Secondary Bitmap Indexes with Vertical and Horizontal Partitioning

Guadalupe Canahuate
Ohio State University
Columbus, OH, USA
canahuat@cse.ohio-state.edu

Tan Apaydin
Ohio State University
Columbus, OH, USA
apaydin@cse.ohio-state.edu

Ahmet Sacan
Ohio State University
Columbus, OH, USA
sacan@cse.ohio-state.edu

Hakan Ferhatosmanoglu
Ohio State University
Columbus, OH, USA
hakan@cse.ohio-state.edu

ABSTRACT

Traditional bitmap indexes are utilized as a special type of primary or clustered indexes where the queries are answered by performing fast logical operations supported by hardware. Answers are mapped to the physical data by using the row id of each tuple. Bitmaps represent the i-th tuple in the original table with the i-th bit position of the index. Run-length compression is used to reduce the size of the bitmaps and it has been shown that ordered data is significantly better compressed. However, for large-scale and dynamic datasets it is infeasible to keep the data always sorted. Partitioning can be used to keep the data in smaller and manageable chunks, where a different bitmap index is built for each chunk. We propose a novel bitmap index design with partitioning which serves as basis for non-clustered bitmap indexes. Individual bitmaps are not stored, only an Existence Bitmap (EB) for the existing ranks of the full table is maintained. This approach improves update performance of sorted bitmaps and does not require maintaining a heap as the underlying table, nor the same ordering for all the partitions. A one dimensional index is used over the ranks to map the bits in the EB to the physical order of the data, which allows queries to run even faster. The proposed approach, called ranked Non-Clustered Bitmaps (rNCB), is compared against traditional bitmaps using FastBit and shows significant performance gains.

1. INTRODUCTION

Bitmap indexes are widely used in data warehouses to speed up the query execution time of selection and aggregate queries. The efficiency gains are mostly due to the bitwise logical operations supported by computer hardware. They have been successfully implemented in commercial Database Management Systems such as Oracle [3, 4], Informix [9, 16], and Sybase [8, 10], among others. Traditional bitmap indexes are built using the row identifiers in the physical order of the table to set the corresponding bit based on the attribute values for the given row. For this reason, bitmap indexes can be considered as a special case of primary or clustered indexes that does not impose a physical order over the base table but rather uses the physical order of the dataset to build the index. A typical primary or clustered index would physically order the data by the index key. As opposed to primary indexes, secondary or non-clustered indexes do not explicitly use the row ids in the table and use a mapping mechanism to locate the data. Although uncompressed or verbatim bitmap indexes built over relatively smaller datasets can be managed efficiently, large scale data sets require bitmap compression to reduce the index size. In addition, the compression technique used needs to enable logical operations over the compressed form of the bitmaps in order to minimize the overhead during query execution [2, 3, 13, 22]. The general approach is to utilize run-length encoding[1] because it effectively compresses columns and does not require explicit decompression during query processing. The two popular run-length encoding based compression techniques are the Byte-aligned Bitmap Code (BBC) [3] and the Word-Aligned Hybrid (WAH) code [22].

The performance of run-length encoders depends on the presence of long runs of the same symbol. Reordering of the data have been successfully applied as a preprocessing step to increase the performance of run length encoding [11, 17]. The overall compression ratio of the bitmaps is considerably improved when data is sorted. The improvements are especially significant for the first few columns, since they contain longer and fewer runs after sorting. However, the sorting has no effect on the later columns which would only compress as if they were in random order. The number of sorted runs grows exponentially with the ordinal position of the column. An immediate implication of this unbalanced compression is that the queries involving the first columns are gaining more significant speed up compared to the others. We aim to achieve comparable performance improvement for *all* the attributes and therefore for *all* the queries over the dataset. The baseline approach is a divide-and-conquer solution that distributes the attributes into smaller subsets and sorts the bitmaps for each *partition* independently. However, in this scenario the row ids can no longer be represented by the bit positions in the bitmaps because the bitmaps are sorted independently of the physical ordering of the dataset. There-

EDBT 2009, March 24–26, 2009, Saint Petersburg, Russia.

[1] Run-length encoding is the process of replacing repeated occurrences of a symbol by a single instance of the symbol and a count.

fore, a mapping table is needed to translate bit positions into row ids. With the use of mapping tables, one can generate as many sorted secondary bitmap indexes as needed. With sorted secondary bitmap (SSB) indexes, there is a clear improvement in the compression performance of the bitmap columns. However, we cannot claim the same for query execution performance. Even when the queries execute faster for a set of attributes sorted together, executing queries involving several partitions would introduce significant overhead. The reason is that one secondary bitmap needs to be translated into a primary bitmap using a bit-by-bit mapping, i.e. each bit set in the secondary bitmap needs to be set in the corresponding position of the primary bitmap. Therefore, in addition to the bitwise operations, we need to account for the mapping into row identifiers of all the bits set to 1 in the partial answers.

In this paper, we investigate whether it is possible to have the best of both worlds: improved compression and improved execution time for *all* the bitmap columns of the attributes in a table. The key to our answer is to remove all redundancy from the bitmaps. We propose a new non-clustered bitmap index organization utilizing advantages of horizontal and vertical partitioning with no significant overhead. With horizontal partitioning, tuples are segmented to ensure that partitions fit into main memory. With vertical partitioning the dataset is divided into sets of attributes and one logical index is created for each set. As a result, each partition has its own secondary bitmap index. Instead of storing each bitmap column in the partition we only store one bitmap column for the whole partition and instead of having one bit per tuple in the bitmap we only have one bit per distinct value or rank. The proposed approach is called ranked Non-Clustered Bitmap (rNCB) as it indexes the rank of a tuple in a sorted bitmap table. As opposed to clustered bitmaps, the bitmap columns are not stored and there is only one bitmap column per partition. The bitmap corresponds to an existence bitmap (EB) of the distinct ranks of the tuples, i.e. the rows from the full bitmap table present in the data, which after partitioning is considerably smaller than the number of tuples in the dataset. By controlling the number of attributes in each partition, we control the size of the full bitmap table and therefore, guarantee faster query execution than primary bitmaps. We formalize and describe the query execution logic over the proposed architecture to ensure correct results. Queries are executed against the full bitmap table which is not stored but rather computed on the fly during query execution time. The resulting bitmap for each partition is then translated into a primary bitmap. The query result for each partition is exactly the same as with traditional bitmaps. Primary bitmaps are ANDed between partitions to obtain the final answer bitmap.

The rest of this paper is organized as follows. Related work and background information is presented in Section 2. The proposed approach is described in Section 3. Experimental results are presented in Section 4. Finally, conclusions are presented in Section 5.

2. BACKGROUND

Bitmap tables are a special type of bit matrices. Each binary row in the bitmap table represents one tuple in the database. The bitmap columns are produced by quantizing the attributes in the database into categories or bins. Each tuple in the database is then encoded based on which bin each attribute value falls into.

For the simple bitmap encoding (also called equality encoding) [16], if a value falls into a bin, this bin is marked "1", otherwise "0". Since a value can only fall into a single bin, only a *single* "1" can exist for each row of each attribute. After binning, the whole database is converted into a 0-1 bitmap table, where rows correspond to tuples and columns correspond to bins. Figure 1 shows an example of the equality encoded bitmap using a table with two attributes, each partitioned into three bins. The first tuple t_1 falls into the first bin in attribute 1, and the third bin in attribute 2. There are several other encoding techniques for bitmaps, such as range [6], interval [7], and workload and attribute distribution oriented [14] encoding.

	Attribute 1			*Attribute 2*		
Tuple	b1	b2	b3	b1	b2	b3
t_1	1	0	0	0	0	1
t_2	0	1	0	1	0	0
t_3	0	0	1	1	0	0
t_4	0	1	0	0	0	1
t_5	0	0	1	0	0	1
t_6	0	0	1	0	1	0
t_7	1	0	0	1	0	0
t_8	0	1	0	0	1	0
t_9	1	0	0	0	1	0

Figure 1: Simple bitmap example for a table with two attributes and three bins per attribute.

