only the rowIDs. The value part is virtual

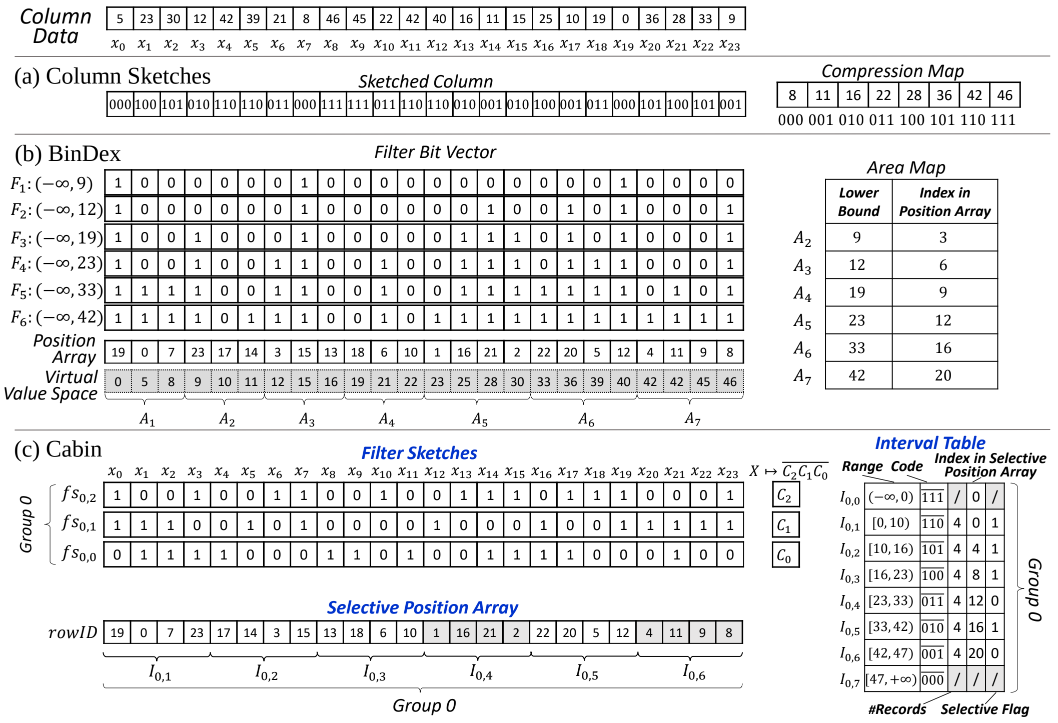


Fig. 2. Scan index structures of (a) Column Sketches, (b) BinDex, and (c) Cabin (with uniform data).

and not physically stored. Second, BinDex divides the position array into even-sized intervals (a.k.a. bins or areas), and keeps the interval bound positions in the area map. Third, there is a filter bit vector per interval. It stores the result for value < interval upper bound. This structure resembles Bitmap Index. When the scan predicate coincides with an interval bound, the associated filter bit vector can be directly returned.

For a scan (e.g., *value* ≤ 10), BinDex uses the area map to locate the corresponding interval (e.g., A_2=[9, 12)). Then, it performs a binary search in the interval, which follows the rowIDs to read the values in the base data. For example, the search finds the predicate value 10 at the second entry in A_2. After that, BinDex chooses and copies a filter bit vector (e.g., F_2 that represents $(-\infty, 12)$) as the draft result. Finally, it refines the draft result by flipping the wrong bits (e.g., bit 14 that corresponds to value 11).

Suppose there are N records and M intervals. Then, there are $M-1$ filter bit vectors. Suppose each rowID takes r bits. Thus, the space cost of BinDex is $\frac{N(r+M-1)}{8}$ bytes. The scan time consists of checking the area map ($O(\log M)$) , searching the interval ($O(\log \frac{N}{M})$), copying the filter bit vector as draft result (i.e., sequentially copying $\frac{N}{8}$ bytes), and refining the draft result (i.e., randomly accessing average $\frac{N}{4M}$ bits). Figure 1 shows the BinDex curve varying the number of intervals M. As M increases, the space cost increases linearly, while the number of random bit accesses decreases, leading to better performance.

Motivation of Our Solution: Cabin. While BinDex achieves the best performance among existing scan indices, it pays substantial space overhead, as shown in Figure 1. We aim to achieve better time-space tradeoff based on three observations:

- *Observation 1*: It is costly to keep a filter bit vector per interval. We propose filter sketches, which represent multiple intervals using sketch codes to reduce space.

- *Observation 2*: The position array takes a fixed amount of space. If space budget is tight, we propose selective position array, i.e. storing position arrays for a fraction of intervals. If the predicate falls into an interval without position array, the scan can use filter sketches in a similar fashion to Column Sketches.
- *Observation 3*: When the number of duplicate values is high, the interval boundaries should be judiciously chosen for time and space efficiency. Hence, we study data-aware intervals.

3 CABIN DESIGN

We propose Cabin, a compressed adaptive binned scan index. In the following, Section 3.1 overviews the design. Then, Section 3.2– 3.4 describe the three distinctive features of Cabin, i.e., filter sketches, selective position array, and data-aware intervals, respectively.

3.1 Cabin Overview

Figure 2(c) depicts Cabin. It is composed of three structures:

- **Filter sketches**: According to Observation 1, we can use w-bit sketch codes (e.g., w=3 in Figure 2(c)) to represent $m = 2^w - 2$ (e.g., 6) intervals. Then we divide all the intervals into groups of m intervals each, and build w filter sketches for every group of m intervals. The j-th interval in group i is denoted as $I_{i,j}$. $I_{i,0}$ and $I_{i,m-1}$ are two special virtual intervals, representing values less than and greater than the entire group I_i. In this way, $I_{i,0}$, $I_{i,1}$, ..., $I_{i,m-1}$ cover the full value range. (This is important for supporting selective position array.) Note that while Figure 2(c) draws only one group of intervals due to paper space constraint, there are usually multiple interval groups in a Cabin.
- **Selective position array**: According to Observation 2, we can remove the portion of the position array for a subset of intervals to save space. As shown in Figure 2(c), the shaded portion of the selective position array is not physically stored. However, an interval without physical position array may be visited by a scan. In such cases, we cannot perform binary search of the predicate value in the position array. Instead, Cabin relies on the filter sketches for evaluating the predicate. Since the filter sketches of an interval group cover the full range of values, they can be used in a similar fashion to Column Sketches.
- **(Data-aware) interval table**: As shown in Figure 2(c), the interval table keeps the metadata of each interval in each group, including the value range, the sketch code word, the number of records in the interval, its start location in the selective position array, and a flag used by selective position array and data-aware intervals to indicate the interval types. To build the interval table, it is natural to divide the value range into even-sized intervals. However, according to Observation 3, even-sized intervals may be sub-optimal when there are a large number of duplicate values. In such cases, Cabin judiciously makes data-aware selection of intervals to further optimize the scan.

In the following subsections, we describe three Cabin designs addressing the three observations progressively:

- $Cabin_F$: A design with filter sketches, even-sized intervals, and fully physical position array (cf. Section 3.2)
- $Cabin_{FS}$: $Cabin_F$ with selective position array (cf. Section 3.3)
- $Cabin_{FSD}$: $Cabin_{FS}$ with data-aware intervals (cf. Section 3.4)

3.2 Filter Sketches

We consider the evaluation of a scan with predicate "$x \leq 10$" using the index as shown in Figure 2(c). (1) Cabin locates the predicate value 10 in the interval table. It falls into $I_{0,2}$. Since $I_{0,2}$ contains 4 records and starts at $pos[4]$, Cabin performs a binary search between $pos[4]$ and $pos[7]$ of the position array. The split point is $pos[5]$, which refers to x_{14}=11. (2) There are two options to generate the draft result: a) compute a bit vector for all values < $I_{0,2}$ (i.e., $x < 10$, or union of $I_{0,0}$ and $I_{0,1}$) then flip the bits to the left of the split point in $I_{0,2}$ (i.e., $pos[4]$); or b) compute the bit vector representing values ≤ $I_{0,2}$ (i.e., $x < 16$, or union of $I_{0,0}$, $I_{0,1}$, and $I_{0,2}$) then flip the bits including and to the right of the split point in $I_{0,2}$ (i.e., $pos[5]$, $pos[6]$, and $pos[7]$). Cabin chooses a) since it minimizes the number of bit flips. (3) Cabin combines the set of filter sketches in group 0 to compute the draft result for $(-\infty, 10)$. (4) Cabin flips the bit for $pos[4]$ (i.e., bit 17) in the draft result to obtain the final result.

Compared to BinDex, Step (1), (2), and (4) are similar. The main novel feature of Cabin$_\text{F}$ resides in Step (3). In the following, we design filter sketches to optimize the draft result computation. Then, we describe the scan algorithm for Cabin$_\text{F}$.

Vertical Bit Layout for Filter Sketches. Let us first consider a horizontal layout for filter sketches. In our design, the optimal sketch code width w is often less than 8. As a result, sketch codes may span byte boundaries, incurring computation costs to reconstruct the codes. Therefore, we store filter sketches in a vertical bit layout, as shown in Figure 2(c). $fs_{0,b}$ (b = 0, ..., $w-1$) stores bit b of all the codes. This layout enables the use of SIMD logical operations to efficiently generate draft results.

Draft Bit Vector Computation Problem. We focus on one of the interval groups. Since the computation is the same regardless of group i, we omit the group ID i below for simplicity. Following the example, we first consider a predicate "$x \le c$".

From Step (2), we see that the draft result is the union of intervals $I_0 \dots I_j$ for some j. Let the code word of interval I_j be $code(j)$. Then the set of code words associated with $I_0 \dots I_j$ is $U^j = \{code(0), \dots, code(j)\}$. For every code C in the filter sketches, we set the corresponding bit in the draft result to 1 iff $C \in U^j$.

