Breaking the Curse of Cardinality on Bitmap Indexes[⋆]

Kesheng Wu, Kurt Stockinger, and Arie Shoshani

Lawrence Berkeley National Lab, University of California, Berkeley, CA, USA
http://sdm.lbl.gov/

Abstract. Bitmap indexes are known to be efficient for ad-hoc range queries that are common in data warehousing and scientific applications. However, they suffer from the curse of cardinality, that is, their efficiency deteriorates as attribute cardinalities increase. A number of strategies have been proposed, but none of them addresses the problem adequately. In this paper, we propose a novel binned bitmap index that greatly reduces the cost to answer queries, and therefore breaks the curse of cardinality. The key idea is to augment the binned index with an Order-preserving Bin-based Clustering (OrBiC) structure. This data structure significantly reduces the I/O operations needed to resolve records that can not be resolved with the bitmaps. To further improve the proposed index structure, we also present a strategy to create single-valued bins for frequent values. This strategy reduces index sizes and improves query processing speed. Overall, the binned indexes with OrBiC great improves the query processing speed, and are 3 – 25 times faster than the best available indexes for high-cardinality data.

1 Introduction

A large data warehouse typically contains high-dimensional data with tens or even hundreds of attributes. Most popular indexing techniques are not effective for answering queries on these datasets; some use the term *the curse of dimensionality* to describe the poor performance [1]. The bitmap index is able to break this curse even when the dimensionality of the dataset is very high [2–4]. Hence, major commercial database systems, such as ORACLE, IBM DB2, and Sybase IQ, have implemented various bitmap indexes. In many cases, bitmap indexes not only take less disk space than the commonly used B-Tree indexes and their variations, but also answer queries much faster [2, 3, 5]. However, the current bitmap indexes also have serious limitations [6]. One of the most serious ones, which we call *the curse of cardinality*, is that both index sizes and query response time increase as the number of distinct values in an attribute increases. In this paper, we propose to break this curse with a novel binned index and demonstrate its effectiveness with both analyses and experimental measurements.

The number of distinct values of an attribute in a dataset is known as the *attribute cardinality*. A number of strategies have been proposed to improve the performance of bitmap indexes on high-cardinality attributes as discussed in the next section; among them we see binning as the most promising [7–10]. Instead of building one bitmap

[⋆] This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

for each distinct value as in the basic bitmap index, a binned bitmap index builds one bitmap for a range (or bin) of values, which reduces the number of bitmaps used in the index. Typically a query range spans multiple *interior bins* and two *edge bins*. For example, an attribute "age" represented by integers between 0 and 100 might be divided into 10 bins uniformly with bin 1 for "age" between 0 and 9, bin 2 for "age" between 10 and 19, and so on. The query "age between 25 and 65" will have bins 4, 5, and 6 as interior bins, bins 3 and 7 as edge bins. The bitmaps can be used to identify rows in the interior bins and edge bins. Those in the interior bins are hits, but the ones in edge bins are only candidates and their base data has to be examined to determine whether they are hits. We call this process of examining the edge bins the *candidate check*. The candidate check is often slow and significantly diminishes the value of a binned index.

In this paper, we propose to solve this problem by augmenting the bitmap index with an Order-preserving Bin-based Clustering (OrBiC) structure. This structure clusters the values for each bin together in the same order as they appear in the bitmap for the bin. The most important reason that the candidate check takes a long time is that the values fall in a bin are scattered in data files. The OrBiC data structure stores the values of each bin together and reduces the time needed for the candidate check. Our analysis and measurements on both synthetic and application datasets show that the total size of the bitmap index with the OrBiC structure can be smaller than the size of the bitmap index without binning. Systematic timing measurements showed that our strategy significantly outperforms both unbinned bitmap indexes and conventional binned bitmap indexes without the OrBiC structure.

The second innovation in this paper is the use of a hybrid of single-valued bins and multi-valued bins. We give an algorithm for creating these single-valued bins for both integer values and floating-point values. This allows us to reduce the time for candidate checks and the size of OrBiC structures. On low-cardinality attributes, this hybrid binning strategy produces mostly single-valued bins. On high-cardinality attributes, it assigns the most frequent values to single-valued bins. It has the advantages of both binned and unbinned bitmap indexes. The combination of the OrBiC structure with hybrid-binning is especially effective for high-cardinality attributes, and results in three-fold to several dozen-fold improvement over well known indexing methods. In summary, our binned bitmap index with OrBiC essentially overcomes the curse of cardinality.

2 Background and Related Work

The strategies for improving the performance of bitmap indexes on high-cardinality attributes can be categorized into three broad categories: compression, encoding and binning. In this section, we briefly review the common strategies from each category and show how our method improves upon existing strategies.

Compression is typically used to reduce the size of each bitmap in a bitmap index. Different compression methods have been used, including general text compression and specialized bitmap compression [11]. General text compression methods are effective in reducing the sizes of bitmaps, but require a long time to decompress bitmaps to answer a query. To improve the query response time, a number of specialized compression methods have been designed, such as Byte-aligned Bitmap Code (BBC) [12] and Word-Aligned Hybrid code (WAH) [13]. In particular, the WAH compression index was shown to have the same theoretical optimality as the B+-Tree index, that is the time required to answer range queries is bounded by linear functions of the number of hits [14, 13]. However, in timing measurements compressed bitmap indexes outperform the B+-tree index significantly [15]. We use WAH compression in this work because WAH compressed indexes were found to be 12 times faster than BBC compressed ones while using about 50% more space [5].

Compression alone does not fully address the difficulty of applying bitmap indexes to high-cardinality attributes. In the extreme case where every value is distinct, the compressed bitmap index can be larger than a typical B-Tree index. For a dataset with N tuples, a WAH compressed index containing N bitmaps requires $5N$ words to store the bitmaps (details in Sec. 4.1), which is larger than the size of a typical B-Tree implementation. The size of a BBC compressed index might be smaller than that of a WAH compressed index, but both indexes would take more time to answer a query than the projection index [3]. Therefore, encoding and binning technique are used in addition to compression especially for high-cardinality attributes.

