BinDex: A Two-Layered Index for Fast and Robust Scans

Linwei Li, Kai Zhang, Jiading Guo, Wen He, Zhenying He, Yinan Jing, Weili Han, X. Sean Wang
Fudan University
{lwli15,zhangk,guojd19,wenhe19,zhenying,jingyn,wlhan,xywangCS}@fudan.edu.cn

ABSTRACT

In modern analytical database systems, the performance of the data scan operation is of key importance to the performance of query execution. Existing approaches may be categorized into index scan and sequential scan. However, both approaches have inherent inefficiencies. Indeed, sequential scan may need to access a large amount of unneeded data, especially for queries with low selectivity. Instead, index scan may involve a large number of expensive random memory accesses when the query selectivity is high. Moreover, with the growing complexities in database query workloads, it has become hard to predict which approach is better for a particular query.

In order to obtain fast and robust scans under all selectivities, this paper proposes BinDex, a two-layered index structure based on binned bitmaps that can be used to significantly accelerate the scan operations for in-memory column stores. The first layer of BinDex consists of a set of binned bitmaps which filter out most unneeded values in a column. The second layer provides some auxiliary information to correct the bits that have incorrect values. By varying the number of bit vectors in the first layer, BinDex can make a tradeoff between memory space and performance. Experimental results show that BinDex outperforms the state-of-the-art approaches with less memory than a B^+-tree would use. And by enlarging the memory space, BinDex can achieve up to 2.9 times higher performance, eliminating the need for making a choice between sequential or index scans.

CCS CONCEPTS

• **Information systems → Data scans**.

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3380563

KEYWORDS

scan; in-memory column stores; indexing

ACM Reference Format:
Linwei Li, Kai Zhang, Jiading Guo, Wen He, Zhenying He, Yinan Jing, Weili Han, X. Sean Wang. 2020. BinDex: A Two-Layered Index for Fast and Robust Scans. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318464.3380563

1 INTRODUCTION

Access path selection for selection operations is one of the critical components in analytical database systems. The choices of the data access path are generally between a sequential scan and an index scan. The performance of the approaches varies according to workloads, where one approach can deliver up to three times higher performance than the other [17]. To use a better approach in query execution, query optimizers generally employ cost models to estimate the scan performance. However, while the complexity of database query workloads has been growing, the cost models often use simple assumptions in the estimation [9]. Consequently, it becomes difficult to predict the optimal approach, resulting in a suboptimal query performance [5].

The huge performance disparities between the index scans and the sequential scans come from their inherent characteristics. The two classes of approaches differ in both the amount of data accessed and the data access pattern. A sequential scan needs to evaluate all the values in a column. Instead, an index scan only accesses the values that satisfy the predicate with the help of some auxiliary information. Therefore, when a vast majority of the values are unneeded for predicates with low selectivities (< ∼1%), an index scan can achieve much higher performance than a sequential scan [17]. However, this is not the case for predicates with higher selectivities (> ∼1%). In this case, because the values satisfying the predicate may scatter over all the column, an index scan may involve a number of random memory accesses to the base data and to the index data structure. The operation time of the random memory accesses is known to be an order of magnitude larger than that of sequential memory accesses [4, 16]. As a result, the overhead of random

accesses would counteract the benefit of avoiding accessing unneeded data incurred with a sequential scan when the selectivity becomes higher, and sequential scans would outperform index scans. Overall, both index scans and sequential scans have their inherent inefficiencies.

We propose BinDex, a two-layered index that significantly enhances the scan performance for in-memory column stores robustly under all selectivities by adopting the main advantages from both existing index scans and sequential scans. The main idea of BinDex is (1) avoid accessing most of unneeded values that don't satisfy the predicate while (2) avoiding or mitigating the overheads caused by random memory accesses. To achieve these goals, BinDex employs a data structure with two layers that work together to perform the scan operation. The first layer of BinDex uses bitmaps with binning. Binning is a technique that partitions the attribute values into a number of ranges, and uses a bit vector to represent each range. In evaluating a predicate, the bit vector in the filter layer that is closest to the final result is selected, where the majority of the bits have the correct values with respect to the predicate. Then the bits of the chosen vector that with incorrect values are corrected to generate the final result for the predicate. In traditional bitmap indexes, the base data needs to be probed to correct the bits in the bit vector, but the probing incurs huge overhead and renders the performance much lower than the other approaches. In BinDex, the second layer uses some auxiliary information to correct the selected bit vector with low overhead. Specifically, the second layer stores all the rowIDs in a sequence with their corresponding values in ascending order. With the selected bit vector, BinDex locates the rowIDs of the values whose bits are incorrectly set/unset. Then the stored rowIDs are used to correct the corresponding bits. Since the rowIDs are sequentially stored, they are accessed with sequential memory reads. Furthermore, the rowIDs are used to prefetch the corresponding bits to write to further alleviate the random write costs. These two layers tightly collaborate to improve the overall performance of the scan operations.

An important characteristic of BinDex is its ability to make tradeoffs between performance and memory usage. With a higher number of bit vectors, fewer values need to be probed in the second layer, which leads to higher performance but larger memory usage. Hence, we introduce memory space as the main dimension in the selection of scan approaches. It is worth noting that, with less memory than a B^+-tree uses, BinDex can outperform state-of-the-art approaches for various workloads. This brings notable advancements for analytical database systems, since it not only leads to higher performance but also avoids the potential performance issues due to inaccurate estimation used in access path selection.

The main contributions of this paper are fourfold.

- We propose BinDex, a two-layered index structure that significantly accelerates the scan operation in in-memory analytical database systems.
- We propose a cost model to help make performance-space tradeoffs in BinDex to maximize the scan performance with a specified memory constraint.
- We intensively evaluate BinDex under various workloads. Our experimental results show that BinDex can achieve up to 2.9× higher performance than the state-of-the-art approaches.
- We propose a policy that uses memory space as the main dimension for selecting the optimal scan approaches in in-memory database systems.

The rest of the paper is organized as follows. Section 2 analyzes the performance of the state-of-the-art approaches. Section 3 demonstrates the methodology and the data structure of BinDex. Section 4 describes the detailed techniques in the design and implementation. Section 5 demonstrates a cost model to help make the tradeoff between space and performance. Section 6 evaluates BinDex with a prototype, and Section 7 discusses the selection of scan approaches based on the memory constraint. Section 8 discusses the related work, and Section 9 concludes the paper.

2 BACKGROUND AND MOTIVATION

In this section, we analyze state-of-the-art index scan and sequential scan approaches for in-memory database systems. As lots of techniques being proposed recently have blurred the line between the two approaches, we establish a rule to distinguish them: If an approach needs to inspect all the values (compressed or not) in a column, we classify it as a sequential scan, while an approach is classified as an index scan if it only accesses values that satisfy a predicate with auxiliary information. We take four main scan approaches to evaluate and analyze their performance, including the traditional secondary index B^+-tree and three sequential scan techniques: early pruning on bit-level storage layout, lightweight indexes, and lossy compression.

2.1 The Scan Operation

Main memory databases usually store data in a compressed format with fixed-length order-preserving schemes [18], where sequential scans are performed on the compressed codes. We use code and value interchangeably in the paper. In addition to integer values, other value types can also be encoded into integer codes, including floating points [11] and strings [2]. The input of a scan operation is a list of N codes and a predicate with a comparison operator, e.g. $<$, $>$, $\leq$, $\geq$, $=$, $\neq$, or BETWEEN. The scan process finds out all matching codes that satisfy the predicate and outputs an N-bit result bit vector or a rowID list. A rowID list stores the rowIDs of all codes

that satisfy the predicate. The result bit vector also indicates the rowIDs of the matching codes, where the ith bit is set as 1 if the code in the ith row satisfies the predicate.

A result bit vector is a common and efficient way for recording scan results, and it has been adopted by state-of-the-art approaches [12, 15, 19]. For complex queries with multiple predicates, predicates on the corresponding columns are first evaluated with a scan. The scan for each predicate generates a result bit vector. Then logical operators such as AND, OR and NOT are performed on these bit vectors for conjunctions or disjunctions, which are more efficient than merging rowID lists [7]. Then the final result bit vector is used to retrieve other columns of interest for this query.

2.2 Secondary Index Scan

A secondary index is a small but structured data structure that filters unneeded values and their other attributes. Traditional secondary indexes are typically in the form of bitmap, B-tree and its variants [10, 28]. They have been widely used in many row-oriented relational database systems.

As an invariant of B-tree, B^+-tree is a self-balancing tree that supports search, insert, and delete operations in logarithmic time. Different with a binary tree, the node in a B^+-tree contains up to tens or hundreds of children, leading to lower height and higher storage efficiency. Based on a B^+-tree, the process of an index scan consists of two main steps. First, according to the requested value range, the tree is traversed to locate the leaf node that contains the first value in the range. Second, leaf nodes are traversed in the B^+-tree to access the values in the range. At the same time, the result (generally a bit vector) is generated with the rowIDs of the indexed data. For in-memory databases, scanning with a B^+-tree involves large amounts of random memory accesses.