Then, two related questions arise: 1) What is the encoding $code(j)$? 2) How to compute $C \in U^j$ efficiently?

Our Solution: MLO. BitWeaving [26] proposed the VBP vertical bit layout. As shown in Figure 3(a), $code(j) = j$ in VBP. To compute $C \in U^j$, VBP checks each bit from the most to the least significant in a loop. Each loop iteration updates m_{lt} and m_{eq}, which represent the less than and the equal to cases. If we omit ¬ (since combined operations, such as andnot, may exist as a single SIMD instruction), VBP performs $5w + 1$ SIMD operations for w-bit codes.

We propose MLO (minimal logical operations), a novel encoding and computation scheme that minimizes the number operations. Unlike VBP, MLO avoids the per-bit loop. Instead, we derive a boolean logical formula to compute $C \in U^j$.

In MLO, we set $code(j) = 2^w - j - 1$. Let $C = \overline{C_{w-1}C_{w-2}\dots C_0}$ be a code word. Our target is the function $f_w(C, U^j)$ s.t. $f_w(C, U^j)$=1 iff $C \in U^j$. The subscript w denotes the number of bits in the code word. We derive the function's formula by recursion. Suppose $f_b(C, U^j)$ is 1 iff $C \in U^j$ when we focus on the lower b bits of the codes. We have the following:

$$f_b(C, U^j) = \begin{cases} C_{b-1} \land f_{b-1}(C, U^j), & \text{if } 0 \le j \le 2^{b-1}, \\ C_{b-1} \lor f_{b-1}(C, U^{j-2^{b-1}}), & \text{if } 2^{b-1} < j < 2^b - 1. \\ 1, & \text{if } j = 2^b - 1. \end{cases}$$

We can prove the following based on the recurrence relations.

Theorem 3.1. *$f_w(C, U^j)$ can be computed with up to $w - 1$ logical operations.*

As shown in Figure 3(b), MLO performs the same computation with only two SIMD logical operations. In general, compared with VBP, which takes $5w + 1$ SIMD operations, our MLO reduces the

(a) VBP

$Code(j) = j$

$Goal\colon Test\ C \in \{\overline{000}, \overline{001}, \overline{010}\}$

```
1. Prepare        2. Compute
$U\colon \overline{010}$       for $i = 2,1,0$
$m_{lt}\colon 0$          $m_{lt} = m_{lt} \vee (m_{eq} \wedge U_i \wedge \neg C_i)$
$m_{eq}\colon 1$          $m_{eq} = m_{eq} \wedge \neg(U_i \oplus C_i)$
                  $Result = m_{eq} \vee m_{lt}$
```

(b) MLO

$Code(j) = 2^3 - j - 1$ $(w{=}3)$

$Goal\colon Test\ C \in \{\overline{111}, \overline{110}, \overline{101}\}$

$Result = f_3 = C_2 \wedge (C_1 \vee C_0)$

Fig. 3. VBP vs. our proposed MLO computation.

number of SIMD operations by 5x.

Scan Algorithm for "≤" Operator. The scan algorithm of Cabin$_F$ for a predicate "$x \le c$" is listed in Algorithm 1:

1. *Prepare (Line 2–4).* Cabin performs a binary search for the predicate value c in the interval table. The target interval $I_{i,j}$ contains *num* records and starts at $pos[start]$. Then, Cabin searches for c between $pos[start]$ and $pos[start{+}num{-}1]$ in the selective position array. The split point is p_c. Note that data associated with $pos[0]$, ..., $pos[p_c - 1]$ satisfy the predicate "$x \le c$".
2. *Find Closest Matching Intervals (Line 5–8).* To compute the result bit vector, there are two options: a) compute the bit vector representing all intervals $< I_{i,j}$ (i.e. $Set_I = \{I_{i,0} \dots I_{i,j-1}\}$) then flip the bits of the records to the left of p_c; b) compute the bit vector representing $Set_I = \{I_{i,0} \dots I_{i,j}\}$ then flip the bits of the unwanted records including and to the right of p_c. Cabin chooses the option with fewer bit flips.
3. *Generate Draft Bit Vector (Line 9–10).* This step calls the MLO computation function for every VL (i.e, vector length, e.g., 256 in Intel AVX2) bit segment in filter sketches. Our implementation pre-generates $f_w(C, U^j)$ function using SIMD operations for all $0 \le j \le 2^w - 1$ and $2 \le w \le 9$. (This is sufficient since the optimal w is often small.) We store the function pointers in a table and invoke the relevant functions for $sketch\_to\_bv$. (Alternatively, SIMD code can be generated online if the database system supports query compilation [14, 19, 21, 28, 36].)
4. *Refine Bit Vector (Line 11–13).* Finally, Cabin performs the bit flips to account for the difference between the interval boundary and the split point. Step 2 has already recorded the incorrect records $pos[p_{start}]$, ..., $pos[p_{end} - 1]$. Cabin simply flips the corresponding bits in a loop. We use software prefetching to accelerate the random memory accesses caused by the bit flips.

Scan Algorithms for Other Comparison Operators. The scan algorithms for the other comparison operators are similar to Algorithm 1. We discuss them in the following:

- "$x < c$": "$x < c$" can be transformed to "$x \le c'$" where c' is the next value[2] less than c. As there are no other values between c and c', the two predicates have the same result.
- "$x \ge c$" and "$x > c$": To compute "$x \ge c$", the basic idea is to compute "$x < c$" and negate the result bit vector. After calling $sketch\_to\_bv$, we perform one additional SIMD operation $bv^s_{draft} = \neg bv^s_{draft}$. In this way, we obtain the negation of the result bit vector after the bit flips. Similarly, "$x > c$" can be computed by negating the result of "$x \le c$". In these cases, MLO performs up to w SIMD operations for every VL code words.
- "$c_1 \le x \le c_2$" : In Cabin, this case is transformed into "$x \ge c_1$" and "$x \le c_2$". Then, Cabin computes the bitwise AND of the result bit vectors of "$x \ge c_1$" and "$x \le c_2$".
- "$x = c$" and "$x \ne c$": "$x = c$" is transformed into "$c \le x \le c$". "$x \ne c$" is supported by negating the result of "$x = c$".

[2]For integer values, we can simply set $c'{=}c{-}1$. For floating-point values, we can get the next representable value c' using the C++ `std::nexttoward` function.

Algorithm 1: Cabin_F for a predicate "$x \le c$".

```
Input  : Input column $X_{1...N}$, predicate "$x \le c$", filter sketches $fs_{i,b}$, sketch code width $w$,
         interval table $TI$, selective position array $pos[]$
Output : Result bit vector $bv$
1  Function Cabin_F Scan("$x \le c$", $X$)
2      ($I_{i,j}$, num, start, sflag) = binary_search_interval($c$, $TI$);
3      /* assume selective flag of $I_{i,j}$ is 1, i.e., $sflag == 1$ */
4      $p_c$ = binary_search_pos($c$, $pos$, $start$, $num$);
5      if $p_c - start \le \frac{num}{2}$ then
6          $Set_I = \{I_{i,0} \ldots I_{i,j-1}\}$; $p_{start} = start$; $p_{end} = p_c$;
7      else
8          $Set_I = \{I_{i,0} \ldots I_{i,j}\}$; $p_{start} = p_c$; $p_{end} = start + num$;
9      for each $VL$-bit segment $s$ in filter sketches do
10         $bv^s_{draft} = sketch\_to\_bv(Set_I, \{fs^s_{i,0} \ldots fs^s_{i,w-1}\})$;
11     for $p_{start} \le r < p_{end}$ do
12         pipeline prefetch;
13         flip_bit($bv[pos[r]]$);
14     return $bv$;
```

Shortcut Optimization. BinDex takes a different approach for evaluating "$x = c$". The idea is to start with an all-zero draft vector, then search the position array to find the small number of records that satisfy the predicate, and finally set the corresponding bits in the draft vector. We call this approach *shortcut optimization.* The shortcut optimization reduces the cost of computing the draft bit vector from filter sketches. We find this beneficial for comparison operators (e.g., BETWEEN) other than "=" when the selectivity is low. Therefore, we extend the use of shortcut optimization for all comparison operators. After searching the selective position array, Cabin knows the selectivity of the scan. If it is lower than a given threshold (e.g., 0.5%), Cabin performs the shortcut optimization.

3.3 Selective Position Array

Let us consider the space cost of the position array. For 4-byte rowIDs, it takes $4N$ bytes, where N is the number of records. Since the rowID size can be similar to or even larger than the data value size, this incurs significant space overhead. For example, the rowIDs of 1 billion records take nearly 4GB memory space.

To save memory space, we propose to selectively store portions of the position array. That is, for a subset of intervals, we do not store their rowIDs, and set their selective flags to 0 in the interval table. This allows more flexible time-space tradeoff.

Scan Algorithm Without Selective Position Array. The scan algorithm of Cabin_FS is listed in Algorithm 2. Compared to Cabin_F, the main difference is that Cabin_FS cannot search the target interval using the selective position array in order to refine the draft bit vector. Instead, we use the filter sketches in a similar fashion to Column Sketches to refine the draft result:

1. *Prepare (Line 2–3).* Cabin finds the target interval $I_{i,j}$ in the interval table. It checks the selective flag. If the flag is 0, meaning that the rowIDs of $I_{i,j}$ are not stored, then Cabin executes Cabin_FS. (If the flag is 1, then Cabin_F is used).

