Bitmap Index Design Choices and Their Performance Implications

Elizabeth O'Neil and Patrick O'Neil
University of Massachusetts at Boston
{eoneil, poneil}@cs.umb.edu

Kesheng Wu
Lawrence Berkeley National Laboratory
kwu@lbl.gov

ABSTRACT

Historically, bitmap indexing has provided an important database capability to accelerate queries. However, only a few database systems have implemented these indexes because of the difficulties of modifying fundamental assumptions in the low-level design of a database system and in the expectations of customers, both of which have developed in an environment that does not support bitmap indexes. Another problem that arises, and one that may more easily be addressed by a research article, is that there is no definitive design for bitmap indexes; bitmap index designs in Oracle, Sybase IQ, Vertica and MODEL 204 are idiosyncratic, and some of them were designed for older machine architectures.

To investigate an efficient design on modern processors, this paper provides details of the Set Query benchmark and a comparison of two research implementations of bitmap indexes. One, called RIDBit, uses the N-ary storage model to organize table rows, and implements a strategy that gracefully switches between the well-known B-tree RID-list structure and a bitmap structure. The other, called FastBit is based on vertical organization of the table data, where all columns are individually stored. It implements a compressed bitmap index, with a linear organization of the bitmaps to optimize disk accesses. Through this comparison, we evaluate the pros and cons of various design choices. Our analysis adds a number of subtleties to the conventional indexing wisdom commonly quoted in the database community.

1. INTRODUCTION

Bitmap indexes have not seen much new adoption in commercial database systems in recent years. While ORACLE has offered bitmap indexing since 1995, other major systems such as DB2 and Microsoft SQL Server do not provide them. Microsoft SQL Server may create bitmaps during hash joins, but not for general indexing; DB2 has adopted an Encoded Vector Index [16], but this is basically an encoded projection index rather than a bitmap index. Sybase Adaptive Server Enterprise (ASE), the major Sybase DBMS, does not have bitmap indexing, although the Sybase Adaptive Server IQ product provides quite competitive bitmap indexing for data warehousing. This situation arises in part because there is no definitive design for bitmap indexes. To investigate such a definitive design, we plan to explore different design choices through a careful study of two research implementations. Since we have control over all aspects of the research implementations, we are able to try out some new techniques for improving performances, such as new forms of compression and careful disk placement. In the process of studying their performance pros and cons, we also find some surprises.

A *basic bitmap index* (more simply, *bitmap index* in what follows) is typically used to index values of a single column **X** in a table. This index consists of an ordered sequence of *keyvalues* representing distinct values of the column, and each keyvalue is associated with a bitmap that specifies the set of rows in the table for which the column **X** has that value. A bitmap has as many bits as the number of rows in the table, and the *k*th bit in the bitmap is set to 1 if the value of column **X** in the *k*th row is equal to the keyvalue associated with the bitmap, and 0 for any other column value. Table 1 shows a basic bitmap index on a table with nine rows, where the column **X** to be indexed has integer values ranging from 0 to 3. We say that the *column cardinality* of **X** is 4 because it has 4 distinct values. The bitmap index for X contains 4 bitmaps, shown as B_0, B_1, …, B_3, with subscripts corresponding to the value represented. In Table 1 the second bit of B_1 is 1 because the second row of X has the value 1, while corresponding bits of B_0, B_2 and B_3 are all 0.

Table 1: A bitmap index for a column named X. Columns B_0 – B_3 are bitmaps.

RID	X	B_0	B_1	B_2	B_3
0	2	0	0	1	0
1	1	0	1	0	0
2	3	0	0	0	1
3	0	1	0	0	0
4	3	0	0	0	1
5	1	0	1	0	0
6	0	1	0	0	0
7	0	1	0	0	0
8	2	0	0	1	0

To answer a query such as "X > 1," we perform bitwise OR (|) operations between successive long-words of B_2 and B_3, resulting in a new bitmap that can take part in additional operations. Since bitwise logical operations such as OR (|), AND (&) and

Table 2: Key differences between RIDBit and FastBit.

	FastBit	RIDBit
Table layout	Vertical storage (columns stored separately)	N-ary storage (columns stored together in row)
Index layout	Arrays of bitmaps	B-tree keyed on keyvalues (improved in project)
Bitmap layout	Continuous	Horizontally partitioned into 32K-bit Segments
Compression	Word-Aligned Hybrid compression	Sparse bitmap converted to RID-list

NOT (~) are well-supported by computer hardware, a bitmap index software could evaluate SQL predicates extremely quickly. Because of this efficiency, even some DBMS systems that do not support bitmap indexes will convert intermediate solutions to bitmaps for some operations. For example, PostgreSQL 8.1.5 has no bitmap index, but uses bitmaps to combine some intermediate solutions [10]. Similarly, Microsoft SQL Server has a bitmap operator for filtering out rows that do not participate in a join operation [5].

Let N denote the number of rows in the table T and C(**X**) the cardinality of column **X**. It is easy to see that a basic bitmap index like the one in Table 1 requires N•C(**X**) bits in the bitmaps. In the worst case where every column value is distinct, so that C(**X**) = N, such a bitmap index requires N^2 bits. For a large dataset with many millions of rows, such an index would be much larger than the table being indexed. For this reason, much of the research on bitmap indexes has focused on compressing bitmaps to minimize index sizes. However, operations on compressed bitmaps are often slower than on uncompressed ones, called *verbatim bitmaps*. There is a delicate balance between reducing index size and reducing query response time, which complicates the design considerations for bitmap index implementations.

Two very different approaches to reducing index sizes are used by the research prototypes we study. FastBit implements the Word-Aligned Hybrid (WAH) compression; the WAH compressed basic bitmap index was shown to be efficient in [18][19]. RIDBit employs a combination of verbatim bitmaps and RID-lists composed of compact (two-byte) Row Identifiers (RIDS). Its unique ability to gracefully switch from verbatim bitmaps to RID-lists based on the column cardinality originated with MODEL 204 [6].

The implementations of FastBit and RIDBit were quite different at the beginning of our study, which made them ideal for contrasting the different implementation strategies and physical design choices. As our study progressed, a number of implementation ideas found to be superior in FastBit were copied in RIDBit; RIDBit software was also modified to better utilize the CPU. The lessons learned in this exercise will be covered in the Summary section. Table 2 gives the key differences between RIDBit and FastBit. We will discuss the detailed design of the two approaches in the next two sections.

The topics covered in succeeding sections are as follows. In Sections 2 and 3, we describe the architecture of FastBit and RIDBit. Section 4 provides a theoretical analysis of index sizes for different columns. Section 5 describes the Set Query Benchmark [7], which is used to compare performance of RIDBit and FastBit. Section 6 presents the detailed experimental measurements. Finally, Section 7 provides a summary and lessons learned.

2. FASTBIT

FastBit started out as a research tool for studying how compression methods affect bitmap indexes, and has been shown since to be an efficient access method in a number of scientific applications [12][17]. It organizes data into tables (with rows and columns), where each table is vertically partitioned and different columns stored in separate files. Very large tables are horizontal partitioned, where each partition typically consisting of many millions of rows. A partition is organized as a directory, with a file containing the schema, followed by the data files for each column. This vertical data organization is similar to a number of contemporary database systems such as Sybase IQ [9], MonetDB [1][2], Kx systems [4], and C-Store [14]. Each column is effectively a projection index as defined in [9] and can sometimes be used efficiently to answer queries without additional indexing structures. FastBit currently indexes only fixed-sized columns, such as integers and floating-point numbers, although it can index low-cardinality string-valued columns through a dictionary that converts the strings to integers. Because of this restriction, the mapping from a row to a row identifier is straightforward.

