RC 20449 (05/02/96)
Computer Science

IBM Research Report

Range-Based Bitmap Indexing for High Cardinality Attributes with Skew

Kun-Lung Wu, Philip S. Yu

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

Research Division
Almaden · T.J. Watson · Tokyo · Zurich

Range-Based Bitmap Indexing for High Cardinality Attributes with Skew[1]

Kun-Lung Wu and *Philip S. Yu*

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Abstract

Bitmap indexing, though effective for low cardinality attributes, can be rather costly for high cardinality attributes due to the large storage requirement for the indexes. In this paper, we study a range-based bitmap (RBM) indexing approach to reducing this storage overhead. The attribute values are partitioned into a small number of ranges and a bitmap vector is used to represent each range, instead of each distinct value. With RBM indexing, however, the number of records assigned to different partitions can be highly uneven, resulting in non-uniform search times for different queries. We present and evaluate a dynamic bucket expansion and contraction (DBEC) approach to simultaneously constructing range-based bitmap indexes for multiple attributes of high cardinality. In DBEC, a number of buckets are used to count the records whose values fall into each bucket. Buckets with a large record count are dynamically expanded into more smaller-range buckets. At the end, multiple buckets with adjacent ranges are combined. Simulations are conducted to compare the DBEC approach with a naive partition, a simple bucket expansion, and the optimal partition approaches. Both synthetic and real data are used in the simulations. The results show that (1) even with only slightly skewed data, the naive partition approach can produce rather uneven buckets; (2) with highly skewed data, DBEC significantly outperforms the simple expansion approach; and (3) DBEC compares favorably with the optimal approach even though the optimal approach requires a complete sorting of the records while DBEC does not.

Index Term: multidimensional data analysis, bitmap indexing, access method, information warehousing, and skew.

[1]This work was supported in part by NASA/CAN grant no. NCC5-101.

1 Introduction

Multidimensional data analysis has become increasingly important for decision support systems as more and more complex queries are posted against ever larger amount of data in a *data/information warehouse* [21, 7, 10]. To efficiently handle multidimensional data analysis, an effective multidimensional index structure is required. Among the many existing multidimensional indexes, such as k-d trees [3, 2], k-d-B trees [16], hB-trees [13, 5], Quad-trees [18], R-trees [9, 19], and grid files [15, 6], bitmap indexing [12, 7] is potentially the easiest to build and maintain.

To build a bitmap index for an attribute of a table of N records, a separate bitmap vector of length N bits is used to represent each distinct attribute value. Each bit in a bitmap vector represents one record, indicating whether or not the record has the matching attribute value represented by the bitmap vector. With bitmap indexes constructed for multiple attributes, it is rather easy to identify the records that satisfy a multiattribute predicate. All it needs is some bitwise logical AND/OR operations using the corresponding bitmap vectors.

As an example, Table 1 shows the trading volumes, closing prices and the exchanges where the stocks are traded, of selected 12 stocks on 10/18/1995. (These data are extracted from the experimental stock market data server at "`http://www.ai.mit.edu/stocks/prices.html`".) Assume that we are to build bitmap indexes for the 3 attributes: "trading volume," "closing price" and "exchange". A bitmap vector of length 12 bits can be used for each distinct attribute value of the 3 indexed attributes. For attribute "exchange", only two bitmap vectors are needed since there are only two distinct values: NASDAQ and NYSE. Therefore, the two bitmap vectors are as follows:

NASDAQ: `(1 0 0 0 0 1 0 0 0 1 1 1);`
NYSE: `(0 1 1 1 1 0 1 1 1 0 0 0).`

The 1's in vector NASDAQ indicate that records 1, 6, 10, 11 and 12 have NASDAQ as their values in attribute "exchange". However, 12 bitmap vectors are needed for "trading volume" and another 12 bitmap vectors are needed for "closing price" because all 12 records have different values. Thus a total of 26 bitmap vectors are needed.

Multidimensional bitmap indexing (i.e., constructing bitmap indexes for multiple attributes) can be very effective if the indexed attributes are of low cardinality, such as the "exchange" attribute in Table 1. However, for high cardinality attributes, the storage requirement can be prohibitively large. If N is the total number of records to be indexed, m is the number of dimensions (attributes)

Table 1: Selected stock trading data on 10/18/1995.

Record ID	Ticker Symbol	Trading Volume	Closing Price	Exchange
1	AAPL	4575000	36.625	NASDAQ
2	ABF	64200	24.500	NYSE
3	AET	369000	72.625	NYSE
4	CPQ	8968800	51.375	NYSE
5	DEC	4461100	49.750	NYSE
6	DELL	2714400	89.750	NASDAQ
7	HWP	3009300	90.250	NYSE
8	IBM	7657700	92.500	NYSE
9	IFMX	3493600	33.000	NYSE
10	INTC	17694400	65.500	NASDAQ
11	LGNT	2600	47.250	NASDAQ
12	MSFT	18288600	91.125	NASDAQ

to be included in the multidimensional index, and D_i is the number of distinct values for attribute i, then the total storage requirement S for a multidimensional bitmap index in bits is

$$S = N \times \sum_{i=1}^{m} D_i.$$

If $\sum D_i$ is small, then S can still be manageable even if N is large. However, if both N and $\sum D_i$ are large, e.g., some attributes may be of continuous data type, then a huge amount of storage space may be needed to store all the bitmap indexes. For example, in Table 1 if the stock data contain all the trading records of both NYSE, NASDAQ and AMEX for all of the 1995 trading days, then the bitmap indexes can become exceedingly large because of the high cardinality in attributes "trading volume" and "closing price." Such a high storage requirement can make multidimensional bitmap indexing very costly.

One approach to reducing the storage overhead due to high cardinality attributes is to partition the attribute values into a smaller number of ranges. A bitmap vector is used to represent a range, instead of a distinct value. This way, the number of bitmap vectors for any attribute can be properly controlled. A 1 is set if the attribute value of a record falls into the specified ranges; otherwise a 0 is set. This range-based bitmap (RBM) indexing, however, may require a longer query processing time as it requires further processing to examine the details of all the records in the matched range. Nevertheless, it represents a good trade-off for reduced storage requirement.

For instance, in Table 1, we can use only 2 range-based bitmap vectors to index attribute "trading volume," instead of 12. Since the records are not pre-sorted before the bitmap indexes are built, one simple approach is to partition the the entire ranges of "trading volume" into two equally spaced buckets, [0, 10,000,000] and (10,000,000, 20,000,000], assuming that the maximum of trading volume is 20,000,000 shares. Thus, the bitmap vector representing the two ranges are as follows:

[10,000,000, 20,000,000]: (0 0 0 0 0 0 0 0 0 1 0 1);
[0, 10,000,000): (1 1 1 1 1 1 1 1 1 0 1 0).

With these two range-based bitmap vectors for attribute "trading volume" and the two bitmap vectors for attribute "exchange", we can process a query like "find all NYSE traded stocks with trading volumes of less than 3 millions shares" in a fairly straightforward fashion. First the bitmap vectors of NYSE: (0 1 1 1 1 0 1 1 1 0 0 0) and [0, 10,000,000): (1 1 1 1 1 1 1 1 1 0 1 0) are bitwise ANDed to produce (0 1 1 1 1 0 1 1 1 0 0 0). Then the 7 records indicated by the 1's are examined in detail to find out which records meet the requirements. In this case, records 2 and 3 are the final results.

However, this simple approach is not a good partition of "trading volumes" into 2 buckets, as it results in 10 records in one bucket and only 2 records in another bucket. This can cause a very uneven processing times for different queries. For instance, a query to find the NYSE traded stocks with trading volumes of less than 3,000,000 shares needs to examine 7 records. On the other hand, another query to find the NASDAQ traded stocks with trading volumes of more than 15,000,000 shares only needs to examine 2 records.