Index scans have been extensively adopted in row-oriented databases, because they avoid accessing all the data in the table. After database systems have evolved to the columnar storage, sequential scans outperform index scans in most cases [17]. The main reason is that scans on columnar storage layouts avoid accessing other unneeded attributes.

2.3 Sequential Scan

In recent years, many approaches have been proposed on accelerating sequential scans on the columnar storage [12, 15, 19, 20, 29, 32]. The state-of-the-art scan approaches take various measures to reduce the amount of accessed data to enhance the overall performance.

2.3.1 Early Pruning on Bit-Level Storage Layout. Bit-level storage layouts including Bitweaving [19] and ByteSlice [12] utilize a technique called early pruning to accelerate sequential scans. The technique physically partitions a code into multiple sub-values, and early pruning offers the opportunity of skipping the least significant bits when all values in a group satisfy (or not satisfy) a predicate with the most significant bits. Taking the Vertical Bit-Parallel method in Bitweaving (called Bitweaving/V) as an example, Bitweaving/V breaks down a column of codes into fixed-length memory blocks, called segments. Each segment contains n codes, where n is the width of a processor word. The n k-bit codes are decomposed into k n-bit words, where the i-th bit in the j-th word equals to the j-th bit of the original i-th code. Bitweaving/V evaluates the n codes in a segment in parallel, from the most significant bit to the least significant bit. To evaluate a predicate $v < (11011)_2$, for instance, it can be known that the code $(10100)_2$ satisfies the predicate at the second bit. After all the results of the codes in a segment are known, the scan stops and the rest bits in the segment do not need to be be accessed. The technique effectively accelerates the sequential scan with no extra memory overhead.

Table 1: The throughputs of data access patterns on one core of Intel Xeon E5-2695 v4

	Seq. Read	Random Read	Seq. Write	Random Write
Throughputs (Gb/s)	4.43	0.27	4.71	0.24

2.3.2 Lightweight indexes. Lightweight indexes are techniques that skips data by using summary statistics over data zones. Such techniques include Zone Maps [20], Column Imprints [29], and Feature Based Data Skipping [32]. For instance, as a widely used technique, Zone Maps partition data into zones and record the metadata of each zone, such as *min* and *max*. With data partitioning, the approach skips zones where all values in the zone satisfy or not satisfy the predicate. When data does not exhibit clustering properties, however, the technique is unable to skip data [15, 25].

2.3.3 Lossy Compression. Hentschel et. al. [15] propose a scan approach based on lossy compression, called Column Sketches. Column Sketches use a lossy compression map to compress the base data to smaller codes and store them as a column called sketched column. Most of the codes are evaluated affirmatively or negatively after a sequential scan on the sketched column, while the base data is further checked for the remaining uncertain codes. Column Sketches reduce the amount of data to be accessed for the lossy compression technique. While Column Sketches achieve much higher performance improvements than other approaches, the technique needs to access all compressed codes with a sequential scan, and random memory accesses to the base data are needed for checking the uncertain codes.

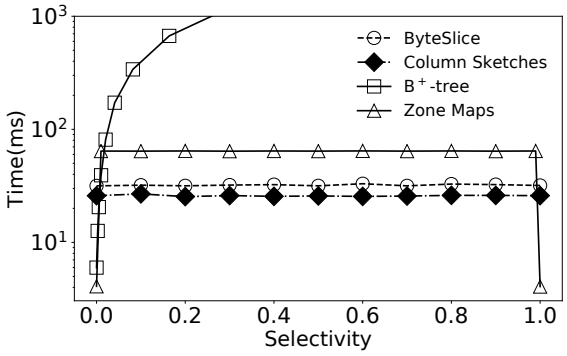


Figure 1: Performance comparison between the state of the art approaches

2.4 Performance Analysis

There is a competition between index scans and sequential scans in access path selection. By varying predicate selectivities, Figure 1 shows the performance of B$^+$-tree, ByteSlice, Zone Maps, and Column Sketches when evaluating a predicate on a column with one billion 32-bit values under uniform distribution. The experimental platform is described in Section 6. In our experiment, the B$^+$-tree demonstrates higher performance than sequential scans with a selectivity of less than 1.0%, while sequential scans outperform the B$^+$-tree for predicates with higher selectivities. There is a huge performance gap between the two scans for different selectivities, which is also verified in [17]. Consequently, access path selection is needed to achieve the best performance.

Memory accesses are the main overhead in in-memory data processing. We evaluate the throughputs of different memory access patterns and show the results in Table 1. The throughput of sequential read is around 15.4× higher than that of random read, while sequential write is 18.6× faster than random write. This is because current CPUs can identify the sequential memory access pattern and prefetch the following data into the cache. Instead, random memory accesses result in cache misses, and a CPU core should wait for the data to be fetched into the cache. Therefore, the memory access pattern significantly influences scan performance.

To gain insights into the performance of scan approaches, we utilize perf_event_open[1] to measure the last level cache (LLC) misses of the four scan approaches, i.e., Column Sketches, ByteSlice, Zone Maps and BinDex. Since each LLC miss results in an memory access, we use it as the metric to demonstrate the memory access overhead of scan approaches. Figure 2 shows the experimental results that are normalized to Column Sketches. The performance numbers reported are the average results of 100 randomly chosen selectivities under uniform distribution. Among the three sequential scan

[1]http://man7.org/linux/man-pages/man2/perf_event_open.2.html

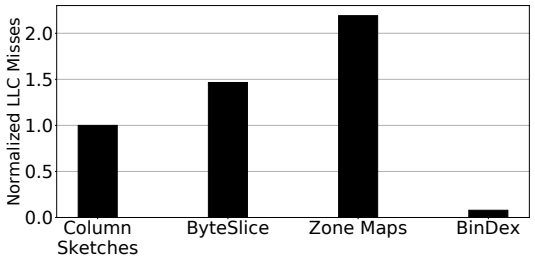


Figure 2: Normalized last level cache misses of four scan approaches

approaches, Column Sketches demand the least number of memory accesses. ByteSlice uses early pruning to reduce the amount of memory to be accessed, while Zone Maps have the highest number of memory accesses because they are unable to skip zones for the evaluated predicate. Compared with other approaches, BinDex has a 11.6-26.8× lower number of LLC misses. This shows that BinDex can dramatically alleviate the expensive memory access overheads.

3 BINDEX: AN OVERVIEW

BinDex is a two-layered index data structure that generates a result bit vector for a scan operation, where each bit indicates whether the corresponding value satisfies the predicate. In this section, we take Figure 3 as an example to demonstrate the methodology of BinDex.

3.1 BinDex Data Structure

A critical concept in BinDex is *virtual value space*. A column indexed by BinDex has a virtual value space, in which all values are sorted in ascending order. Given N values $x_{1...N}$ in a column, the virtual value space is partitioned into K *virtual areas*, denoted as $A = \{A_1, A_2, ..., A_K\}$. Each virtual area serves as a bin and contains around N/K sorted values. In Figure 3, the 16 values in the virtual value space are partitioned into four virtual areas, each of which contains four values. Please note that the values in the column are still stored in the original order, where the virtual value space is only a concept in BinDex which is not stored separately.

Based on the virtual value space, there are three main data structures in BinDex, i.e., an area map, a set of filter bit vectors, and a position array. The area map $S = \{S_1, S_2, ..., S_{K-1}\}$ maps a value to a virtual area and is used by both the two layers. Each element in the area map S_i contains a $(value, count)$ pair, which denotes the start value of the virtual area A_{i+1} and the number of values contained from A_1 to A_i. If a constant c satisfies $S_{i-1}.value \leq c < S_i.value$, it locates in the virtual area A_i. In our example, there are three elements in the area map. The second element (114, 8) denotes that the first value in virtual area A_3 is 114, and there are 8 values in A_1 and A_2.