Algorithm 2: Cabin$_{\text{FS}}$ for a predicate "$x \le$ c".

```
Input  : Input column $X_{1...N}$, predicate "$x \le c$", filter sketches $fs_{i,b}$, sketch code width $w$,
         interval table $TI$, selective position array $pos[\,]$
Output : Result bit vector $bv$
1  Function Cabin$_{\text{FS}}$Scan("$x \le c$", $X$)
2      ($I_{i,j}$, num, start, sflag) = binary_search_interval($c$, $TI$);
3      /* assume selective flag of $I_{i,j}$ is 0, i.e., $sflag == 0$ */
4      $Set_I = \{I_{i,0} \dots I_{i,j-1}\}$;
5      for each $VL$-bit segment $s$ in filter sketches do
6          $bv^s_{draft} = sketch\_to\_bv(Set_I, \{fs^s_{i,0}, \dots, fs^s_{i,w-1}\})$;
7          $bv^s_{eq} = sketch\_to\_bv(I_{i,j}, \{fs^s_{i,0}, \dots, fs^s_{i,w-1}\})$;
8          $eqbits[\,] = bv\_to\_pos(bv^s_{eq})$ ;
9          for each $r$ in $eqbits[\,]$ do
10             $bv^s = bv^s_{draft} | ((X_{pos[s*VL+r]} \le c) \ll r)$;
11     return $bv$;
```

2. *Find Matching Interval (Line 4).* Since the portion of position array for $I_{i,j}$ is not stored, we cannot search $I_{i,j}$. Cabin$_{\text{FS}}$ simply sets Set_I to all intervals < $I_{i,j}$.
3. *Generate Draft Bit Vector (Line 5–6).* This step is the same as Cabin$_{\text{F}}$. $sketch\_to\_bv$ is invoked to obtain the draft bit vector for every VL code words.
4. *Refine Bit Vector (Line 7–10).* For a value $\in I_{i,j}$, filter sketches alone cannot tell the predicate outcome. In such cases, Cabin$_{\text{FS}}$ visits the base data to evaluate the predicate. Accordingly, the algorithm obtains all the positions whose sketch code is equal to the code of $I_{i,j}$ (Line 7–8), then checks the corresponding base value against the predicate (Line 9–10).

Sketch Code Equality Comparison. We develop a boolean logical function $eq_w(C, C^T)$, where C^T is the code of the target interval $I_{i,j}$. Let $C = \overline{C_{w-1}C_{w-2}\dots C_0}$ be a code word. $eq_w(C, C^T)$ =1 iff $C = C^T$. We also derive the formula by recursion:

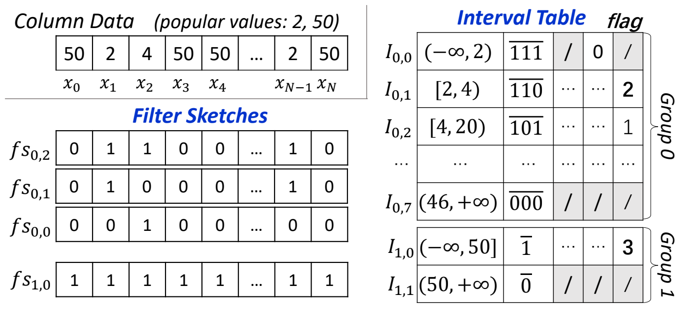
$$eq_b(C, C^T) = \begin{cases} \neg C_{b-1} \wedge eq_{b-1}(C, C^T), & C^T_{b-1} = 0, \\ C_{b-1} \wedge eq_{b-1}(C, C^T), & C^T_{b-1} = 1. \end{cases}$$

For example, $eq_3(C, \overline{010}) = \neg C_2 \wedge C_1 \wedge \neg C_0$. Note that Intel processors support andnot as a single SIMD instruction. Therefore, $eq_w(C, C^T)$ takes up to w SIMD operations to compute for each SIMD segment.

3.4 Data-aware Intervals

In Cabin$_{\text{F}}$ and Cabin$_{\text{FS}}$, we divide the value range into equal-sized intervals. This strategy works well when the number of duplicates for each distinct value is small. However, real-world data may contain a lot of duplicates. Two representative scenarios are as follows:

- *Skewed data distribution*: Real-world data distribution is often skewed. For example, power-law distribution is common in real-world graphs, such as social networks [5]. In this scenario, a few values may have a large number of duplicates, while the number of duplicates of most values is small.
- *Small number of distinct values (NDV)*: The number of distinct values can be small in a data column. Examples are categorical attributes. In this scenario, the number of duplicates can be quite large for each distinct value.


Column Data (popular values: 2, 50)
50 2 4 50 50 ... 2 50
x_0 x_1 x_2 x_3 x_4 x_{N-1} x_N
Filter Sketches
$fs_{0,2}$ 0 1 1 0 0 ... 1 0
$fs_{0,1}$ 0 1 0 0 0 ... 1 0
$fs_{0,0}$ 0 0 1 0 0 ... 0 0
$fs_{1,0}$ 1 1 1 1 1 ... 1 1
Interval Table
flag
$I_{0,0}$ $(-\infty, 2)$ $\overline{111}$ / 0 /
$I_{0,1}$ $[2, 4)$ $\overline{110}$ 2
$I_{0,2}$ $[4, 20)$ $\overline{101}$ 1
... ...
$I_{0,7}$ $(46, +\infty)$ $\overline{000}$ / / /
$I_{1,0}$ $(-\infty, 50]$ $\overline{1}$ 3
$I_{1,1}$ $(50, +\infty)$ $\overline{0}$ / / /
Group 0
Group 1

Interval Type	flag = 0	interval without position array	flag = 2	skew interval
	flag = 1	interval with position array	flag = 3	skew interval group

Fig. 4. Cabin supporting data with duplicates.

We call values with a large number of duplicates *popular values*. Cabin$_{\text{FSD}}$ judiciously chooses intervals, interval groups, and/or sketch code widths to better support popular values.

In the following, we describe the steps of index building with an emphasis on data-aware intervals:

Index Building Step 1: Popular Value Discovery. Given a base data column, we consider three alternative methods for discovering popular values. First, the sort-based method allocates a temporary sort buffer and sorts the (value, rowID) pairs in value order. It scans the sorted array and computes the number of duplicates for each value. Second, the hash-based method allocates a temporary hash table and counts the number of duplicates per value using the hash table. Third, the sample-based method collects a small (e.g., 1%) sample from the base data. Then, it applies the sort-based method on the sample. Since popular values appear in the sample with high probability, we can use the sample based estimates to discover popular values. We record a value as a candidate popular value if its number of occurrences $\geq N_{pl}$. The bound N_{pl} is computed as the record count N divided by the *maximal* number of intervals given the memory space budget of Cabin. We also collect/estimate other meta-information about the base data, such as the value range and the number of distinct values.

Index Building Step 2: Optimal Design Selection. Given memory space budget and the data characteristics, we compute the optimal design parameters (cf. Section 4). Then, we compute N_p, the average interval size, as N divided by the number of intervals in the optimal parameters. Since $N_p > N_{pl}$ in most cases, we remove unqualified candidates to obtain the popular values. For each popular value, we put the value into its own interval. We call such interval *skew interval*. If the number of duplicates is larger than the average interval group size $N_p(2^w - 2)$, then we put the value into its own interval group. We call such interval group *skew interval group*. The code width of all skew interval groups is set to 1. As for unpopular normal values, we create even-sized intervals.

Index Building Step 3: Structure Building. Given the selected design parameters in Step 2, we build the interval table, the selective position array, and the filter sketches. For skew intervals and skew interval groups, we do not store the associated portions of the position array. As a skew interval contains only a single value, scans do not need to search the position array in the interval.

Optimizing for Popular Values. Figure 4 illustrates data-aware intervals for popular values. There are two popular values in the base data: 50 and 2. First, 50 occupies its own skew interval group (i.e., Group 1). A skew interval group contains only two virtual intervals $(-\infty, v_p]$ and $(v_p, +\infty)$, where v_p is the popular value. We set flag=3 to indicate a skew interval group. Second, 2 has its own skew interval in Group 0 (i.e., $I_{0,1}$). We set flag=2 for skew intervals.

Table 2. Terms used in cost analysis.

w	sketch code width	$I_{i,j}$	j-th interval in Group i
g	number of groups	$p_{i,j}$	probability of accessing $I_{i,j}$
sp	percentage of stored pos[]	Set_x	Set of intervals w/ property x
N	number of records	t_{seqr}	time to sequentially read a byte
M	number of intervals	t_{seqw}	time to sequentially write a byte
$\|I_{i,j}\|$	number of records in $I_{i,j}$	t_{ranr}	time for a random read
r	rowID bits	t_{ranw}	time for a random write
VL	SIMD vector length	t_{simd}	time of a SIMD logical operation

For a scan with predicate $x \leq c$, Cabin$_{\text{FSD}}$ finds the target interval in the interval table. There are two cases for popular values. First, the target is a skew interval group with flag=3 (e.g., c=50). Then the scan directly returns the single filter sketch (e.g., $fs_{1,0}$). Second, the target interval is a skew interval in a normal group (e.g., c=2). Then the scan performs a simplified Cabin$_{\text{F}}$Scan. Specifically, since the skew interval contains a single value, there is no need to refine the draft result. The draft result bit vector can be returned directly. In both cases, Cabin$_{\text{FSD}}$ simplifies the scan procedure, thereby improving scan performance for popular values.

4 OPTIMAL DESIGN SELECTION

We model the time and space cost of Cabin, then describe how to compute the optimal design parameters for Cabin.

Design Parameters of Cabin. There are three important design parameters in Cabin. (Table 2 lists the terms used in the analysis.)