In general, without complete knowledge of the distribution of attribute values, the range-based bitmap index can result in very uneven buckets and cause access times for different queries to vary substantially. This problem can be further aggravated if the attribute values are highly skewed. Ideally, one would want to design an access method that is "balanced", namely, the execution times are more or less equal for different search queries. However, in order to know the complete distribution of attribute values, one may have to sort the data. Unfortunately, sorting is usually costly due to I/O overhead, especially for external sorting of a large table.

In this paper, we present a dynamic bucket expansion and contraction (DBEC) approach to constructing buckets for range-based bitmap indexing for high cardinality attributes. DBEC does not require pre-sorting before building the bitmap indexes. It consists of 2 phases. In the first phase, data records are sequentially scanned into the memory and B_i initial equally-spaced buckets are used to count the number of records falling into each bucket. Here, $B_i > B$ and B is the final number of ranges for which bitmap indexes are to be built. This can be done simultaneously for multiple high cardinality attributes. However, for simplicity, from now on we will focus on a single attribute even though the same algorithm can be simultaneously applied to multiple attributes. Whenever a bucket has accumulated more than a certain number of records, it is dynamically expanded into more smaller-range buckets. Upon completion of counting, multiple small-range buckets are then contracted into the final B buckets such that each bucket contains about the same number of records. In order to support range queries, we focus on bucket contraction that only combines those buckets representing continuous ranges. We study three heuristics to combine multiple adjacent smaller-range buckets into the final B larger-range buckets. In the second phase, data are then sequentially scanned again to actually build the bitmap indexes, with a bitmap vector representing the range of each final bucket.

Simulations are conducted to evaluate the DBEC approach and compare it with a naive partition, a simple expansion and the optimal partition approaches. The naive partition approach assumes the attribute values are uniformly distributed, and it simply partitions the attribute range into B equally-spaced buckets and constructs the bitmap indexes in a single I/O scan. It needs the least I/O cost for building the bitmap indexes. However, if the attribute values are skewed the resulting distribution of bucket sizes can be highly uneven. Similar to DBEC, the simple expansion approach requires two phases; but, different from DBEC, the initial B_i buckets are not expanded. Bucket contraction is also used by the simple expansion approach to contract B_i small-range buckets into the final B buckets. Without dynamic expansion of buckets, the simple expansion approach cannot handle the cases where the attribute values are highly skewed. The optimal partition approach first sorts the records in order and then they are optimally partitioned into B buckets.

Both synthetically generated data and real data containing the trading data of some 296 stocks traded in the 3 major US stock markets on 10/18/1995 were used to evaluate DBEC and compare it with the naive partition and simple expansion approaches. The results show that (1) with only slightly skewed attribute values, the naive partition approach can result in highly uneven bucket sizes; (2) with high skew, DBEC substantially outperforms the simple expansion approach; (3) using a good heuristic for bucket contraction, DBEC compares favorably with the optimal partition approach even though DBEC does not require a pre-sorting of the entire attribute values. These results are verified against the stock trading data in which the distribution of trading volumes is highly skewed while that of closing prices is less so.

Note that the issue of bucket expansion and contraction discussed here is, to some extent, similar to that of bucket partitioning in the *hash-partitioned join* algorithms, such as the GRACE hash-join and Hybrid hash-join algorithms [4, 20, 14, 11]. Both are to partition a relation into multiple buckets based on the values of a particular attribute. However, their constraints and objectives are distinctively different. In the hash-partitioned join algorithms, the goal is to hash a relation R into disjoint buckets such that each bucket can be fit into main memory to build a hash table for join operations with another relation S. The range of a bucket is not important, and thus a tuple can be arbitrarily hashed into any bucket. Bucket tuning can therefore be used to combine any number of small buckets into a larger one so long as the resultant bucket can be fit into the main memory [11]. On the other hand, in the bucket expansion and contraction for bitmap indexing discussed in this paper, the range of a bucket is an important piece of information. Tuples cannot be hashed into any bucket. In order to support range queries, only contiguously ranged small buckets can be combined in bucket contraction. Besides, the bucket size can be flexible. As a result, the solutions to both problems are quite different.

In addition, the concept of counting object occurrence has also been used to identify hot spots in databases and file systems [17, 1]. However, unlike hot spot identification, the objective of occurrence counting in bucket partition for bitmap indexes is to ensure that the attribute values are equally partitioned. Finally, the results of this paper on equally partitioning a table based on its attribute values are also applicable to distributing workload to different processors in a parallel machine, such as an IBM SP/2.

The rest of the paper is organized as follows. Section 2 describes the details about bucket construction and contraction. Section 3 presents the simulation model and workloads used. Section 4 then provides in detail the simulation results. Section 5 summarizes the paper.

2 Bucket construction

Assuming that the minimum (V_{min}) and maximum values (V_{max}) of an attribute are given, the simplest approach to constructing B buckets for an attribute is to partition the entire attribute range into B equal-distance ranges, each with the same width of $(V_{max} - V_{min})/B$. This approach is called **naive partition**. It assumes that the distribution of the attribute values is uniform between V_{min} and V_{max}. However, the distributions of many real-life data are not uniform. They tend to be skewed to certain ranges. Thus, this naive partition approach is likely to construct highly uneven buckets even if the data are only slightly skewed.

2.1 Bucket expansion

The drawback of the naive partition approach is due to the fact that buckets are constructed without knowing the distribution of attribute values. We propose here a bucket expansion and contraction approach to roughly constructing buckets according to the distribution of the data records. It does not require a pre-sorting of the data, but it consists of two phases of sequential I/O scans. In the first phase, the data are scanned and a large number of B_i small-range buckets, where $B_i > B$, are used to count the number of records falling into each bucket. Here, the initial B_i buckets are constructed simply by partitioning the entire attribute range equally into B_i intervals. These small-range buckets are then contracted into the final B larger-range buckets such that the size of each final bucket is about equal.

There are two different expansion alternatives. One is called **simple expansion** and the other **dynamic expansion**. The simple expansion uses B_i as the initial buckets and simply collects the data counts in each bucket during the first phase. Namely, the expansion is done initially and uniformly. The larger B_i is, the more effective the simple expansion approach becomes. However, if attribute values are highly skewed, B_i needs to be extremely large in order for the simple expansion scheme to be effective. Large B_i takes memory space and thus is less desirable if multiple attributes are to be partitioned simultaneously.

On the other hand, the dynamic expansion scheme dynamically expands a growing bucket into more buckets, each with a smaller range. Assuming that `b[j]` represents a bucket j during the dynamic expansion phase and `a[i]` the attribute value of tuple i. Figure 1 shows the pseudo code for dynamic bucket expansion. For each tuple i, we find the bucket `b[j]` such that `a[i]` falls into the range of `b[j]` and increase its record count. If needed, a bucket is dynamically expanded into more smaller-range buckets. The criterion for expansion is when a bucket count is greater than a threshold, T. The checking of threshold can be periodical or whenever a bucket count is updated. Since some of the buckets may be newly expanded, the threshold for triggering a further expansion of a small-range bucket should take into consideration the age of the bucket since its creation. After every tuple is scanned, the original record counts of all the expanded bucket are re-distributed into the smaller-range buckets proportional to the respective counts of the smaller-range buckets.

For dynamic bucket expansion, there might be some cases where the same attribute values

Dynamic Bucket Expansion:

```
for every tuple i
{
    find bucket b[j] such that a[i] falls into b[j] and increase its record count;
    push a[i] into the primary stack;
    push a[i] into the secondary stack;
    if bucket counts need to be checked for expansion
    {
        find all the buckets that need to be expanded;
        for each bucket b[k] to be expaned;
        {
            if there is an isolated skew value in b[k]
                allocate a separate bucket for it and adjust neighboring buckets;
            else
                expand b[k] into more smaller-range buckets;
            maintain the count and range of b[k];
        }
    }
}
re-distribute the occurrence counts of all the expanded buckets;
```

Figure 1: Pseudo code for dynamic bucket expansion.

appear repeatedly for many times. Namely, there are some isolated skew values. In this case, dynamic bucket expansion may result in some buckets with exceptionally large occurrence counts and many other empty buckets. Too many empty buckets are not desirable since they take memory space. To prevent excessive empty buckets from occurring, we can identify these isolated skew values in the process of bucket expansion.