Bitmap indexes can provide very efficient performance for point and range queries thanks to fast bit-wise operations over the bitmaps, which are efficiently supported by hardware.

With equality encoded bitmaps a point query is executed by ANDing together the bit vectors corresponding to the values specified in the search key. For example, finding the data points that correspond to a query where Attribute 1 is equal to 3 and Attribute 2 is equal to 5 is only a matter of ANDing the two bitmaps together. Equality Encoded Bitmaps are optimal for point queries [7]. Range queries are executed by first ORing together all bit vectors specified by each range in the search key and then ANDing the answers together. If the query range for an attribute queried includes more than half of the cardinality then the query is executed by taking the complement of the ORed bitmaps that are not included in the range queried.

No matter which bitmap encoding we use, the bitmap index table is a 0-1 table. This table needs to be compressed to be effective on a large database. General purpose text compression techniques are clearly not suitable for this purpose since they significantly reduce the efficiency of queries [12, 21]. Specialized bitmap compression schemes have been proposed to overcome this problem. These schemes are based on run-length encoding, i.e., they replace runs of 0's or 1's in the columns by a single instance of the symbol and a run count. These methods not only compress the data but also enable fast bitwise logical operations, which translates into faster query processing.

Run length encoding [18] can therefore be used over every column to compress the data when long runs of "0" or "1" blocks become available. Pure run length encoding is not a good strategy because of its accessing inefficiency. The two most popular compression techniques for bitmaps are the Byte-aligned Bitmap Code (BBC) [3] and the Word-Aligned Hybrid (WAH) code [22]. Unlike tradi-

tional run length encoding, these schemes mix run length encoding and direct storage. BBC stores the compressed data in Bytes while WAH stores it in Words. WAH is more CPU efficient because it only has two types of words, literal and fill words, and it only accesses the index by words. Let w denote the number of bits in a word, the lower $(w-1)$ bits of a literal word contain the bit values from the bitmap. If the word is a fill, then the second most significant bit is the fill bit, and the remaining $(w-2)$ bits store the fill length. WAH imposes the word-alignment requirement on the fills. This requirement is key to ensure that logical operations only access words.

FastBit [20] is an open source software tool that uses WAH compressed and verbatim bitmaps to support SQL-like queries, such as selection queries. The design choices of FastBit are proven to be effective when compared to other bitmap indexing methods [15].

rNCB can be implemented within any relational database system regardless of the approach taken to store the base tables. rNCB is not restricted to traditional row stores, column store systems can also adopt rNCB. For example consider C-Store [19]. C-Store is a read-optimized relational DBMS such that the storage of data is by columns rather than rows, and data is stored in an overlapping collection of attribute-projections. C-Store already implements bitmap indexes to speed up the query execution. NCB can improve query performance of C-Store as a multidimensional index over each projection. Our mapping table can be implemented as a join index. Join indexes are used to obtain results across projections and can be easily adapted to our needs.

3. APPROACH

In this section, we present the proposed approach that combines the advantages of the representation of the sorted data and the efficient query execution of bitmap indexes. We first provide the technical motivation for the proposed approach and then formally describe the components of our system.

3.1 Technical motivation

Consider a relational table D with d attributes and n tuples. The cardinality, i.e., number of distinct values in the range, of attribute A_i is denoted by c_i. Often, the domain of A_i is quantized into bins before creating the bitmap index. In those cases c_i refers to the number of bins for attribute A_i.

For traditional bitmaps, a bitmap table B is created using one column for each attribute bin and one row for each tuple. The number of rows in the bitmap table is denoted by n, and the number of columns B_c is given by the sum of the cardinalities:

$$B_c = \sum_{i=1}^{d} c_i$$

A bitmap table B is called a **full bitmap table** if the number of the distinct tuples B_n is equal to the Cartesian product of all the values of the attributes in the table. The number of distinct tuples in a full bitmap table is given by the product of the cardinalities:

$$B_n = \prod_{i=1}^{d} c_i$$

For example, consider the full bitmap table presented in Figure 2, all possible *distinct ranks* for a table with 3 attributes each with

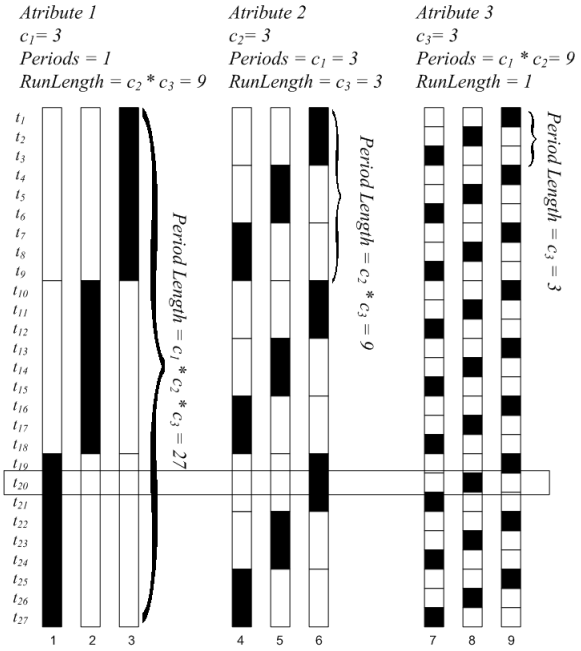

Figure 2: Full Sorted Bitmap Table with 3 attributes and 3 bins per attribute. The white blocks represent 0s and black blocks represent 1s.

cardinality of 3. The number of columns B_c = 3+3+3 = 9, and the number of tuples (*distinct ranks*) B_n = 3·3·3 = 27.

A full bitmap table B is sorted if the tuples are in lexicographic order of the columns, e.g. the table in Figure 2 is a sorted full bitmap table. The sorting order of the columns refers to the order in which the columns were evaluated in the ordering. Different sorting orders of the columns produce different permutations of the tuples under the same ordering criteria. Unless otherwise noted, the sorting order of the columns used in this paper is the order in which the attributes appear in the dataset.

In the full bitmap table B, a tuple t is identified by the set of bin numbers of each attribute value, i.e. $t = \{b_{t1}, b_{t2}, ..., b_{td}\}$, where b_{ti} refers to the bin number of the value of attribute A_i. For instance, in Figure 2 the tuple with rank 20 can be identified as $t_{20} = \{1, 3, 2\}$. The **rank** $\pi(t)$ of tuple t refers to the position of t in the full ordered bitmap table B and is given by:

$$\pi(t) = 1 + \sum_{i=1}^{d}((c_i - b_{ti}) \cdot \prod_{j>i}^{d} c_j)$$

For example, the rank of t_{20} can be computed as $\pi(t_{20}) = 1+(3-1)\cdot(3\cdot 3)+(3-3)\cdot(3)+(3-2)\cdot 1 = 20$.

The main consideration is that given the attribute cardinalities and their sorting order, the actual value of the attributes can be derived from the rank of the tuple. In other words, the rank function is one-to-one and this property, i.e. mapping from a given rank to an attribute value, is used in our query execution algorithm which is described later.

Attribute	Column	PL	$Start_{Run1}$	End_{Run1}
1	1	27	19	27
1	2	27	10	18
1	3	27	1	9
2	1	9	7	9
2	2	9	4	6
2	3	9	1	3
3	1	3	3	3
3	2	3	2	2
3	3	3	1	1

Table 1: Summary for the Full Sorted Bitmap Table in Figure 2.

Summarizing the Sorted Full Bitmap Table. Few elements are needed to summarize the sorted full bitmap table as there is a clear pattern on the organization of the table when lexicographic order is used. For simplicity of the analysis let us consider that all the attributes have the same cardinality.

Columns can be considered to be *periodic*, i.e. the pattern repeats itself after a certain number of bits. The number of periods in a column depends on the number of attributes preceding this column. For columns 1-3 (first attribute) in Figure 2, there is only one period since the runs never repeat. For columns 4-6 (second attribute) there are three periods (c_1), and for columns 7-9 (third attribute) there are 9 periods ($c_1 \cdot c_2$). Note that the number of periods p_i for an attribute A_i is given by the product of the cardinalities of the preceding attributes. The period length (number of tuples in a period) can be computed as B_n/p_i. For example, for the columns of *Attribute 2* in Figure 2, the period length is $27/3 = 9$.