FastBit implements a number of different bitmap indexes with various binning, encoding and compression strategies [13]. The index used in this study is the WAH compressed basic bitmap index. All bitmaps of an index are stored in a single file as shown in Table 3. Logically, an index file contains two sets of values: the keyvalues and the compressed bitmaps. FastBit stores both the keyvalues and

Table 3: Content of a FastBit index file.

N	Number of rows
C	Column cardinality
keyvalues[C]	Distinct values associate with each bitmap
starts[C+1]	Starting position of each compressed bit-map (final position is end of all bitmaps)
bitmaps[C]	WAH Compressed bitmaps

bitmaps in arrays on disk. Since each keyvalue is the same size, it can be located easily. To locate the bitmaps, FastBit stores another array starts[] to record the starting position of all compressed bitmaps in the index file (in bitmaps[ ]). To simplify the software, one extra values is used in array starts[] to record the ending position of the last bitmap.

FastBit generates all bitmaps of an entire index for one partition in memory before writing the index file. This dictates that the entire index must fit in memory and imposes an upper bound on how many rows a horizontal partition can hold on a given computer system. Typically, a partition has no more than 100 million rows, so that a small number of bitmap indexes may be built in-memory at once.

In general, FastBit stores the array keyvalues[ ] in ascending order so that it can efficiently locate any particular value. In some cases, it is possible to replace this array with a hash function. Using hash functions typically requires fewer I/O operations to answer a query than using arrays do, but using arrays more easily accommodates arbitrary keyvalues. FastBit uses memory maps to access the array keyvalues[] and starts[] if the OS supports it; otherwise it reads the two arrays entirely into memory. Since the index for a partition has to fit in memory when built, this reading procedure does not impose any additional constraint on the sizes of the partitions.

One advantage of the linear layout of the bitmaps is that it minimizes the number of I/O operations when answering a query. For example, to answer the range predicate "3 < KN < 10", FastBit needs to access bitmaps for values 4 through 9. Since these bitmaps are laid out consecutively in the index file, FastBit reads all these bitmaps in one sequential read operation.

The linear layout of bitmaps means that FastBit is not in any way optimized for update. An update that might add or subtract a 1-bit to one of the bitmaps would require modification of the bitmap, followed by a reorganization of all successive bitmaps in the set. In scientific applications, changes in real time between queries are unusual, so this limitation is not a serious drawback, and it is not a problem for most commercial data warehousing applications either. Furthermore, we will see in Section 7 that the new Vertica database product [14] provides a model where a fixed index for stable data can be maintained on disk while new data is inserted to a memory resident dynamic store that takes part in all queries.

FastBit reconstitutes a C++ bitmap data structure from the bytes read into memory. This step makes it easy to use the bitwise logical operation functions implemented in C++; however, it introduces unnecessary overhead by invoking the C++ constructor and destructor. Additionally, since FastBit aggregates the read operations for many bitmaps together, a certain amount of memory management is required to produce the C++ bitmap objects.

3. RIDBIT

RIDBit was developed as a pedagogical exercise for an advanced database internals course, to illustrate how a bitmap indexing capability could be developed. The RIDBit architecture is based on index design first developed for the Model 204 Database product from Computer Corporation of America [6]. We can view bitmaps, representing the set of rows with a given value for a column, as providing an alternative form for RID-lists commonly used in indexes. The column values are represented as keyvalues in a B-tree index, and row-sets that follow each column value are represented either as RID-lists or bitmaps. Bitmaps are more space-efficient than RID-lists when the bitmap is relatively dense, and bitmaps are usually more CPU-efficient as well. To create Bitmaps for the N rows of a table T = {r_1, r_2, …, r_N}, we start with a 1-1 mapping m from rows of T to Z[M], the first M positive integers. In what follows we avoid frequent reference to the mapping m: when we speak of the *row number* of a row r of T, we will mean the value m(r).

Note that while there are N rows in T = {r_1, r_2,..., r_N}, it is possible that the number of bits M in the bitmap representation of RIDBit is somewhat greater than N, since it associates a fixed number of rows p with each disk page for fast lookup, even when the rows are somewhat varying in size. The advantage of this is that for a given row r with row number j, the page number accessed to retrieve row r is j/p and the page slot is j%p, where % denotes the modulo operator. This usually means that rows are assigned row numbers in disk-clustered sequence during load, a valuable property. The RIDBit architecture stores the rows in an N-ary organization, where all column values of a row are stored together. Since the rows might have varying sizes and we may not always be able to accommodate an equal number of rows on

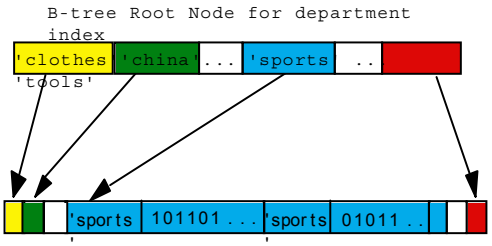

Figure 1: A RIDBit Index on department, a column of the SALES table.

each disk page, the value p must be chosen as a maximum; thus for a page of larger rows, some slots on a page will not accommodate the full set of p rows, and we will find that $m^{-1}(j)$ for some row numbers j in Z[M] are undefined.

RIDBit organizes its indexes as B-trees. A bitmap index for a column A with values v_1, v_2, . . ., v_k, is a B-tree with entries having these keyvalues and associated data portions that contain bitmaps or RID-lists for the properties A = v_1, ..., A = v_k. Bitmaps in this index are just a different way to specify lists of RIDs, and when the density of a bitmap becomes too small to be efficient, a RID-list is used instead. Note in particular that when we speak of a bitmap index in RIDBit, we admit the possibility that some bitmaps are in fact RID-lists. See Figure 1 for an index example with low cardinality, where all row-sets are represented by verbatim bitmaps. RIDBit actually stores each verbatim bitmap as a series of successive bitmap fragments, called *segments*. Each box in Figure 1 is an illustration of multiple bitmap segments for "department = 'sports'".

Recall that bitmaps are called *dense* if the proportion of 1-bits in the bitmap is relatively large. A bitmap index for a column with 32 values will have bitmaps with average density of 1/32. In this case the disk space to hold a column index will be comparable to the disk space needed for a RID-list index in products with 32-bit RIDs. While the verbatim bitmap index size is proportional to the number of column values, a RID-list index is about the same size for any number of values (as long as we can continue to amortize the key size with a long block of RIDs). For a column index with a very small number of values, the bitmaps will have high densities (such as 50% for predicates such as GENDER = 'M' or GENDER = 'F'), and the disk savings is enormous. On the other hand, when average bitmap density for a bitmap index becomes too low, the bitmaps can be efficiently compressed. The simplest compression method, and the one used in RIDBit, is to translate the bitmap back to a RID list (albeit a special small-sized RID in the case of RIDBit). Boolean operations on these mixtures of bitmaps and RID lists can be found in [6][11]. To account for the fact that some of the page slots are not used, we use an *Existence bitmap* (designated *EBM*), which has exactly those 1 bits corresponding to existing rows[1]. Now when RIDBit needs to performs a NOT on a bitmap B, it loops through a long int array performing the ~ operation, then AND's the result with the corresponding long int array from EBM.