To identify isolated skew values, two stacks can be maintained. The primary stack is maintained by the recency of occurrence, while the secondary stack is maintained by the frequency of occurrence. Namely, the attribute value with the highest occurrence count is placed on the top of the secondary stack, and the bottom of the stack gets flushed out first. For each record scanned, its attribute value will be added or moved to the top of the primary stack, similar to a conventional LRU stack. It will also be added to the secondary stack, if not yet there. The occurrence count of the secondary stack is incremented. Therefore, the primary stack identifies attribute values that are repeated reasonably recently, and recommends that the secondary stack track the occurrence counts for those values. The secondary stack will stop considering those attribute values whose occurrence counts do not increase fast enough. With these two stacks, we can control the storage requirement for the two stacks and avoid tracking every possible attribute value. As a result, in the expansion algorithm, we first check if a bucket contains some high occurrence attribute values in the secondary stack before a bucket is expanded. If a sizable fraction of a bucket count is due to a high occurrence attribute, a single bucket is assigned to this attribute value. This single value bucket will not be split any further no matter how large the occurrence count grows.

Note that bucket construction for a high cardinality attribute can be piggybacked with the bitmap vector construction of other low cardinality attributes. Thus, no additional I/O overhead may be incurred for the first phase. This is particularly true if multiple passes are needed to build the complete bitmap vectors because there are many attributes to be included in the multi-dimensional bitmap indexes. In any case, the collection of occurrence counts by range for all high cardinality attributes can be done together in one scan. In contrast, if sorting is used to collect the distribution of attribute values, each high cardinality attribute has to be sorted separately, requiring substantially more I/O overheads.

2.2 Bucket contraction

After bucket expansion, there are more small-range buckets than the required number of buckets, B. So, a combination of multiple buckets with smaller counts into a bigger one is needed such that the total number of buckets is B and the resulting bucket sizes are about equal. If attribute values do not have range semantics, e.g., no range queries are allowed against the attribute, then non-adjacent smaller size buckets can be combined. In this case, bucket contraction is relatively easy. We can first sort the small-range buckets based on the bucket count in a non-increasing order. We can then pack the largest of the remaining small-range buckets to one of the large-range B buckets with the smallest count. This heuristic is referred to in the literature as LPT (longest processing time first) due to [8]. The heuristic optimization algorithm solves the so called minimum makespan or multiprocessor scheduling problem. Although the minimum makesapn problem is known to be NP-complete, LPT is a very fast heuristic which has reasonably good worse-case performance and excellent average-case performance.

Since most attributes of continuous data types do have range semantics and typically allow range queries, we will focus on the case that only adjacent buckets can be combined. In this case, it is more difficult for bucket contraction. The problem is to combine them so that the standard deviation of the sizes of the final B buckets is minimal. We evaluate 3 different heuristics to bucket contraction in our experiments. For simple exposition, let us call the buckets after the expansion process *small-range buckets* and the final B buckets *large-range buckets*. Also, assuming that the data structure representing a bucket maintains the ranges of the bucket and its record count, and small-range buckets are represented by `b[i], i = 0, ..., E-1`, while large-range buckets are represented by `B[j], j = 0, ..., B-1`. Here, E is the total number of small-range buckets after the expansion phase. If the total number of records to be indexed is N, then the ideal large-range bucket size should be N/B.

Iterative one-bar: The first heuristic uses a bar to guide the decision of whether or not to include the next adjacent small-range bucket into the current large-range bucket. If the total record count of the resulting large-range bucket is less than the bar, then include the next small-range bucket to the current large-range bucket; otherwise, include it to a new large-range bucket. The bar is first set to be N/B, the ideal mean for a large-range bucket. Then it is raised upward by an increment of 1 in each iteration. In each iteration, the standard deviation of the B large-range bucket sizes is computed. The iteration stops when the standard deviation stops decreasing.

The iterative one-bar heuristic may cause the first few large-range buckets to be packed with too many small-range buckets such that the last few large-range buckets end up with too few small-range buckets, especially as the bar is raised. Since our objective is to have approximately equal sized large-range buckets, a second bar probably should be used to limit the first few large-range buckets to be around N/B while the second bar is raised.

Iterative two-bar: The second heuristic uses two bars. The first bar is initially set at $N/B-\delta$, while the second bar is initially set at N/B and raised by 1 at each iteration. Now the next small-range bucket can only be included in the current large-range bucket, if (a) before the inclusion, the large-range bucket count is under the first bar and (b) after the inclusion, the resultant large-range bucket count is also under the second bar. For each first bar, the iteration on the second bar stops when the standard deviation of the B bucket sizes stops decreasing. Thus, for each first bar, we obtain a set of bucket partition. The process repeats until the first bar value changes from $N/B-\delta$ to $N/B+\delta$. In our experiments, $\delta = 20\%$ of N/B, and it is incremented each time by 1. After the final value of first bar is tested, we choose the set of bucket partition with the minimal standard deviation.

One-pass zigzag: The third heuristics uses only one pass. The next small-range bucket is included in the current large-range bucket if (i) the resulting large-range bucket count is smaller than N/B or (ii) the *excess* of the resulting large-range bucket count over N/B is smaller than

Synthetic Workload Generation:

```
i = 0;
while (i < 10,000)
{
    generate a data point d, where d is a real number and 100 < d < 9900;
    i = i + 1;
    randomly generate a number p, 0 < p < 1;
    if (p > 0.999)
    {
        j = 0;
        while (j < 500 and i < 10,000)
        {
            generate a data point d_j from a normal distribution with mean d and
             standard deviation σ;
            if ((isolated skew values are needed) and (|d_j − d| < 2σ));
                d_j = d;
            i = i+1; and j = j+1;
        }
    }
}
hash the 10,000 data points into another sequence;
```

Figure 2: Pseudo code for synthetic workload generation.

the *deficit* of the current large-range bucket count from N/B if the next small-range bucket is not included. Namely, `(B[j].count + b[i].count - N/B) < (N/B - B[j].count)`, where `B[j]` is the current large-range bucket and `b[i]` is the next small-range bucket. In fact, this heuristic implicitly uses a floating upper bar and lower bar to force the large-range bucket count to zigzag around the ideal N/B records.

3 Simulation model

We implemented the naive partition, the simple bucket expansion and contraction, the dynamic bucket expansion and contraction and the optimal partition approaches. All three heuristics for bucket contraction were implemented for both simple expansion and dynamic expansion approaches. When using synthetic workloads, we conducted 100 experiments and plotted the standard deviations of the sizes of the final B buckets from each experiment. Each experiment generated 10,000 data points ranging from 100 to 9,900. These 10,000 points were to be partitioned into 20 buckets.

Fig. 2 shows the steps for generating the data points in the synthetic workloads. Note that data skew was introduced by generating 500 data points from a normal distribution with standard deviation of σ. The degree of skew was determined by σ. A σ of 400 was used to generate the low skew workload while a σ of 10 was used to generate the high skew workload. If isolated skew values are needed, we forced those data points within 2σ of the mean of the normal distribution to be the same value (see steps 12 and 13 in Fig. 2).