Notice that there is only a single run of 1s in each period. The total number of runs in a period is either 2 (when the period starts or ends with 1s) or 3 (when the run of 1s is somewhere in the middle of the period). The length of the run of 1s in each period depends on the cardinalities of the following attributes. For columns 7-9, the run length is 1 (as this is the last attribute), for columns 4-6 the run length is 3 (c_3), and for columns 1-3 the run length is 9 ($c_2 \cdot c_3$). So far, the number of periods, the length of the periods, the length of the runs of 1s are all the same for the columns of an attribute. The only difference between the columns of one attribute is the position of the runs of 1s.

For each bitmap column i of Attribute j, three values are stored that enable efficient computation of the bit values of a rank:

- $Period\ length$ (PL), which is the number of tuples before the pattern of the runs repeats itself.

$$PL = \prod_{k=j}^{d} c_k$$

- $Start_{Run1}$, which is the start position of the run of 1s within a period.

$$Start_{Run1} = 1 + (c_j - i) \prod_{k=j+1}^{d} c_k$$

- End_{Run1}, which is the end position of the run of 1s within a period.

$$End_{Run1} = (c_j - i + 1) \prod_{k=j+1}^{d} c_k$$

The summary of the bitmap table consists of the cardinalities of the attributes, the sorting order of columns, and the previously described elements. Table 1 shows the summary of the sorted full bitmap table of Figure 2. With this structure, finding the bit value of a column for a given rank is actually no more complicated than translating a number from decimal to binary format. The main difference is that the base used to translate the number changes as we move between the attributes, i.e. the base used is derived from the cardinalities of the following attributes.

Deriving the bit value b for column i of attribute A_j for rank $\pi(t) = r$ reduces to testing whether the rank is within the run of 1s for the corresponding period:

$$b = ?(\ PL \cdot p + Start_{Run1}\ \leq\ r\ \leq\ PL \cdot p + End_{Run1})$$

where the period p for rank r is computed as $\lfloor r/PL \rfloor$. As an example, let us consider the summary in Table 1. Given the rank of a tuple, e.g. $\pi(t) = 10$ (t_{10} in Figure 2), the bit value of a bitmap column, e.g. the first bin for the second attribute (fourth row in Table 1), can be derived by computing $p = \lfloor 10/9 \rfloor = 1$ and evaluating whether r falls into the run of 1s for that period. Since 10 is not between 16 $(9 + 7)$ and 18 $(9 + 9)$, the bit value is 0.

As can be seen, it is possible to store the summary of the full sorted bitmap table and answer queries about the values of a particular tuple efficiently from the rank of the tuple. By storing the ranks of the tuples and the mapping from the tuples to the original row ids, we can answer selection queries by scanning the existing ranks in only one pass.

Partitioning. Given a database D with d attributes, there are two possible partitioning directions: vertical and horizontal partitioning.

Vertical partitioning, $A = V(D)$, also called a projection, refers to a subset of the attributes in D such that $|A| \leq d$.

Horizontal partitioning, $T_A = H(D)$, refers to a condition to distribute the tuples in D for the attributes in A. Horizontal partitioning can be done, for example, on ranges of some attribute, the row identifier, or hash values.

A **Partition** $P = \{A, T_A\}$ is a pair of a set of attributes A and a horizontal partitioning criterion T_A. Each set of tuples produced by T_A in partition P is called a **segment**.

For simplicity, we assume that each attribute is in exactly one partition and there is no overlap between partitions, therefore the union of all partitions produces D.

3.2 System Overview

In this section, we describe the system components of the proposed approach. Figure 3 shows the system overview that includes an index and a query engine. The index has two parts: a mapping table and an Existence Bitmap (EB). One index is built for each partition. The mapping table translates ranks into row identifiers. The EB indicates which ranks are present in the dataset. The query engine has three functional units: the query planner, the query executor, and the bitmap translator. The query planner identifies the bitmaps needed to answer the query and the partitions involved in the query. The query executor computes the values queried in the logical bitmap on-the-fly, as the individual bitmaps are not stored.

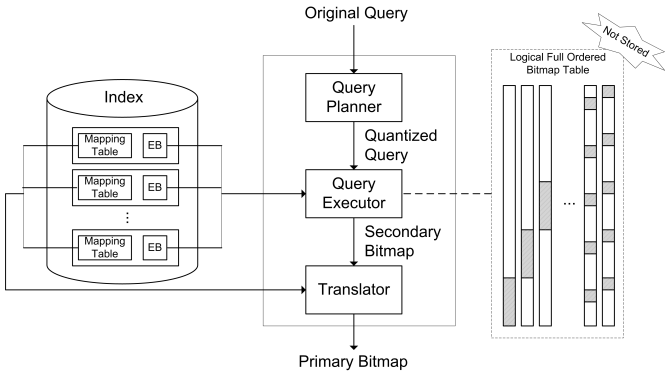

Figure 3: System Overview of the proposed approach.

The translator converts the secondary (logical) bitmap into a primary bitmap, i.e. the bit positions in the bitmap refer to the row identifiers in the physical order of the data. Each component is described in detail below.

3.3 Index Structure

The ranked non-clustered bitmap index (rNCB) has two elements: a mapping table and an Existence Bitmap.

The **Mapping Table** (MT) enables the translation from bit positions to row identifiers. This mapping mechanism is mandatory for any secondary index as, for most queries, it is necessary to access the actual data to answer the query. There are many possible implementations for the mapping table. A simple and update efficient way is to use a B+Tree [5]. The size of the mapping table is $O(N \log N)$, where N is the number of rows in the table.

The **Existence Bitmap** (EB) indicates which tuples from the full ordered bitmap exist in the partition. The EB can be stored in two different formats: as a WAH compressed bitmap or as a sorted list of ranks (*rankList*). The EB is stored as a rankList when it is sparse, i.e. there is only one bit set in each literal word. Otherwise, the EB is stored as a one-sided WAH compressed bitmap, i.e. only the zeros are compressed. The reason for this kind of compression is that the query is executed inplace over the EB for improved performance. The criterion to decide how to store the EB is simple, yet effective. The decision is made based on the comparison of the number of words of the compressed EB and the number of distinct ranks. If the number of words in the compressed EB is less than the number of distinct ranks, then the EB is stored as a compressed bitmap. The implication is that, when the EB is stored as WAH compressed bitmap, on the average there are at least two bits set in each literal word, as we can assume that for large datasets there would be a fill word of zeros between every literal word.

No matter how the EB is stored, queries are answered with a simple sequential scan of the EB. When the EB is stored as a rankList one rank is processed at a time. However, when the EB is stored as a WAH compressed bitmap, we operate at the word level and potentially evaluate $w - 1$ existing ranks at a time.

The size of the EB for a partition with N_s distinct ranks when the EB is stored as a rank list is $O(N_s \log_2 N)$ and $O(N_s)$ when stored as a WAH compressed bitmap.

To create the rNCB index, we generate the summary of the full sorted bitmap table using the order of the attributes and the attribute cardinalities. Then, by scanning the dataset we compute the rank of each tuple. The rank is set in the EB and the row id is inserted into the mapping table using the rank as the key value.

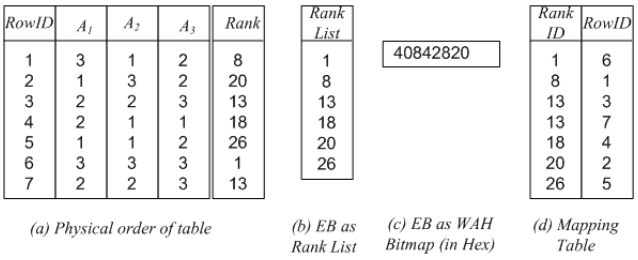

Figure 4: An example of (a) a table with 7 tuples, the corresponding EB stored as (b) a rankList and as (c) a WAH compressed bitmap, and (d) the mapping table.