Bitmap encoding methods are applied to reduce the number of bitmaps used in a bitmap index. Among the different encoding methods, the bit-sliced index [3] (also called the binary encoding [16]) produces the least number of bitmaps. One shortcoming of this encoding method is that it needs to access nearly every bitmap to answer any query. There is a number of other encoding methods that produce more bitmaps than the binary encoding, but only have to access a small number of bitmaps to answer a query [17, 18]. Still, using encoding methods alone also does not fully address the performance issues of high-cardinality attributes. In the worst case, even the most compact binary encoding produces an index that is as large as the projection index. Furthermore, the projection index usually outperforms such a binary encoded index because the operations on the bitmap index are more complex than operations on the projection index.

For attributes with extremely high cardinality, combining compression and encoding methods does not produce indexes that are competitive with projection indexes. This is because bitmaps produced by compact encoding schemes such as the binary encoded bitmaps typically do not compress well. In this case, compression does little to reduce the index sizes, but could increase the query response time. Thus, binning may be the most promising technique for high cardinality attributes.

Binning places multiple distinct values into a single bitmap and therefore reduces the number of bitmaps required for the bitmap index [7, 8, 10]. It allows a user to control how many bitmaps to use in an index. As mentioned earlier, the disadvantage is that one needs to perform the candidate check to resolve the edge bins. Performing the candidate check usually ends up touching a majority of the disk pages storing the base data even though the number of false positives may be small. The reason is that reading data from disk is performed in pages (typically, 4 KB or 8 KB)[1] and the candidates are usually scattered throughout the base data. Therefore the time required to answer a query is usually longer than that of the projection index. Our challenge here is to reduce the time needed for candidate check, or eliminate such checks when possible.

[1] Most I/O system also performs read-ahead, which reads 128 KB or more in one operation.

There are a number of recent papers that address the issue of how to reduce the number of candidate checks [19, 20]. They optimize the placement of bin boundaries to minimize the average query response time by taking into account of data distribution and query workloads. However, their strategy do not reduce the time required for each candidate check procedure. In this work, we present an auxiliary data structure that allows us to directly reduce the candidate check time and thus the overall query processing time. This complements the existing work. In practice, because the exact query workload is usually not available before indexes are built, our approach is likely to be more effective because it does not rely on knowing the query workload.

In this work, we assume the base data is not modified or infrequently modified. Such data are common in extremely large data warehouses, where the only updates are bulk loading of a large number of new records. Similarly, most scientific applications generate or collect data records never modify their data records either [21]. For this reason, a number of research database systems such as C-Store [22] and MonetDB [23] make similar assumptions. Usually these systems can efficiently append new records, but they also implement a number of strategies to accommodate a small number of updates. For example, a special mask for deleted entries can be maintained and an update to a row can be treated as a deletion followed by an append. Using these strategies, many datasets can be treated as read-only.

3 The New Binned Index Structure

In this section, we explain the new binned index structure. The two key elements of this index are: an Order-preserving Bin-based Clustering (OrBiC) data structure and a hybrid-binning strategy that uses single-valued bins together with multi-valued bins. Before describing them, we first briefly review the basic binned bitmap index.

To build a typical binned bitmap index, one first chooses the bin boundaries. Next, each value in a bin is represented in the corresponding bitmap by setting a bit to 1. Figure 1(a) shows an illustration of a bitmap index with two bins for an attribute whose values can be between 0 and 1. Bin 0 is for values between 0 and 0.5 (not including 0.5) and bin 1 is for values between 0.5 and 1 (including 0.5). For a query requesting the rows with values greater than 0.3, all rows in bin 1 satisfy the condition – an interior bin that we need to access the bitmap; some rows in bin 0 (an edge bin) also may satisfy the query condition. We need to examine the base data for bin 1 to determine if they are actually hits. Because the base data in bin 1 is usually scatter on disk, the *candidate check* process is typically expensive. The proposed new data structure is to cluster the values according to the bin number and reduce the I/O cost.

3.1 Order-preserving Bin-based Clustering

During a candidate check, all values that fall in the edge bin must be examined. The rows for a given edge bin are known as soon as the index is built. Therefore, it is possible to reorganize the data so that the values in an edge bin are stored consecutively. We call this additional data structure for storing the reorganized data as Order-preserving Bin-based Clustering (OrBiC).

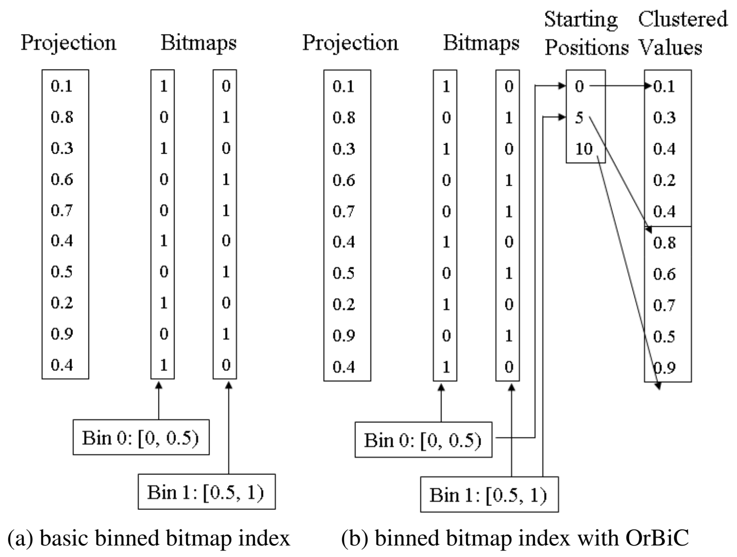


Fig. 1. An illustration of bitmap indexes with two bins.

Typically, a bitmap index is built for a single attribute of a relation (a column of a table) at a time. In this setting, the OrBiC data structure can be thought of as a projection of the attribute, reordered according to the bin numbers, and preserving the relative order of values in each bin. To use these values efficiently, we also need to record the starting and ending positions of each bin. Because we preserve the relative order of values in each bin, their relative positions are the same as those bits that are 1 in the bitmap for the bin. This allows us to use the clustered values without their row identifiers.