Note that in Step 11 a data point from the normal distribution may become greater than 9,900 or less than 100. In that case, a 9,900 was subtracted from it or a 100 was added to it. Step 18 was used to remove the position dependency introduced because of 500 data points were continuously generated by steps 6–17. Fig. 3 shows the data distributions of an instance of 10,000 data points for the low skew and high skew workloads without isolated skew values and the high skew workload with isolated skew values. For the low skew case, the histogram is relatively smooth. However, for the high skew cases, there are five spikes that really contain a lot of data points. For the case of high skew with isolated skew values, same values appeared many times in those spikes. On the other hand, for the case of high skew without isolated skew values, data points representing the spikes follow a normal distribution with $\sigma = 10$. The histogram was generated by partitioning the range between 0 and 10,000 equally into 500 bins and counting the number of data points that fall into each bin.

4 Simulation results

4.1 Synthetical workload

First we examine the impact of skew on the effectiveness of the expansion schemes assuming that there are no isolated skew values. Fig. 4 shows the three expansion schemes for the case of low skew without isolated skew values. We plotted the histogram of the standard deviations of the resultant 20 bucket sizes from each experiment. There were 100 experiments in each histogram, each experiment partitioned 10,000 data points into 20 buckets. Therefore, the optimal large-range bucket size was 500 data points. For both expansion schemes the one-pass zigzag heuristic was used for bucket contraction (the impact of the three heuristics will be discussed later on). The initial number of buckets for both simple expansion and dynamic expansion B_i was set to be $10B$, namely 200 (the impact of B_i will also be discussed later). Note that for the dynamic expansion scheme, bucket expansions were triggered when bucket counts exceeded a threshold T. For all the experiments conducted in this paper, we set $T = 0.2 \times N/B$. Namely, whenever a bucket grew over 20% of the ideal mean N/B, the bucket was ready to be expanded into smaller-range buckets.

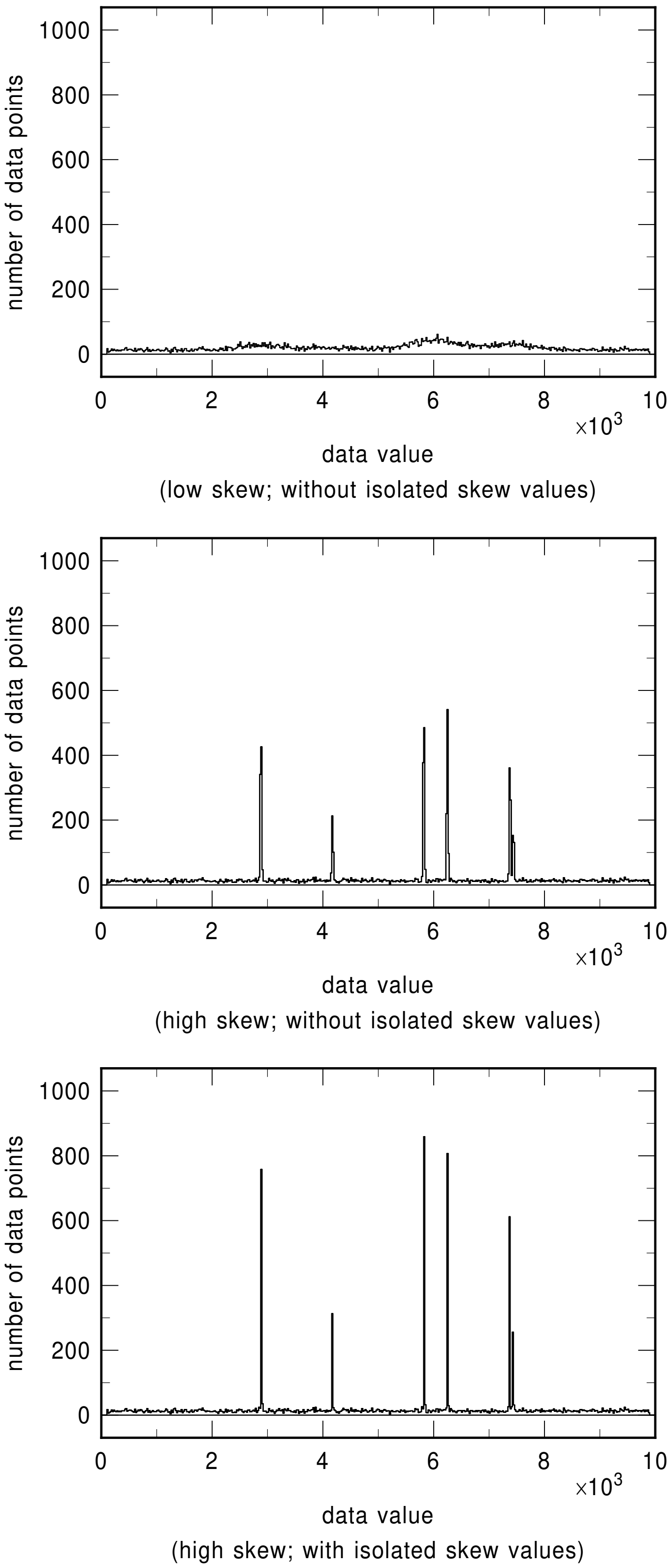


Figure 3: Data distributions for low skew and high skew cases.

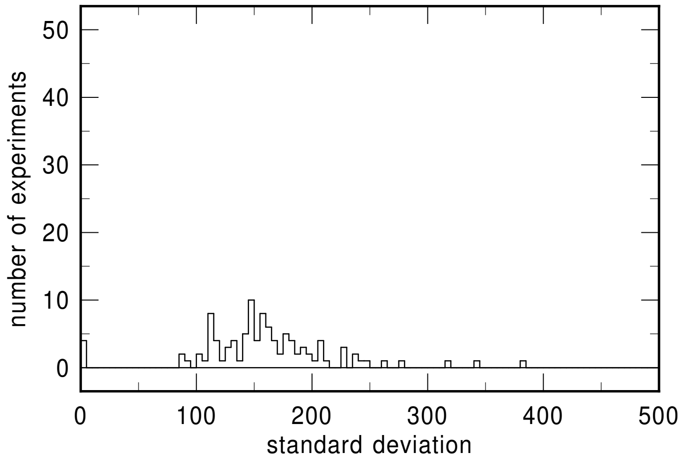
But T was adjusted according to the age of the bucket since its creation (e.g, if a new bucket was created after 10% of the data was scanned, the threshold for this bucket at the time when 20% of the data was scanned was $10\% \times 0.2 \times N/B$). Bucket sizes were checked to see if an expansion was needed after every 10% of the total data was scanned. In our experiments, every time a bucket was expanded it was expanded into s smaller buckets. We set s to be 10 for all the experiments in this paper.

From Fig. 4, it is very clearly demonstrated here that the naive partition scheme cannot even handle the low skew case because the majority of the standard deviations is spread between 100 and 300. On the other hand, the simple expansion scheme performs comparably with the dynamic expansion scheme for this low skew case. The standard deviations of bucket sizes for both cases are concentrated at around 20.

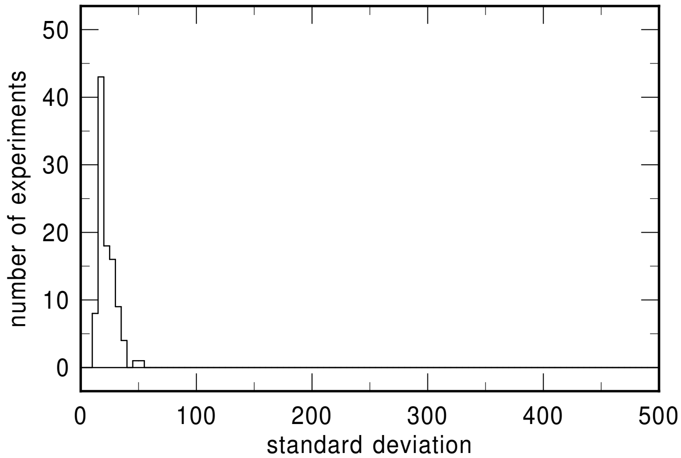
However, in the case of high skew without isolated skew values, the simple expansion scheme can no longer compete with the dynamic expansion scheme. This is illustrated very sharply in Fig. 5. In this figure, the histogram of the standard deviations for the simple expansion scheme is spread between 50 and 300, while that for the dynamic expansion scheme is still concentrated at around 20. In this high skew case, the naive partition approach is clearly not usable as it simply cannot partition the buckets evenly. For these experiments, B_i was also set to $10B$ and the one-pass zigzag heuristic was used.