As an illustration of the proposed index, consider the structures in Figure 4. Figure 4(a) presents a table with 7 rows and 6 distinct ranks out of the all possible 27 ranks in Figure 2. Figure 4(b) and (c) depict the corresponding EB stored as a rankList and a WAH compressed bitmap, respectively. Duplicates are not stored in the rank list. For the EB stored as a WAH compressed bitmap, the bit positions set correspond to the existing ranks. Finally, the mapping table shown in Figure 4(d) enables the translation between the ranks and the actual row ids in the table.

3.4 Query Engine

A selection query is a set of conditions of the form A *op* v, where A is an attribute, *op* in $\{=, <, \leq, >, \geq\}$ is the operator, and v is the value queried. We refer to point queries as the queries that use the equal operator ($A = v$) for all conditions and range queries to the queries using a *between* condition ($v_1 \leq A \leq v_2$).

In a bitmap based system, queries can be thought as a set of bitmaps for a subset of the attributes. The queried values, v_i, are quantized to identify the corresponding bitmap bins, b_i. If the bitmaps correspond to the same attribute then the resulting bitmaps are ORed together, otherwise they are ANDed together.

The **Query Planner** quantizes the query and transforms it into an execution plan by identifying the attribute bins that need to be accessed and determining the number of partitions and/or segments involved in the query.

The **Query Executor** takes the parsed query and scans the corresponding EB.

In contrast to traditional bitmaps, the query execution is done row wise within a partition, not column wise. For simplicity of the following discussion, let us assume that the EB is stored as a rankList. When a rank is read from the EB, the query executor decides whether that rank satisfies all query conditions or not by evaluating each attribute queried. If the rank satisfies all conditions, then the rank is translated into row ids and the corresponding bit positions are set in a primary bitmap. However, if the rank does not satisfy any one of the query conditions, it is discarded and no further evaluation is needed.

A powerful optimization comes from the fact that the ranks are

```
Execute_RankList(P,Q)

1:  B = All zeros primary bitmap
2:  colValue = {0, −1}^qDim
3:  for each rank r in the EB
4:      isAnswer = true
5:      for each attribute A_i in Q
6:          if r > colValue[i].validityRank
7:              p = r/A_i.PL
8:              start1s = A_i.PL * p + A_i.Start_Run1
9:              end1s = A_i.PL * p + A_i.End_Run1
10:             if start1s ≤ r ≤ end1s
11:                 colValue[i]= {1,end1s}
12:             else if r <start1s
13:                 colValue[i] = {0, start1s-1}
14:             else
15:                 colValue[i] = {0,start1s + A_i.PL - 1}
16:         if colValue[i].bit = 0
17:             isAnswer = false
18:     if isAnswer
19:         C = translate(r)
20:         B = B OR C
21: return B
```

Algorithm 1: Query execution algorithm for EB stored as rankList. P is the partition and Q is the query representation.

processed in sorted order and the start and end positions of the run of 1s are known. When a rank is evaluated for a given column, the derived bit value is stored together with a *validity rank*. The validity rank is nothing more than the rank at which the current run ends for that column. For example, consider again the rNCB in Figure 4 and a query asking for the first column of Attribute 1. The first rank evaluated is 1. Rank 1 evaluates to 0 in that column because the run of 1s start at position 19 (see Table 1). The value 0 is stored together with a validity rank of 18, i.e. the position at which the run of 0s end. When the next rank (8) is evaluated, it is first compared with the validity rank of the previous result. Since 8 is less than 18, it means that the result is still valid and can be used as the answer.

To fully take advantage of this optimization, attributes in the query are arranged in the same order as the sorting order during query parsing. The reason is that the earlier columns would have larger validity ranks. The worse case performance of rNCB is given when the previous result is never reused and computation of the values is needed for each rank, e.g. when the columns queried involve only the last attribute in the sorting order.

Algorithms 1 and 2 present the pseudocode for query processing when the EB is stored as a rankList and a WAH compressed bitmap, respectively. Both algorithms are simplified for clarity of the presentation assuming only one segment in the partition.

In Algorithm 1, the primary bitmap B is initialized as an all zeros bitmap (Line 1) and the column values (colValue) are initialized to bit value 0 and validity rank -1 (Line 2). Then, the EB is scanned (Line 3). For each rank r, the variable isAnswer is initialized to true (Line 4) and is used to indicate whether the tuples with rank r satisfy the query. Each attribute in the query is evaluated (Line 5). If the column value is not valid for rank r then a new value is derived (Line 6). The start and end positions of the run of 1s is computed for period p (Line 7) in which the rank r lies (Lines 8-9). Then, if r lies within the run of 1s (Line 10) the result for this column is set to 1 and the validity of the result is set to end of the

```
Execute_WAH(P,Q)

1:  B = EB
2:  wordNumber = 0
3:  colValue = {0, −1}^qDim
4:  for each word w_i in the EB
5:      decode (w_i)
6:      if w_i.isFill
7:          wordNumber = wordNumber + w_i.nWords
8:      else
9:          word = w_i
10:         for all attributes A_j ∈ Q
11:             if wordNumber > colValue[j].validity
12:                 startRank = wordNumber*w+1
13:                 endRank = wordNumber*(w+1)
14:                 if startRank and endRank fall into the same period
15:                     colValue[j].word = getWord (A_j, startRank, endRank)
16:                     if (colValue[j].word==0)
17:                         if endRank mod A.PL < A.Start_Run1
18:                             colValue[j].validity = word where
                                the run of 1s starts
19:                         else
20:                             colValue[j].validity = word where
                                the period ends
21:                     else if (colValue[j].word = 0x7FFFFFFF) //All 1s
22:                             colValue[j].validity = word where
                                the run of 1s ends
23:                 else //More than one period
24:                     colValue[j].word = 0
25:                     iterate over each period
26:                         compute startRank and endRank for the period
27:                         colValue[j].word = colValue[j].word OR
                            getWord(A_j, startRank, endRank)
28:             word = word AND colValue[j].word
29:         wordNumber++
30:         if word = 0
31:             break
32:         B[i] = word
33:     return translate(B)
```

Algorithm 2: Query execution algorithm for EB stored as WAH compressed bitmap. P is the partition and Q is the query representation.

runs of 1s (Line 11). However, if r lies outside the run of 1s then the answer is set to 0 and r is evaluated to decide whether it falls within the first run of 0s (in which case the validity rank is set to the start position of the runs of 1s) or it falls within the second run of 0s (in which case the validity rank is set to the start of the run of 1s in the next period) (Lines 12-15). If a bit value is 0 for a given attribute then the loop stops and r is not an answer to the query (Lines 16-17). If after evaluating all the queried columns isAnswer is still true, then the rank is translated into a primary bitmap C.

The **Bitmap Translator** (*translate* method in Line 19) is given a rank and returns a primary bitmap, i.e. a bitmap with set bits corresponding to the positions of the row ids of the tuples that map to the given rank. The bitmap translator is the only process that accesses the mapping table or B+Tree. For clarity of the presentation, we made C a bitmap, but for efficiency we implemented it at the word level, and the OR is only done between B and the non-zero words of C. Finally, the primary bitmap B is returned as the answer. For performance reasons, the row ids in the mapping table are stored as two numbers, word number and bit offset, to facilitate the translation of the rank into the bit positions in the primary bitmap. The word number is given by $\lfloor row\_id/word\_size \rfloor$ and the bit offset is given by ($row\_id \mod word\_size$).

```
getWord(A,start,end)

1:  word = 0
2:  runs = {0x1, 0x3, 0x7, 0xF, 0x1F, ..., 0x7FFFFFFF }
3:  period = start/A.PL
4:  startRun = A.PL*(period)+A.Start_Run1
5:  endRun = A.PL*(period)+A.End_Run1
6:  if (start <= endRun AND end >= startRun)
7:     start = Max(start,startRun)
8:     end = Min(end,endRun)
9:     if (start mod w == 0)
10:       word = runs[end mod w]
11:    else
12:          word = runs[end mod w] AND NOT(runs[(start mod w)-1])
13: return word
```

Algorithm 3: getWord algorithm returns an integer (word) with the bit values for ranks between start and end.