- *Sketch code word width (w).* As w increases, the number of bit vectors in filter sketches increases. The cost of draft bit vector generation becomes proportionally expensive. On the other hand, the number of intervals per group ($2^w - 2$) increases exponentially. Interval size becomes smaller, leading to fewer bit flips in the refine step.
- *Number of groups (g).* Every contiguous $2^w - 2$ intervals form a group. The total number of non-virtual intervals is $M = g(2^w - 2)$. As g rises, M increases. The interval size decreases, and therefore the refine step improves. However, the space for storing all filter sketches grows as g increases. The optimal g is often larger than 1 (cf. Section 5.4).
- *Stored proportion of selective position array (sp).* $sp \in [0, 1]$ is the fraction of rowIDs physically stored in the selective position array. For even-sized intervals, $sp \cdot M$ computes the number of intervals, whose flag=1 in the interval table. In this work, we assume that every data record is equally likely to be the query predicate value. Hence, the probability that the predicate value falls into each equal-sized interval is the same. Therefore, we randomly pick intervals that do not store the position array.

Scan Performance. Let $p_{i,j}$ be the probability that the scan predicate value falls into interval $I_{i,j}$. Let $|I_{i,j}|$ be the number of records in $I_{i,j}$. If we assume that every base data record is equally likely to be queried, then $p_{i,j} = \frac{|I_{i,j}|}{N}$. Let Set_0 be the set of all intervals whose flag=0 (i.e., the rowIDs are not stored). Let Set_1 be the set of all intervals whose flag=1 (i.e. with stored position array). Let Set_{skew} be the set of all skew intervals. Then the average scan time of Cabin can be modeled as follows:

$$T_{Cabin} = \sum_{Set_1} \frac{|I_{i,j}|}{N} T_{CabinF}(I_{i,j}) + \sum_{Set_0} \frac{|I_{i,j}|}{N} T_{CabinFS}(I_{i,j}) + \sum_{Set_{skew}} \frac{|I_{i,j}|}{N} T_{CabinFSD}(I_{i,j})$$

T_{CabinF}, the scan time of Cabin$_{\text{F}}$, consists of four components:

1. *Prepare* ($T_{searchF}$). Cabin$_F$ searches the interval table containing M intervals, and then the target interval with an average $\frac{N}{M}$ records. Hence, this step takes $O(\log N)$ time.
2. *Find Closest Matching Intervals* (T_{fcmi}). Step 2 involves $O(1)$ comparison and assignment operations.
3. *Generate Draft Bit Vector* (T_{genbv}). This step reads a group of w N-bit filter sketches, performs an average w SIMD logical operations for every VL code words, and writes a N-bit result vector. Hence, $T_{genbv} = \max\{\frac{wN}{8}t_{seqr}, \frac{wN}{VL}t_{simd}, \frac{N}{8}t_{seqw}\}$ where t_{seqr}, t_{simd}, and t_{seqw} are the time to sequentially read a byte, perform a SIMD logical operation, and sequentially write a byte, respectively. The max takes into consideration the overlapping of computation and memory accesses [15, 33].
4. *Refine Bit Vector ($T_{refineF}$).* As Step 2 finds the closer interval end to the split point, the number of bit flips is at most half of the interval. On average, $\frac{1}{4}$ of the positions need to be corrected. For each position, Step 4 reads the rowIDs in the position array then flips the bit in the draft result. Hence, $T_{refineF} = \frac{|I_{i,j}|}{4}(\frac{r}{8}t_{seqr} + t_{ranw})$, where $|I_{i,j}|$ is the size of interval $I_{i,j}$, r is the number of bits of rowID, t_{ranw} is the time for a random write.

It follows from the above discussion:

$$\begin{aligned}T_{CabinF}(I_{i,j}) &= T_{searchF} + T_{fcmi} + T_{genbv} + T_{refineF}\\ &= O(\log N) + \max\{\tfrac{wN}{8}t_{seqr}, \tfrac{wN}{VL}t_{simd}, \tfrac{N}{8}t_{seqw}\} + |I_{i,j}|(\tfrac{r}{32}t_{seqr} + \tfrac{1}{4}t_{ranw})\end{aligned}$$

$T_{CabinFS}$, the scan time of Cabin$_{FS}$, also consists of four components. The costs of the first three steps are similar to those of Cabin$_S$ except that it does not search the interval. The main difference is the cost of the refine step. Since the portion of the selective position array is not stored for the interval, Cabin$_{FS}$ computes the equal-code positions with SIMD logical operations, and reads all $|I_{i,j}|$ base data values in interval $I_{i,j}$ to evaluate the predicate. On average, half the values satisfy the predicate, and the scan flips $\frac{|I_{i,j}|}{2}$ bits in the draft result. Hence, $T_{refineFS} = \frac{wN}{VL}t_{simd} + |I_{i,j}|(t_{ranr} + \frac{1}{2}t_{ranw})$, where t_{ranr} is the time for a random read. Hence, we have:

$$\begin{aligned}T_{CabinFS}(I_{i,j}) &= T_{searchFS} + T_{fcmi} + T_{genbv} + T_{refineFS}\\ &= O(\log M) + \max\{\tfrac{wN}{8}t_{seqr}, \tfrac{wN}{VL}t_{simd}, \tfrac{N}{8}t_{seqw}\} + \tfrac{wN}{VL}t_{simd} + |I_{i,j}|(t_{ranr} + \tfrac{1}{2}t_{ranw})\end{aligned}$$

Finally, we compute $T_{CabinFSD}$ for popular values. The scan searches the interval table in $O(\log M)$ time. Then it computes the result. There are two cases. For a skew group, the scan returns the filter sketches with negligible cost. For a skew interval, the scan generates and returns the draft bit vector in T_{genbv} time.

$$\begin{aligned}T_{CabinFSD}(I_{i,j}) &= T_{searchFSD} + T_{fcmi} + T_{genbv}\\ &= \begin{cases} O(\log M) + \max\{\frac{wN}{8}t_{seqr}, \frac{wN}{VL}t_{simd}, \frac{N}{8}t_{seqw}\}, & flag = 2,\\ O(\log M) + \max\{\frac{N}{8}t_{seqr}, \frac{N}{VL}t_{simd}, \frac{N}{8}t_{seqw}\}, & flag = 3.\end{cases}\end{aligned}$$

Space Overhead. We first consider all equal-sized intervals:

- *Interval table* (S_{TI}). It stores meta information of intervals. Its space cost is negligible.
- *Filter sketches* (S_{fs}). There are g interval groups. Each group has w N-bit vectors. Hence, $S_{fs} = \frac{gwN}{8}$ bytes.
- *Selective position array* (S_{pos}). The total size is $\frac{Nr}{8}$ bytes. Since sp portion is physically stored, $S_{pos} = sp\frac{Nr}{8}$ bytes.

Thus, we have the following for equal-sized intervals:

$$S_{Cabin} = S_{TI} + S_{fs} + S_{pos} = \frac{gwN}{8} + sp\frac{Nr}{8}$$

Algorithm 3: Optimal design selection algorithm.

```
Input   : Column data $X_{1...N}$, space budget $B$
Output  : Sketch code width $w$, number of groups $g$, stored proportion of selective position
          array $sp$
1  Function DesignSelection($X$, $B$)
2      $N = |X|$; $CP$ = CandidatePopularValues ($X$, $B$);
3      $T_{min} = +\infty$;
4      for $2 \leq w \leq 9$ do
5          $g_{max} = \frac{8B}{w \cdot N}$ /* $sp = 0$ */; $g_{min} = \frac{8B - N \cdot r}{w \cdot N}$ /* $sp = 1$ */;
6          for $g_{min} \leq g \leq g_{max}$ do
7              $M = g \cdot (2^w - 2)$;
8              $Set_{skew}$ = DataAwareInterval($X$, $CP$, $M$);
9              $Set_{normal}$ = EqualInterval($X$, $Set_{skew}$, $M$);
10             $sp$ = ComputeSP(N, $Set_{skew}$, $Set_{normal}$, $g$, $w$);
11             $(Set_0, Set_1)$ = GetSelPosArray($Set_{normal}$, $sp$);
12             Compute $T_{Cabin}$ given $Set_0$, $Set_1$, and $Set_{skew}$;
13             if $T_{Cabin} < T_{min}$ then
14                 $T_{min} = T_{Cabin}$; update ($w$, $g$, $sp$);
15     return ($w$, $g$, $sp$);
```

For popular values, Cabin does not store the rowIDs. Therefore, we subtract the number of records with popular values from N in the computation of S_{pos}. As for filter sketches, there are two cases. For a skew interval group, Cabin stores only a single N-bit vector. For a skew interval in a normal group, the size of filter sketches is the same as that of normal intervals.

Optimal Design Selection. Given a memory space budget, Algorithm 3 finds the design parameters that optimize the scan time of Cabin. It begins by getting the candidate popular values (Line 2, cf. Section 3.4). Then, it enumerates every pair of (w, g). We restrict w to be up to 9, which is sufficient in our experiments with up to a billion records. The range of g is bounded by the cases where the selective position array is entirely stored or removed. Therefore, the procedure is efficient because the search space is limited.

For each pair of (w, g), the algorithm computes the set of skew intervals from popular values (Line 8, cf. Section 3.4). Then, for unpopular values, it computes equal-sized intervals (Line 9). After that, the algorithm examines the space cost of filter sketches and selective position array by taking into account of both skew intervals and normal intervals. It computes sp (Line 10) and chooses a subset of intervals to remove the associated portion of the selective position array (Line 11). This completes the tentative design for all intervals. The scan time is estimated using the above formulas[3] (Line 12). Finally, the algorithm returns the design parameters that obtain the minimal scan time.

5 EVALUATION

We perform extensive experiments to compare Cabin with existing optimized scan solutions in this section.

[3]The asymptotic terms, i.e., $O(\log M)$ and $O(\log N)$, in the formulas refer to the time of searching the interval table and/or the position array. Since the search time is tiny compared to the $(O(N))$ cost of draft bit vector generation and refinement (cf. Figure 11(a) and Figure 12), we omit these terms in the computation.