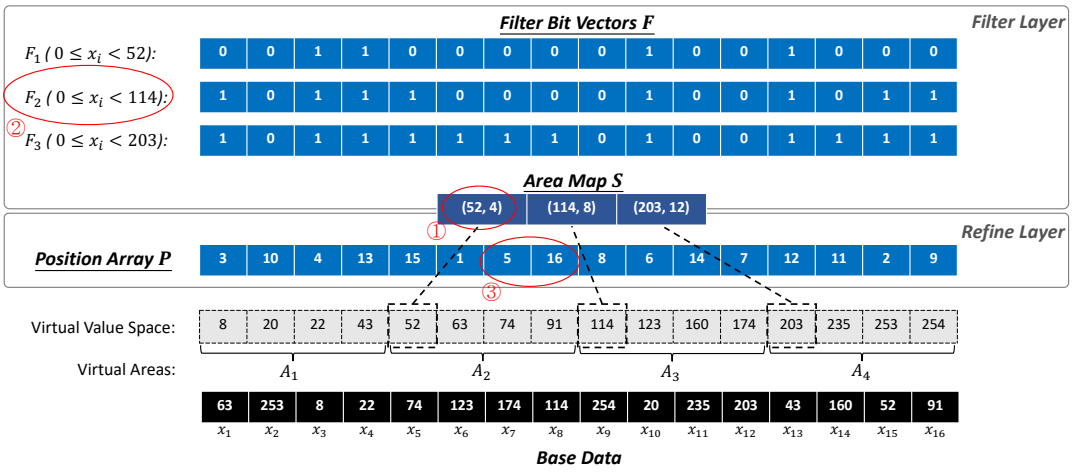


Figure 3: The data structure of BinDex

The first layer of BinDex is called *filter layer*, which utilizes binning to construct a set of bit vectors. The bit vectors are used to generate draft scan results for a predicate. Based on the virtual area, the filter layer consisting of $K-1$ *filter bit vectors*, denoted as $F = \{F_1, F_2, ..., F_{K-1}\}$. The ith filter bit vector F_i is an order-preserving array consists of N bits, where each bit indicates whether the corresponding value is in the first i virtual areas, i.e., a bit in the filter bit vector F_i is set as 1 if the corresponding value is less than $S_i.value$. In Figure 3, the bits for values less than 114 (the start point of the virtual area A_3) are set as 1 in the bit vector F_2. When evaluating a predicate, the filter layer first selects a bit vector that is closest to the final result as the candidate result bit vector. Since the bits for most values that satisfy the predicate are set as 1 in the candidate result bit vector, only a small portion of the values need to be probed.

The second layer is called *refine layer*. It refines and corrects the draft result from the filter layer to deliver the final result bit vector. In the refine layer, the rowIDs of all values in the virtual value space are sequentially stored in a data structure, called *position array*. In a candidate result bit vector, only part of the values in a virtual area do not satisfy the predicate. We first locate the range of values whose bits need to be corrected with a binary search (details in Sec. 4.2), then their rowIDs in the position array are found. As the values in the virtual value space are sorted, their rowIDs are stored sequentially in the position array. With the range of rowIDs, the corresponding bits in the filter bit vector are corrected. In BinDex, the expected number of values that need to be probed for a predicate is 1/4 the number of values in a virtual area (see Section 4.1). Please note that BinDex is flexible, where any index (e.g. B$^+$-tree) that supports efficient range query to quickly locate the uncovered value range in the filter bit vector can be used in the refine layer. Because BinDex aims at accelerating the select operator in analytical databases where data updating is not frequent, we design the refine layer as a position array to improve refine efficiency and save memory space.

Overall, BinDex maintains the benefits of traditional indexes since it avoids accessing most of the data that does not satisfy a predicate in the filter layer. The refine layer mitigates the random memory access overheads by reorganizing the rowIDs of data for sequential accesses. The two layers tightly collaborate to accelerate the scan operation.

3.2 Example: Scan with BinDex

In this subsection, we use the example case in Figure 3 and take the predicate "$x < 69$" to illustrate the scan procedure with BinDex. We are going to explain how to perform operations including $>$,$\le$, $\ge$, $=$ and BETWEEN in Section 4.3.

Since the constant 69 falls in the range [52, 114), S_1 is located through a binary search on the area map (①). Then the filter bit vector F_2 is copied as the candidate result bit vector (②). Since all the bits for values in [8, 114) are 1 in F_2, the bits for values in [69, 114) should be unset as 0 to get the correct result bit vector. Please note that F_1 can also be selected because it contains partial results except for values in the range [52, 69), which should be set as 1 in the refine layer. We will elaborate the selection of filter bit vectors to minimize the number of probes in Section 4.1.

In the refine layer, we first locate the first value that is larger than 69, which is 74. Then the rowIDs for values in [69, 114) are located in the position array. As shown in Figure 3, there are two values in the range, i.e., 74 and 91, whose rowIDs are 5 and 16, respectively (③). Then the rowIDs in the position array are sequentially accessed to unset the corresponding bits (5th and 16th) in the result bit vector as 0. When the procedure completes, the final result bit vector for the predicate "$x < 69$" is generated.

4 BINDEX DESIGN AND IMPLEMENTATION

BinDex adopts three main design decisions to enhance the scan performance. First, the selection of the optimal filter bit vector minimizes the number of probes for a predicate (Section 4.1). Second, sequentially accessing the rowIDs addresses the random read overheads. Meanwhile, rowIDs are used to prefetch the bits to be written in the result bit vector, mitigating the random write overheads (Section 4.2). Third, evenly partitioning the number of values among virtual areas controls the number of probes in each virtual area (Section 4.4). With the techniques adopted, BinDex achieves consistently high performance for predicates with different operators, selectivities, and data distributions.

4.1 The Selection of Filter Bit Vector

In BinDex, a filter bit vector is selected in the filter layer as a draft of the result bit vector. Given a predicate $x < c$, there are generally two candidate filter bit vectors. We use A_{k_c} to denote the virtual area that the predicate constant c belongs to. Values locate in A_{k_c} are all "uncertain values" which could not be affirmed or negated by the filter bit vector. Then both F_{k_c-1} and F_{k_c} can be chosen as the draft result bit vector since values in area A_{k_c} are set as 1 in F_{k_c} but unset as 0 in F_{k_c-1}. Therefore, they are treated differently in the refine layer to obtain the final result bit vector. If F_{k_c-1} is chosen, the corresponding bits of values that satisfy the predicate in A_{k_c} should be set as 1. On the contrary, the corresponding bits of values that do not satisfy the predicate in A_{k_c} should be unset as 0 in F_{k_c}.

In the example case (Figure 3), to evaluate the predicate $x < 60$, either F_2 or F_1 can be selected since 60 locates in A_2. If F_2 is selected, the bits of x_1, x_5 and x_{16} should be reset to 0, since they don't satisfy the predicate but are set as 1 in F_2. However, if F_1 is selected, only the bit of x_{15} should be set as 1. This example demonstrates that there can be a huge variance in the number of probed values with different filter bit vectors. The number of probes influences the number of memory accesses, which exerts a huge impact on the overall performance. Therefore, a filter bit vector that is as similar as possible to the result bit vector should be selected.

To evaluate a predicate $x < c$ in BinDex, we firstly conduct a binary search over the area map to locate the corresponding virtual area A_{k_c}. Then we use a binary search over the area A_{k_c} to find the last value x_{i_c} that is less than c, here i_c denotes the corresponding position in the virtual area space. Here the binary search is performed via the position array, where the rowIDs are used to access the corresponding values in the base data for comparison. After locating x_{i_c}, the number of values to be probed in F_{k_c-1} and F_{k_c} can be calculated as $i_c - S_{k_c-1}.count$ and $S_{k_c}.count - i_c$, respectively. F_{k_c-1} is selected as the candidate result bit vector if $i_c < \frac{1}{2}(S_{k_c-1}.count + S_{k_c}.count)$ or F_{k_c} for otherwise.

With K virtual areas, there are on average N/K values in each virtual area. The number of values need to be probed (N_m) to get the result bit vector is calculated as

$$N_m(c) = \begin{cases} i_c - (k_c - 1) \cdot N/K, & \text{if } \frac{i_c-(k_c-1)\cdot N/K}{N/K} < \frac{1}{2} \\ k_c \cdot N/K - i_c, & \text{if } \frac{1}{2} \le \frac{i_c-(k_c-1)\cdot N/K}{N/K} \le 1 \end{cases} \tag{1}$$

We assume that the predicate constants in queries have a uniform distribution among the base data, thus i_c is an uniform random variable. The mathematical expectation of N_m is

$$\mathbb{E}(N_m) = \frac{K}{N} \cdot \sum_{i_c=\frac{(k_c-1)N}{K}}^{\frac{k_c N}{K}} N_m = \frac{K}{N}\frac{N^2}{4K^2} = \frac{N}{4K} \tag{2}$$

We can see that the expected number of probed values is 1/4 of the values in a virtual area.

In BinDex, we also use another two bit vectors in query evaluation, i.e., bit vectors with all bits set/unset as 0 or 1. For instance, to evaluate the predicate $x < 20$ in Figure 3, the all-0 bit vector instead of F_1 is used to minimize the number of probes. The all-1 bit vector is selected when evaluating the predicate $x < 254$. We do not store the two bit vectors, but generate them with *calloc()* and *memset()*, which overhead is lower than that of copying a stored one.

4.2 Result Refining

On average, $\frac{N}{4K}$ values need to be verified with a filter bit vector. The positions of the values, i.e., either $\{P_{S_{k_c-1}.count+1}, ..., P_{i_c}\}$ or $\{P_{i_c+1}, P_{i_c+1}, ..., P_{S_{k_c}.count}\}$, are stored sequentially within the position array. Therefore, the rowIDs in the position array can be accessed with sequential memory accesses. However, the to-be-modified bits in the selected filter bit vector are not sequentially stored. As a result, writing the result bit vector involves random memory accesses. As discussed in Section 2.4, the overhead of random memory accesses can seriously impact the scan performance. In BinDex, the overhead has been significantly mitigated since only $\frac{1}{4K}$ of the values need to be probed. Moreover, we utilize the rowIDs stored in the position array to perform software pipelining to further mitigate the random write overhead. The rowIDs are used to calculate the addresses of the bits in the candidate result bit vector, with which *prefetching* is adopted to load the subsequent bits into the CPU cache to address the random write overhead.