In RIDBit, the sequence of rows on a table as well as the bitmaps referencing them are broken into equal-sized fragments, called ***segments***, so that a verbatim bitmap segment will fit on a single disk page. In its current architecture, a RIDBit segment fits on a 4KByte page and a verbatim bitmap contains about 32K bits; thus a table is broken into segments of about 32K rows each. As for the bitmaps, there are three different ways to store them. The bitmaps with the highest densities are stored as segmented verbatim bitmaps. As the bit density decreases, the bitmaps are stored as segment-relative RID-lists, as explained in the next paragraph. At extreme low density, the segment-relative RIDs are directly stored as full-sized RIDs in the space normally used to store segment pointers to bitmaps or RID-lists in the leaf level of the B-tree. Since these RIDs are directly stored in the B-tree nodes, they are called "local" RID-lists.

RIDs used to access a row in a RIDBit segment, known as segment-relative RIDs (following the design of MODEL 204) are represented by integers from 1 to 32K - m (where m bits are used to contain a count of 1-bits in a bitmap), and thus only require two bytes each, or a short int in a typical C program. RIDBit supports verbatim bitmaps down to a density of 1/50, and a verbatim bitmap of that minimum density will thus require only 32K/50 = 655 short ints = 1310 bytes for RID-list representation. Thus several RID-lists with maximum size 1310 bytes or less are likely to fit on a single disk page. At the beginning of each segmented bitmap/RID-list pointer at the leaf level of the B-tree, the segment number will specify the higher order bits of a longer RID (4 bytes or perhaps more), but the segment-relative RIDs only use two bytes each. This is an important form of prefix compression, which greatly speeds up most index range searches.

A second implication of segmentation involves combining predicates. The B-tree index entry for a particular value in RIDBit is made up of a series of pointers to segment bitmaps or RID-list, but there are no pointers for segments that have no representative

1 It was pointed out by Mike Stonebraker that a "non-existence bitmap" would be more efficient, and this change is planned.

rows. In the case of a clustered index, for example, each particular index value entry will have pointers to only a small sequence of row segments. In MODEL 204, if several predicates involving different column indexes are ANDed, the evaluation begins segment-by-segment. If one of the predicate indexes has *no pointer* to a bitmap segment for a segment, then the segments for the other indexes can be ignored as well. Queries such as this can turn out to be very common in a workload, and the I/O saved by ignoring I/O for these index segments can significantly improve performance. This optimization, while present in MODEL 204, was not implemented for the RIDBit prototype product, meaning that certain queries measured for the current paper did not take advantage of it. A number of other improvements in RIDBit were implemented during the course of these experiments, but this one was considered too difficult to complete in the time allotted.

We note that a RIDBit index can contain bitmaps for some index keyvalues and RID-lists for other values, or even for some segments within a value entry, according to whether the segment's bit density falls over or under the current division point of 1/50. In what follows, we will assume that a bitmap index combines verbatim bitmap and RID-list representations where appropriate, and continue to refer to the hybrid form as a bitmap index. When we refer to the *bitmap* for a given value v in a bitmap index, this should be understood to be a generic name: it may be a bitmap or a RID-list, or a segment-by-segment combination of the two forms; the term *verbatim bitmap* however, specifically stands for a bitmap that is not in RID-list form.

To retrieve the selected values from the table data, RIDBit needs to read the disk pages containing them. Due to the horizontal data organization, the whole row is read if any value from the row is needed.

4. ANALYSIS OF INDEX SIZE

FastBit and RIDBit implement different bitmap compression algorithms. Here we look at their effectiveness in reducing index size. For FastBit, index sizes are extensively treated in [19], where Figure 6 summarizes the index size per row over various kinds of data. Here we consider only the simple uniform random case for both FastBit and RIDBit. Note that RIDBit compression efficiency is not much affected by local clusters of bits in the bitmap, so the uniform random case is a good predictor of the general case. FastBit can take advantage of local runs of bits, as shown in [19].

For FastBit in the uniform random case, we have the following index size expression which is derived from equation (4) of Section 4.2 of [19], converted from size in words to size in bits. Here w = 32, for 32 bits per word in the current experiments.

FastBit index size per row, in bits = $(Cw/(w-1))\,(1 - (1 - 1/C)^{2w-2} - (1/C)^{2w-2})$ [19]. This expression can be further simplified for small C and large C as follows:

$(w/w-1)\,C$	small C
$(Cw/(w-1))(1 - e^{-(2w-2)/C}) \approx 2w$	$1 \ll C \ll N$
$5w = 160$	$C \sim N$

In the extreme case where C=N, each compressed bitmap is represented with 5 words, 3 of which are used to represent the bulk of the bits and the remaining 2 are used to represent the N%31 leftover bits.

The RIDBit implementation uses pages of size $4096 = 2^{12}$ bytes, holding 2^{15} bits (actually, 2^{15} - 16). A segment covers 2^{15} rows, using a bitmap or a segmented RID-list. The segment-relative RIDs of a segmented RID-list are 16 bits long and can start at any byte in a disk page (for current 4 KByte pages, and also for larger pages up to 8 KByte). A bitmap index has segment-relative bitmaps or RID-lists for each of C column values, ignoring cases with only one row per segment where segmentation is not used. Multiple RID-lists may share a page. If an index is composed entirely of RID-lists, the total size is 16 bits per row, for one segment-relative rid, while if it is entirely bitmaps, the total size is C bits per row, for 1 bit in each of C bitmaps. In the extreme case of C between a value on the close order of 32,000 and N: $k \bullet 32{,}000 \le C \le N$, where k is some small integer, a few rows for each column value per segment, each value requires 8 bytes for its local list entry. In summary, the number of bits per row used by RIDBit is as follows:

C	Bitmap segments; good for small C
16	Segmented RID-lists, $C < k \bullet 32{,}000$
64	Local RID-lists, $k \bullet 32.000 \le C \le N$

Of course an index can have a mixture of bitmaps

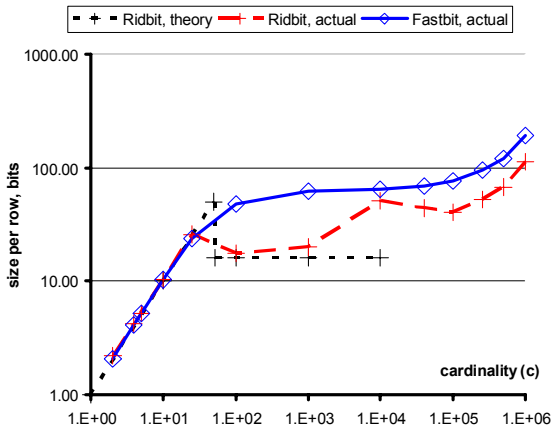

Figure 2: Index sizes vs. column cardinality.

and RID-lists, but this happens only in edge cases, since the decision is made based on a whole segment. The above simple formulas provide straight lines on both linear and log-log plots, and we use the latter in Figure 2 to cover more ground and allow easy comparison to Figure 6 in [19]. The above formula does not include the size of B-trees, which add a contribution of O(C) with a small constant of proportionality based on the number of per-key records in a leaf node.