An illustration of a bitmap index with two bins and the OrBiC structure is shown in Figure 1(b). In addition to the clustered values, we also store starting and ending positions of every bin. Since we assume the base data is read-only, the reordered values are of course also read-only. Their starting positions are never modified. This allows us to pack the clustered values in an array. We also pack the starting and ending positions together into one short array alongside the clustered values.

To perform a candidate check, we first read the starting positions to determine which values are needed. Since the values corresponding to each bin are packed consecutively on disk, they can be read sequentially. This significantly reduces the I/O cost associated with candidate checks as shown later in this paper. Additionally, since OrBiC data structure is part of a bitmap index, it does not affect the ordering of the base data and allow different attributes to have their own bitmap indexes with OrBiC.

3.2 Single-Valued Bins

Next we consider the issue of reducing the storage requirement for the OrBiC structure. Our approach is based on the observation that in real applications the distributions of data is hardly ever uniform, but skewed, that is some of the values appear much more frequently than the rest. For example, in a typical store, the sales records contain many more sales of lower priced items than higher priced ones. Similarly, in a climate simulation, there are fewer records with very high or very low temperature values. In these cases, removing the most frequent values from OrBiC significantly reduces the number of entries stored.

To facilitate the removal of frequent values, we store the actual minimum and maximum values in each bin. If a bin has the same minimum and maximum value, i.e., representing a single value, then no candidate check is ever needed for this bin. We call these bins the *single-valued bins*.

Knowing the minimum and the maximum also helps reduce the need for candidate check. In the example used in Figure 1(b), a query to find all records with values greater than 0.4 appear to have bin 0 as the edge bin. However, since the actual maximum of bin 0 is 0.4, we do not need to perform the candidate check.

3.3 Creating Single-Valued Bins

Now we discuss the mechanics of generating the single-valued bins. We first describe the procedure to specify bin boundaries to ensure that a single-valued bin actually holds only one value, and then describe a heuristic for assigning values to single-valued bins.

As mentioned before, our binning procedure starts with a set of bin boundaries. To create single-valued bins we need to specify the bin boundaries precisely so that the intended bins actually contain only one value each. Given a set of bin boundaries $\{b_0, b_1, \ldots, b_B\}$, we define a set of bins with closed left ends and open right ends. For example, the first bin contains values satisfying the following conditions $b_0 \leq x < b_1$ and the second bin contains values satisfying $b_1 \leq x < b_2$. Given this definition, to have single-valued bin for value b_i, we need to make sure that the next bin boundary is b_{i+1}, the smallest possible value that is larger than b_i. In digital computers, all numbers are discrete and it is possible to compute b_{i+1} quickly.

For integer attributes, the smallest possible value that is larger than b_i is $b_{i+1} = b_i + 1$. To compute the same for floating-point numbers, we rely on a parameter known as the *machine epsilon* (or the unit round off error) ϵ, which is defined to be the smallest number such that $1+\epsilon > 1$ in floating-point arithmetic [24]. For a normal floating-point number b_i, we can compute b_{i+1} as $b_{i+1} = b_i(1 + \epsilon)$.

Now that we know how to specify the bin boundaries to make single-valued bins, the remaining challenge is to decide when to put a value in its own bin. In this work, we use a heuristic to produce approximate *equal-weight bins*. The overall goal is to make each bin have the same number of rows. Because there is no way to further divide a single value into multiple bins, a frequent value should be in its own bin. To make this decision, we need to know how many times each value appears in a dataset (i.e., the frequency). We may compute the exact frequencies or approximate them with sampling [25]. With the exact frequency counts, we can make more precise decisions, but it may take more time and space to collect the counts. We generate equal-weight bins by first identifying the most frequent value. If its frequency is no less than the average count for a bin, we place it in its own bin, otherwise we only have multi-valued bins. Once a single-valued bin is identified, the procedure is recursively applied to the left side and the right side of the single-valued bin. This heuristic requires the number of bins to be specified first, a topic we discuss in the next section.

4 Performance Analysis

In this section, we compute the worst case sizes and query processing costs of binned indexes with OrBiC. Our analyses use some earlier results on sizes of WAH compressed bitmaps [13]. In addition to understanding the performance characteristics of binning with OrBiC, we also use this study to explore options for deciding the number of bins to use.

4.1 Curse of Cardinality

To start with, we recall the main results about the most difficult case for compressed bitmap indexes, which is random data. We also use this opportunity to explain exactly what we mean by the curse of cardinality.

Let C denote the attribute cardinality of C and d_i denote the probability of value i in the dataset. Assuming each d_i is a constant independent of others, then the bitmaps generated for the basic bitmap index and the binned bitmap index are all *uniform random bitmaps* as defined in [13]. The key results we use for our analysis is the formula for the size of such a uniform random bitmap.

Following the definitions used in [13], we define w to be the number of bits in a word, N to be the number of rows in a dataset (also the number of bits in a bitmap of a bitmap index), and d to be the fraction of the bits that are 1 in a bitmap. The size of a uniform random bitmap is given by

$$\begin{aligned} m(d) &= \left\lfloor \frac{N}{w-1} \right\rfloor + 2 - \left(\left\lfloor \frac{N}{w-1} \right\rfloor - 1 \right) \left((1-d)^{2w-2} + d^{2w-2} \right) && (1) \\ &\approx 3 + \frac{N}{w-1} \left(1 - (1-d)^{2w-2} - d^{2w-2} \right). && (2) \end{aligned}$$

The first part on the right-hand side of Equation (1), $\lfloor \frac{N}{w-1} \rfloor + 2$, is the maximum number of words that can be used by a WAH compressed bitmap. The remaining of the right-hand side is the expected number of words that can be removed by WAH compression [13]. Knowing the size of each bitmap, we can sum them up to give the total size of the bitmap index as $\sum m(d_i)$.

Note that Equation (1) is the exact formula from [13], while Equation (2) is a modification of the approximation given in [13]. This approximation is more accurate, particularly for very low bit densities. Since the bitmaps with only 0s are not stored in a bitmap index, the minimum bit density is $d = 1/N$. In this case, the two formulas from Equations (1) and (2) give the same value, 5, which is accurate in our experience.