4.3 Scan Algorithms

Algorithm 1 shows the pseudocode of the BinDex scan procedure in evaluating predicates with the "<" operator. Given

Algorithm 1: BinDex scan for a predicate with "<" operator

Input : a predicate "$x < c$" on column $X_{1...N}$, filter bit vector $F_{1...K}$, area map S, position array P
Output : the result bit vector

```
1  k_c = binary_search_area(c, S);
2  i_c = binary_search_pos(c, P, X);
3  if i_c < 1/2(S_{k_c-1}.count + S_{k_c}.count) then
4  |  k_s = k_c − 1, p_start = S_{k_s}.count + 1, p_end = i_c;
5  else
6  |  k_s = k_c, p_start = i_c + 1, p_end = S_{k_s}.count;
7  end
8  F_r = bit_vector_build(k_s);
9  for i ← p_start to p_end do
10 |  if i + h <= p_end then
11 |  |  prefetch (&F_r[ get_pos(i + h)]);
12 |  end
13 |  flip_bit(F_r[get_pos(i)]);
14 end
15 return R;
```

a predicate $x < c$, we first locate the virtual area A_{k_c} that contains the last value that is less than c. A binary search is performed on the *value* attribute of the area map S (Line 1). Then the rowID of the value (denoted as i_c) is located with binary search in A_{k_c} (Line 2). In the binary search on line 2, since values are sorted in the virtual value space, their sequentially-stored rowIDs in the position array are used to access the values in the base data for comparison.

Then we select the filter bit vector that needs the smallest number of probes and record the range of rowIDs whose bits need to be modified in the result bit vector (Line 3-7). Line 8 generates the result bit vector, where an all-0 bit vector is allocated if $k_s = 0$, or an all-1 bit vector is built if k_s equals to the total number of virtual areas. Otherwise, F_{k_s} is copied as the draft of the result bit vector (Line 8). In the end, the copied bit vector is refined with prefetching (Line 9-14). The instruction on line 11 prefetches the *get_pos(i+h)*th bit of F_r into the cache. *get_pos()* is a function that fetches the ith rowID in the position array. Then each prefetched bit is flipped to get the final result bit vector (Line 13). The prefetched data will be accessed after h iterations, where h is the prefetching stride to interleave computation and memory accesses. In this way, data manipulation pipelines with data loading to mitigate the random write overheads.

BinDex can be parallelized to utilize multi-core resources to enhance the overall performance. The best way to parallelize BinDex is to have multiple threads cooperatively copying different parts of the selected filter bit vector to form the result bit vector. After that, a thread identifies the

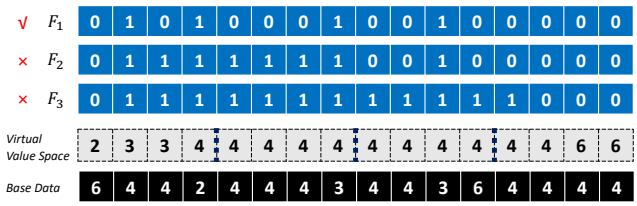


Figure 4: Index Compression

range of the values to be refined in the position array, and evenly partitions the range among the threads. Then each thread refines the result bit vector with the atomic compare-and-swap (CAS) instruction.

Not only for the < operator, BinDex can achieve similar performance for operators including >, ≥, and ≤ on the same ascendingly sorted virtual value space. In the following we describe how BinDex handles predicates with other operators.

"$x > c$" : With the values in the virtual value space sorted in ascending order, the general idea to perform the > operator is to conduct a bit-wise NOT operation on the result bit vector of the predicate $x \leq c$. As the operation would result in large number of memory accesses, we perform the bit-wise NOT operation when duplicating the candidate filter bit vector (*bit_vector_copy*). This significantly mitigates the overhead, leading to a similar performance with the < operator.

"$x \leq c$" and **"$x \geq c$"** : A predicate in the form of $x \leq c$ is transformed to the predicate $x < (c + 1)$. As the values are compressed into order-preserving codes in modern database systems, the two predicates have the same result. Same with $x \leq c$, the predicate $x \geq c$ is transformed to $x > (c - 1)$.

"$c_1 < x < c_2$" : We treat the predicate as two predicates $x < c_2$ and $x > c_1$, where a bit-wise AND operation is performed on their result bit vectors. Instead of naively performing the procedure, the draft result bit vector is only generated once. We record the filter bit vector for $x < c_1$ and $x < c_2$ as F_{c_1} and F_{c_2}, respectively. For each word in the bit vectors, we first perform bitwise NOT on F_{c_1} and then perform bitwise AND between the result and F_{c_2}. The scheme writes the approximate bit vector only once, which enhances the performance of the BETWEEN operator.

"$x = c$" : Considering the distribution of column data may be highly skewed, in order to balance the execution time for different selectivities, we first calculate the number of matched values with the area map and the position array (denoted as n_m). If $n_m < \frac{N}{K}$, we create an N-bit draft bit vector with all values unset as 0, then we search over the virtual value space to get the rowIDs of the column data that equals to c, and set the corresponding bits in the bit vector as 1. If $n_m \geq \frac{N}{K}$, a bit-wise XOR is performed on the bit vectors of predicates $x < c$ and $x < (c + 1)$ to generate the result. The predicate "$x \neq c$" is processed in the opposite way.

4.4 Index Building

Initial Index Building To build BinDex, the virtual value space is firstly generated by sorting the base data in ascending order. Please note that the sorted data is removed after the building procedure. The original rowIDs of the sorted values are stored in the position array. Then the virtual value space is evenly partitioned so that each area contains the same number of values. For the area map S, $S_i.value$ is the first value of A_{i+1}, and $S_i.count$ is the number of values contained from A_1 to A_i. With $K-1$ elements in the area map, $K-1$ filter bit vectors are built. For the kth filter bit vector F_k, the value of the ith bit is set as 1 if $x_i < S_k.value$, or unset as 0 for otherwise.

Index Compression BinDex adopts various approaches to compress its memory space. Besides generating all-0 and all-1 bit vectors at run time, BinDex compresses bit vectors for specific workloads. For workloads with skewed data distribution, a large amount of values in the column are the same, which brings the opportunity for bitmap compression. For instance, as shown in Figure 4, 4 appears 11 times and spans three virtual areas. When evaluating the predicate $x < 4$, F_1 is used as the candidate result, while an all-1 bit vector is used with $x < n$ $(n > 4)$. In fact, F_2 and F_3 will never be used in predicate evaluation, which are redundant.

We propose a scheme to compress BinDex: If a value c spans A_i and A_{i+1} (i>0), and the number of c is more than half the number of values in both the areas, F_i and S_i are removed. Based on the scheme, F_2 and F_3 are removed, and the area map only contains S_1. The compression scheme can save a significant amount of memory while still constraining the number of probes in the refine layer to be within $\frac{N}{4K}$.

Index Updating In analytical databases, data updating is not frequent and generally in a bulk appending manner. To avoid frequently rebuilding the index when appending new data, the position array of BinDex can be designed as a set of loosely coupled memory blocks to store rowIDs. When new data is loaded after the initial index building, the data is appended to columns. Correspondingly, their bits are appended to the filter bit vectors, and their rowIDs are inserted into the memory blocks in the position array to alleviate the updating overhead. When the relative standard deviation (RSD) of the virtual area size reaches a specified threshold, which means the value distribution in virtual areas is imbalance, the index is rebuilt. The threshold is an adjustable number based on performance requirements in database systems, which is used to tradeoff between the worst-case query performance and index rebuilding cost.

4.5 System Integration

For integrating BinDex into existing systems, we note a fact that modern column-oriented databases generally adopt late materialization in query processing, where a system creates intermediate results and postpones retrieving tuples. If the intermediate results of the select operator are stored in the form of a bitmap, BinDex can be directly integrated by substituting the original select operator. If the intermediate results in the DBMS are stored in other forms such as a rowID list, we may replace the rowID list with our bitmap and implement a new fetch operator. The new fetch operator reads the bitmap generated by select operator and retrieves column value so that other operators can access.

In using BinDex, access path selection for the select operator may be disabled. Indeed, in Section 6 we show that BinDex with 32 or more virtual areas is always faster than all existing sequential scan and index scan approaches. BinDex can be directly used for columns with integer values. For columns with other types such as floating point or string, values should be encoded into integer codes with the corresponding techniques [2, 11], which is a common practice in modern database systems. For columns with encoded integer values, BinDex makes no change to the base data layout, thus all other operators can remain unchanged.