We could minimize RIDBit index size by simply choosing the minimum of C and 16 bits per row, which suggests one to switch from bitmap to RID-list when $C \geq 16$. However, index size is not the most important criterion in deciding when to switch storage scheme. Experimentally, we have found that bitmaps are faster than segmented RID-lists for answering queries for C well above 16, even in face of the extra I/O needed. We choose to store the data in bitmaps for C up to 50. Possibly we should store the data on disk as RID-lists and construct bitmaps as we bring them into memory, but we have not tested this option.

The internal structures of RIDBit indexes for columns of various cardinalities C fall into three categories:

Low cardinality: $C \leq 16$, where segmented bitmaps are no larger than segmented RID-lists and faster.

Medium cardinality: $16 < C < 50$, where segmented bitmaps are larger than segmented RID-lists but still faster.

High cardinality: $C > 50$, where segmented RID-lists are much smaller and generally more efficient.

In the extreme case where $k \bullet 32{,}000 \leq C \leq N$, a few rows of each value per segment, segmentation is dropped in favor of unsegmented RID-lists of full-size local RIDs.

This analysis is easily generalized to using arbitrary p bits for segment-relative RIDs: just replace 16 with p above. With many of today's processors, including the Pentium 4 and its descendents, it is efficient to access groups of bits in memory without byte-alignment. Thus we could use p=12 bits to access any byte address in a 4 KByte page, and further compress the RID-lists.

Figure 2 shows the actual RIDBit and FastBit index sizes for the experiments reported in Section 6. The RIDBit index sizes shown here include the sizes of the B-trees, which were not included in our simple analysis, but became important for large cardinality (C = 1,000,000 for example). Similarly, the index sizes for FastBit are the total size of index files including the array starts[ ]. Overall, we see that the RIDBit indexes are never larger than FastBit indexes. When $C \leq 16$, both indexes have the same sizes; for larger C, FastBit indexes take about 64 bits per row while RIDBit index takes about 16 bits per row; when C is close to N ($=10^6$), FastBit index takes 192 bits per row and RIDBit index takes about 113 bits per row. As noted before, when C = N, each WAH compressed bitmap takes 5 words and one word is needed to store the starting position of the bitmap, which gives the total of 6 words per row. In this extreme case, RIDBit essentially stores a B^+-tree, which takes about 3.5 words per row.

5. THE SET QUERY BENCHMARK

We use a number of queries from the Set Query Benchmark to study performance of RIDBit and FastBit. The Set Query Benchmark was designed for query-mostly applications [7][8], and predates the Star-Schema data warehouse design. Since bitmap indexes are primarily used for high-performance queries, it is a natural choice. Due to the lack of support for join operations in both FastBit and RIDBit (we expect to add join capability to both products in the future), we only implemented the first five queries from the Set Query Benchmark. This lack of join support also ruled out the well-known TPC-H Benchmark [15].

The Set Query benchmark was defined on a BENCH table of one million 200-byte rows, containing a clustering column KSEQ with unique values 1, 2, 3, ..., in order of the rows, and a number of randomly generated columns whose names indicate their cardinalities, as shown in Table 4. For example, K5 has 5 distinct values appearing randomly on approximately 200,000 times each.

Queries of the Set Query Benchmark were modeled on marketing analysis tasks. We briefly describe the five SQL queries used for our timing measurements.

Q1: SELECT count(*) FROM BENCH WHERE KN=2; KN is one of KSEQ, K500K, …, K2. There are 13 different instances of Q1. Since it involves only one column at a time in the WHERE clause, we call Q1 a one-dimensional (1-D) query.

Q2A: SELECT count (*) FROM BENCH WHERE K2=2 and KN = 3; KN is one of KSEQ, K500K, …, K4. There are 12 instances of Q2A.

Q2B: SELECT count (*) FROM BENCH WHERE K2=2 and NOT KN = 3;

Table 4: Set Query Benchmark columns and their column cardinalities.

Name	Cardinality
KSEQ	1,000,000
K500K	500,000
K250K	250,000
K100K	100,000
K40K	40,000
K10K	10,000
K1K	1,000
K100	100
K25	25
K10	10
K5	5
K4	4
K2	2

KN is one of KSEQ, K500K, …, K4. Both Q2A and Q2B are two-dimensional queries since each WHERE clause involves conditions on two columns. At one time, the "NOT KN = 3" clause was difficult to support efficiently.

Since the above three queries only count the number of rows satisfying the specified conditions, we say they are *count queries*. Both FastBit and RIDBit answer count queries using INDEX ONLY.

Q3A: SELECT sum(K1K) FROM BENCH WHERE KSEQ between 400000 and 500000 and KN=3; KN is one of K500K, K250K, …, K4. There are 11 instances of Q3A.

Q3B: SELECT sum(K1K) FROM BENCH WHERE (KSEQ between 400000 and 410000 or KSEQ between 420000 and 430000 or KSEQ between 440000 and 450000 or KSEQ between 460000 and 470000 or KSEQ between 480000 and 500000) and KN=3; KN is one of K500K, K250K, …, K4.

Q3A0 and Q3B0: We include a variation of the above two queries by replacing the SELECT clause with "SELECT count(*)", making them count queries like Q1 and Q2.

Q4: SELECT KSEQ, K500K FROM BENCH WHERE constraint with 3 or 5 conditions. The constraints come from the Table 5. Queries Q4A selects 3 consecutive conditions from Table 5, such as, 1-3 and 2-4, and Q4B selects 5 consecutive conditions, such as 1-5 and 2-6. Our tests use 8 instances of Q4A and Q4B, where the last two instances of Q4B uses the first two conditions when there are no more conditions at the end of the list. To answer these queries, multiple indexes are needed and results from each index have to be combined.

The original Q4A and Q4B had a select clause with two columns, KSEQ and K500K. In our tests, we vary the number of columns selected from 0 (an index-only query retrieving count(*)) to 13. This creates more test cases for a better comparison between different data organizations.

Q5: SELECT KN1, KN2, count (*) GROUP BY KN1, KN2; for each (KN1, KN2) in {(K2, K100), (K4, K25), (K10, K25)}. There are three instances of Q5.

In the following tests, this query is implemented as a set of queries of the form "SELECT count (*) WHERE KN1=x and KN2=y," where x and y are distinct values of KN1 and KN2. We choose to answer Q5 this way mainly to exercise the indexing performance of FastBit and RIDBit, even though FastBit can support this query directly [12]. Thus, this is a count query using index only.

Table 5: Range conditions used for Q4.

(1)	K2 = 1
(2)	K100 > 80
(3)	K10K between 2000 and 3000
(4)	K5 = 3
(5)	K25 in (11, 19)
(6)	K4 = 3
(7)	K100 < 41
(8)	K1K between 850 and 950
(9)	K10 = 7
(10)	K25 in (3, 4)

6. INDEX PERFORMANCE EXPERIMENTS

We present the performance measurements in three parts, the time to construct the indexes, time to answer the count queries and time to answer the retrieval queries. Before presenting the timing measurements, we briefly describe the test setup.