It is possible that every bitmap has a bit density of $1/N$ if every value of an attribute is distinct. In that case, there are N such bitmaps and the total size of bitmaps is $5N$ words[2]. This total size is larger than a typical B-Tree index which is observed to be 3 - $4N$ words, and it is also larger than the size of a projection index which uses exactly N words. This is one aspect of the curse of cardinality: even with an effective compression, the bitmap index size can be larger than commonly used indexes.

Associated with the increase in index sizes, the query processing time would also increase because of the increased time to perform I/O operations and to operate on the compressed bitmaps. For an attribute with typical high cardinality, say, $C < N/10$, the compressed index size is about $2N$ words, and WAH compressed indexes never take more time than scanning the vertical projections (also known as projection indexes). However, as the attribute cardinality further increases, the indexes would have more than $2N$ words and the WAH compressed indexes would take more time than the projection indexes for increasingly more queries. This is the second aspect of the curse of cardinality for bitmap indexes.

4.2 Sizes of Binned Indexes

Equation (1) gives us a way to compute the expected sizes of bitmaps used in an index. Since we assume the base data is read-only, these bitmaps will not change and therefore can be densely packed together one after another [26]. In an index file containing such a set of packed bitmaps, we also need to store the starting positions of bitmaps and bin boundaries. Since the bitmaps follow each other, we need to know the starting position of bitmap i and bitmap $i + 1$ in order to read the content of ith bitmap. To allow the last bitmap to be handled the same way as the rest, we store the position just after the last byte of the last bitmap as the starting position of a nonexistent bitmap. Altogether we store $B + 1$ starting positions for B bitmaps.

To define B bins for a variable that can take values between 0 and 1, we need to define $B - 1$ bin boundaries between 0 and 1. One may choose to store bin boundaries with or without the values 0 and 1. In our test software, we choose to store the value 1, but not the value 0. This allows us to easily count values less than b_i. This way, we store B bin boundaries for B bins. We also store the actual minimum and maximum value of each bin, which leads to an additional $2B$ values. Assuming that each value is stored in one word, the total size of the bin boundaries and the minimum and maximum values is $3B$ words.

Assuming no single-valued bins, the total number of the clustered values is exactly the number of rows, N. We need $B + 1$ values to record the starting position of each bin. Altogether, the total size (in number of words) of our binned index is

$$S = \sum m(d_i) + N + 5B + 2. \qquad (3)$$

[2] There are N words out of these $5N$ words that have the same value in each bitmap, therefore it is possible to replace these N words with one word [13]. However, doing so makes all the bitmaps depending on this single parameter which makes it more difficult to create the bitmap indexes. To simplify the testing software, we have chosen to keep a counter in each bitmap.

In later experiments, we use a series of synthetic data following Zipf distribution, where the value i, between 0 and $C-1$, has the probability that is proportional to $(i+1)^{-z}$. The constant z is known as the Zipf exponent ($z \geq 0$). When $z=0$ and C is an integer multiple of B, we can simplify the formula for the total size of our binned index as follows.

$$\begin{aligned} S_{z=0} &\approx \left(3 + \frac{N}{w-1}\left(1-(1-\frac{1}{B})^{2w-2} - (\frac{1}{B})^{2w-2}\right)\right) B + N + 5B + 2 \quad (4) \\ &\approx 3N + 8B. \quad (5) \end{aligned}$$

Note that the approximation in Equation (5) is accurate when $1-(1-1/B)^{2w-2} \approx (2w-2)/B$ is accurate, which is true for large B, say $B > 1000$. For smaller B, we need to go back to Equation (4) to compute the index size accurately. The expressions for the equal-weight bins and for non-zero Zipf exponents can be similarly computed even though they are not as easily simplified as in this special case.

4.3 Query Processing Cost

Next we compute the number of words accessed when answering an average query. To simplify the discussion, we only consider 1-sided range queries of the form $x > c$, where c is a constant we call the *query boundary*. The analysis we carry out here can be similarly applied to equality queries and 2-sided range queries. In this analysis, we only consider the amount of data to be read from disk in order to answer a query. To further simplify the discussion, we assume the query boundaries are a uniform sample of all distinct values that appears in the dataset. The main measure we use to judge the effectiveness of an indexing method is the average number of words needed to answer such a 1-sided range query.

To answer a range query with a binned bitmap index, two steps are needed. Step 1 operates on the bitmaps to identify which bins are fully contained in the query range and which are edge bins that require a candidate check. Step 2 performs the candidate check. For any 1-sided range query, there can be at most one edge bin. For simplicity, we assume a candidate check is always necessary. Before the evaluation can start, we always read the starting positions of the bitmaps, the bin boundaries, and the minimum values and the maximum values of each bin to memory. This process reads $4B+1$ words. Clearly, this is the worst case scenario; one could cache these values to reduce the query response time.

In Step 1, the main cost is reading the bitmaps from disk. If we need to read more than half of the bitmaps (as measured by the number of words accessed), we can evaluate the complement of the query instead. This allows us to read no more than half of the words in the bitmap index. Given that the query boundaries are uniformly distributed in the domain of the attribute, the average number of words accessed in Step 1 is one quarter of the total size of the bitmaps ($0.25 \sum m(d_i)$).

In Step 2, the main cost is reading the values in the edge bin. We simply take this cost to be N/B words. The cost of reading the starting and ending positions of the bin could be taken as two words. However, because the underlying file system reads at least one page, we approximate it by assuming that all $B + 1$ starting positions are read. Overall, the total number of words read from disk is

$$R = 2 + 5B + N/B + 0.25 \sum m(d_i). \tag{6}$$

With equal-weight bins and for nonuniform data, we can similarly compute the expected cost to process an average query. Because of the use of single-valued bins, the expressions for these cases are much longer than Equation (6). Instead of giving analytical expressions, we plotted them in Figure 2.

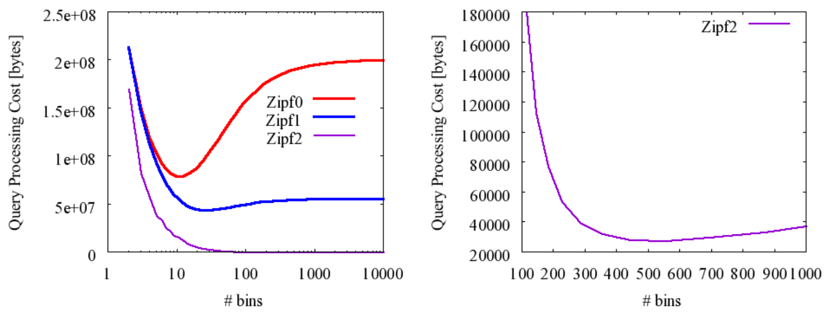


Fig. 2. The average number of words accessed to answer a 1-sided range query using bitmap indexes with equal-weight bins and OrBiC ($N = 10^8$, $C = 10^6$, $w = 32$).