As an example, consider the previous query asking for the first column of Attribute 1. The first four ranks, {1,8,13,18}, evaluate to 0. When rank 20 is evaluated the result is 1 with a validity rank of 27. The query translator seeks rank 20 in the mapping table to obtain row id = 2 (word 0, offset 2). Then the first word in the bitmap B is ORed with a word with only the second bit set. Finally, rank 26 uses the result computed for rank 20 and it is translated into row id 5. The final answer (in hex) is the primary bitmap B= $\{24000000\}$.

Algorithm 2 presents the pseudocode for the query execution when the EB is stored as a WAH compressed bitmap. The bitmap B is initialized as the EB (Line 1), *wordNumber* is initialized to 0 (Line 2), and the column values (colValue) are initialized to word value 0 and validity word -1 (Line 3). The EB is accessed by words (Line 4). The key is to keep the word number (*wordNumber*) in the verbatim bitmap of the current word, w_i, in order to be able to correctly derive the ranks included in this word. If the word is a fill word then *wordNumber* is incremented by the number of words encoded by the fill (Lines 5-7) and the next word is read from the EB. If w_i is a literal (Line 8), then the validity rank of the previous result is compared against *wordNumber* (Line 11). If the result is not valid, a new value needs to be computed. *wordNumber* is used to compute the start and end ranks in the word (Lines 12-13). If both ranks fall into the same period then the bit values for the ranks between start and end can be derived in a single call to the *getWord* method. The validity word is derived, similarly to the validity rank when the EB is stored as a rankList, to be either the start or end of the run of 1s or the end of the period (Lines 16-22). If the start and end ranks fall into different periods (Line 23), then for each period we compute start and end ranks and call the *getWord* method. These *partial* results are ORed together to obtain the final word value. In this case, the validity word for the result is only the current word.

Algorithm 3 presents the pseudocode of the *getWord* method. This method compares the start and end positions with the start and end positions of the run of 1s in this column. If the queried positions have no overlap with the run of 1s, then we can return an all 0s word. However, if they overlap, then we produce a word with bits set for the positions that overlap with the run of 1s. To compute this efficiently we have an array *runs* that stores integers with increasing runs of 1s.

Let us consider again the query asking for the first column of Attribute 1. There is only one word in the EB, which means that all 6 distinct ranks would be evaluated at the same time. Since *wordNumber*=0 falls completely in the first period of column 1, the getWord method needs to be called only once with start=1 and end=27. The result (in hex) returned is word 00001FF0, which is ANDed with the word 0 of the EB to obtain: 00000820. Since the resulting word is not 0, translation into the primary bitmap is needed. The position of each non-zero bit is computed and the rank is translated just as in the case of the rankList, producing the same answer.

The previous pseudocodes only consider a partition with one segment. Having several segments would increase the number of bitmaps returned as no operations are done across segments. In the case that there are more than one partition involved in the query, the translated bitmaps B from each partition are combined together, e.g. ANDed together, before returning the final answer to the user.

3.5 Partitioning

For rNCB, the goal of horizontal partitioning is to ensure that each segment fits into main memory and the goal of vertical partitioning is to guarantee the performance of the ordering for all the attributes in the partition.

The following is the criteria used to decide how many vertical partitions to build for rNCB. The improved performance of rNCB relies on processing smaller EBs than the traditional bitmaps. There are two factors that can make the EB have less ranks/words than the verbatim bitmaps. The first one is the number of duplicate ranks in the partition and the second one is the number of all possible distinct ranks in the full bitmap table.

In order to guarantee improved query execution time over the traditional bitmaps, we need to impose two conditions over the rNCB partitions. The first one refers to the number of distinct ranks in the EB and the second one, to the number of bits required to encode a rank.

It is clear that rNCB query execution time depends on the size of the EB as a sequential scan of the EB is performed for each query. Therefore, by constraining the number of possible distinct ranks in the EB we bound the time required to process the EB. Since our baseline is traditional bitmaps, it is only intuitive to make the EB always smaller than a corresponding verbatim bitmap. The size in words of a verbatim bitmap is $\frac{n}{w}$, where n is the number of rows in the table and w is the word size, e.g. 32 bits[2]. For WAH compressed bitmaps, the maximum size (when the column is incompressible) is $\frac{n}{w-1}$, as 1 bit is used in each word to indicate the type of word.

Without considering any other factors, in the general case, if we constrain the EB to have at most $\frac{n}{w}$ ranks, then each rank/word in the EB would be evaluating w dataset tuples simultaneously on the average. This is assuming that all the ranks in a partition are distinct, which is not often the case. When duplicates occur, then even more tuples are simultaneously processed.

In addition to this, since queries are often performed over a subset of attributes and more than one bitmap column is involved in answering a query, we could compare the EB size to the average total size of the bitmaps columns involved in the query. For these

[2]For our implementation in Java, the word size is set to 31 bits since Java does not have unsigned data types.

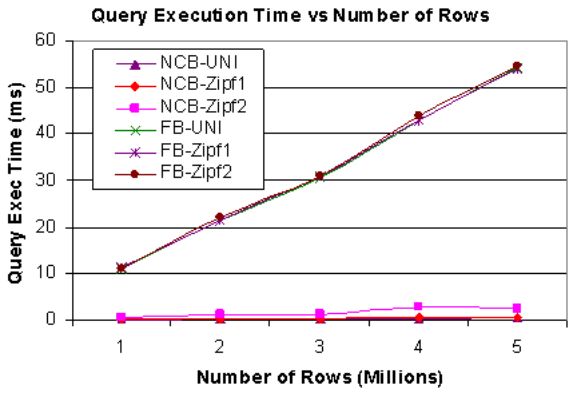

Figure 5: Execution of point queries for datasets with 5 attributes each with cardinality 5, different distributions, and varying number of rows.

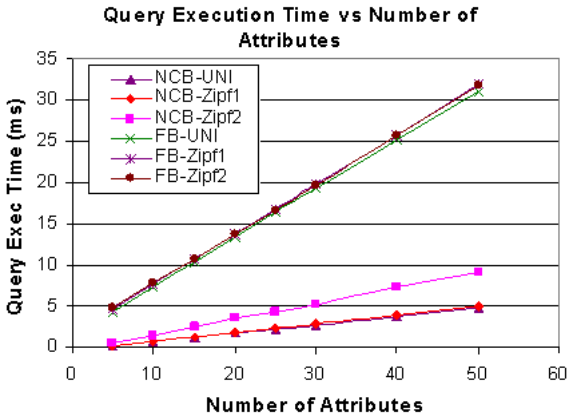

Figure 6: Execution of point queries for datasets with varying number of attributes each with cardinality 5, different distributions, and 1M rows.

Cardinality	UNI	Zipf1	Zipf2
5	2,978	3,125	2,978
10	100,000	95,293	33,299
15	758,302	457,842	74,029
20	2,526,975	974,746	109,797
30	4,518,787	1,931,186	162,716
40	4,878,900	2,613,087	199,400
50	4,960,228	3,082,957	224,688

Table 2: Number of distinct ranks for different data distributions as attribute cardinality increases. The datasets have 5 attributes and 5M rows.

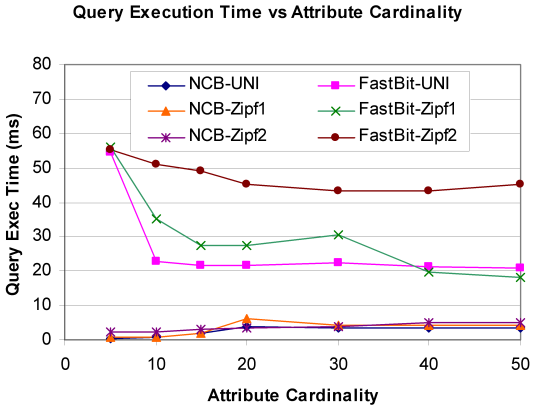

Figure 7: Execution of point queries for datasets with 5M rows, 5 attributes, and different distributions with varying cardinality.

reasons, w can be a smaller number and still produce improved performance. In our experiments we set the constraint factor to 5. The reason is that our datasets are relatively small and larger number would produce more partitions. Within the constraints, less partitions are always preferred because of the space requirement of the mapping tables.