Table 2: Notations in the performance model

Dataset	N	number of values
	cw	rowID width (bits)
Hardware	C_M	time of LLC miss (sec)
	C_H	time of LLC hit (sec)
	P	number of processor cores
	C_R	time of a random write with prefetching (sec)
	B_r	data reading bandwidth(Gb/s)
	B_w	data writing bandwidth(Gb/s)
BinDex	K	number of virtual areas
	L	number of rowIDs need to be read

5 PERFORMANCE MODELING

In BinDex, a higher number of filter bit vectors / virtual areas leads to higher performance and larger memory space. In production systems with memory constraints or specific performance requirements, tradeoffs should be made between the memory space and performance. We develop a performance model to help users achieve the goal. In the model, the scan process with BinDex is divided into four phases. Table 2 shows the parameters and notations used in the model.

Search on the area map In area search, binary search is performed on the area map to locate the virtual area. Since the area map is frequently used whose size is small (less than 1 KB), we take it as residing in the last level cache, and we calculate the search time as $T_A = \log_2 K \cdot C_H$.

Copy filter bit vector The copying process involves sequential reads to the original filter bit vector and sequential

writes to the allocated result bit vector. Since the operation continually saturates memory bandwidth, we calculate the time of copying the filter bit vector as $T_C = N/B_r + N/B_w$.

Search on the position array After a virtual area A_k is chosen, the portion of the position array that belongs to A_k is searched to locate the range of values to be probed. Since the size of the position array is large, we take the binary search on the position array and the reference to the base data as random memory accesses. The time of searching on the position array is calculated as $T_P = \log_2(N/K) \cdot C_M$.

Refine result bit vector The time of refining result bit vector depends on the number of virtual areas. We model this phase as two parts. The first part is reading the rowIDs in the position array, and the second part is randomly writing to the corresponding positions in the result bit vector. BinDex cannot saturate the memory bandwidth in the refine phase. With multiple cores performing the operation in parallel, the execution time is calculated as: $T_R = L \cdot C_H/P + L \cdot C_R/P$.

The overall scan time with BinDex is calculated as follows.

$$T(N,K) = T_A + T_P + T_C + T_R$$
$$= \log_2 K \cdot C_H + \log_2 \frac{N}{K} \cdot C_M + \frac{N}{B_r} + \frac{N}{B_w} + (C_H + C_R) \cdot L/P$$

When L equals to $N/4K$, the equation calculates the average performance of BinDex, since on average 1/4 of the values in a vritual area are probed. $T(N,K)$ estimates the worst case performance when L equals to $N/2K$, i.e., half of the values in a virtual area need to be accessed. In a production system with a given memory space constraint M, the maximum number of virtual areas without index compression is calculated as $k = M/N - cw + 1$. As BinDex adopts a compression scheme, we use equi-depth histograms to sample and estimate the number of bit vectors that can be compressed (denoted as k'). Then the performance of BinDex with memory constraints M can be estimated by substituting $K = k + k'$ into $T(N,K)$.

6 EXPERIMENTAL ANALYSIS

In this section, we evaluate the performance of BinDex under a variety of workloads and configurations.

6.1 Experimental Setup

Hardware and Software We run experiments on a machine equipped with an Intel Xeon E5-2695 v4 Broadwell processor running at 2.1GHz. The processor has 18 cores and a 45MB L3 cache. The server is equipped with a 1TB disk and 128 GB DDR4 DRAM. The operating system is 64-bit Ubuntu Server 18.04 with Linux Kernel version 4.15.0-54. The programs are compiled using g++ 6.5 with optimization flag -O3. The needed data are preloaded into memory before experimental runs. All experiments are performed with the data residing in the main memory, where the needed data are preloaded before experiments.

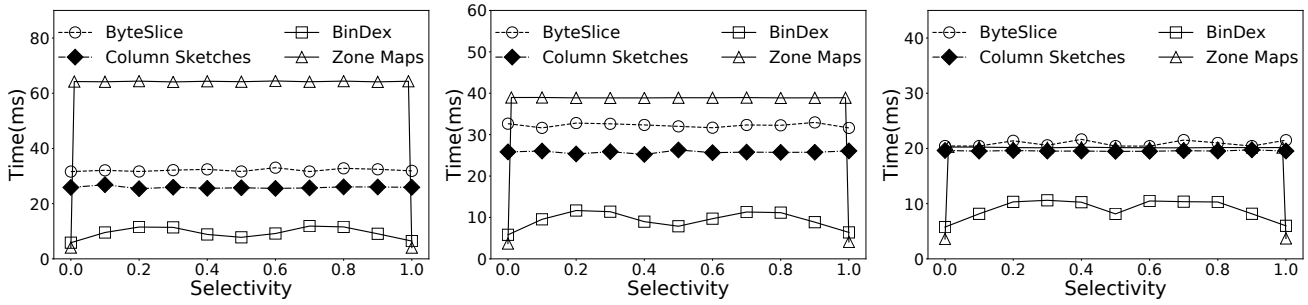
Benchmark In the experiments, we create a table with one billion values in each column. Both uniform and skewed workloads are used in the evaluation. For the uniform workload, values are uniformly distributed integers between $[0, 2^k)$, where k is the code width ranging from 4 bits to 32 bits. The popularity of the values in the skewed workload follows a Zipf distribution of skewness 1 and 2. We control the selectivities of the predicates in the experiments by varying the predicate constant c.

The number of virtual areas in BinDex is set as 128 by default, and the performance with different number of virtual areas is reported in Section 6.7. When conducting predicate evaluation, each approach makes use of all the 18 cores. Except for the experiments that vary the query selectivities, the numbers reported in the experiments are the average performance under 100 randomly chosen selectivities. For each selectivity, the performance are the average results with 20 experimental runs.

Competitors We compare BinDex against ByteSlice, Zone Maps, Column Sketches, and B$^+$-tree. Besides the open source ByteSlice project, we write the code for other baselines according to the original paper and try to optimize the performance by adopting all the optimizations mentioned. For Column Sketches, we follow the experimental settings and use single byte column sketch. The B$^+$-tree has a fanout of 200 and adopts prefetching in the implementation. All the approaches (including B$^+$-tree) output a bit vector as the scan result. The experimental results of these approaches are consistent with their original papers.

6.2 Performance with Uniform Data Distribution

In this subsection, we evaluate the performance of the $<$ operator on uniformly distributed numerical data. Figure 5 compares the performance of BinDex with other approaches by varying the selectivity from 0 to 1 with different code widths. For all workloads, BinDex significantly outperforms all the competitor approaches. For instance, for 32-bit code width BinDex achieves 2.1×, 2.9× and 5.6× higher performance than Column Sketches, Byteslice and Zone Maps, respectively. Column Sketches outperform other sequential scan approaches for 16-bit and 32-bit code widths. For the 8-bit code width, Column Sketches exhibit no performance advantage as at least a one-byte sketch should be used, thus the amount of accessed data is not reduced. For very low and high selectivities, Zone Maps are effective in accelerating scan operations. This is because most of the values satisfy or do not satisfy the predicate, and Zone Maps are able to skip



(a) Code width = 32 bits (b) Code width = 16 bits (c) Code width = 8 bits

Figure 5: Scan performance on datasets with uniform distributions

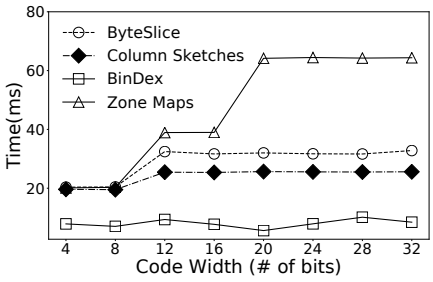


Figure 6: Performance by varying code width from 4 to 32 bits

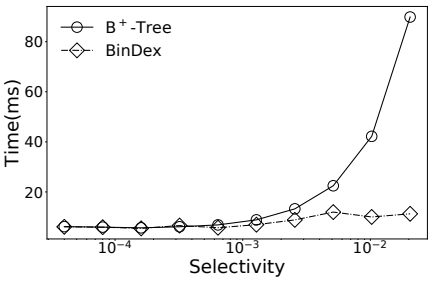


Figure 7: Performance comparison with B$^+$-tree

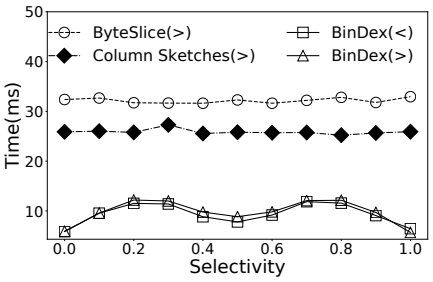


Figure 8: Performance of the > operator

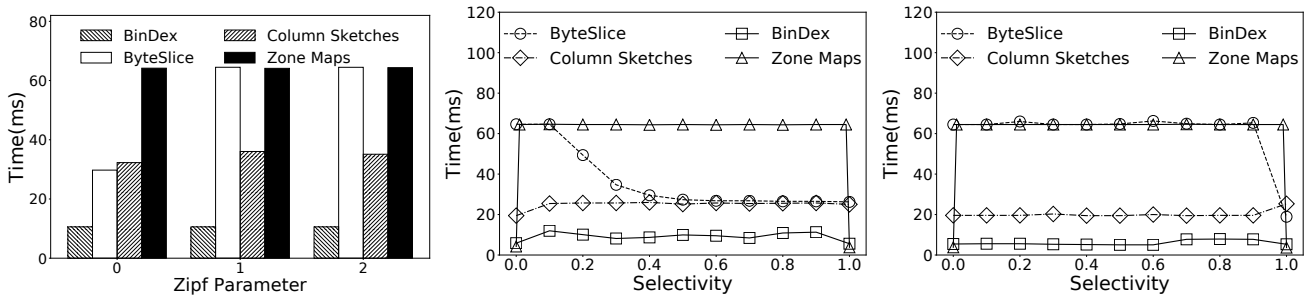
large number of zones by only comparing their min/max values. However, Zone Maps become inefficient for most cases as zones are hard to skip under the uniform data distribution.