The impact of B_i

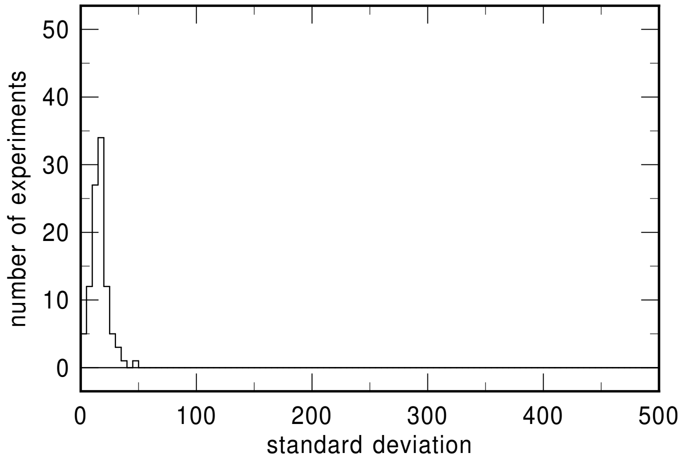
Note that in simple expansion and dynamic expansion, the initial number of buckets B_i was set to be larger than the final number of buckets B. The simple expansion scheme used these B_i buckets to collect data records in the first phase and then they were combined into B final buckets. To understand the impact of B_i on the performance of both expansion schemes, we set B_i to $2B$ and $15B$ and run the experiments. Fig. 6 shows the histograms of standard deviations of the B bucket sizes for the simple expansion scheme while Fig. 7 shows the histograms of standard deviations of the B bucket sizes for the dynamic expansion scheme. For both figures, the workloads were high skew without isolated skew values, and one-pass zigzag heuristic was used. From these two figures, we observe that B_i has *little, if any*, impact on the performance of the dynamic expansion scheme. But, it does have a significant impact on the performance of the simple expansion scheme. Certainly, the dynamic expansion scheme is less sensitive to B_i since buckets can be dynamically



(naive partition, low skew w/o isoloated skew values)

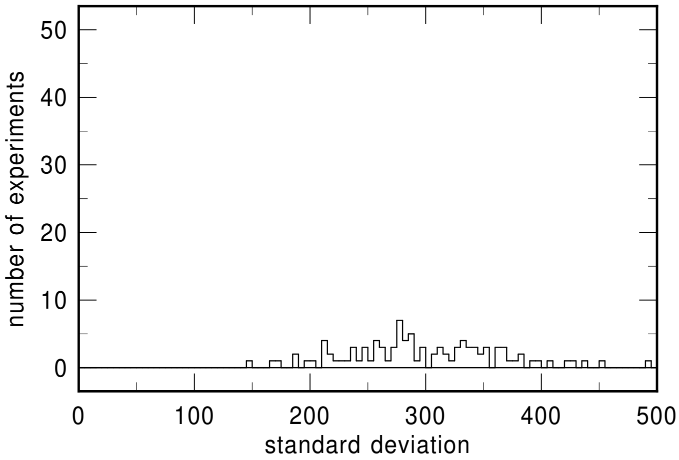


(simple expansion, low skew w/o isolated skew values)

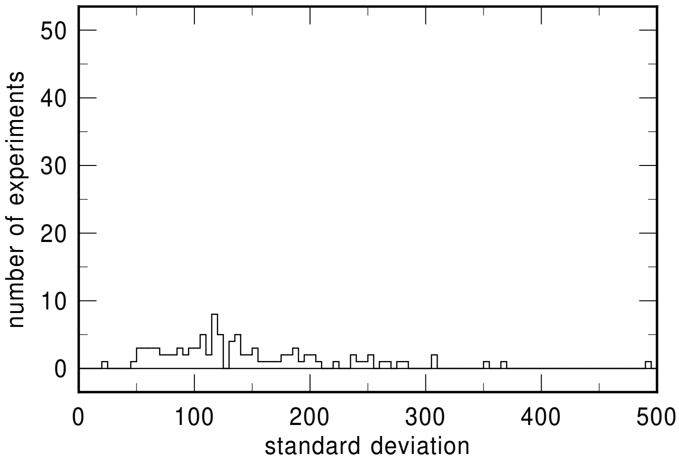


(dynamic expansion, low skew w/o isolated skew values)

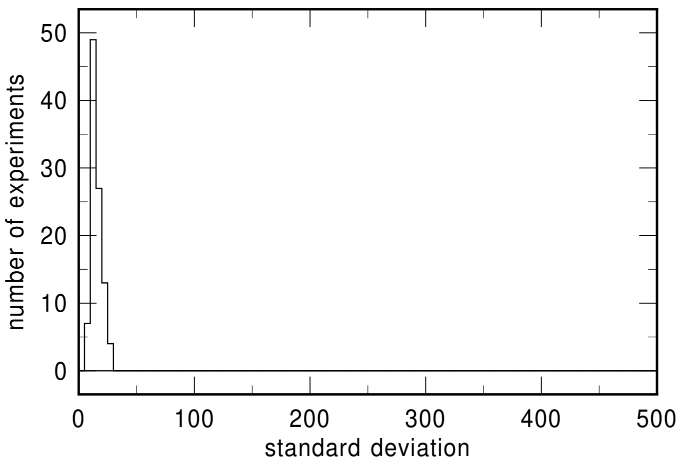
Figure 4: Comparisons of three bucket construction approaches for the case of low skew without isolated skew values.



(naive partition, high skew w/o isolated skew values)



(simple expansion, high skew w/o isolated skew values)



(dynamic expansion, high skew w/o isoloated skew values)

Figure 5: Comparisons of bucket construction approaches for the case of high skew without isolated skew values.

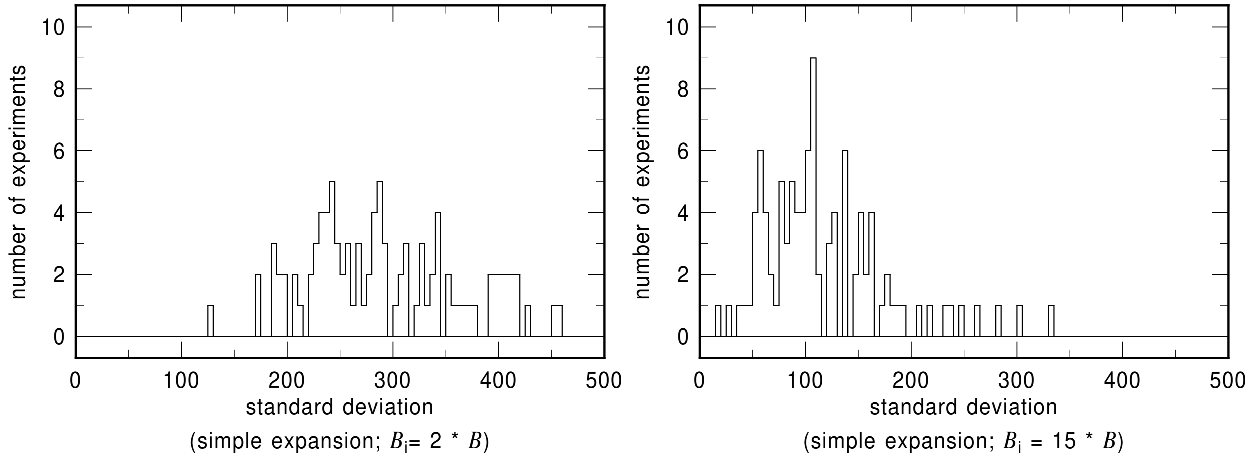


Figure 6: Impact of B_i on standard deviation for simple expansion.