5.1 Experimental Setup

Machine Configuration. The experimental machine is equipped with a 3.0GHz Intel Core i7-9700 processor (8 cores, 12MB L3 cache), 32GB DDR4 DRAM, and 16TB disks. The processor supports 256-bit SIMD instructions. The operating system is 64-bit Ubuntu Server 20.04 with Linux kernel 5.13.0-30. All testing programs are compiled using g++ 9.4 (i.e., the default version in Ubuntu 20.04) with optimization flag -O3 and SIMD flag -mavx, -mavx2, -mbmi1, -mbmi2. All experiments are performed in main memory.

Solutions to Compare. We compare Cabin against five optimized scan solutions: ByteSlice (*BS*) [13], Zone Maps (*ZM*) [27], Column Sketches (*CS*) [16], B+-tree [11], and BinDex [25]. We obtain the code of ByteSlice and BinDex from their github repositories. We select a widely used in-memory B+-tree implementation, STX B+-Tree [3]. For Zone Maps and Column Sketches, we follow the papers to faithfully implement the techniques. We use single-byte column sketches as in the original experimental setting. Moreover, we try our best to optimize all scan solutions with software prefetching and SIMD techniques.

Both BinDex and Cabin can leverage more space to construct more filter bit vectors or filter sketches for achieving better scan performance. We set a reasonable space limit by considering the space cost of the B+-Tree. In our experimental setting, a rowID is 32-bit large and each column data value takes d bits, where d=8, 16, 32, or 64. Hence, the data column takes dN bits, where N is the number of records. Since the B+-Tree's leaf level contains N pair of (value, rowID), the B+-Tree size is at least $(d+32)N$ bits. (Note that the B+-Tree consumes more space for storing non-leaf nodes as well as other fields, such as sibling pointers, in leaf nodes.) Consequently, we set the default space upper limit of BinDex and Cabin to be $\frac{d+32}{d}\times$ column size (i.e., 5x for d=8, 3x for d=16, 2x for d=32, and 1.5x for d=64), which is smaller than the B+-Tree.

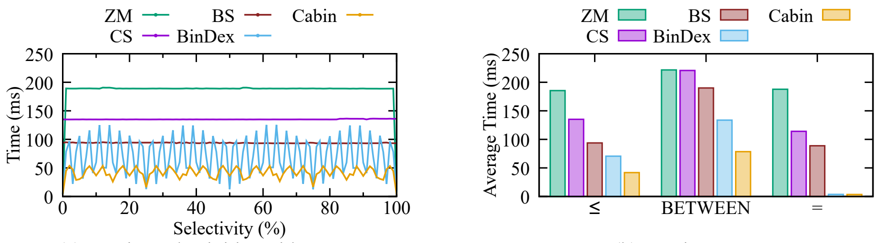
Note that this space limit is a lower bound of the space used in tree-structured indices in general, including not only B+-Trees, but also more recent proposals, such as ART [24] and learned index [20]. This is because values are not sorted and a tree-structured index has to store (value, rowID)s for the data column.

Synthetic Workload. We generate each synthetic data column with one billion random values. We consider four cases: 1) integer data following uniform distribution over the full value range; 2) integer data following Zipf distribution to model skew data; 3) integer data with NDV distinct values uniformly distributed in $[0, NDV)$ to model domain encoded categorical attributes; and 4) skewed floating point data following skewed normal distribution (location=0, scale=1, shape=4) or pareto distribution (scale=1, shape=4) generated with the Boost library. Note that there are few duplicate values in case 1 and 4, but a large number of duplicates in case 2 and 3.

We perform scan operations with different types of predicates on the synthetically generated data columns. We run all experiments in a single-threaded test program. The output of all scan solutions is a result bit vector except the B+-tree, which outputs a rowID list.

Real-World Data Sets. We also conduct experiments on two widely used real-world data sets, DBLP [1] and IMDb [2]. The DBLP data set contains 5.26M records. We use the `n_citation` column in the scan experiment. The IMDb data set contains 10.2M records. We use the `startYear` column in the scan experiment. Both data sets contain a lot of duplicates.

TPC-H and SSB Workloads. Besides the above stand-alone tests, we evaluate Cabin in a full fledged main memory database system, MonetDB. To add Cabin to MonetDB, our implementation follows the approach of BinDex. Specifically, the Cabin-based select operator supports the same API as the original select operator, which takes a single column and a predicate as input. A new fetch operator supports the result bit vector instead of the rowID list in the original fetch operator. To run



(a) Varying selectivities with ≤ operator. (b) Varying operators.

Fig. 5. Scan performance (uniform, 32-bit data).

a SQL query, we first invoke MonetDB to generate the corresponding MAL script, which consists of operator calls to execute the selected query plan for the query. Then, we modify the MAL script to call either BinDex or Cabin enhanced scan operations. We run the original, BinDex-enhanced, and Cabin-enhanced MAL scripts to compare the query performance of MonetDB, MonetDB+BinDex, and MonetDB+Cabin, respectively. In the MonetDB experiments, we still bound the memory usage of BinDex and Cabin by $\frac{d+32}{d}\times$ column size for each indexed column, where d is the bit width of data values.

We run TPC-H and SSB (Star Schema Benchmark) workloads on MonetDB. For TPC-H, we generate data with scale factor=20. We follow previous study [16, 25] to run TPC-H Q1 and Q6, which scan the Lineitem table with various filters to compute aggregates. We also run Q12, which performs a single join between Lineitem and Orders tables in addition to scans and aggregates. For SSB, we generate data with scale factor=60. We run all the SSB queries.

5.2 Scan Performance on Uniform Data

In this subsection, we perform scan experiments on uniform data with few duplicates. We vary the predicate comparison operator, the data width, and the predicate selectivity in the experiments.

Scan with ≤ Operator. Figure 5(a) compares the scan solutions with ≤ operator on 32-bit integer data. The x-axis varies the selectivity from 0 to 100% with 1% increments. The y-axis reports scan time in ms. Each reported point is the average over 10 runs.

First, we see that Zone Maps, Column Sketches, and ByteSlice show three flat lines as the selectivity increases. Zone Maps fall back to plain scans in most cases, reading $32N$ bits of data and displaying a flat (green) curve. Only at very low or very high selectivities can Zone Maps effectively skip entire buckets. Column Sketches out-performs Zone Maps by reducing the amount of data to read. Regardless of selectivities, it reads $8N$ bits in the sketched column to compare the predicate code with the 1-byte sketch codes, and performs $\frac{N}{256}$ random accesses where sketch code = predicate code. ByteSlice reads the byte-level columnar data with an efficient SIMD algorithm. It accesses similar amount of data for all selectivities.

Second, both BinDex and Cabin see wave-like curves. As the selectivity increases, the predicate value moves across the intervals in the scan indices. The number of bits to refine depends on how far the predicate value is from the closest interval boundary, and thus varies periodically as the predicate value moves across multiple interval boundaries. This causes the wave shapes.

Third, the space of BinDex and Cabin are bounded by twice the column data size (i.e., $32N$) as discussed in Section 5.1. Hence, the space limit is $64N$. For BinDex, the position array takes $32N$ for storing N 32-bit rowIDs. Therefore, BinDex uses the remaining space to store 32 N-bit filter vectors. There are 33 value intervals. In comparison, the optimal design of Cabin has $g = 6$ interval groups, each containing $2^w - 2 = 30$ intervals where $w = 5$. That is, Cabin supports 180 intervals in total, over 5x as many as that supported by BinDex. As a result, Cabin reduces the interval size and the average number of bits to refine (i.e., $\frac{1}{4}\times$ interval size) by over 5x, attaining significant better

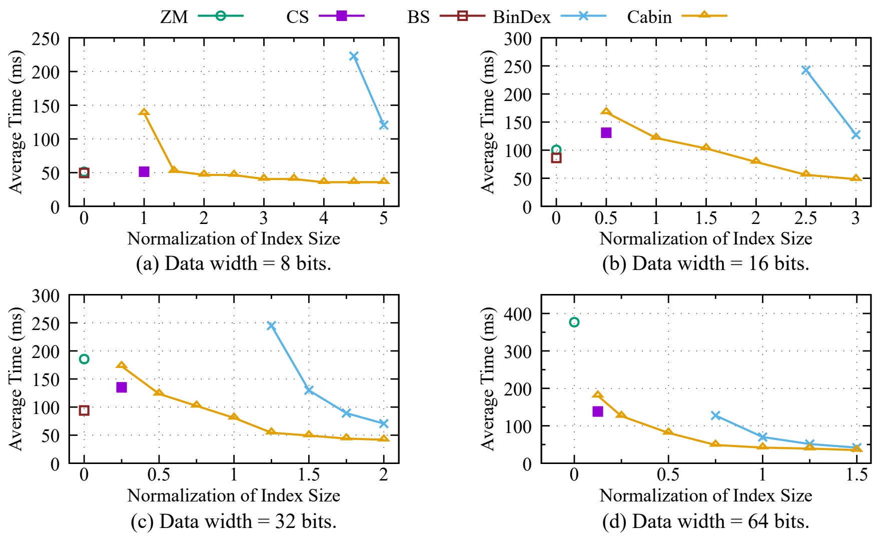


Fig. 6. Space-time analysis for scans.

performance than BinDex.

Overall, Cabin achieves up to 11.7x, 8.38x, 5.86x, and 3.51x better performance than Zone Maps, Column Sketches, ByteSlice, and BinDex for 1%–99% selectivities, respectively. (The improvement over Column Sketches and ByteSlice at 0% and 100% is over 20x because of the shortcut optimization.) We compute the average scan time over all the measured selectivities. Compared to Zone Maps, Column Sketches, ByteSlice, and BinDex, Cabin improves the average scan time by 4.48x, 3.27x, 2.27x, and 1.70x, respectively.