Figure 2 plots the average number of words accessed to answer a query. The figure shows the query processing costs for three synthetic attributes, Zipf0, Zipf1 and Zipf2 (named after their Zipf exponents). We identify the optimal number of bins to be about 13 for Zipf0, 25 for Zipf1 and 550 for Zipf2. We also note that the average query processing cost is not sensitive to the number of bins for Zipf1 and Zipf2. For example, in the case of Zipf2 any number of bins from 400 to 1000 leads to nearly the same query processing cost.

Next we compare the query processing cost of our new method against that of the unbinned bitmap index and the projection index. We first consider the unbinned bitmap index. Again we assume that one quarter of words has to be read on average. Thus, the query processing costs of an attribute where every value is distinct is $1.25N$ words. In Figure 2, we assumed $N = 10^8$, which means the query processing cost is at worst 5×10^8 bytes. The query processing cost of the projection index is always N words. In Figure 2, this corresponds to a query processing cost of 4×10^8 bytes. We clearly see that query processing of our binned index with OrBiC costs less than 4×10^8 bytes. With the optimal number of bins, the expected query processing cost for Zipf0 is about 8×10^7 bytes, which is about 1/5th of that of the projection index. In a more realistic case, where the unbinned index is close to $2N$ words, the average query processing cost is about 2×10^8 words, the binned index with OrBiC is about 3 ($\sim 2 \times 10^8/8 \times 10^7$) times as fast as the unbinned index. The query processing cost for nonuniform data is much less than that for Zipf0 as evidenced by experiments in the next section.

In Figure 3, we show a comparison between the query processing cost of a binned bitmap index with OrBiC and one without OrBiC. It is easy to see that without OrBiC we need to use more than 1000 bins in order for the query processing cost to be less than that of the projection index. However, with OrBiC, the query processing cost is always below that of the projection index.

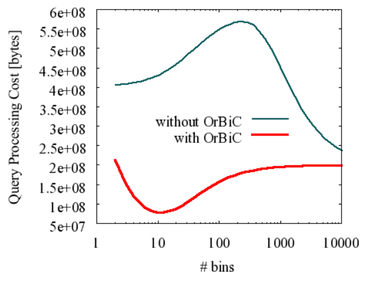


Fig. 3. The average number of words accessed to answer a 1-sided range query on Zipf0 using binned bitmap indexes with and without OrBiC.

Overall, the analyses here show that the new binned bitmap index with OrBiC and equal-weight binning outperforms well-known methods for answering range queries on read-only data. Since our analyses do not include the CPU time or I/O overhead such as disk seek time, the actual observed query response time could show different performance characteristics. Next, we conduct a number of tests on both synthetic and real application data to measure the actual performance.

5 Data Sets and Index Sizes

In this section we describe the synthetic and application datasets used for our performance evaluation. We also discuss the sizes of bitmap indexes with different numbers of bins and compare their sizes with unbinned bitmap indexes and the expected values computed in the previous section.

All experiments were conducted on a computer with dual 2.8 GHz Pentium 4 processors, 2 GB of main memory, and an IDE RAID storage system capable of sustaining 60 MB/sec for reads and writes. Our bitmap index software is implemented with C++ and compiled with gcc 4.1.0 using the compiler optimization flag -O5.

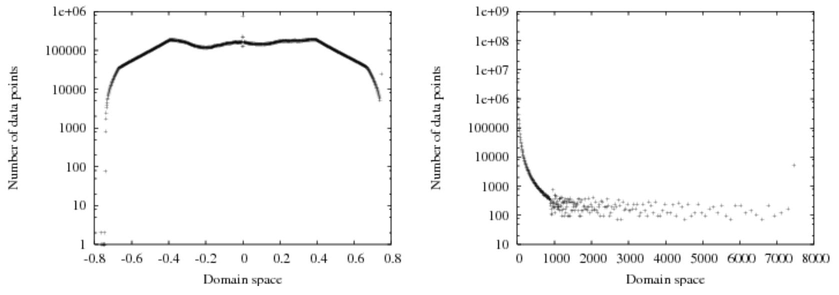
5.1 Zipf Data

The synthetic data set consists of three high-cardinality attributes following the Zipf distribution with Zipf exponents 0, 1, and 2. We refer to these three attributes as Zipf0, Zipf1 and Zipf2. The number of rows is 100 million. The total size of the data set is 1.2 GB. The three synthetic attributes have non-negative values less than 1 million. Their cardinalities are much higher than those used in the earlier tests [11, 13].

5.2 Astrophysics Data

Our application data set is from an astrophysics application that studies supernova explosions. The data consists of 6 high-cardinality floating-point valued attributes with 110 million rows. The average attribute cardinality of these attributes is about 25 million. The total size of the data set is about 2.6 GB.

The distributions of two attributes are shown in Figure 4. Note the log scale on the y-axis. We show the distribution of x-velocity as the representative of the three velocity components. These attributes have some infrequent values, but the majority of the values have frequencies within the same order of magnitude. The other three attributes, density, entropy and pressure, have much higher skew in their distribution. We show the distribution of density as the representative. In this case, we see that the frequencies of values span eight orders of magnitudes.



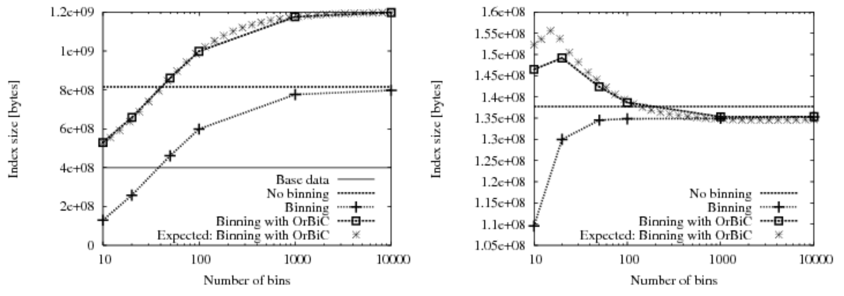
(a) x-velocity (b) density

Fig. 4. Distribution of astrophysics data. Note the log scale on the y-axis.