Figure 6 compares the average performance of the approaches by varying the code width from 4 bits to 32 bits. ByteSlice and Column Sketches show the same performance with the code widths of less than or equal to 8 bits, because these approaches can not reduce data accesses with small code widths. Instead, BinDex is the only approach that can accelerate scans for small code widths, which delivers up to 1.8 times higher performance than other approaches. With code widths larger than 8 bits, the execution time of Zone Maps increases since the data needs to be accessed is proportional to the code width. Column Sketches deliver consistent performance with varying code widths, which is around 2.2 times lower than BinDex.

B$^+$-tree outperforms sequential scan approaches for predicates with extremely low selectivity (< 1%). In Figure 7, we compare the performance of B$^+$-tree and BinDex. As shown in the figure, BinDex achieves similar performance with B$^+$-tree under low selectivity (0.001% - 0.4%). This is because both approaches initialize an all-zero bit vector, while the number of accessed rowIDs and flipped bits are the same. When selectivity increases (> 0.4%), the filter bit vectors in BinDex take effect which largely reduces the number of rowIDs to be accessed, therefore leading to a huge performance improvement.

6.3 Performance with Skewed Data Distributions

In this subsection, we evaluate the performance of the approaches with skewed data distributions. With the skew factor varying from zipf = 0 (uniform distribution) to zipf = 2 (heavily skewed distribution), Figure 9a shows the execution time of predicates with the < operator. As shown in the figure, BinDex consistently outperforms all the other approaches under the skewed data distributions. On average, BinDex delivers 1.7 - 6.5 times speed up than the other approaches for the skewed data distributions with zipf=1, and 2.3 - 10.2 times speed up with zipf=2. When the skew factor increases, the execution time of ByteSlice increases correspondingly. This is because for the Zipf distribution, increasing the skew factor has generated large number of same values. For most of the queries, the constant c lies in the dense region of the zipfian curve. As a result, the most significant bits in a segment would be mostly the same while the least significant bits need to be accessed to know if the values satisfy the predicate or not. Hence, early stopping is less effective with skewed data distribution, which is consistent with the experimental results reported in previous



(a) Performance comparison by varying skewness (b) Performance with skewed data distribution (zipf = 1) (c) Performance with skewed data distribution (zipf = 2)

Figure 9: Scan performance on datasets with Zipf distributions

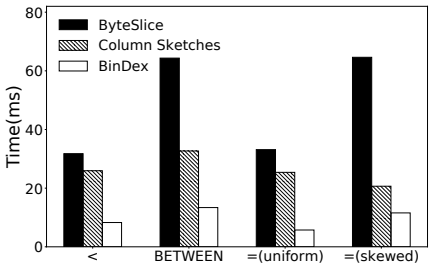


Figure 10: Performance comparison of different operators

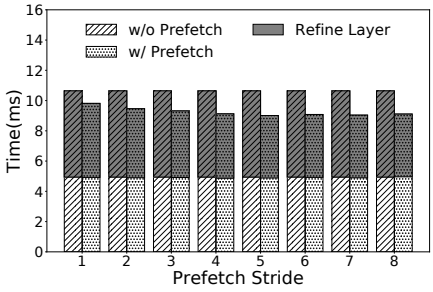


Figure 11: BinDex performance with different prefetch strides

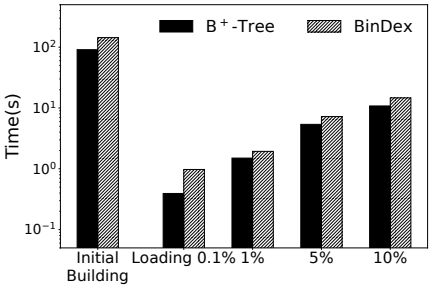


Figure 12: The time of index building and updating

studies [12]. On the contrary, the execution time of Column Sketches and BinDex decreases. This is because the values can be efficiently filtered as lots of them are the same, and the number of probes are significantly reduced, leading to higher performance.

Figure 9b and Figure 9c compare the performance of the approaches with skewed workloads (zipf=1 and zipf=2), respectively. The early stopping technique, i.e., ByteSlice, behaves differently with the two skew factors. With skew factor zipf=1, the performance of ByteSlice improves with the increase of selectivity. This is because, the query constant lies in the sparse region of the zipfian curve for high selectivities, which is larger than most values in the column. Therefore, early stopping works as most values can be known if they satisfy the predicate by only accessing the most significant bits. Figure 9c shows that ByteSlice works poorly in heavily skewed data distribution, because the query constant locates in the dense area of the Zipfian curve for selectivities less than 0.9. Column Sketches obtain slight performance improvement for skewed data as frequent values are mapped to unique sketch codes, therefore reducing the amount of probes. To conclude, skewed workloads have little impact on the performance of BinDex.

6.4 Other Operators

Figure 8 shows the performance of BinDex, Column Sketches and ByteSlice on predicates with the > operator under uniform workloads. All the approaches achieve similar performance for the < and > operators. For sequential scan approaches such as Column Sketches and ByteSlice, the < and > operators make no difference as all values in the column need to be accessed and evaluated, leading to the same costs for both computation and memory accesses. For BinDex's virtual value space where values are sorted in ascending order, the filter bit vectors need to be bitwise reversed to serve as the approximate results for the > operator. We perform bitwise NOT when copying the filter bit vectors which avoids introducing extra memory accessing overhead. As shown in the figure, the performance of the > operator is almost the same with that of the < operator in BinDex.

Figure 10 compares the efficiency of BETWEEN and = operators to the < operator. For the BETWEEN operator, evaluations are conducted under uniform workloads. The number of rowIDs accessed in the refine layer for BETWEEN operator is twice that of < operator. However, as introduced in Section 4.3, we merge the process of bit vector copying of the two predicates into one, which significantly alleviates the memory access costs. With this optimization, the execution

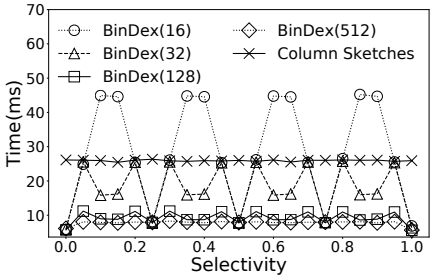


Figure 13: The performance of BinDex with different number of virtual areas

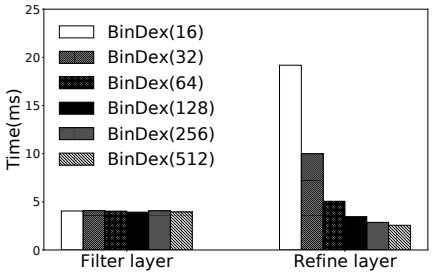


Figure 14: The execution time of filter layer and refine layer with varying number of virtual areas

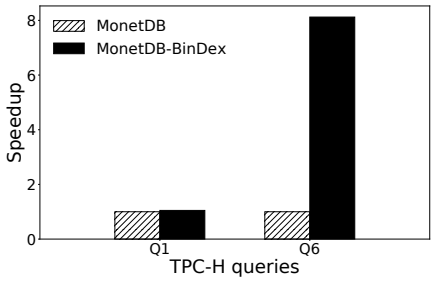


Figure 15: The performance improvement of MonetDB with BinDex on TPC-H benchmark

time of the filter layer only increases by 41%, and the total execution time of the BETWEEN operator is 61% higher than that of the < operator, which is still 3.8× and 1.5× faster than that of ByteSlice and Column Sketches, respectively.

For the "=" operation, as described in Section 4.3, BinDex adopts two different approaches under low and high selectivities. We use uniform (selectivity $\ll 0.001$) and skewed (zipf=1, selectivity>$\frac{1}{K}$) data distributions to evaluate the performance of the two approaches. For the uniform distribution, the number of bits to be flipped in the refine layer is very small, and the performance of BinDex is 3.5× and 4.9× higher than that of Column Sketches and ByteSlice, respectively. For such predicates, initializing the all-zero result bit vector is the main overhead in BinDex. When the selectivity is larger than $\frac{1}{K}$ with the skewed distribution, the performance of processing a "=" operation is similar to that of processing a "BETWEEN" operation, since the result of predicate "$c-1<x<c+1$" is equivalent to that of "$x=c$". For such workloads, BinDex achieves 0.75× and 5.3× speedup over Column Sketches and ByteSlice, respectively.