Experiment Setup

We performed our tests on a number of different Linux 2.6 machines with ext3 file systems on various types of disk systems. Table 6 shows some basic information about the test machines and the disk systems. To make sure the full disk access time is accounted for, we un-mount the file system and then mount the file system before each query. Under Linux, this clears the file system cache. To avoid spending an excessive amount of time on mount/un-mount, we duplicated the test data four times to generate a total of five sets of the same data files. This allows us to run each query five times on different data between each pair of mount/un-mounts. Since the timing measurements are performed on five copies of the data files, we also avoid potential performance traps related to any peculiar disk placement of the files. All of these operations are repeated six times to give a total of 30 runs for each query, and the time we report is the median elapsed time for all 30 runs, measured by the function

Table 6: Information about the test systems.

	CPU		disk		
	Type	Clock (GHz)	Type	Latency (ms)	Speed (MB/s)
HDA	Pentium 4	2.2	EIDE	7.6	38.7
MD0	Pentium 4	2.2	Software RAID0 (2 disks)	9.4	58.8
SDA	Pentium 4	2.8	Hardware RAID0 (4 disks)	15.8	62.2
SDB	PowerPC 5	1.6	SCSI	8.3	54.4

Table 7: Total index sizes (MB) and the time (in seconds) needed to build them.

		RIDBit	FastBit
	Size	64.2 MB	93.5b MB
time	**HDA**	75.7 sec	21.7 sec
	MD0	8.3 sec	27.2 sec
	SDA	3.5 sec	34.8 sec
	SDB	4.0 sec	41.7 sec

gettimeofday.

Index Building

The total time used by RIDBit and FastBit to build indexes is shown in Table 7. In Figure 3 we examine in detail how the time is spent in building different indexes, taking the **MD0** system as representative of the four systems measured. The elapsed time shown in Figure 3 is the medium value of building indexes for five separate copies of the test data. The total time reported in Table 7 is the sum of these medium values.

In Figure 3 we see that RIDBit requires slightly more time to build low-cardinality indexes and FastBit requires considerably more time to build high-cardinality indexes. In high-cardinality cases, FastBit generates a large number of small bitmap objects and spends much time in allocating memory of these bitmaps. RIDBit maintains a pre-allocated stack of page-sized buffers, and thus avoid the same pitfall.

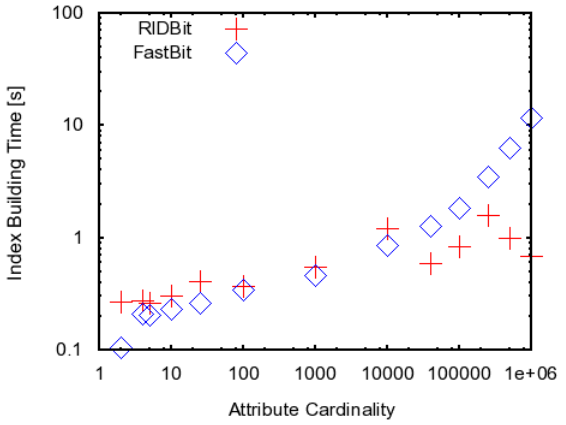

Figure 3: Time (in seconds) required to build each individual index on system MD0.

An overview of all the timing measurements on count queries is presented in Table 8. In this table, we show the total time of all instances of each query, for example, the row for Q1 is the sum of the median elapsed time for 13 instances of Q1. The last row in the table shows the total time of all count queries. On three of the four test systems, the total time used by FastBit and RIDBit are within 10% of each other, with FastBit taking less time Q1, Q2 and Q4 while RIDBit taking less time on Q5. The performance of RIDBit was improved during this joint measurement effort by emulating some of features of FastBit, as we will explain in Section 7. On the fourth system, **SDB**, the performance difference between RIDBit and FastBit was traced to an unexpected overhead for per I/O operation at the lower levels of that I/O system apparently impacted RIDBit more than FastBit.

Table 8: Total elapsed time (seconds) to answer the count queries on four test systems.

	HDA		**MD0**		**SDA**		**SDB**	
	RIDBit	FastBit	RIDBit	FastBit	RIDBit	FastBit	RIDBit	FastBit
Q1	0.39	0.23	0.50	0.25	0.34	0.26	0.52	0.22
Q2A	0.74	0.42	0.68	0.51	0.50	0.47	0.85	0.53
Q2B	0.71	0.42	0.66	0.49	0.53	0.46	0.88	0.52
Q3A0	2.28	2.18	2.00	1.91	1.79	1.73	1.97	2.06
Q3B0	2.08	2.46	1.76	1.90	1.49	1.41	1.87	1.81
Q4A0	1.39	0.94	1.22	0.83	0.97	0.77	2.10	1.03
Q4B0	2.20	1.46	1.75	1.31	1.67	1.21	2.98	1.65
Q5	1.13	1.44	1.09	1.46	0.81	1.21	1.03	1.50
Total	10.92	9.55	9.66	8.66	8.10	7.52	12.20	9.32

Both FastBit and RIDBit (modified during the joint work) have arranged index data so that most of the range predicates performed use sequential reads of a relatively large number of disk sectors; thus the total execution time of these accesses should be dominated by the time to read the disk sectors. To verify this is indeed the case, we show the number of disk sectors read in Table 9. Since the numbers of disk sectors read on different systems are nearly identical[2], we

Index-Only Query Performance

Here we review the time required to answer the count queries. These timing measurements directly reflect the performance of indexing methods. We start with an overview of the timing results then drill down the details as we find various aspects of interest.

2 The precise number of disk sectors read may differ because there are potential differences in the number

Table 9: Number of disk sectors (in thousands) needed to answer count queries.

	RIDBit	FastBit
Q1	10.7	4.9
Q2A	15.0	9.2
Q2B	15.0	9.2
Q3A0	52.0	64.4
Q3B0	34.4	48.8
Q4A0	27.8	35.9
Q4B0	41.3	56.5
Q5	23.0	27.4
Total	219.2	256.3

only show the values from system **MD0**. We see that the indexing method that reads more disk sectors does not always uses more time, therefore we have to investigate further.

We next examine the performance on Q1 in detail. In Figure 4 the medium query response time is plotted against the number of hits for Q1. Since each instance of Q1 involves only one bitmap from one index, it is relatively easy to understand where the time is spent. The time used by FastBit is primarily for two read operations: first, to read the starting positions of the index structure shown in Table 3, and second, to read the selected bitmap. These two read operations may each incur 9.4 ms I/O latency, which leads to a total elapsed time of about 0.02 s, unless the selected bitmap happen to be in the read-ahead buffer of the first read operation, which leads to a total time of about 0.01 s. Among the 13 instances of Q1, most are either 0.01 s or 0.02 s. When the number of hits is very small and the cardinality of the column is high, it takes more time to complete the first read operation. The I/O time of RIDBit can also be divided into two parts: first to read the tablespace index blocks involved (listing the positions of pages in the tablespace), and at the same time access the B-tree root node and a few index nodes, and second (in all KN=2 cases where N is 100K 250K, 500K, and SEQ, the cases using unsegmented RID-lists) to read in the bitmap or RID-list that will determine the count to be retrieved. Since each of these operations requires at least 9.3 ms (and in fact the first one to read in the index blocks of the tablespaces requires 17 ms), the total time used by RIDBit is nearly 0.028 s (28 ms) in most cases.