Scan with Different Comparison Operators. Figure 5(b) shows the scan performance of different filter operators. Note that ≤ performs a single in-equality comparison, representing <, >, and ≥ operators. For BETWEEN, Cabin decomposes $c_1 \leq x \leq c_2$ into $x \geq c_1$ and $x \leq c_2$. Hence, BETWEEN takes nearly twice as much time as ≤ in Cabin. Compared with Zone Maps, Column Sketches, ByteSlice, and BinDex, Cabin improves the average performance of scans with BETWEEN by 2.82x, 2.81x, 2.42x, and 1.70x, respectively.

For =, Cabin and BinDex use the shortcut optimization that directly sets the matching bits in the result bit vector[4]. As a result, Cabin and BinDex take a few ms to evaluate scans with =, which are orders of magnitude faster than the other solutions.

Index Space-Time Analysis Varying Data Width. Figure 6 analyzes the time-space trade-off of the scan solutions for queries with ≤ operator while varying data width. The x-axis is the index size normalized to that of the column data. The space bound is set as discussed in Section 5.1. The y-axis reports the scan time averaged across all the measured selectivities. In the figures, Zone Maps, Column Sketches, and ByteSlice are single points. Cabin and BinDex can exploit larger space to achieve better scan performance, showing two curves. Note that the ByteSlice implementation does not support 64-bit data and thus is not tested on 64-bit data.

From the figures, we see that Cabin achieves the best performance (at 2x–5x sizes for 8 bits, at 2x–3x sizes for 16 bits, at 1x–2x sizes for 32 bits, at 0.25x–1.5x column size for 64 bits). Compared to BinDex, Cabin's higher performance mainly comes from filter sketches, which support more intervals given the same space budget. Moreover, Cabin enjoys a much wider range of space

[4]BinDex implements the shortcut optimization only for =. Cabin extends it to optimize all comparison operators, including BETWEEN, when the selectivity is low.

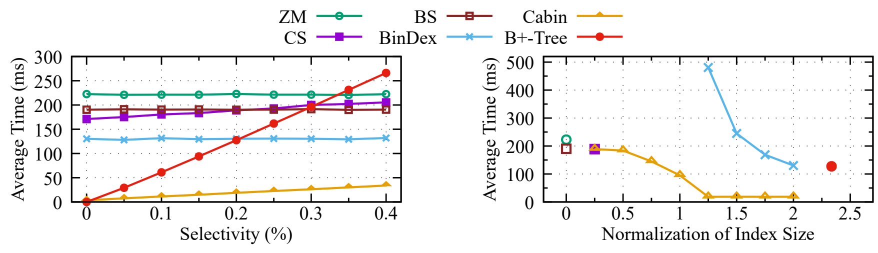


Fig. 7. Comparison with B+-Tree at low selectivities.

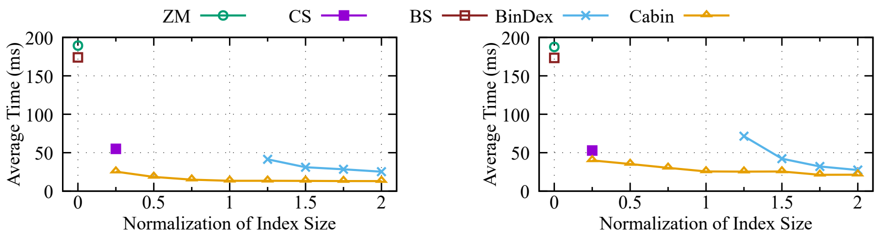


Fig. 8. Scan performance on data with duplicates.

configurations than BinDex. BinDex is at least as large as its position array, which takes 4x–0.5x as much space as the data column for 8-bit – 64-bit values. In contrast, the selective position array can effectively reduce Cabin's size with graceful performance degradation.

Comparison with B+-Tree When Selectivities are Low. We find that the B+-Tree performs poorly at medium to high selectivities. To show good B+-Tree performance, we run scans with BETWEEN and focus on low selectivities from 0% to 0.4%. Given a specific selectivity, we randomly pick 1000 value pairs as the BETWEEN predicate values that satisfy the target selectivity, and report the average time of the 1000 scans in Figure 7(a).

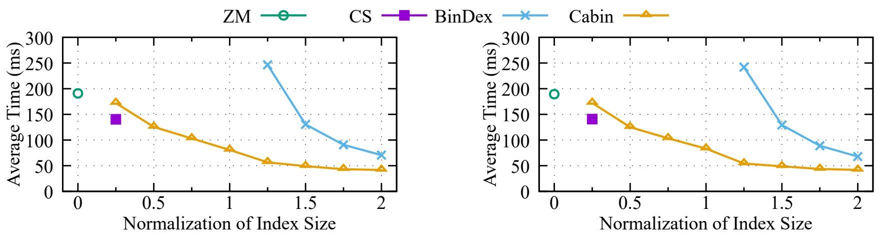
From Figure 7(a), we see that the B+-Tree is the second best solution when the selectivity is lower than 0.2%. However, as the selectivity increases, the performance of the B+-Tree quickly degrades because the B+-Tree leaf scan incurs random memory accesses and costly pointer chasing. At 0.4% selectivity, the B+-Tree becomes the slowest solution. In comparison, Cabin achieves the best performance by using the shortcut optimization for low selectivities[4].

Figure 7(b) shows the space-time scatter plot for low selectivities from 0% to 0.4%. We see that Cabin achieves the best time-space trade-off among all solutions. Note that compared to the B+-Tree, Cabin is both smaller and faster. When the index size is reduced, Cabin employs the selective position array. If the BETWEEN range falls in intervals where position arrays are not stored, the shortcut optimization cannot be applied. Hence, the scan performance degrades gracefully, leading to the arc shape for smaller index sizes.

5.3 Data with Different Characteristics

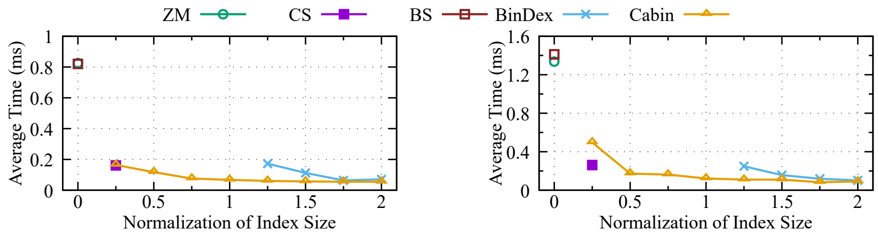
Scan Performance on Data with Duplicates. We study two representative scenarios where there can be a lot of duplicates in the column data: 1) Skew data with Zipf distribution, which models the power-law distribution in real-world data sets; and 2) Data with a small number of distinct values, which models categorical attributes that are domain encoded.

Figure 8(a) shows the space-time scatter plot for skew data with the Zipf parameter = 2. Figure 8(b) shows the space-time scatter plot when there are 100 distinct values. In both cases, the data column contains popular values that span entire value intervals. Our technique of data-aware intervals



(a) skewed normal distribution. (b) pareto distribution.

Fig. 9. Scan performance on skewed floating point values.



(a) DBLP. (b) IMDb.

Fig. 10. Scan performance on real-world datasets.

both reduces the space cost and improves the scan performance for supporting the popular values. As a result, Cabin achieves the best time-space trade-off in both cases, as shown in Figure 8.

Scan Performance on Skewed Floating Point Data. Figure 9(a) and (b) show the time-space trade-off of scan solutions for 32-bit floating point data that follow skewed normal distribution and pareto distribution, respectively. (Please note that ByteSlice does not support floating point data.) In both data sets, value ranges contain skewed amount of data but the number of duplicates is low. Therefore, equal-sized intervals, which contain equal number of records per interval, still work well. Cabin sees similar benefits as in the case of 32-bit uniform integer data in Figure 6(c).

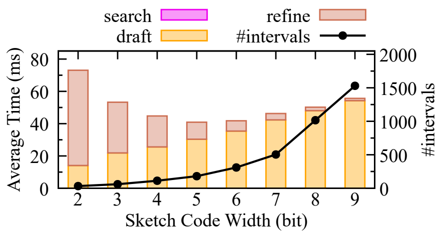
Scan Performance on Real-World Datasets. Figure 10 shows the scan performance on DBLP's `n_citation` column and IMDb's `startYear` column. Since both data sets contain a lot of duplicates, Cabin constructs data-aware intervals to achieve good performance. From Figure 10, we see that Cabin achieves the best scan time at all sizes for DBLP and at 0.5–2x column size for IMDb.

5.4 Benefit of Proposed Techniques in Cabin

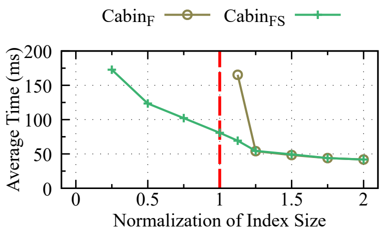
We study the benefit of the proposed techniques of Cabin. All experiments in this subsection use 32-bit data and the $\leq$ operator.

Filter Sketches. Figure 11(a) shows the scan time while varying the sketch code width w from 2 to 9, and limiting Cabin's size by $2 \times$ column size. The scan time is broken down into three parts of the Cabin$_F$ algorithm: 1) prepare and search (*search*), 2) generate the draft bit vector (*draft*), and 3) refine the bit vector (*refine*). Note that the search cost is tiny and hardly visible. As w increases, the number of intervals increases. Hence, the interval size and the number of bits to refine decrease accordingly, leading to the decreasing refine cost. On the other hand, a draft result bit vector is constructed from w filter sketches with MLO computation. As w increases, the draft cost increases linearly as guaranteed by theorem 3.1. Our DesignSelection algorithm computes the optimal design parameter. In this case, $w = 5$ obtains the best scan time, which is consistent with the output of the DesignSelection algorithm.