Another constraint imposed over rNCB partitions is that the number of bits required to encode a given rank in the full bitmap table could not exceed $w + \lceil \log_2 w \rceil$ bits. In our experiments w was set to 31 because integers were used to represent the word number in the bitmap.

For all real datasets our constraints were satisfied using 2 partitions with equal number of attributes. Since in our case the query patterns are unknown, we produced equi-sized partitions to balance the performance of all queries and avoid worse case execution times for some queries. In the case when the query patterns are known, attributes that are often queried together can be placed in the same partition to avoid the overhead of translating bitmaps and ANDing the partial results. Moreover, attributes could be replicated into two or more partitions as long as the constraints are met. Another consideration is to place attributes in their order of query frequency within the partition as, for point queries, earlier columns have a slight advantage over subsequent columns as it would become evident in the experimental results.

4. EXPERIMENTAL RESULTS

We performed experiments over both real and synthetic datasets. For the real datasets we used `HEP` data, which comes from high energy experiments with over 2 million rows and 12 attributes with cardinality ranging from 2 to 12 bins, and three other datasets from the UCI data repository [1]. `Adult` dataset is census data with 48,842 rows and 14 attributes with cardinality ranging from 2 to 42. For the `Adult` dataset, the 5 continuous attributes were quantized using equi-populated partitions into 3 bins. `Nursery` is data derived from a hierarchical decision model originally developed to rank applications for nursery schools. `Nursery` has 12,960 rows and 8 attributes with cardinality ranging from 3 to 6. In poker hand (`poker`) each record is an example of a hand consisting of five playing cards drawn from a standard deck of 52. `Poker` dataset has 1,025,010 rows and 10 attributes half with cardinality 4 (suit) and half with cardinality 13 (rank). For the synthetic datasets we generated uniform (`UNI`) and Zipf distributions for datasets with varying number of attributes (5 to 50) with varying cardinality from 5 to 50 and different number of rows starting from 1 million to 5 million. Zipf distributions were generated using s=1 (`Zipf1`) and

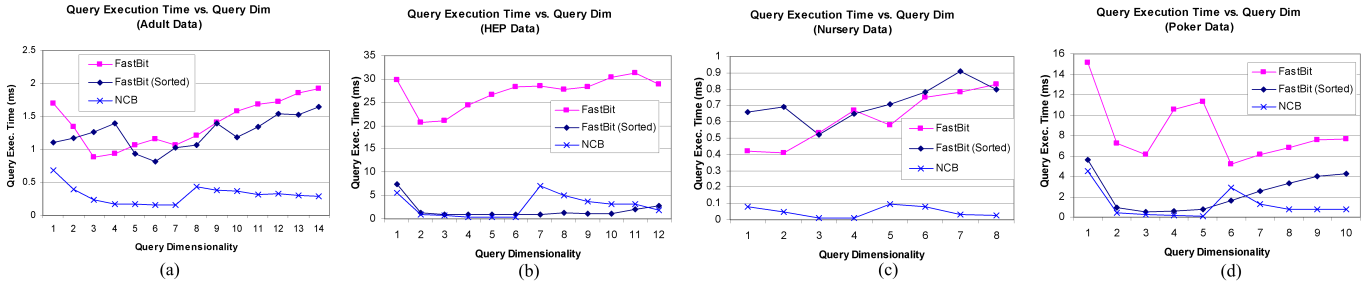

Figure 10: Execution of point queries for real datasets as the query dimensionality increases. Query attributes are selected in the same order of the dataset attributes.

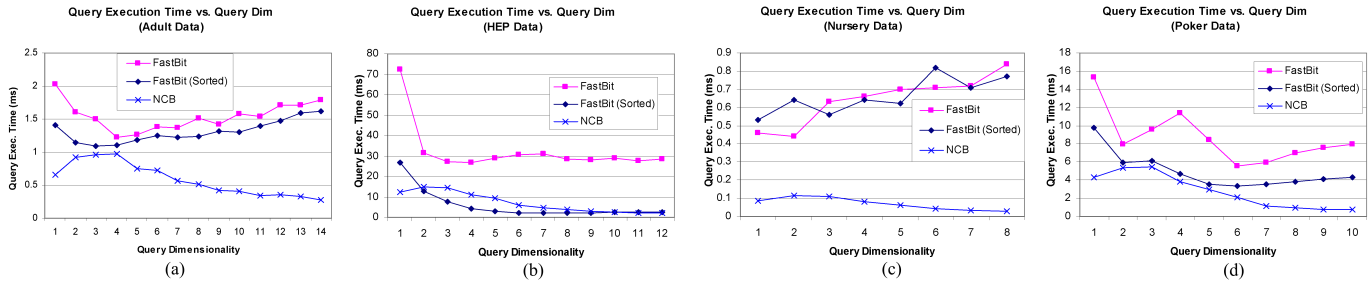

Figure 11: Execution of point queries for real datasets as the query dimensionality increases. Query attributes are randomly selected.

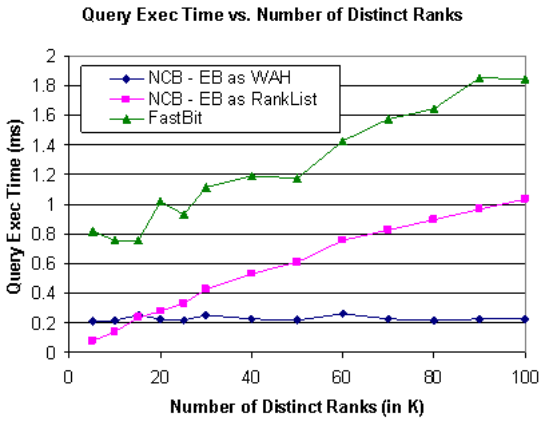

Figure 8: Execution time of point queries for increasing number of distinct ranks in the EB.

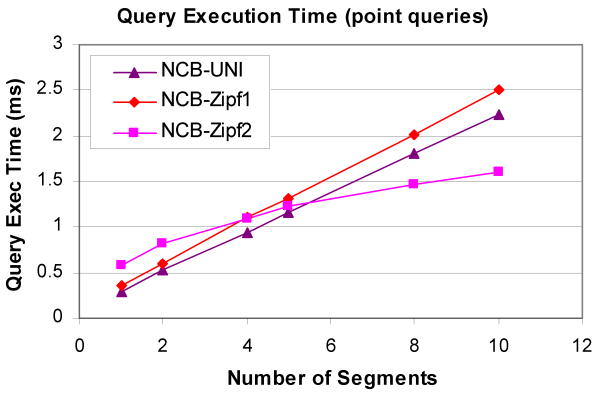

Figure 9: Execution time of point queries for datasets with 5 attributes, each with cardinality 5, 1M rows, and different distributions. rNCBs are built with varying number of horizontal partitions (segments).

2 (`Zipf2`) as the value of the exponent characterizing the distribution.

Point queries are generated by randomly sampling 100 tuples from the dataset. Range queries are selected the same way but the attribute value of the tuple is expanded into both directions (higher and lower values) until the number of bitmaps queried is equal to the percent of *cardinality-selectivity* specified. When not specified, point queries and range queries refer to queries over all the attributes in the dataset.

The implementation of the ranked Non-Clustered Bitmaps (rNCB) is done in Java. We compare query execution time with FastBit. The queries for FastBit are specified as selects of the first attribute in the dataset and the time reported corresponds to the CPU time output by the program. This CPU time (as opposed to the elapsed time) does not include I/O time.

The experiments are run in a computer with 3.00GHz CPU, 2GB RAM, Linux Ubuntu 7.10 operating system. To measure query execution time for rNCB and FasBit we run each set of 100 queries 5 times and drop the highest value to avoid outliers produced by process preemption, or java virtual machine garbage collection. The remaining 4 runs are averaged together and reported in this section.