The time needed to answer higher dimensional count queries is dominated by the time needed to answer each of the one-dimensional conditions. For example, the two-dimensional queries Q2A and Q2B involve two conditions of same form as Q1; we expect Q2A and Q2B to take about twice as much time as that of Q1. We see from the measurements on **MD0** in Table 8 that this estimate is accurate (a ratio of 0.51/0.25 = 2.04); on the other hand, RIDBit has a much smaller increment of elapsed time (0.68/0.50 = 1.36), presumably because the initialization of a tablespace for a second index is easily combined with the initialization of the first tablespace. This observation holds for Q4A0 and Q4B0 as well. For example, the total time for Q4B0 on **MD0** is 1.31 s which is about 1.6 times of that for Q4A0. This relative difference is close to 5/3, the ratio of dimensions of the queries.

of I/O nodes involved in different file systems. In addition, the software RAID may require additional disk accesses to resolve the file content.

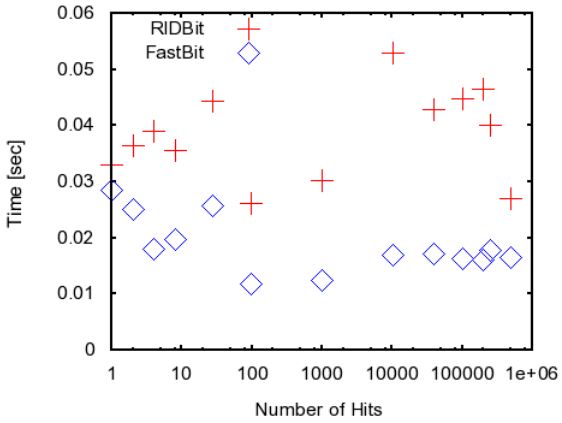

Figure 4: Elapsed time (seconds) to answer Q1 on MD0.

We see that the query response time for queries Q1, Q2A, Q2B, Q4A0 and Q4B0 follows our expectation. In these cases, the time used by FastBit is slightly less than that used by RIDBit. Table 10 shows the CPU time used to answer the count queries on **MD0**. Compared with the elapsed time reported in Table 8, we see the CPU time is usually 1/5th of the elapsed time or less for queries Q1, Q2A, Q2B, Q4A0 and Q4B0. The query processing time follows our expectation partly because the I/O time is so much more than the CPU time. Next we examine the cases for Q3 and Q5.

Figure 5 shows the elapsed time to answer each instance of Q3A0 on **MD0**. We notice that the time values fall in a very narrow range; the maximum and minimum values are within 20% of each other. This is because the time to resolve the common condition on KSEQ dominates the total query response time. To resolve this condition on KSEQ, FastBit reads 100,001 compressed bitmaps of about 5 words each,

Table 10 Total CPU time (seconds) to answer count queries on MD0.

	RIDBit	FastBit
Q1	0.016	0.045
Q2A	0.028	0.059
Q2B	0.038	0.061
Q3A0	0.500	0.672
Q3B0	0.307	0.521
Q4A0	0.137	0.111
Q4B0	0.192	0.165
Q5	0.701	0.795
Total	1.919	2.429

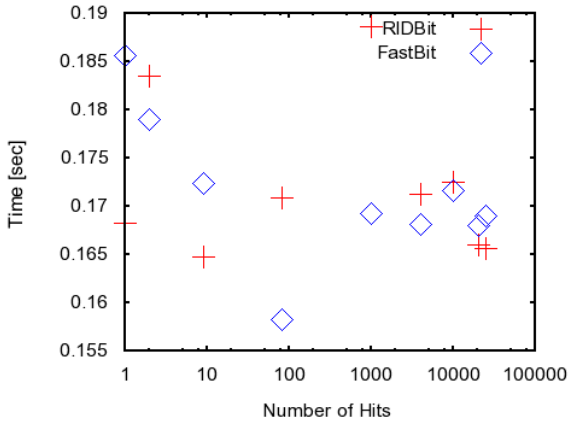

Figure 5 Elapsed Time (seconds) to answer Q3A0 on MD0.

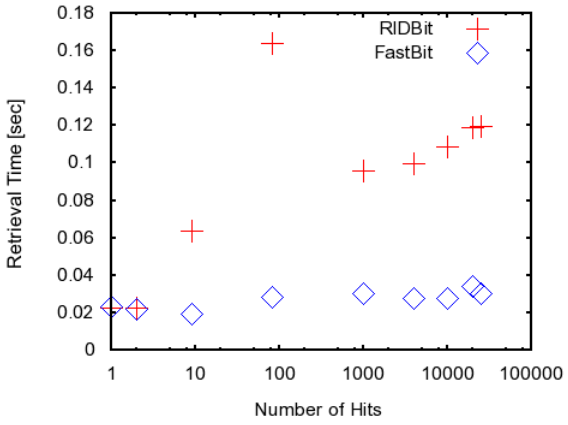

Figure 6 Time spent to retrieve the selected records to answer Q3A on MD0.

while RIDBit reads 100,001 leaf nodes of the B-tree with an average size about 3.5 words each. Even though FastBit reads more data than RIDBit, it doesn't always use more I/O time because it reads all bitmaps in one sequential read operation. Since the bitmaps selected by the conditions on KSEQ in Q3B0 can not be read in one operation, FastBit usually uses more time than RIDBit. From Table 10, we see that Q3A0 and Q3B0 also require more CPU time than Q4A0, Q4B0 and other. In FastBit, this CPU time is primarily spent on reconstructing the large number of C++ bitmap objects. On Q3A0 and Q3B0, RIDBit uses less CPU time than FastBit.

Another query where RIDBit is faster than FastBit is Q5. From Table 10 we see that RIDBit requires about 13% less CPU time on **MD0**, which again suggests that RIDBit is more CPU efficient than FastBit. The difference in elapsed time is larger (about 25% on **MD0**) than that in CPU time because FastBit indexes are larger than RIDBit indexes.

Table Retrieval Query Performance

Next we present measurements of table retrieval queries. We present the measurements on Q3 before those on Q4 because Q3 only retrieves a sum of values from one column, while Q4 retrieves a varying number of columns.

Figure 6 shows the time required to retrieve the column selected in Q3A. The time values shown are the differences between query response time of Q3A and that of Q3A0. Overall, we see that the time required by FastBit slowly rises as the number of hits increases. RIDBit uses about the same amount of time as FastBit when one or two records are retrieved; but it uses more time when the number of hits is larger. The time required to retrieve the values for Q3B has similar trend as that for Q3A.

Because of the condition on KSEQ, the records selected by Q3A and Q3B are from between row 400,000 and 500,000. The second condition in Q3A controls how many records are selected and how they are distributed. Since all columns in test data are uniform random numbers, the selected records are uniformly scattered among rows 400,000 to 500,000. The Operating Systems on our test machines all retrieve data from disk in pages (of either 4 KB or 8KB). To better understand the retrieval time, we compute how many pages are accessed assuming 4 KB pages.