6.5 Performance Impact by Prefetching

BinDex mitigates the random accesses in the refine layer with prefetching. BinDex prefetches bits that are h iterations ahead in the loop, where h is called as prefetch stride. The setting of the prefetch stride considers both the cache miss penalty and the execution time of each iteration. Figure 11 shows the performance improvement by prefetching of BinDex(128) on uniform 32-bit codes. As shown in the figure, prefetching improves the performance of the refine layer and BinDex for 14.1-29.9% and 7.6-15.5%, respectively. By varying the prefetch stride from 1 to 8, the performance improvement increases with a larger prefetch stride and stays relatively stable when exceeds a certain threshold (6 in our experiments). We find that setting the prefetch stride at 6 works for all workloads on our platform, and we use the setting in our evaluations. It can be optimized on other targeted hardware platforms via the same experiments.

In the experiment, the filter layer takes 52% of the overall processing time, but it generates the correct results for around 99.8% values. Instead, the refine layer only flips the bits for 0.2% values but takes 48% of the time. Therefore, the filter layer is the main factor in enhancing the overall performance, while prefetching mitigates the random access overhead in the refine layer.

6.6 Index Building and Incremental Data Updating

In Figure 12, we evaluate the index building time and the incremental index updating time when new data is loaded. For a column with one billion values, the initial index building takes 143 seconds, which is 36.3% slower than a B$^+$-tree.

When new data is loaded, BinDex supports incremental updating to avoid rebuilding the index from scratch. With the initial index built for one billion values, we measure the time of updating the index by appending 0.1%, 1%, 5% and 10% of the values in the base data and compare the efficiency with B$^+$-tree. As shown in Figure 12, the average updating efficiency of BinDex is 33.4% lower than that of B$^+$-tree. The reason mainly lies in the filter layer, where initializing or expanding the bitmaps introduces non-trivial overheads.

6.7 Performance Impact by Number of Virtual Areas

The number of virtual areas in BinDex is the main factor that influences the performance, since it determines the number of probes in the refine layer. Figure 13 demonstrates the performance of BinDex with 16-512 virtual areas under different selectivities. BinDex(K) in the figure denotes the performance of BinDex with K virtual areas. As demonstrated in the figure, the performance of BinDex varies in a large range with 16-32 virtual areas. This is because predicates with different selectivities lead to a different number of probes in the refine layer, resulting in the performance variation. For instance, for 16 virtual areas on a column with N values, each virtual area contains $6.25\%N$ values. The filter bit vector F_2

is chosen with a predicate of 10% selectivity, because 3.75%N (10% − 6.25%) values need to be probed if F_1 is chosen, while F_2 only needs to probe 2.5%N (6.25% × 2 − 10%) values. When the number of virtual areas increases to 32, each virtual area contains 3.125%N values. Therefore, the number of the values to be probed is only 0.625%N (10% − 3 × 3.125%) with F_3 selected. Since the number of probed values dramatically drops from 2.5%N to 0.625%N, the execution time of BinDex decreases from 44.9 ms to only 15.8 ms. The performance of Column Sketches is also included in Figure 13, which is lower than BinDex(32) in all cases, and the average performance of BinDex exceeds Column Sketches with 16 virtual areas. With 512 or more virtual areas, the performance of BinDex becomes relatively stable with different selectivities. BinDex(512) achieves around 2.9× higher performance than Column Sketches.

Figure 14 shows the execution time of the refine layer and the filter layer with different number of virtual areas. When the number of virtual areas increases from 16 to 512, the time of the filter layer hardly changes as the execution time of its main task stays constant, i.e., copying the filter bit vector. However, the execution time of the refine layer dramatically drops with the increase of the number of virtual areas.

6.8 TPC-H Evaluation

We integrate BinDex into MonetDB and evaluate its performance improvement on TPC-H benchmark. To integrate BinDex into MonetDB, we implement a new select operator that takes a single column and a predicate as inputs, which makes the API stay the same with the original select operator. MonetDB uses a rowID list as its intermediate scan result, but BinDex outputs a bitmap. For compatibility, we implement a new fetch operator that works with bitmap, and the rest of the operators in MonetDB are left unchanged. We use the explain command to create plans and edit the plans to use BinDex. Then we feed the revised plan into the MAL interface of MonetDB to perform the query processing.

Figure 15 compares the performance improvement brought by BinDex. Comparing with the original MonetDB, MonetDB with BinDex achieves 4.9% and 7.1× performance improvement for Q1 and Q6 in the TPC-H benchmark, respectively. Because the majority of the query execution time of Q1 is spent in performing aggregation, its performance improvement is not apparent. For Q6, in contrast, much more time is spent on performing scan, thus there comes a much higher performance improvement.

Currently, the performance is evaluated with one thread, and we leave the parallelization of BinDex in MonetDB as a future work. To parallelize BinDex in MonetDB, we need to revise the query plan generated by the MonetDB optimizer. The default optimizer in MonetDB uses a “mitosis" way to parallelize query execution, where it horizontally slices a column and replicates the query plan per slice. Multiple threads evaluate replicated plan fragments concurrently, and MonetDB merges their intermediate results in the end. However, as stated in Section 4.3, the best way to parallelize BinDex is to have multiple threads cooperatively copying different parts of the selected filter bit vector and refining in the position array. Therefore, to integrate the process into MonetDB, the select operator needs to be parallelized separately, and the bit vector generated by BinDex should be sliced and distributed among the threads as their inputs. Correspondingly, the query plan generated by the optimizer should be revised to support the execution flow.

7 PERFORMANCE-SPACE TRADEOFFS

Different with other indexes such as B^+-tree, BinDex is able to tradeoff between performance and memory space. We introduce memory space as the main dimension in the selection of scan approaches. According to the additional memory space needed, we classify the scan techniques as three categories: (1) Early pruning on bit-level storage layouts [12, 19] that demands no additional spaces; (2) Column Sketches [15] that use lossy compression to build a sketch for a column; (3) BinDex that demands less memory than a B^+-tree but larger memory than other approaches.

We compare the memory space taken by the approaches with 1 billion values of 32-bit codes. For early pruning techniques on bit-level storage such as Bitweaving and ByteSlice, they rearrange the storage layouts and require no extra memory space. Since Column Sketches mainly use 8-bit or 16-bit sketched columns [15], thus the extra space for the sketched column is 1-2 GB. Therefore, early pruning technique is the fastest approach when the memory is less than 1 GB, while Column Sketches should be chosen with more memory. For 32-bit rowIDs, a B^+-tree takes around 8.7 GB extra memory space for 1 billion values. The memory space taken by BinDex(32) is around 8.0 GB, which is smaller that a B^+-tree. Installing 32 virtual areas is enough for BinDex to outperform other approaches under all selectivities, and the average performance of BinDex(32) is 24.7% higher than that of Column Sketches. With 128 virtual areas, BinDex achieves up to 2.5 times higher performance than Column Sketches, and the memory taken is around 20 GB. For skewed workloads, BinDex can further compress the memory space it takes. The memory of BinDex with 128 virtual areas is 19.6 GB and 5.9 GB for workloads with zipf=1 and zipf=2, respectively.

Based on the memory consumption of each approach and the system memory constraints, database systems can choose the appropriate scan approach.The performance model in Sec. 5 can be used to estimate the performance of BinDex, which shows an error rate of 12.3% in our evaluation. With

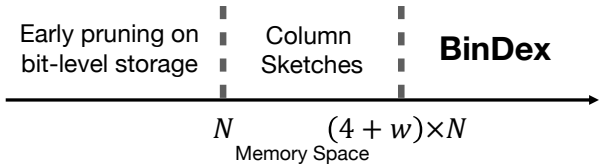


Figure 16: The selection of scan approaches with memory constraints

the estimated performance of other approaches, database systems can choose the appropriate select operator given a specified memory budget. Since sequential scan approaches generally derive cycle/code instead of execution time in their performance models, we leave it as a future work to utilize performance models to choose scan approaches.

According to our evaluations on several x86-64 servers, Figure 16 shows the general conceptual graph of the guidance on selecting scan approaches according to available memory space on commodity off-the-shelf (COTS) x86-64 platforms. The following equations formally describe the figure, where the memory budget is denoted as M in bytes.

$$\begin{cases} BinDex, & \text{if } M \geq (4+w) \cdot N \\ Column\ Sketches, & \text{if } N \leq M < (4+w) \cdot N \\ Bitweaving/ByteSlice & \text{if } M < N \end{cases} \quad (3)$$

In the equations, N denotes the number of values and w denotes the rowID width in bytes. We use $4 \cdot N$ bytes as the basic memory space required for filter bit vectors, since 32 virtual areas are enough for BinDex to outperform competitors on our platforms under various selectivities. To conclude, database systems can use memory space as the main dimension for selecting the best scan approaches in system setup.