5.3 Index Sizes

Next we examine the bitmap index sizes for both data sets. We start our discussion with the Zipf data.

For each of the three Zipf attributes Zipf0, Zipf1 and Zipf2, we generated bitmap indexes with 10, 20, 50, 100, 1000 and 10000 equal-weight bins, where each bin has about the same number of values. Each of these variants also include an instance with OrBiC and an instance without OrBiC. In addition, we also generated bitmap indexes with no binning. In total we generated 13 different bitmap indexes per attribute. The sizes of the bitmap indexes for the Zipf data are shown in Figure 5. We label the indexes with equal-weight binning (without OrBiC) as "binning", and the indexes with both equal-weight binning and OrBiC as "binning with OrBiC". For references, we also plotted the size of the base data as the solid horizontal line, the unbinned index size as the dashed line, and the expected sizes according to the analyses from the previous section as 'x'.

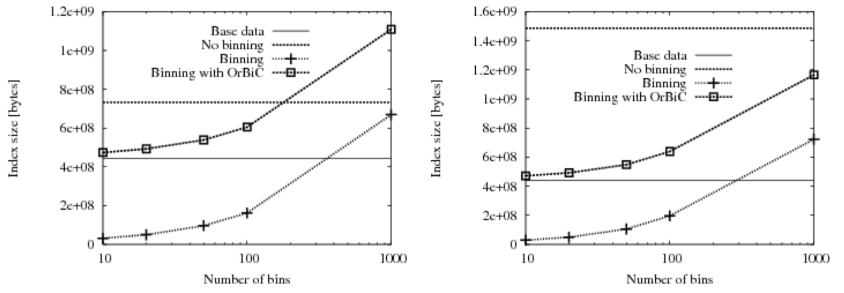


(a) Zipf0 (b) Zipf2

Fig. 5. Sizes of the bitmap indexes with different numbers of bins for Zipf data.

Note that the expected sizes of the bitmap indexes agree very well with the actual sizes in Figure 5. We see that the size of the bitmap index with binning (without OrBiC) is always smaller than the bitmap index without binning. As the number of bins increases, the curves marked "binning" and "binning with OrBiC" become closer for Zipf2, because the OrBiC data structure stores less and less values as more and more bins become single-valued.

The sizes of the bitmap indexes for the astrophysics data are shown in Figure 6. Again we see that the bitmap index with binning is always smaller than the bitmap index without binning[3]. We also note that the bitmap index with binning and OrBiC is only larger than the unbinned bitmap index in one case, namely for the attribute x-velocity with 1000 bins with OrBiC. These results clearly demonstrate that our novel binned bitmap index is able to take advantage of the non-uniformity present in the data to reduce the sizes of bitmap indexes.



(a) x-velocity (b) density

Fig. 6. Sizes of the bitmap indexes with different numbers of bins for astrophysics data.

[3] Due to extensive resource requirements of bitmap indexes without binning we had to build the unbinned indexes on a server with more than 2GB of main memory.

6 Query Processing Time

In this section we report an experimental evaluation of our binned indexes on both synthetic and application data. The experiments are structured as follows. We first compare the performance of range queries for binned bitmap indexes with and without OrBiC. The results show that binned bitmap indexes with OrBiC are about a factor of 3 to 25 faster than those without OrBiC. Next, we run a set of tests to measure the relative performance of binned index with OrBiC against unbinned indexes and projection indexes. Because these indexes are known to significantly outperform the more popular B-Tree index [2, 3, 5], we do not compare with B-Tree indexes directly.

All experiments are based on so-called aggregation queries that are common in data warehousing and scientific applications. These types of queries provide statistical information on the result set rather than returning result records. A typical example of an aggregation query in an astrophysics application is as follows: "Count the number of cells where pressure $>$ X". Note that before we executed each set of queries, we unmounted and remounted the file system containing both the data and the indexes in order to ensure cold cache behavior. The timing values reported here are elapsed time in seconds.

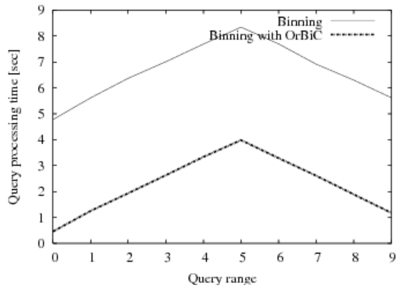
In our first set of experiments we evaluate the query performance of binned bitmap indexes with and without OrBiC. To simplify the following discussion, we fix the number of bins at 100. We start our performance evaluation with range queries over synthetic data. The cardinality of each synthetic attribute is 1,000,000. For each attribute we ran 10 range queries with ranges uniformly distributed over the entire attribute domain space. In particular, the query workload is as follows: $a \geq C/Q * q + \delta$ where a is the query attribute, C is the attribute cardinality, Q is the total number of queries, q is the query number and δ is a small value to make sure that the query range does not fall on a bin boundary. Note that if a query range falls on a bin boundary there is no need for a candidate check. Hence, the query performance for binned bitmap indexes with OrBiC is equal to those without OrBiC.

Figure 7(a) shows the query response time for binned bitmap indexes with and without OrBiC for Zipf0, i.e. uniformly distributed data. We see that binned bitmap indexes with OrBiC are about a factor of 3 faster than binned bitmap indexes without OrBiC. This agrees with Figure 3 for 100 bins. In Figure 7(a), we notice that both timing curves show a characteristic "A shape", because we evaluate the complement of the query if more than half of the bitmaps are involved.