Let m denote the number of rows in a data page. The RIDBit and FastBit use different organization for the table data, which leads to different number of records to be placed on a page. RIDBit uses a horizontal organization; FastBit uses a vertical data organization. The number of records per 4-KB page for RIDBit is 75. The number of records per 4-KB page for FastBit is 1024. We use m_h to denote the number of records per page for the horizon data organization, and use m_v to denote the number of records per page for the vertical data organization. Let n_h denote the number of pages for the 100,000 rows between 400,000 and 500,000 in the horizontal organization, $n_h = 100{,}000/m_h = 1{,}334$. Let n_v denote the number of pages for 100,000 records in vertical organization, $n_v = 100{,}000/m_v = 98$. If every page is touched, clearly, there is an advantage to use vertical data organization. Next, we examine a more general case, where s records are randomly selected. Assuming that s is much smaller than 100,000, we can use the following formulae to estimate the number of pages to be accessed [8]: $p_h = n_h (1-\exp(-s/n_h))$ and $p_v = n_v(1-\exp(-s/n_v))$.

Figure 7 shows the number of disk sectors accessed for the retrieval operation. The number of disk sectors shown is the difference of the number of disk sectors accessed to answer Q3A and that to answer Q3A0. In the same plot, we also show the number of disk sectors to be accessed using the above

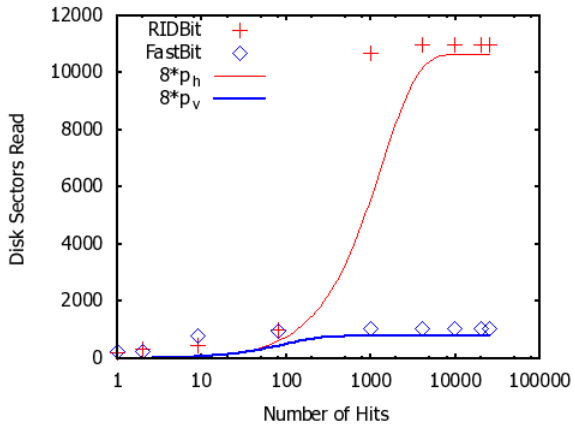

Figure 7 Number of disk sectors accessed to retrieve the records for Q3A.

formulae for p_h and p_v. The multiplying factor of 8 is to translate the 4 KB pages to 512-byte disk sectors. In general, the actual number of disk sectors accessed agrees with predictions. The actual disk sectors accessed is typically more than the prediction because the I/O system performs read-ahead.

Comparing Figure 6 and Figure 7, we see that the time used for retrieval generally follows the number of disk sectors accessed. We note two deviations. When the number of disk sectors accessed is small, the I/O overhead, in particular, the disk seek time, dominates the total retrieval time. As the number of disk sectors accessed increases, the retrieval time increases proportionally until nearly all of the disk sectors are accessed. In which case, the retrieval time may actually be less because the data file can be read into memory with large sequential read operations. This can either be accomplished by the OS or the database software.

Our modified versions of Q4 retrieve 0, 1, 3, 5, 7, 9, 11 and 13 column values. In Figure 8 and Figure 9, we show the total query response time against the number of columns selected for Q4A and Q4B on **MD0**. In these figures, each symbol shows the total time of 8 instances of Q4A (or Q4B) with the same number of columns selected.

From Figure 8, we see that the total time used with FastBit's vertical data organization increases linearly with the number of columns selected. In Figure 8, the slope of the line form by FastBit is about 0.8, which indicates that in 0.8 seconds it can read 8 copies (8 instances of Q4Ax) of the 4-MB data file. This reading speed of about 40 MB/s is about 68% of the asymptotic reading speed of 58.8 MB/s shown in Table 6.

The timing measurements of RIDBit show the expected behavior for horizontal data organization. It takes the same amount of time as long as some columns are retrieved. In Figure 8, we see that retrieving data in the vertical data organization usually takes less time than those in horizontal organization. The line for the vertical data organization intersects that for the horizontal organization around 11. When more than 11 (out of 13) columns are retrieved, using the horizontal data organization takes less time.

The maximum number of hits from Q4A is about 1,100. In horizontal organization, there are 13,334 pages for the table data. Therefore, RIDBit does not need to access all pages. For FastBit, Each data file in the vertical data organization takes up 977 pages and nearly all these pages are accessed by FastBit. In this case, FastBit uses one sequential read on each data file. In contrast, RIDBit is reading one page at a time or a small number of pages at a time. Depending on the relative performance of random reads to sequential reads, the line for vertical data organization may cross the one for horizontal data organization at different locations. Of course, this

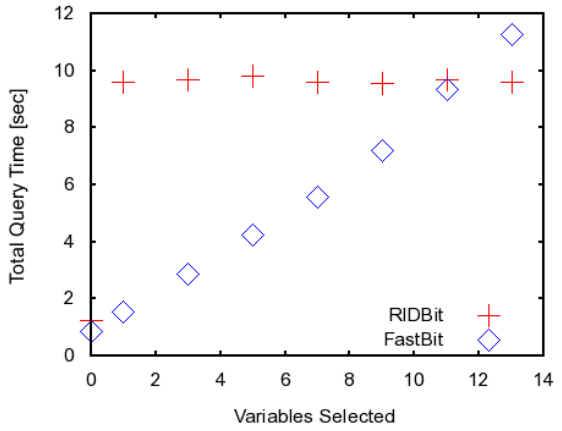

Figure 8 Total time used to answer Q4A on MD0.

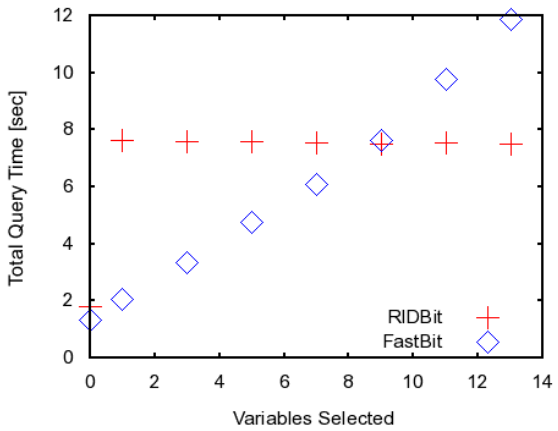

Figure 9 Total time used to answer Q4B on MD0.

cross over point also depends on the number records selected as illustrated in Figure 9.

Each Q4B query selects about 100 hits on average. In this case, RIDBit only needs to access 100 pages no matter how many columns are selected as shown in Figure 9. In contrast, FastBit accesses about 100 pages per column. We expect the horizontal data organization to have an advantage over the vertical data organization in this case. From Figure 9, we see that if less than 7 columns are selected, FastBit in fact uses less time. This is because FastBit decides to read the whole data file if more than $1/16^{th}$ of the pages are randomly selected. This option reads more pages than necessary; however, because the sequential reads are much more efficient than random reads, reading more data sequentially actually take less time in this case.

7. SUMMARY AND CONCLUSIONS

We outline what lessons we have learned from our performance tests. To be of value, these lessons should indicate how we would proceed if we were implementing a new bitmap index on a commercial database product. Though it took some time for these lessons to become clear to us, we believe the results are worth the effort.