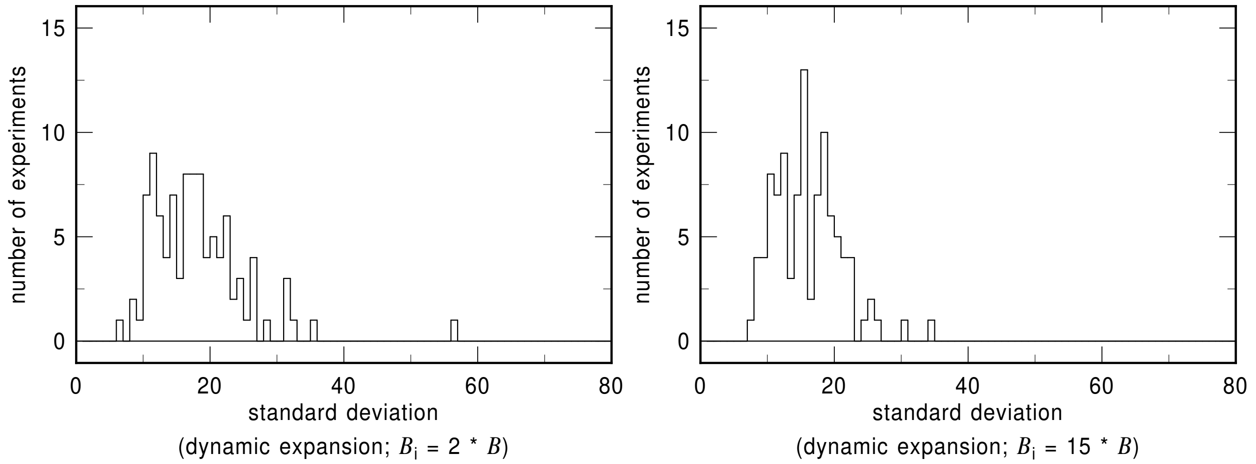


Figure 7: Impact of B_i on standard deviation for dynamic expansion.

expanded. But, it illustrates an important design issue here. Since we don't know the distribution of attribute values beforehand, we don't know how to set B_i in order for the simple expansion scheme to perform acceptably. Of course, we can always use a bigger B_i. However, if memory usage is an issue, as in the case of simultaneously partitioning the ranges of multiple attribute values, then B_i cannot be arbitrarily set. In such a case, the dynamic expansion scheme possesses a unique advantage since a small B_i can be used and still the performance will not be degraded noticeably.

Comparisons of contraction heuristics

Now we come to compare the three bucket contraction heuristics. We used the dynamic expansion approach and tested for the high skew without isolated skew values. Fig. 8 shows the histograms of standard deviations of the bucket sizes for the three heuristics. The iterative one-bar heuristic performs the worst while the one-pass zigzag the best. Note that for the best scheme, the standard deviations of bucket sizes are mostly less than 20. Since the ideal bucket size is 500, such low standard deviations show that the dynamic bucket expansion and contraction approach compare favorably with the optimal partition approach. Moreover, the dynamic expansion approach does not require costly pre-sorting while the optimal partition approach does.

The impact of isolated skew values

In this section, we look at the impact of isolated skew values. If not identified, repeated isolated skew values can result in a large number of useless expansions. These useless expansions create many empty buckets and thus waste memory space. Fig. 9 and 10 show the impact of identifying isolated skew values on the total number of bucket expansions and the effectiveness of DBEC, respectively, for the high skew workload. Without identification of isolated skew values, the number of expansions is about 250. In contrast, with identification, the number of expansions is about 10 (see Fig. 9). In addition to the reduction in unnecessary bucket expansions, bucket contraction can also be better accomplished after isolated skew values are identified. Knowing the sizes of the isolated skew values, we can allocate a separate final bucket for each of them if necessary and try to combine the rest of the smaller buckets. Fig. 10 illustrates the substantial improvement in the effectiveness of identifying isolated skew values. Without identification, the standard deviations spread between 50 and 350; with identification, they spread between 50 and 150.

4.2 Real workload

In addition to the synthetically generated workloads, we also evaluated the different bucket construction approaches using stock trading data. The trading data contain some 296 selected stocks traded on 10/18/1995 on the 3 major US stock markets. These data were obtained from the MIT experimental stock market data server at ``http://www.ai.mit.edu/stocks/prices.html''. We downloaded the trading data and removed the non-stock entries, such as the S&P 500 and other indexes. Also, we removed the stock entries that had zero trading volume. The trading data contain ticker symbols, trading volumes, and prices. The distribution of trading volumes of these 296 stocks are highly skewed but the closing prices are less skewed (see Fig. 11). Most of the trading volumes are less than 3 millions shares. But, there are two stocks with over 15 millions shares changed

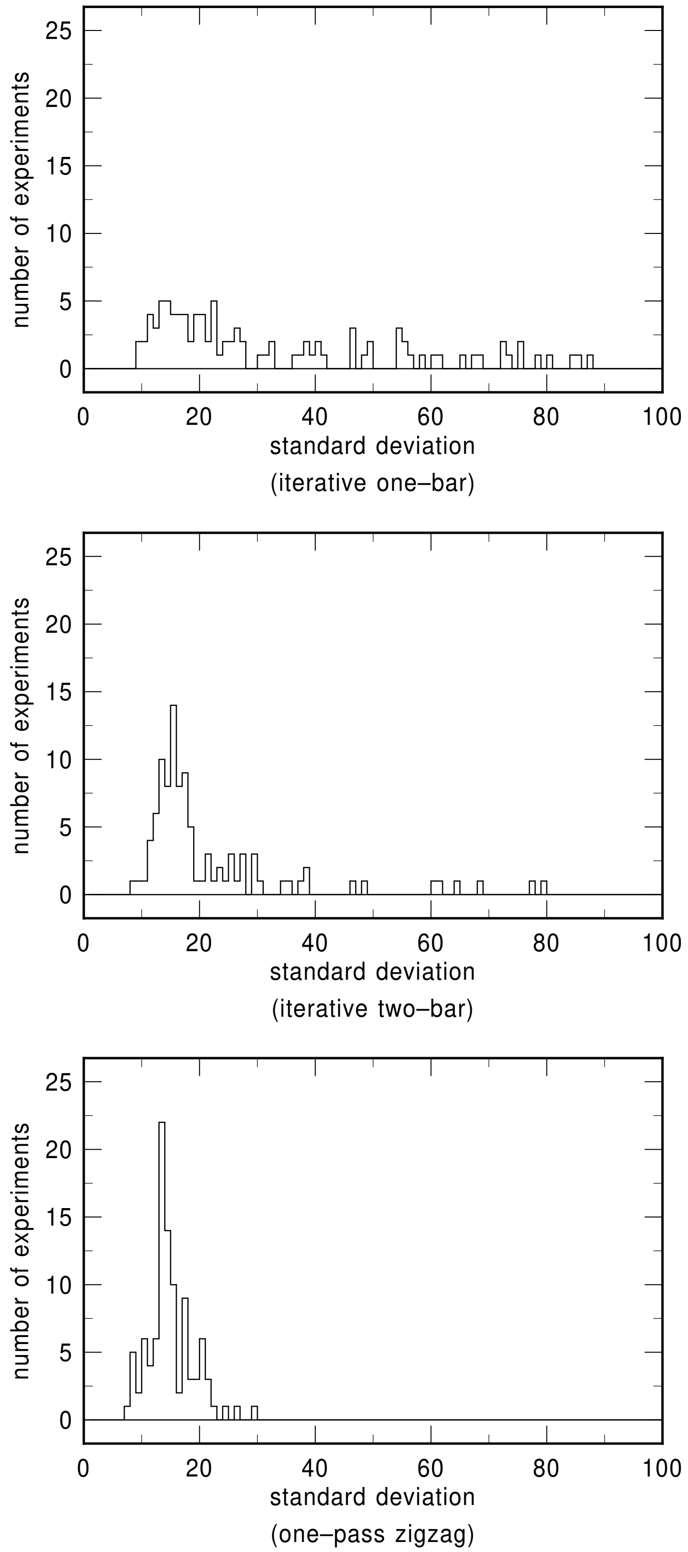


Figure 8: Comparisons of three heuristics for bucket contraction for the case of high skew without isolated skew values.