8 RELATED WORK

Sequential Scan: Techniques including compression [1, 3, 31], SIMD [22, 35, 36], and scan sharing [6, 13, 14, 23, 24, 37] have been studied to accelerate the sequential scans. The early pruning techniques, together with lightweight indexes [25, 29, 32, 33] and Column Sketches [15], are all specialized sequential scans that reduce the amount of data to be accessed, in different ways. The nature of sequential scans requires to access all the column codes, even not all bits of each code. Instead, BinDex trades memory for performance and uses filter vectors to avoid accessing the entire column.

Bitmap indexes: Bitmap indexes work well for categorial data with low cardinality, where each distinct value is mapped to a bitmap with all the corresponding bits being set as 1 [34]. Such bitmap indexes perform poorly for high-cardinality predicates because of the huge amount of bitmaps that need to be stored and the high overhead of merging bitmaps in query execution. Range encoding can overcome the inefficiencies of traditional bitmap to accelerate the range scan operations [7, 8, 21]. Based on range encoding, binning further optimizes the scan performance on columns with high cardinality, where multiple values are covered in a bin/bitmap to reduce the number of bitmaps [26, 27, 30]. Previous bitmap approaches with binning is similar with the filter layer of BinDex. However, they do not have the refine layer and have to probe the base data, which involves expensive random memory accesses. Instead, the virtual value space adopted by BinDex partitions sorted data into bins, based on which the position array dramatically mitigates the probing overheads with sequential reads and prefetching.

Access Path Selection: [17] studies the access path selection in columnar memory databases, demonstrating that sequential scans perform better in most cases while index scans outperform sequential scans on predicates with low selectivity. [17] also shows that concurrency is an important factor that should be taken into account in access path selection. We plan to evaluate BinDex with high query concurrency and compare it with sequential scans in the future. To address the inaccuracy and inefficiency of cost model estimation in access path selection, Smooth Scan [5] adaptively morphs its behavior between index scan and sequential scan according to selectivity. Since BinDex can outperform all the state-of-the-art scan approaches for in-memory column stores, access path selection is no longer needed when the required memory space for BinDex is available in the system.

9 CONCLUSION

We presented BinDex, a two-layered index that significantly enhances scan performance for in-memory column stores by incorporating the main merits from the existing index scan and sequential scan approaches. Our evaluation results showed that BinDex can achieve up to 2.9× higher performance than state-of-the-art scan approaches, regardless of data distribution and query selectivity. The robust and consistently-high performance delivered by BinDex eliminates the need for access path selection in query executions, and we proposed a policy that uses the memory space as the main dimension for selecting the optimal scan approach in in-memory database systems. For future work, we plan to integrate BinDex to in-disk database systems.

10 ACKNOWLEDGEMENT

We would like to thank anonymous reviewers of SIGMOD '20 for their insightful comments and suggestions. This work was supported in part by National Key R&D Program of China (Grant No. 2018YFB1004404, 2018YFB1402602), NSFC (Grant No. 61802066, 61572136), and Shanghai Sailing Program 18YF1401300.

REFERENCES

[1] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating Compression and Execution in Column-oriented Database Systems. In *SIGMOD*. 671–682.

[2] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. 2009. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*. ACM, 283–296.

[3] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. 2009. Dictionary-based Order-preserving String Compression for Main Memory Column Stores. In *SIGMOD*. 283–296.

[4] Peter A Boncz, Stefan Manegold, Martin L Kersten, et al. 1999. Database architecture optimized for the new bottleneck: Memory access. In *Proc. VLDB Endow.*, Vol. 99. 54–65.

[5] Renata Borovica-Gajic, Stratos Idreos, Anastasia Ailamaki, Marcin Zukowski, and Campbell Fraser. 2018. Smooth Scan: robust access path selection without cardinality estimation. *The VLDB Journal* 27, 4 (2018), 521–545.

[6] George Candea, Neoklis Polyzotis, and Radek Vingralek. 2011. Predictable Performance and High Query Concurrency for Data Analytics. *The VLDB Journal* 20, 2 (2011), 227–248.

[7] Chee-Yong Chan and Yannis E. Ioannidis. 1998. Bitmap Index Design and Evaluation. In *SIGMOD*. 355–366.

[8] Chee-Yong Chan and Yannis E. Ioannidis. 1999. An Efficient Bitmap Encoding Scheme for Selection Queries. In *SIGMOD*. 215–226.

[9] S. Christodoulakis. 1984. Implications of Certain Assumptions in Database Performance Evauation. *ACM Transaction on Database Systems* 9, 2 (June 1984).

[10] Douglas Comer. 1979. Ubiquitous B-Tree. *Comput. Surveys* 11 (1979), 121–137. Issue 2.

[11] Wenbin Fang, Bingsheng He, and Qiong Luo. 2010. Database compression on graphics processors. *Proc. VLDB Endow.* 3, 1-2 (2010), 670–680.

[12] 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 *SIGMOD*. ACM, 31–46.

[13] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2012. SharedDB: Killing One Thousand Queries with One Stone. *Proc. VLDB Endow.* 5, 6 (2012), 526–537.

[14] Georgios Giannikis, Darko Makreshanski, Gustavo Alonso, and Donald Kossmann. 2014. Shared Workload Optimization. *Proc. VLDB Endow.* 7, 6 (2014), 429–440.

[15] Brian Hentschel, Michael S Kester, and Stratos Idreos. 2018. Column Sketches: A Scan Accelerator for Rapid and Robust Predicate Evaluation. In *SIGMOD*. ACM, 857–872.

[16] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. 2010. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Transactions on Parallel & Distributed Systems* 1 (2010), 105–118.

[17] Michael S. Kester, Manos Athanassoulis, and Stratos Idreos. 2017. Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe?. In *SIGMOD*. 715–730.

[18] Jens Krueger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. 2011. Fast Updates on Read-optimized Databases Using Multi-core CPUs. In *Proc. VLDB Endow.*, Vol. 5. 61–72.

[19] Yinan Li and Jignesh M Patel. 2013. BitWeaving: fast scans for main memory data processing. In *SIGMOD*. ACM, 289–300.

[20] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proc. VLDB Endow.* 476–487.

[21] Patrick O'Neil and Dallan Quass. 1997. Improved Query Performance with Variant Indexes. In *SIGMOD*. 38–49.

[22] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *SIGMOD*. 1493–1508.

[23] Iraklis Psaroudakis, Manos Athanassoulis, and Anastasia Ailamaki. 2013. Sharing Data and Work Across Concurrent Analytical Queries. *Proc. VLDB Endow.* 6, 9 (2013), 637–648.

[24] Lin Qiao, Vijayshankar Raman, Frederick Reiss, Peter J. Haas, and Guy M. Lohman. 2008. Main-memory Scan Sharing for Multi-core CPUs. *Proc. VLDB Endow.* 1, 1 (2008), 610–621.

[25] Wilson Qin and Stratos Idreos. 2016. Adaptive data skipping in main-memory systems. In *SIGMOD*. ACM, 2255–2256.

[26] Doron Rotem, Kurt Stockinger, and Kesheng Wu. 2005. Optimizing Candidate Check Costs for Bitmap Indices. In *CIKM*. 648–655.

[27] D. Rotem, K. Stockinger, and Kesheng Wu. 2006. Minimizing I/O Costs of Multi-Dimensional Queries with Bitmap Indices. In *SSDBM*. 33–44.

[28] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *SIGMOD*. 23–34.

[29] Lefteris Sidirourgos and Martin Kersten. 2013. Column imprints: a secondary index structure. In *SIGMOD*. ACM, 893–904.

[30] Kurt Stockinger, Kesheng Wu, and Arie Shoshani. 2004. Evaluation Strategies for Bitmap Indices with Binning. In *Database and Expert Systems Applications*. 120–129.

[31] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-store: A Column-oriented DBMS. In *Proc. VLDB Endow.* 553–564.

[32] Liwen Sun, Michael J Franklin, Sanjay Krishnan, and Reynold S Xin. 2014. Fine-grained partitioning for aggressive data skipping. In *SIGMOD*. ACM, 1115–1126.

[33] Liwen Sun, Michael J Franklin, Jiannan Wang, and Eugene Wu. 2016. Skipping-oriented partitioning for columnar layouts. *Proc. VLDB Endow.* 10, 4 (2016), 421–432.

[34] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2017. An Experimental Study of Bitmap Compression vs. Inverted List Compression. In *SIGMOD*. 993–1008.

[35] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-scan: Ultra Fast In-memory Table Scan Using On-chip Vector Processing Units. *Proc. VLDB Endow.* 2, 1 (2009), 385–394.

[36] Jingren Zhou and Kenneth A. Ross. 2002. Implementing Database Operations Using SIMD Instructions. In *SIGMOD*. 145–156.

[37] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. 2007. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *Proc. VLDB Endow.* 723–734.