Vertical Data Organization Has Better Performance

It seems clear that vertical data organization (columns stored separately) has an important architectural advantage over row-store for queries. Certainly, were we to make a major modification of RIDBit, the first thing we would do is to adopt this format. From Figure 8 and Figure 9, we see that the queries that retrieve one column take a much longer elapsed time than those simply counts the number of hits, even though the WHERE clause contains five different range conditions. This is the case even if as few as 100 records are retrieved as shown in Figure 9. Our tests also showed that for queries retrieving a small number of columns in a table the vertical data organization is much more efficient. Only if nearly all columns are retrieved is the row-oriented organization more efficient. Most queries that occur in commercial applications do not retrieve a large percentage of the columns in a row, for example, most queries in TPC-H retrieve two to five columns, so it seems clear that the vertical data organization is preferred.

Clustered Index Organization Has Better Performance

In terms of bitmap index organization, the linear organization of FastBit shown in Table 3 is more efficient for processing range queries because the bitmaps can be read into memory with a single sequential scan of the bitmaps[ ] array, once the starts[ ] array has determined the start and end position of the bitmaps on disk. As it stands, this approach trades flexibility of the index data structure for performance. The most severe limitation of this index organization is that any modification to the index will cause the whole index file to be reorganized, which would be exceedingly expensive.

We found in modifying RIDBit to reduce the number of disk scans for a single range query that we could read the appropriate leaf nodes of the B-tree into memory (in a single sequential scan, once the initial and terminal keyvalue leaves of the B-tree are determined), then learn the positions of the initial and terminal bitmap/RID-list in the range. This is simple because during the initial load of the index, successive keyvalues K and successive segments S within each keyvalue are placed in lexicographic order by (K,S) and the B-tree is built in left-to right order while the bitmaps/RID-lists are also placed on disk in that order. Therefore it is possible to use approximately the same approach to the RIDBit B-tree/Bitmap layout that FastBit does, performing a few long sequential scans to access all bitmaps/RID-lists. Furthermore, since the leaf level of the B-tree is present in memory, we can validate if some newly inserted rows lie outside the range and access them as well; we still cannot insert an arbitrary number of new rows in the middle of the sequence (because of the risk of a RID-list becoming too large and requiring re-positioning), but we can insert such rows up to that point and afterward place them in a new position at the end of all the segments, where this will be detected by an examination of leaf pages in the desired range. While this approach is not perfect, it is comparable to what DB2 does in terms of clustered indexes.

We note that the problems with inserts disappear entirely in the case of a product such as Vertica, where Read-Optimized Store remains unchanged and new rows are added to Write-Optimized Store until enough additions require a merge-out to form a new Read-Optimized Store.

Modifications for Modern Processors Are Needed

There are a number of ways in which older indexing methods are inefficient on modern processors. Oracle's index compression approach, known as Byte-Aligned Bitmap Code (BBC), uses a type of compression/decompression that requires a good deal of branching; this can be terribly inefficient because it causes pipeline stalls that are much more expensive on modern processors than

they were when BBC was introduced. Indeed we found that using Branch-Avoiding C code on Pentium 4 to precompute conditions rather than using if-then-else forms was important for improved performance. Another change over the past fifteen years or so is that sequential scans have become much more efficient, requiring much smaller filter factors (by a factor of fifty) before a list prefetch of pages becomes more efficient than a sequential scan that simply picks up more rows. It because of this that clustering has become more important, leading to such new and important capabilities as DB2's Multi-Dimensional Clustering (MDC). All of this should be born in mind in implementing a new indexing method for today's processors.

FastBit indexes are usually larger than RIDBit indexes, but it can answer many queries in less time because it accesses the needed bitmaps in less I/O operations. Obviously, when a large fraction of the bitmaps is needed, FastBit will take more time. FastBit typically spends more CPU time in answering queries than RIDBit, though the CPU time differences are small compared with those of I/O time.

In summary, we recommend the vertical organization for base data and the linear (or packed) organization for the bitmap indexes to achieve good query performance. To insulate the indexes from changes in the base data, we suggest using separate Read-Optimized Store and Write-Optimized Store as with Vertica.

8. REFERENCES

[1] P. A. Boncz, M. L. Kersten. Monet: An Impressionist Sketch of an Advanced Database System. In Proceedings Basque International Workshop on Information Technology, San Sebastian, Spain, July 1995.

[2] P. A. Boncz, F. Kwakkel, M. L. Kersten. *High Performance Support for OO Traversals in Monet*. Technical Report CS-R9568, CWI, Amsterdam, The Netherlands, 1995.

[3] T. Johnson. Performance Measurements of Compressed Bitmap Indexes. In *VLDB,* Edinburgh, Scotland, September 1999. Morgan Kaufmann.

[4] Kx Systems. http://kx.com. 2006.

[5] Microsoft. SQL Server Database Engine: Logical and Physical Operators Reference. http://msdn2.microsoft.com/en-us/library/ms191158.aspx.

[6] P. O'Neil. Model 204 Architecture and Performance. In 2nd International Workshop in High Performance Transaction Systems, Asilomar, California, USA, September 1987. Springer-Verlag.

[7] P. O'Neil. The Set Query Benchmark. In *The Benchmark Handbook For Database and Transaction Processing Benchmarks*, Jim Gray, Editor, Morgan Kaufmann, 1993.

[8] P. O'Neil and E. O'Neil. *Database Principles, Programming, and Performance*. 2nd Ed. Morgan Kaufmann Publishers. 2001.

[9] P. O'Neil and D. Quass. Improved Query Performance with Variant Indexes. In *SIGMOD*, Tucson, AR, USA, May 1997. ACM Press.

[10] PostgreSQL: PostgreSQL 8.1.5 Documentation, Chapter 13. Performance Tips. http://www.postgresql.org/docs/8.1/interactive/performance-tips.html.

[11] D. Rinfret, P. E. O'Neil and E. J. O'Neil. Bit-Sliced Index Arithmetic. In *SIGMOD*, Santa Barbara, CA, USA, May 2001. ACM Press.

[12] K. Stockinger, E. W. Bethel, S. Campbell, E. Dart, K. Wu. Detecting Distributed Scans Using High-Performance Query-Driven Visualization. *Supercomputing 06*. 2006.

[13] K. Stockinger and K. Wu. Bitmap Indices for Data Warehouses. In *Data Warehouses and OLAP*. 2007. IRM Press. London.

[14] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran and S. Zdonik. C-Store: A Column Oriented DBMS. *VLDB*, pages 553-564, 2005.

[15] TPC-H Version 2.4.0 in PDF Form from http://www.tpc.org/tpch/default.asp

[16] R. Winter. Indexing Goes a New Direction. 1999. http://www.wintercorp.com/rwintercolumns/ie_9901.html.

[17] K. Wu, J. Gu, J. Lauret, A. M. Poskanzer, A. Shoshani, A. Sim, and W.-M. Zhang. Grid Collector: Facilitating Efficient Selective Access from Data Grids. In *International Supercomputer Conference* 2005, Heidelberg, Germany.

[18] K. Wu, E. J. Otoo, and A. Shoshani. Compressing bitmap indexes for faster search operations. In *SSDBM'02*, pages 99-108, 2002.

[19] K. Wu, E. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems*, v 31, pages 1-38, 2006.