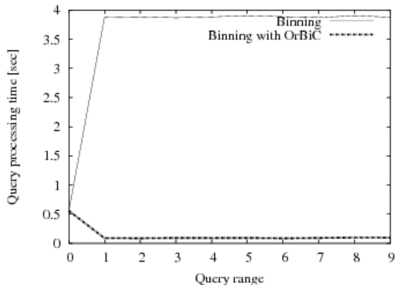
Figure 7(b) show the query response time for binned bitmap indexes on Zipf2. Again we see that the binned bitmap indexes with OrBiC are significantly faster than those without OrBiC. In particular, the average performance improvement is a factor of 25.6 for Zipf2 (and 5.5 for Zipf1). These speedup values are larger than that for Zipf0 indicating that the advantage of using OrBiC increases as the skewness of data increases.

Figure 8 shows the query response time for range queries over two attributes of the astrophysics data set. Again we see a significant performance increase for binned bitmap indexes with OrBiC compared with bitmap indexes without OrBiC.

Table 1 summarizes the average performance improvements of binned bitmap indexes with OrBiC over binned bitmap indexes without OrBiC. The advantage of using

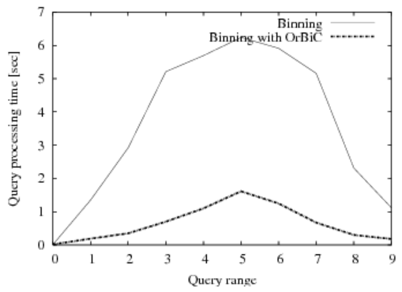


(a) Zipf0

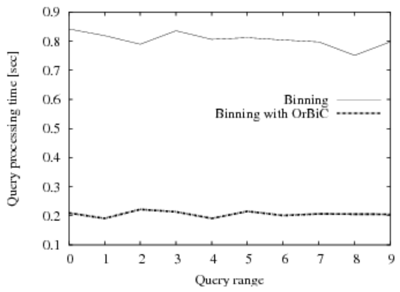


(b) Zipf2

Fig. 7. Processing time of range queries over Zipf data. The figure shows a significant performance advantage of binned bitmap indexes with OrBiC over binned bitmap indexes without OrBiC.



(a) x-velocity



(b) density

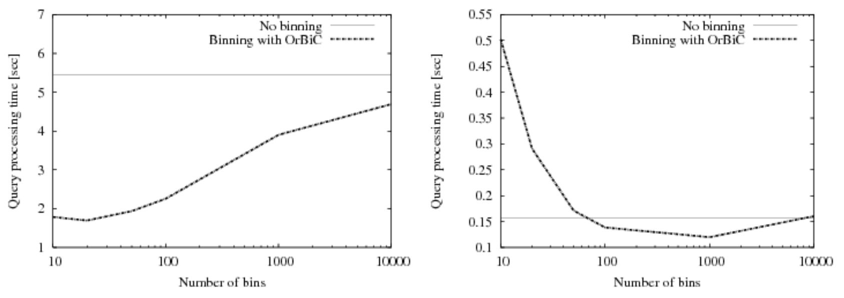
Fig. 8. Processing time of range queries over astrophysics data. The figure shows a significant performance advantage of binned bitmap indexes with OrBiC over binned bitmap indexes without OrBiC.

OrBiC is the least for uniform data, which has a speedup value of 3. On our application data and synthetic data with skew, the speedup values are larger.

Table 1. Average speedup of binned bitmap indexes with OrBiC over binned bitmap indexes without OrBiC.

Synthetic		Astrophysics			
Attribute	Speedup	Attribute	Speedup	Attribute	Speedup
Zipf0	2.94	density	3.91	x_velocity	5.65
Zipf1	5.50	entropy	12.61	y_velocity	4.82
Zipf2	25.62	pressure	4.40	z_velocity	4.28

The previous set of tests clearly confirms the advantage of using OrBiC. In the following tests, we compare binned indexes with OrBiC with two other types of indexes, the unbinned bitmap index and the projection index. In this set of tests, we use the average query response time over all queries to compare different indexing methods. We start with a comparison against the unbinned index in Figure 9.



(a) Zipf0 (b) Zipf2

Fig. 9. The average query response time with the new binned index with OrBiC and the unbinned bitmap index.

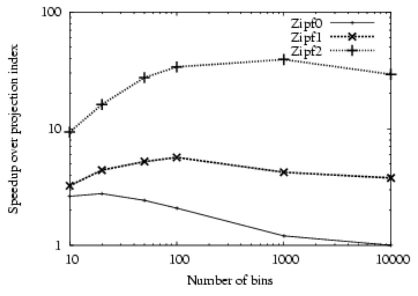
In Figure 9, the vertical axis shows the average query response time to answer the same 10 range queries used in our previous tests. The horizontal axis shows the number of bins used by the binned index. Figure 9(a) is for the uniform data Zipf0 and Figure 9(b) is for the highly skewed data Zipf2. In the case of the uniform data, where the unbinned index require about 5.4 seconds to answer a query[4], the binned index with OrBiC is always better. In the best case, the new binned index with OrBiC is about three times faster than unbinned index, which agrees with the analysis given in Section 4.3. On highly skewed data, the unbinned index performs very well, using about 0.15 seconds instead of 5.4 seconds. However, even in this case, the new binned index with OrBiC can outperform the unbinned index with a wide range of choices as the number of bins.

Figure 10 shows the relative performance of the new binned index against the projection index. The vertical axis is the speedup of range queries using binned bitmap indexes over the projection index and the horizontal axis is the number of bins used by the binned index. Overall, we see that the speedup values are always greater than 1, indicating the new binned index with OrBiC is always faster than the projection index.

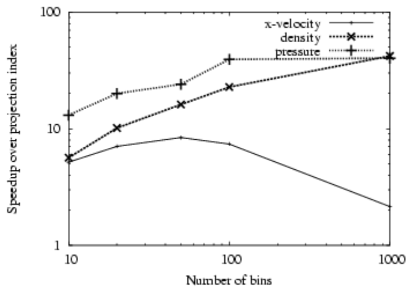
Figure 10(a) shows the results for synthetic data. For Zipf0 with 20 bins the speedup over the projection index is about a factor of 3. For Zipf1 and Zipf2, the highest speedup is about a factor of 6 and 40, respectively.