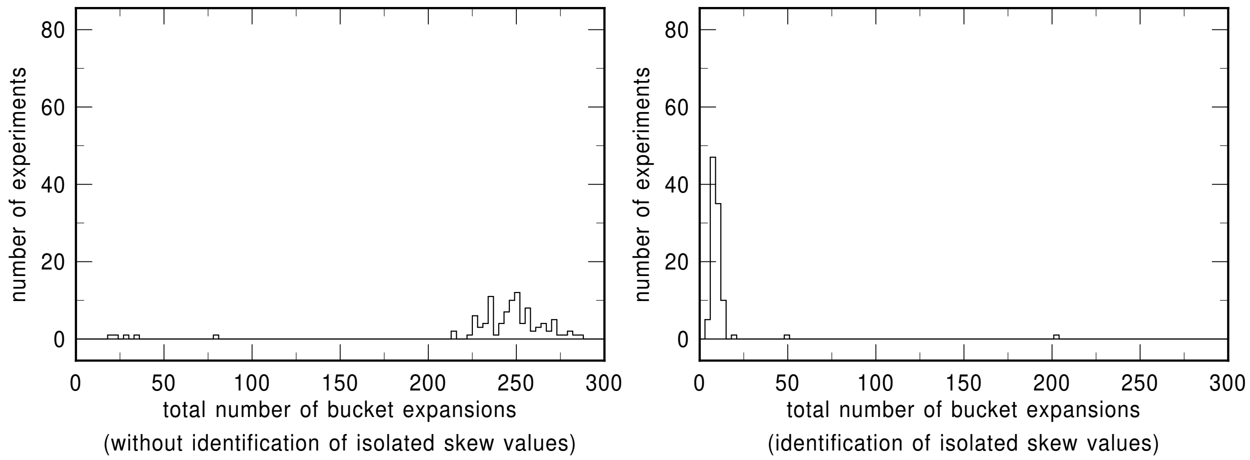


Figure 9: The impact of isolated skew values on the number of expansions.

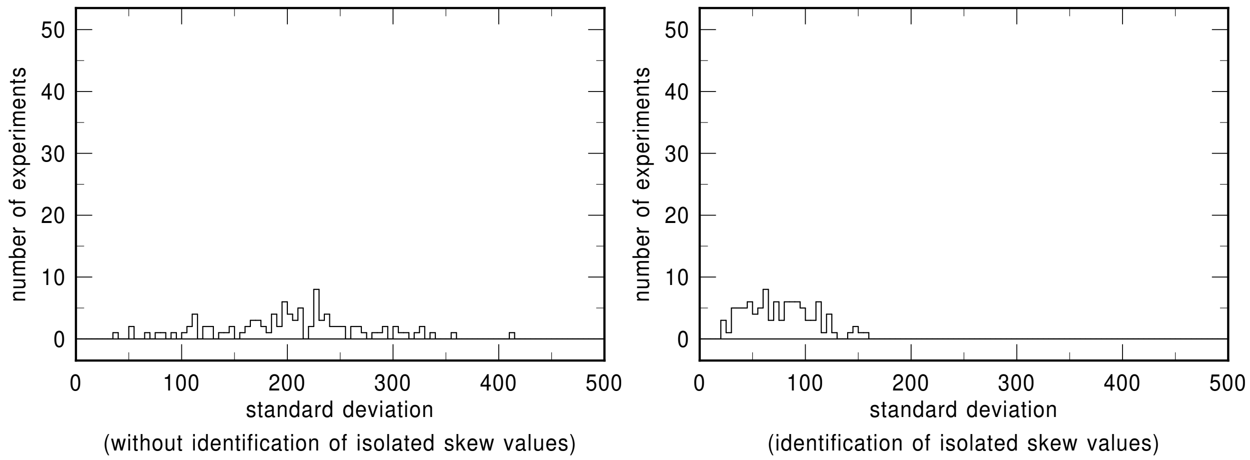


Figure 10: The impacts of identifying isolated skew values on the effectiveness of DBEC.

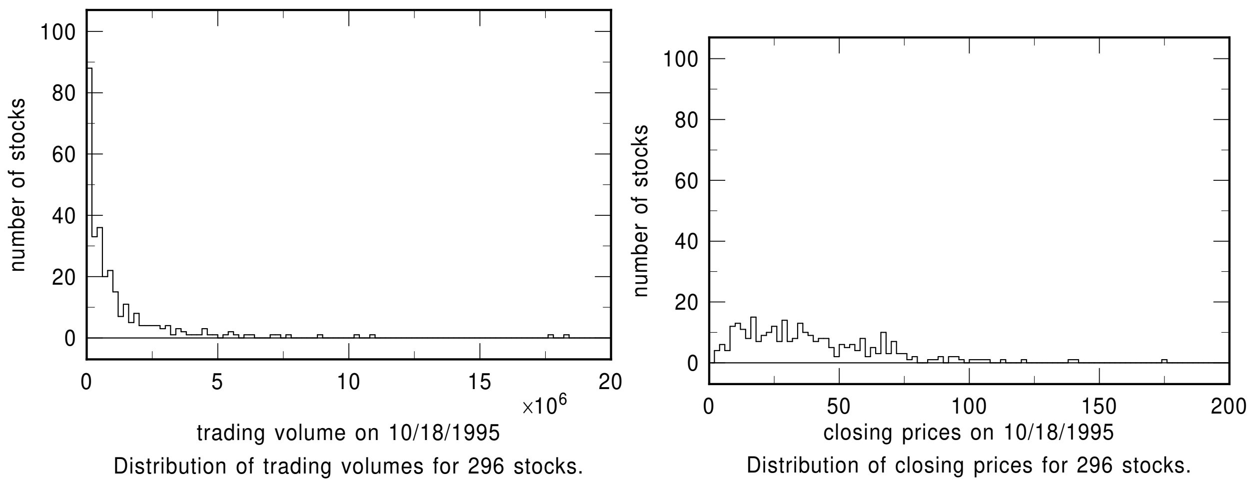


Figure 11: Distribution of stock trading volumes and closing prices of selected 296 stocks on 10/18/1995.

hands, and a small number of stocks with between 5 millions and 10 millions shares changed hands. The closing prices however are much less skewed, but still most of them are less than $75.00.

We used the trading volumes and the closing prices of the 296 stocks as input and partitioned them into 5 buckets. The results of the actual bucket sizes are shown in Fig. 12, and the actual bucket ranges are shown in Table 2 and 3. For these set of experiments, the one-pass zigzag heuristic was used in the bucket contraction for both simple expansion and dynamic expansion approaches. B_i for both simple and dynamic expansion was $10B$. Similar to the experiments using synthetic workload presented in previous section, a bucket was expanded into 10 smaller-ranged buckets in the dynamic expansion approach. Clearly indicated in Fig. 12, the naive partition scheme produces the most unbalanced buckets for both trading volumes and closing prices (bucket 1 has the most data while bucket 5 has almost none). The simple expansion scheme produces a fairly balanced buckets for the closing prices, but not for the trading volumes, which are highly skewed. The dynamic expansion approach, on the other hand, indeed produces a comparable performance with the optimal partition approach. The actual bucket ranges shown in Tables 2 and 3 give further evidence that the dynamic partition approach does partition the bucket ranges closely resemble the optimal partition approach.