Scan Time Breakdown Varying g and sp. Figure 12 depicts the scan performance of Cabin with $w = 5$ and varying g and sp to satisfy the space limit. The line shows the average scan time,



(a) Filter sketches.



(b) Selective position array.

Fig. 11. Benefit of proposed techniques in Cabin.

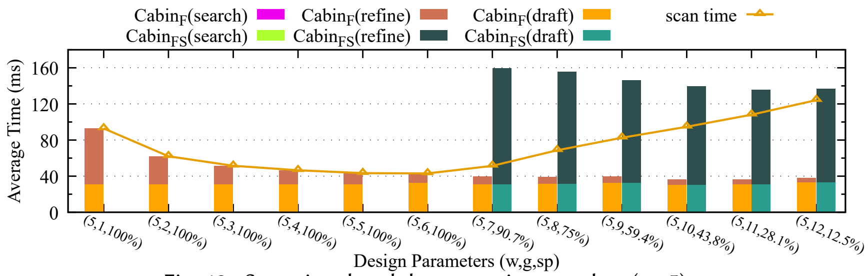
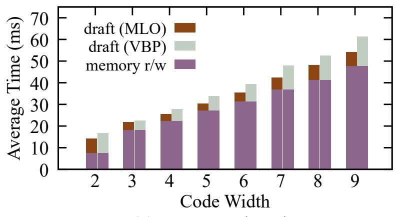
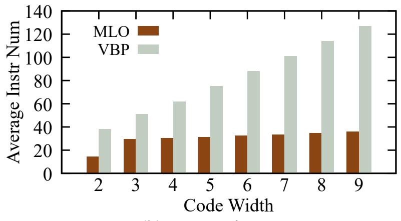


Fig. 12. Scan time breakdown varying g and sp (w=5).



(a) Computation time.



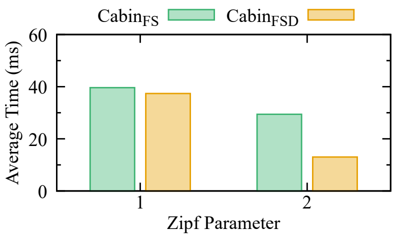
(b) Instruction count.

Fig. 13. Benefit of MLO in draft bit vector generation.

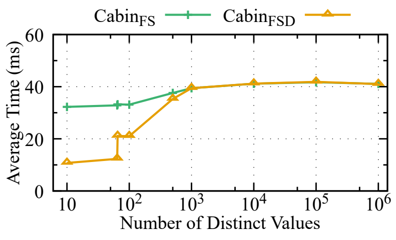
while the bars show the time breakdown into three components, i.e., search, draft, and refine, as in Figure 11(a). When sp < 100%, a query may hit an interval without the stored position array. In such cases, Cabin$_{FS}$ is executed instead of Cabin$_F$. From the figure, we see that (1) the search time is tiny; (2) refine is more costly in Cabin$_{FS}$ since Cabin$_{FS}$ has to visit the base data if filter sketches cannot tell the predicate outcome; and (3) (5,6,100%), which is chosen by Algorithm 3, achieves the best performance.

MLO. We compare MLO vs. VBP to understand the benefit of MLO in draft bit vector generation. Figure 13(a) and (b) compare the execution time and instruction count of MLO and VBP varying the code width. We see that compared to VBP, MLO significantly reduces the number of instructions for computing the draft bit vectors. While memory accesses account for a large portion of the execution time, the instruction reduction by MLO attains up to 19.6% improvement (at w=2) in execution time.

Selective Position Array. Figure 11(b) compares Cabin$_F$ and Cabin$_{FS}$ to understand the benefit of the selective position array. The red dotted line indicates the size of the full position array that stores N 32-bit rowIDs. Since it stores the full position array, Cabin$_F$'s size is always larger than 1 × column size. In comparison, by setting sp to be less than 1, the technique of selective position array stores the position array only for sp fraction of the intervals. In this way, Cabin$_{FS}$ supports space budges lower than 1 × column size. The selective position array substantially extends the range of index size that can be supported by Cabin.

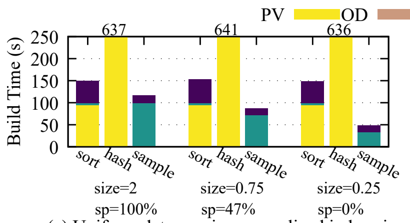


(a) Data with high skewness.

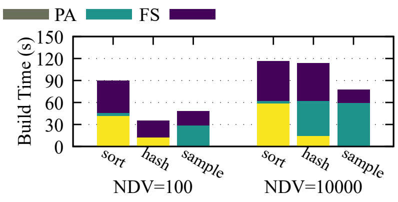


(b) Data with a low NDV.

Fig. 14. Benefit of data-aware intervals.



(a) Uniform data varying normalized index size.



(b) Data with duplicates varying NDV.

Fig. 15. Cabin build time breakdown.

Table 3. Build time of scan solutions (in seconds).

Solution	Vary data width (10^9 values)				Vary data size (32-bit)			
	8-bit	16-bit	32-bit	64-bit	10^6	10^7	10^8	10^9
Zone Maps	0.074	0.12	0.22	0.48	0.00023	0.0023	0.022	0.22
ByteSlice	1.65	1.87	2.17	/	0.0022	0.022	0.22	2.17
B+-tree	42.3	52.3	85.9	/	0.059	0.68	7.67	85.9
Column Sketches	30.2	80.4	108	111	0.082	0.91	9.86	108
BinDex	48.2	61.2	88.9	101	0.056	0.67	7.80	88.9
Cabin	52.7	81.9	117	132	0.089	0.99	10.8	117

Data-aware Intervals. We compare Cabin$_{FS}$ and Cabin$_{FSD}$ to understand the benefit of data-aware intervals. Similar to Section 5.3, we consider two cases: 1) skew data with Zipf distribution, and 2) data with a small number of distinct values. Cabin$_{FSD}$ employs data-aware intervals to optimize for popular values.

For case 1), Figure 14(a) shows the scan time while varying the Zipf parameter. We see that as the data is more skewed, Cabin$_{FSD}$ achieves more significant improvement over Cabin$_{FS}$. For case 2), Figure 14(b) reports the scan performance while varying the number of distinct values. We see that Cabin$_{FSD}$ works better than Cabin$_{FS}$ when there are fewer than 1000 distinct values. In such cases, entire intervals contain the same values, and the optimization of data-aware intervals is applicable. Cabin$_{FSD}$ improves the scan time and reduces the space cost at the same time.

5.5 Build Time

Popular Value Discovery and Build Time Breakdown. Figure 15 evaluates sort-based (*sort*), hash-based (*hash*), and sample-based (*sample*) methods for discovering popular values during Cabin index building. The build time is decomposed into five components: popular value discovery (*PV*), optimal design selection (excluding PV) (*OD*), and structure building for interval table (*IT*), position array (*PA*), and filter sketches (*FS*). We see that the optimal design selection using Algorithm 3 is fast (i.e. 3–41 microseconds) in all cases. Compared to sort and hash, sample reduces the cost of PV by using a small sample. It also avoids the sorting cost in PA for intervals whose position array is not stored (i.e., *sp* < 100%). Hash works well for data with duplicates, but incurs prohibitively high hashing cost for uniform data because the hash table is much larger than the CPU cache. Overall,

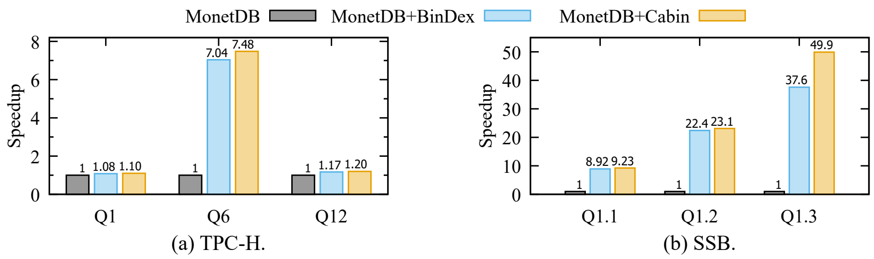


Fig. 16. Comparing MonetDB with scan index enhanced MonetDB using TPC-H and SSB workloads.

Table 4. Total and scan time in ms of TPC-H and SSB queries.

Queries	MonetDB	MonetDB+BinDex	MonetDB+Cabin
TPC-H Q1	12670 / 330.9	11699 / 10.3	11548 / 7.7
TPC-H Q6	521.7 / 468.9	74.1 / 22.3	69.8 / 19.1
TPC-H Q12	610.2 / 176.8	523.1 / 18.5	508.8 / 11.4
SSB Q1.1	10286 / 9699	1154 / 30.1	1115 / 17.3
SSB Q1.2	9214 / 9016	410.8 / 33.7	399.1 / 19.1
SSB Q1.3	9359 / 9205	248.7 / 40.5	187.5 / 20.5
SSB Q2.1	8560 / 4.84	8629 / 0.42	8613 / 0.12
SSB Q2.2	8444 / 1.43	8346 / 0.082	8363 / 0.043
SSB Q2.3	8333 / 1.89	8246 / 0.22	8309 / 0.11
SSB Q3.1	30815 / 7.63	28204/ 0.65	28328 / 0.18
SSB Q3.2	28955 / 5.52	26670 / 0.62	26540 / 0.19
SSB Q3.3	1517 / 7.13	1424 / 1.52	1467 / 1.35
SSB Q3.4	169.3 / 6.83	176.5 / 1.57	173.2 / 1.30
SSB Q4.1	21710 / 12.6	21596 / 1.74	21369 / 0.26
SSB Q4.2	4353 / 12.8	4411 / 1.80	4401 / 0.35
SSB Q4.3	2452 / 8.60	2468 / 1.22	2463 / 0.31

sample achieves good performance in all cases. Hence, we choose sample in Cabin building.