Figure 10(b) shows the average speedup for the astrophysics dataset. We can see that for attribute x-velocity bitmap indexes with about 50 bins have the highest speedup of about a factor of 8. Note that the attribute x-velocity is of moderate skewness. However, for the highly skewed attributes such as density and pressure, the optimal number of bins is much higher and is around 1000. Additional analyses are required to fully understand

[4] Assuming that 5.4 seconds were used to read a quarter of bitmaps, totaling about 8×10^8 bytes, the effective reading speed is 37 MB/s, which is about 2/3 of the maximum reading speed.



(a) Synthetic



(b) Astrophysics

Fig. 10. Speedup of range queries using binned bitmap indexes over projection index.

the dependency of the number of bins on the skewness of data. However, the benefit of using OrBiC is clear; the highest speedup over the projection index for the attributes density and pressure is about a factor of 40.

7 Conclusions

The basic unbinned bitmap indexes used in major commercial database systems are efficient for querying low-cardinality attributes and highly skewed data. However, they suffer from the *curse of cardinality*, i.e. their effectiveness decreases as attribute cardinality increases. We solve this important problem by using a novel binned bitmap index structure that performs efficiently even with extremely high-cardinality attributes. One key idea is to augment the bitmaps with an *Order-preserving Bin-based Clustering (OrBiC)* data structure. This data structure significantly reduces the cost of candidate checks. In addition, we use a hybrid-binning strategy that employs single-valued bins for frequent values to eliminate the need for candidate checks for these single-valued bins. This further enhances the search performance. We performed detailed analytical and experimental evaluations of our bitmap index structure and showed that our binned bitmap indexes are in most cases smaller in size and more efficient in answering queries than the unbinned bitmap indexes and the projection indexes.

In the worst case for a unbinned bitmap index, that is when the attribute has very high cardinality and uniform data distribution, our analysis provides definitive guidance on the number of bins to use. In addition, the predicted advantage has been verified experimentally as well. For more realistic data, where the single-valued bins enhance performance, a precise analysis of query processing cost (Equation (6)) is more complicated, and how to determine the optimal number of bins remains a challenge. Nevertheless, we experimentally demonstrated that our technique effectively broke the curse of cardinality.

The software that implements our binning technique has been released under LGPL and can be accessed at `https://codeforge.lbl.gov/projects/fastbit`.

References

1. Berchtold, S., Böhm, C., Kriegal, H.P.: The pyramid-technique: Towards breaking the curse of dimensionality. SIGMOD Record **27**(2) (June 1998) 142–153
2. O'Neil, P.: Model 204 architecture and performance. In: Second International Workshop in High Performance Transaction Systems, Springer Verlag (1987)
3. O'Neil, P., Quass, D.: Improved query performance with variant indices. In: SIGMOD. ACM Press (1997)
4. Wu, K., Otoo, E.J., Shoshani, A.: On the performance of bitmap indices for high cardinality attributes. In VLDB. Morgan Kaufmann (2004) 24–35
5. Wu, K., Otoo, E., Shoshani, A.: A performance comparison of bitmap indices. In: CIKM. ACM Press (2001)
6. Lewis, J.: Bitmap indexes - part 1: Understanding bitmap indexes. http://www.dbazine.com/oracle/or-articles/jlewis3 (2006)
7. Koudas, N.: Space efficient bitmap indexing. In: CIKM. ACM Press (2000)
8. Shoshani, A., Bernardo, L.M., Nordberg, H., Rotem, D., Sim, A.: Multidimensional indexing and query coordination for tertiary storage management. In: SSDBM. (1999) 214–225
9. Stockinger, K., Duellmann, D., Hoschek, W., Schikuta, E.: Improving the performance of high-energy physics analysis through bitmap indices. In: DEXA. Springer Verlag (2000)
10. Wu, K.L., Yu, P.: Range-based bitmap indexing for high cardinality attributes with skew. Technical Report RC 20449, IBM Watson Research , New York (1996)
11. Johnson, T.: Performance Measurements of Compressed Bitmap Indices. In: VLDB. Morgan Kaufmann (1999)
12. Antoshenkov, G.: Byte-aligned Bitmap Compression. Technical report, Oracle Corp. (1994) U.S. Patent number 5,363,098.
13. Wu, K., Otoo, E., Shoshani, A.: Optimizing bitmap indices with efficient compression. ACM Transactions on Database Systems **31** (2006) 1–38
14. Comer, D.: The ubiquitous B-tree. Computing Surveys **11**(2) (1979) 121–137
15. Wu, K., Otoo, E.J., Shoshani, A.: Compressing bitmap indexes for faster search operations. In: SSDBM. (2002) 99–108
16. Wong, H.K.T., Liu, H.F., Olken, F., Rotem, D., Wong, L.: Bit transposed files. In: Proceedings of VLDB 85, Stockholm. (1985) 448–457
17. Chan, C.Y., Ioannidis, Y.E.: Bitmap Index Design and Evaluation. In: SIGMOD. ACM Press (1998)
18. Chan, C.Y., Ioannidis, Y.E.: An Efficient Bitmap Encoding Scheme for Selection Queries. In: SIGMOD. ACM Press (1999)
19. Rotem, D., Stockinger, K., Wu, K.: Minimizing I/O costs of multi-dimensional queries with bitmap indices. In: SSDBM. IEEE (2006)
20. Rotem, D., Stockinger, K., Wu, K.: Optimizing candidate check costs for bitmap indices. In: CIKM. ACM Press (2005)
21. Gray, J., Liu, D.T., Nieto-Santisteban, M., Szalay, A., DeWitt, D., Heber, G.: Scientific data management in the coming decade. CTWatch Quarterly (2005)
22. Stonebraker, M., et al: C-store: A column-oriented dbms. In: VLDB. (2005) 553–564
23. Boncz, P.A., Zukowski, M., Nes, N.: Monetdb/x100: Hyper-pipelining query execution. In: CIDR. (2005) 225–237
24. Golub, G.H., van Loan, C.F.: Matrix Computations. third edn. The Johns Hopkins University Press. (1996)
25. Thaper, N., Guha, S., Indyk, P., Koudas, N.: Dynamic multidimensional histograms. In: SIGMOD. ACM (2002) 428–439
26. O'Neil, E., O'Neil, P., Wu, K.: Bitmap index design choices and their performance implications. In: IDEAS. (2007) 72–84