5 Summary

As multidimensional data analysis is becoming heavily used for decision support systems, an efficient multidimensional access method (or index) has become increasingly important. Among

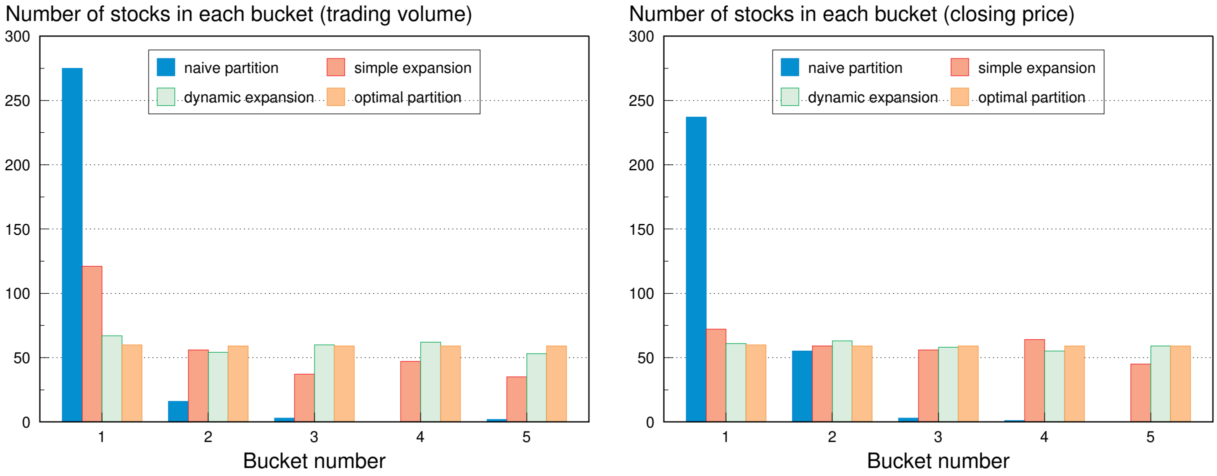


Figure 12: Actual bucket sizes for different bucket construction schemes.

Table 2: Bucket ranges produced by different bucket construction approaches on trading volumes.

	naive partition	simple expansion	dynamic expansion	optimal partition
1	[1000, 4000800]	[1000, 400980]	[1000, 132993.4]	[2600, 117300]
2	(4000800, 8000600]	(400980, 800960]	(132993.4, 400980]	(117900, 369000]
3	(8000600, 12000400]	(800960, 1200940]	(400980, 824958.8]	(394300, 801000]
4	(12000400, 16000200]	(1200940, 2800860]	(824958.8, 1888905.6]	(803800, 1773300]
5	(16000200, 20000000]	(2800860, 20000000]	(1888905.6, 20000000]	(1811500, 18288600]

Table 3: Bucket ranges produced by different bucket construction approaches on closing prices.

	naive partition	simple expansion	dynamic expansion	optimal partition
1	[0.00, 60.00]	[0.00, 18.00]	[0.00, 16.20]	[2.25, 16.00]
2	(60.00, 120.00]	(18.00, 30.00]	(16.20, 28.98]	(16.00, 27.75]
3	(120.00, 180.00]	(30.00, 42.00]	(28.98, 40.80]	(27.88, 39.50]
4	(180.00, 240.00]	(42.00, 66.00]	(40.80, 60.00]	(39.62, 59.75]
5	(240.00, 300.00]	(66.00, 300.00]	[60.00, 300.00]	[60.25, 213.62]

the multidimensional indexes, bitmap indexing is potentially the easiest to construct and maintain. However, bitmap indexing can cause a significant storage overhead if the attributes to be indexed have high cardinality. One solution to this problem is to use range-based bitmap indexing. Namely, the attribute values are partitioned into a small number of ranges and a bitmap vector is constructed to represent each range, not each distinct value. But, without knowing the distribution of the attribute values, the partition ranges can be highly uneven, resulting in highly unbalanced query access times.

In this paper, we have presented and evaluated a dynamic bucket expansion and contraction approach to constructing range-based multidimensional bitmap indexes for high cardinality attributes with skew. The data are first scanned into the buffer to construct the bucket ranges by counting the data points falling into each bucket. If a bucket grows beyond a threshold, it is expanded into more smaller-range buckets. After the scan, adjacent buckets are combined into the final required number of buckets with approximately balanced count. The bitmap vectors are then built with another scan of the data. The dynamic expansion scheme was compared with a naive partition, a simple expansion and the optimal partition approach. The results have shown that the naive partition approach cannot handle even a slightly skewed workload. The simple expansion scheme, though acceptable for low skewed data, cannot handle a highly skewed workloads. However, the dynamic expansion approach performs comparably with the optimal partition approach for any kind of workloads. Both synthetically generated workload and read data were used in the evaluations and the same results were observed for both kinds of workloads.

References

[1] S. Akyurek and K. Salem. Adaptive block rearrangement. In *Proc. of Int. Conf. on Data Engineering*, pages 182–189, 1993.

[2] J. L. Bentley. Multidimensional binary search trees in database applications. *IEEE Trans. on Software Engineering*, 5(4):333–340, 1979.

[3] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, Sept. 1975.

[4] D. J. DeWitt and R. Gerber. Multiprocessor hash-based join algorithms. In *Proc. of Very Large Data Bases*, pages 151–164, 1985.

[5] G. Evangelidis, D. Lomet, and B. Salzberg. The hB^{II}-tree: A modified hB-tree supporting concurrency, recovery and node consolidation. In *Proc. of Very Large Data Bases*, pages 551–561, 1995.

[6] M. Freeston. The BANG file: a new kind of grid file. In *Proc. of 1987 ACM SIGMOD*, pages 260–269, 1987.

[7] C. D. French. "One size fits all" database architectures do not work for DSS. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 449–450, 1995.

[8] R. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Computing*, 17:416–429, 1969.

[9] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of 1984 ACM SIGMOD*, pages 47–57, 1984.

[10] G. Hallmark. The oracle warehouse. In *Proc. of Very Large Data Bases*, pages 707–709, 1995.

[11] M. Kitsuregawa, M. Nakayama, and M. Takagi. The effect of bucket size tuning in the dynamic hybrid GRACE hash join method. In *Proc. of Very Large Data Bases*, pages 257–266, 1990.

[12] D. E. Knuth. *The Art of Computer Programming, Vol 3: Sorting and Searching.* Addison-Wesley, 1973.

[13] D. B. Lomet and B. Salzberg. The hB-tree: A multiattribute indexing method with good guaranteed performance. *ACM Trans. on Database Systems*, 15(4):625–658, 1990.

[14] M. Nakayama, M. Kitsuregawa, and M. Takagi. Hash-partitioned join method using dynamic destaging strategy. In *Proc. of Very Large Data Bases*, pages 468–478, 1988.

[15] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. on Database Systems*, 9(1):38–71, 1984.

[16] J. Robinson. The K-D-B-tree: A search structure for large multidimensional dynamic indexes. In *Proc. of 1981 ACM SIGMOD*, pages 10–18, 1981.

[17] K. Salem, D. Barbara, and R. J. Lipton. Probabilistic diagnosis of hot spots. In *Proc. of Int. Conf. on Data Engineering*, pages 30–39, 1992.

[18] H. Samet. *Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990.

[19] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A dynamic index for multi-dimensional objects. In *Proc. of 13th VLDB Conference*, pages 507–518, 1987.

[20] L. D. Shapiro. Join processing in database systems with large main memories. *ACM Trans. on Database Systems*, 11(3):239–264, Sept. 1986.

[21] Y. Zhuge et al. View maintenance in a warehousing environment. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 316–327, 1995.