Next, we present the experiments conducted over synthetic data

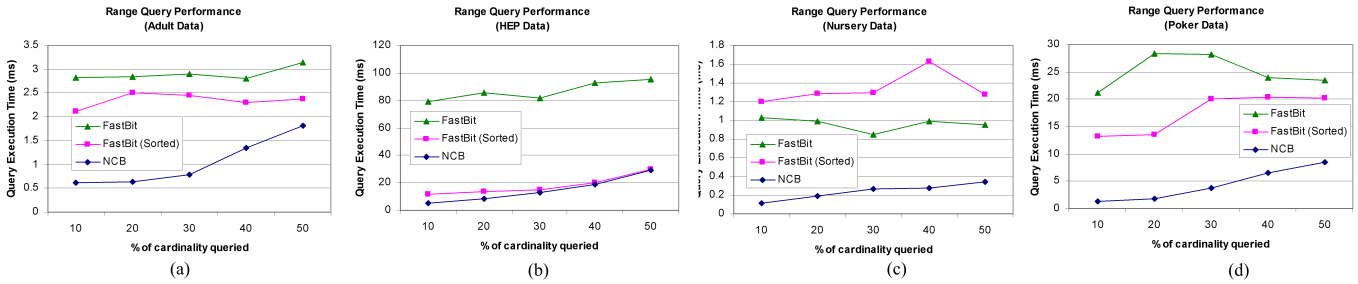

Figure 12: Execution of range queries with varying percentage of attribute cardinalities queried.

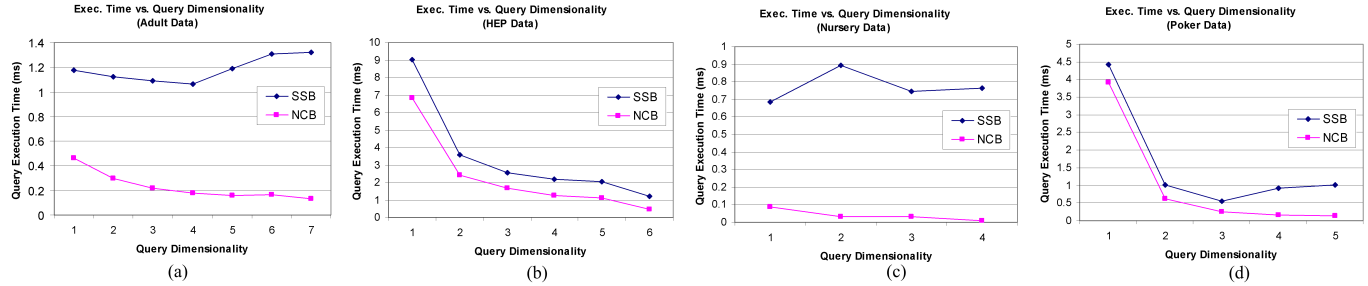

Figure 13: Execution of point queries with varying dimensionality in sorting order for Sorted Secondary Bitmaps (SSB) and ranked Non-Clustered Bitmaps (rNCB).

	SSB			rNCB		
Dataset	Part 1	Part 2	Tot (KB)	Part 1	Part 2	Tot (KB)
Adult	64.49	15.51	80	3.95	26.56	30.51
Hep	37.03	118.06	155.09	21.14	3.06	24.2
Nursery	4.98	1.46	6.44	0.29	0.05	0.34
Poker	208.64	447.6	656.24	1.41	4.58	5.99

Table 4: Bitmap Size comparison of SSB and rNCB.

to analyze and evaluate the performance of rNCB (just NCB in the figures). Then, we present the query execution times for real datasets.

4.1 Number of Rows

Figure 5 shows the average query execution time as the number of rows increases (from 1 to 5 million). Note that FastBit execution time is linear with the number of rows in the dataset as the size of each bitmap and the number of hits in the queries increases. However, rNCB execution time does not depend on the number of rows but rather on the number of distinct ranks, which remains constant since the number of attributes and their cardinality do not change.

4.2 Number of Attributes

Figure 6 shows the average query execution time as the number of attributes increases. For this experiment, rNCB is built using 5 attributes per partition. As can be seen, both approaches grow linearly but the slope of rNCB is smaller than FastBit.

4.3 Attribute Cardinality

Figure 7 shows the average query execution time as the cardinality increases. rNCB performs considerably faster than FastBit for all distributions. It is worth noting that depending on the cardinality rNCB is using 1 or 2 partitions. For UNI and Zipf1, two partitions are created after cardinality 20 and 30, respectively. For Zipf2, the number of distinct ranks never violates the partition constraint and therefore only one partition is used for Zipf2 for all cardinalities. The number of distinct ranks in the dataset for the three distributions is presented in Table 2.

4.4 Number of Distinct Ranks

Figure 8 shows the average query execution time as the number of distinct ranks in the EB increases. As can be seen, the execution time is linear to the number of distinct ranks when the EB is stored as a rankList, however, when the EB is stored as a WAH compressed bitmap the query execution increases until the EB is not compressible and then stabilizes. Recall that we only compress the zeros in the EB.

4.5 Number of Segments

Figure 9 shows the average query execution time as the number of horizontal partitions (segments) used to create the rNCB increases. Partitions are built with 1M, 500K, 250K, 200K, 125K, and 100K rows, to produce 1, 2, 4, 5, 8, and 10 segments, respectively. As can be seen, the execution time is linear with respect to the number of segments.

4.6 rNCB Performance over Real Datasets

Figure 10 presents the average query execution time as the number of queried attributes increases for real datasets. For these experiments, query attributes are selected in the same order of the dataset attributes. For example, query dimensionality 2 means the first two attributes of the dataset are queried. In general, the execution time of rNCB decreases as query dimensionality increases because queries are executed row-wise and the answer is returned as soon as one query condition is not satisfied. Since two partitions are used for each dataset, when the query ask for the first 8

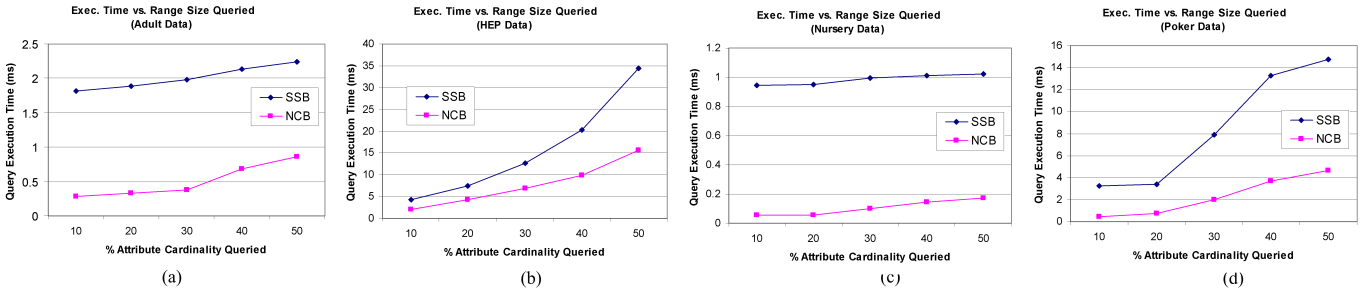

Figure 14: Execution of range queries with varying percentage of attribute cardinalities queried for Sorted Secondary Bitmaps (SSB) and ranked Non-Clustered Bitmaps (rNCB).

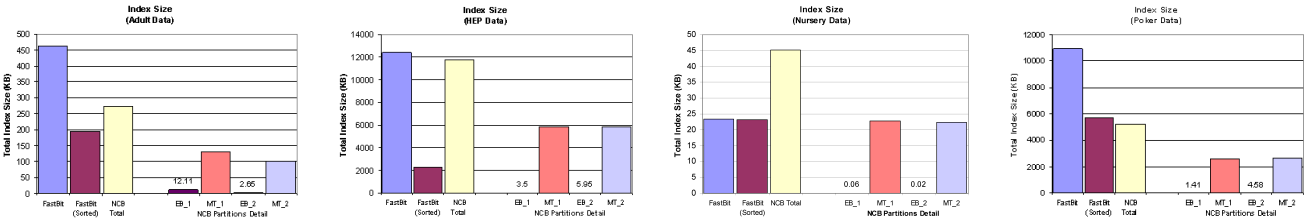

Figure 15: Index Size Comparison for FastBit, Sorted FastBit and rNCB for real datasets.

attributes, for `Adult` data for example, there is a sudden increase in query execution time. The reason is that partitions are built using 7 attributes for `Adult` data and therefore, when a query asks for 8 attributes there is an extra cost of translating two bitmaps and ANDing them together to produce the final answer.