Build Time Varying Data Width and Data Size. Table 3 compares the build time of scan solutions. We vary the data width from 8-bit to 64-bit, and the data size (N) from 10^6 to 10^9 records. We see that Cabin takes modestly longer time than B+-Tree, Column Sketches, and BinDex because of its relatively complex structure. Since scan indices can be used by a large number of queries after building, it is beneficial to pay the build cost for better query performance in OLAP.

5.6 TPC-H and SSB Workloads

TPC-H. Figure 16(a) shows the query performance of TPC-H Q1, Q6, and Q12. Table 4 lists the total query time and the scan time of the TPC-H queries. We see that compared to MonetDB, MonetDB+Cabin achieves 1.10x, 7.48x, and 1.20x performance improvement for Q1, Q6, and Q12, respectively. Q1 and Q12 see less significant improvement than Q6. This is because scan accounts for only 2.6% of Q1's query time and 29.0% of Q12's query time, while scan takes 89.9% of Q6's query time. Focusing on the scan time in Table 4, we see that compared to MonetDB+BinDex, MonetDB+Cabin reduces the scan time significantly (by a factor of 1.17–1.62x), showing the benefits of filter sketches and data-aware intervals.

SSB. Figure 16(b) shows the query performance of SSB Q1.1, Q1.2, and Q1.3, and Table 4 shows the total query time and scan time for all SSB queries. Each SSB query performs at least one join operation. We see that in Q1.1–Q1.3, scan plays an important role in the query evaluation. Compared to MonetDB, MonetDB+Cabin achieves 9.23–49.9x performance improvement for Q1.1–Q1.3. In contrast, in queries Q2.1–Q4.3, scan accounts for only a small fraction of the total query time. As a result, the index enhanced solutions have similar performance as MonetDB. Overall, if we focus on the scan time, we see that MonetDB+Cabin improves the scan performance in all SSB queries by a factor of 5.2–560x compared with MonetDB and 1.1–6.7x compared with MonetDB+BinDex.

6 DISCUSSION

PAX Layout. In this work, Cabin is constructed on a whole data column. In addition to the column layout, PAX is another popular data layout in OLAP systems [4]. In PAX, a table is divided into row groups, and each row group employs the column layout. To support PAX, we can simply build a Cabin per row group, and use the per-row-group Cabins to accelerate scans.

Data Updates. For deletes, we can record the deleted rows in a delete bit vector. Then, a scan computes the bit-wise AND of the scan result bit vector and the delete bit vector. For inserts, we can employ the main+delta approach. We build Cabin indices for the main data, and append newly inserted values to the delta data. When merging the main and delta into the new main data, we re-construct Cabin indices. In this way, a scan consists of the Cabin-enhanced scan on the main data, and a plain scan on the delta data. An update can be supported as a delete followed by an insert.

Support for Strings. Scan indices, including Cabin, mostly focus on numeric values and filter predicates (e.g., $<$, $>$, $\leq$, $\geq$, $=$, $\neq$, or BETWEEN) that specify value ranges. In many cases, strings can be encoded as numeric values and effectively supported for such predicates. However, string matching operations (e.g., LIKE) cannot be easily supported by scan indices. There is no clear sort order for string matching operations, but the sort order is the basis for the design of many scan indices. It would be interesting to study how to combine inverted indices and scan indices to improve OLAP queries with string matching operations.

Optimization Based on Query History. In this work, our Cabin design does not rely on any query history. Here, we consider potential optimizations if the query history is available. (1) Index selection: We can identify frequent queries and analyze the importance of scans to query performance. Then, we can choose a subset of columns to build Cabin indices in order to maximize the performance benefit given memory space budget. (2) Query-aware selective position array: Instead of randomly selecting intervals, we can choose which intervals to remove position arrays based on the distribution of query predicate values.

7 CONCLUSION

In this paper, we propose and evaluate a novel scan index, Cabin. Extensive experiments show that Cabin achieves better time-space tradeoff than state-of-the-art scan solutions. Cabin is a promising scan index for main-memory analytical databases.

ACKNOWLEDGMENTS

This work is partially supported by Natural Science Foundation of China (62172390). We thank the anonymous meta-reviewer and reviewers for their insightful comments and suggestions. Shimin Chen is the corresponding author.

REFERENCES

[1] 2023. DBLP Citation Network Dataset. https://www.aminer.cn/citation.
[2] 2023. IMDb Datasets. https://developer.imdb.com/non-commercial-datasets/.
[3] 2023. stx::B+tree(tlx::B+tree). https://github.com/tlx.

[4] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*. Morgan Kaufmann, 169–180.

[5] Albert-László Barabási and Réka Albert. 1999. Emergence of scaling in random networks. *science* 286, 5439 (1999), 509–512.

[6] Ronald Barber, Peter Bendel, Marco Czech, Oliver Draese, Frederick Ho, Namik Hrle, Stratos Idreos, Min-Soo Kim, Oliver Koeth, Jae-Gil Lee, Tianchao Tim Li, Guy M. Lohman, Konstantinos Morfonios, René Müller, Keshava Murthy, Ippokratis Pandis, Lin Qiao, Vijayshankar Raman, Richard Sidle, Knut Stolze, and Sandor Szabo. 2012. Business Analytics in (a) Blink. *IEEE Data Eng. Bull.* 35, 1 (2012), 9–14.

[7] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. 2009. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*. ACM, 283–296.

[8] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the memory wall in MonetDB. *Commun. ACM* 51, 12 (2008), 77–85.

[9] David Broneske, Sebastian Breß, and Gunter Saake. 2014. Database Scan Variants on Modern CPUs: A Performance Study. In *Proceedings of the 2nd International Workshop on In Memory Data Management and Analytics, IMDM 2014, Hangzhou, China, September 1, 2014*. 1–15.

[10] Chee Yong Chan and Yannis E. Ioannidis. 1998. Bitmap Index Design and Evaluation. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*. ACM Press, 355–366.

[11] Douglas Comer. 1979. The Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (1979), 121–137.

[12] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2011. SAP HANA database: data management for modern business applications. *SIGMOD Rec.* 40, 4 (2011), 45–51.

[13] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. 2015. ByteSlice: Pushing the Envelop of Main Memory Data Processing with a New Storage Layout. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. ACM, 31–46.

[14] Craig Freedman, Erik Ismert, and Per-Åke Larson. 2014. Compilation in the Microsoft SQL Server Hekaton Engine. *IEEE Data Eng. Bull.* 37, 1 (2014), 22–30.

[15] John L. Hennessy and David A. Patterson. 2012. *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann.

[16] 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, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 857–872.

[17] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*. www.cidrdb.org, 68–78.

[18] Ryan Johnson, Vijayshankar Raman, Richard Sidle, and Garret Swart. 2008. Row-wise parallel predicate evaluation. *Proc. VLDB Endow.* 1, 1 (2008), 622–634.

[19] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive Execution of Compiled Queries. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 197–208.

[20] 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, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 489–504.

[21] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. 2010. Generating code for holistic query evaluation. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*. IEEE Computer Society, 613–624.

[22] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, Juan Loaiza, Neil MacNaughton, Vineet Marwah, Niloy Mukherjee, Atrayee Mullick, Sujatha Muthulingam, Vivekanandhan Raja, Marty Roth, Ekrem Soylemez, and Mohamed Zaït. 2015. Oracle Database In-Memory: A dual format in-memory database. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*. IEEE Computer Society, 1253–1258.

[23] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, 311–326.

[24] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013.* IEEE Computer Society, 38–49.

[25] Linwei Li, Kai Zhang, Jiading Guo, Wen He, Zhenying He, Yinan Jing, Weili Han, and X. Sean Wang. 2020. BinDex: A Two-Layered Index for Fast and Robust Scans. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020.* ACM, 909–923.

[26] Yinan Li and Jignesh M. Patel. 2013. BitWeaving: fast scans for main memory data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013.* ACM, 289–300.

[27] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA.* Morgan Kaufmann, 476–487.

[28] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.

[29] Patrick E. O'Neil and Dallan Quass. 1997. Improved Query Performance with Variant Indexes. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA.* ACM Press, 38–49. https://doi.org/10.1145/253260.253268

[30] Holger Pirk, Stefan Manegold, and Martin L. Kersten. 2014. Waste not... Efficient co-processing of relational data. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014.* IEEE Computer Society, 508–519.

[31] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015.* ACM, 1493–1508.

[32] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.* ACM, 1981–1984.

[33] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter R. Mattson, and John D. Owens. 2000. Memory access scheduling. In *27th International Symposium on Computer Architecture (ISCA 2000), June 10-14, 2000, Vancouver, BC, Canada.* IEEE Computer Society, 128–138.

[34] Lefteris Sidirourgos and Martin L. Kersten. 2013. Column imprints: a secondary index structure. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013.* ACM, 893–904.

[35] Liwen Sun, Michael J. Franklin, Sanjay Krishnan, and Reynold S. Xin. 2014. Fine-grained partitioning for aggressive data skipping. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014.* ACM, 1115–1126.

[36] Skye Wanderman-Milne and Nong Li. 2014. Runtime Code Generation in Cloudera Impala. *IEEE Data Eng. Bull.* 37, 1 (2014), 31–37.

[37] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Faerber. 2013. Vectorizing Database Column Scans with Complex Predicates. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2013, Riva del Garda, Trento, Italy, August 26, 2013.* 1–12.

[38] Marcin Zukowski, Peter A. Boncz, Niels Nes, and Sándor Héman. 2005. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Eng. Bull.* 28, 2 (2005), 17–22.

Received July 2023; revised October 2023; accepted November 2023