Figure 11 presents the same set of experiments with the only distinction that the attributes are randomly selected from the dataset. In this case a query with dimensionality of 2 can query *any* two attributes in the data, including two attributes from different partitions. As can be seen, when comparing the two graphs, earlier columns in the sorting order have a slight advantage in query execution time over latter columns. Nevertheless, the execution time continues to be consistently faster than FastBit.

Figure 12 shows the query execution time of range queries when the percentage of cardinality queried is increased from 10 to 50%. rNCB execution time is faster than FastBit for all four datasets.

4.7 Index Size

Figures 15 shows a comparison between the random ordered compressed bitmaps (FastBit), the sorted compressed bitmaps (FastBit Sorted) and rNCB for the real datasets. The rNCB size is also presented for each component: the EBs and the MTs for each partition. As can be seen, the rNCB size is completely dominated by the size of the mapping tables and in most cases the rNCB size is worse than the ordered WAH compressed bitmaps (FastBit Sorted) but better than the total size of random ordered WAH compressed bitmaps.

Table 3, shows the effect of vertical partition on the size of the full bitmap table. By creating 2 partitions, the size of the full bitmap is not reduced by half, but rather by the product of the cardinalities of the attributes involved. This is the reason why low number of partitions satisfy the partition constraints that guarantee faster query execution of rNCB when compared against random ordered compressed bitmaps.

4.8 rNCB vs. Secondary Sorted Bitmaps (SSB)

With the following set of experiments we compare the performance of rNCB against SSB, where the sorted bitmap columns are explicitly stored. To produce the effect of SSB, we partitioned each dataset using the same partitions than rNCB. Then we sorted the data physically and used FastBit to create WAH compressed bitmaps over this dataset. For the experiments, we do not use queries involving attributes across partitions. We ran the same set of queries for rNCB and SSB and compared execution time.

Figure 13 shows the query execution time of point queries with varying dimensionality for SSB and rNCB. For this experiment we ran point queries with dimensionality ranging from 1 to the number of attributes in the partition. We ran queries over the two partitions and averaged the running times. As can be seen, rNCB performs better than SSB because there is no redundancy in the EB.

Figure 14 shows the query execution time of range queries with varying percentage of cardinality queried. Again, both techniques show the same trend but rNCB performs consistently faster than SSB.

In Table 4 we compare the sizes (in KB) of the two approaches. The reported sizes are only for the EB of rNCB and the bitmap columns of SSB, mapping table sizes are omitted since it would be the same for both approaches. As can be seen the EB compact representation is more space efficient than the compressed bitmaps, even when they are sorted.

Another advantage of rNCB over SSB is the update performance. While SSB needs to update several bitmap columns, sometimes all

		Whole Dataset			rNCB (P=2) (Aggregated)		
Dataset	Rows	B_n	$\log_2 B_n$	Distinct Ranks	Possible Ranks	Distinct Ranks	EB Size (KB)
HEP	2,173,762	505,107,472,320	39	353,889	2,180,178	12,063	30.51
Adult	48,842	148,142,131,200	38	24,057	2,245,320	9,345	24.2
Nursery	12,960	115,200	17	12,960	792	294	0.35
Poker	1,025,010	380,204,032	29	1,022,771	45,968	45,084	5.99

Table 3: Size statistics for real datasets and rNCB with two partitions.

of the bitmap columns, rNCB only needs to update one bitmap, the EB. Moreover, if the rank being inserted is already set in the EB, then only the mapping table structure needs to be updated.

5. CONCLUSION

In this paper we propose a novel bitmap index design with vertical and horizontal partitioning that serves as basis for non-clustered bitmap indexes. As opposed to traditional bitmap indexes, where one bitmap column is stored for each attribute value and each tuple is represented by one bit in each bitmap, the proposed scheme generates only one bitmap column for a set of attributes (partition) to encode the rank of the tuples in the full sorted bitmap table. The full sorted bitmap table is not stored, but rather logically queried using the rank of the tuples. The proposed approach is compared against Sorted Secondary Bitmap Indexes, Sorted Clustered Bitmap Indexes, and FastBit, a recent bitmap index implementation. Experiments show that query execution time is greatly reduced when rNCB is used.

One positive side effect of having only one EB per partition as opposed to one bitmap per column in the partition is that the update cost is greatly reduced. In sorted secondary bitmaps, in addition to updating the mapping table, the bitmap indexes need to be rebuilt with every update or batch update as the insertion of a new bit in the middle of the bitmap would shift the bit positions and the alignment of the words would no longer be valid. In rNCB, the EB is stored as a WAH compressed bitmap on the ranks. In the case when the rank of the new tuple is already in the partition only the mapping table needs to be updated. And even in the case when the rank of the tuple is new and the word for the rank is compressed in the EB, we would break the run and increase the size of the EB at most 3 words. Rebuilds can be avoided because the bit position for the rank is already in the EB.

Acknowledgments

This research is partially supported by US National Science Foundation grants IIS-0546713, and DBI-0750891.

6. REFERENCES

[1] http://mlearn.ics.uci.edu/MLRepository.html.

[2] S. Amer-Yahia and T. Johnson. Optimizing queries on compressed bitmaps. In *The VLDB Journal*, pages 329–338, 2000.

[3] G. Antoshenkov. Byte-aligned bitmap compression. In *Data Compression Conference*, Nashua, NH, 1995. Oracle Corp.

[4] G. Antoshenkov and M. Ziauddin. Query processing and optimization in oracle rdb. *The VLDB Journal*, 1996.

[5] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.

[6] C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. In *SIGMOD*, pages 355–366, 1998.

[7] C.-Y. Chan and Y. E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *SIGMOD*, pages 215–226, 1999.

[8] H. Edelstein. Faster data warehouses. Information Week, December 1995.

[9] I. Inc. Informix decision support indexing for the enterprise data warehouse. http://www.informix.com/informix/corpinfo/-zines/whiteidx.htm.

[10] S. Inc. *Sybase IQ Indexes*, chapter 5: Sybase IQ Release 11.2 Collection. Sybase Inc., March 1997.

[11] D. Johnson, S. Krishnan, J. Chhugani, S. Kumar, and S. Venkatasubramanian. Compressing large boolean matrices using reordering techniques. In *VLDB 2004*.

[12] T. Johnson. Performance measurement of compressed bitmap indices. In *VLDB*, pages 278–289, 1999.

[13] T. Johnson. Performance measurements of compressed bitmap indices. In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 278–289. Morgan Kaufmann, 1999.

[14] N. Koudas. Space efficient bitmap indexing. In *Proceedings of the ninth international conference on Information and knowledge management*, pages 194–201. ACM Press, 2000.

[15] E. O'Neil, P. O'Neil, and K. Wu. Bitmap index design choices and their performance implications. *Database Engineering and Applications Symposium, 2007. IDEAS 2007. 11th International*, pages 72–84, Sept. 2007.

[16] P. O'Neil. Informix and indexing support for data warehouses, 1997.

[17] A. Pinar, T. Tao, and H. Ferhatosmanoglu. Compressing bitmap indices by data reorganization. *ICDE*, pages 310–321, 2005.

[18] D. Salomon. *Data Compression 2nd edition*. Springer Verlag, New York, 2000.

[19] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: A column-oriented dbms. VLDB, 2005.

[20] K. Wu. Fastbit: an efficient indexing technology for accelerating data-intensive science. *J. Phys.: Conf. Ser.*, 16:556–560, 2005.

[21] K. Wu, E. Otoo, and A. Shoshani. A performance comparison of bitmap indexes. In *Proceedings of the 2001 ACM CIKM International Conference on Information and Knowledge Management*, pages 559–561, Atlanta, Georgia, November 2001.

[22] K. Wu, E. Otoo, and A. Shoshani. Compressing bitmap indexes for faster search operations. In *SSDBM*, pages 99–108, Edinburgh, Scotland, UK, July 2002.