DOI: 10.1145/1866480.1866497
IDEAS '10 Proceedings of the Fourteenth International Database Engineering & Applications Symposium
Pages 114-116

Multi-Level Bitmap Indexes for Flash Memory Storage*

Kesheng Wu, Kamesh Madduri and Shane Canon
Lawrence Berkeley National Laboratory, Berkeley, CA, USA
{KWu, KMadduri, SCanon}@lbl.gov

July 23, 2010

Abstract

Due to their low access latency, high read speed, and power-efficient operation, flash memory storage devices are rapidly emerging as an attractive alternative to traditional magnetic storage devices. However, tests show that the most efficient indexing methods are not able to take advantage of the flash memory storage devices. In this paper, we present a set of multi-level bitmap indexes that can effectively take advantage of flash storage devices. These indexing methods use coarsely binned indexes to answer queries approximately, and then use finely binned indexes to refine the answers. Our new methods read significantly lower volumes of data at the expense of an increased disk access count, thus taking full advantage of the improved read speed and low access latency of flash devices. To demonstrate the advantage of these new indexes, we measure their performance on a number of storage systems using a standard data warehousing benchmark called the Set Query Benchmark. We observe that multi-level strategies on flash drives are up to 3 times faster than traditional indexing strategies on magnetic disk drives.

1 Introduction

The technology for storing data on spinning magnetic disks has been in use for many decades. Over the years, while disk capacity has increased dramatically, the data transfer rate to and from the disks has increased at a slower pace, and the disk access latency has remained the same for most consumer products in recent years. The slow data transfer rates and long access latencies are bottlenecks for data analysis tools. The good news is that the technology for manufacturing flash memory for persistent storage has matured significantly, and flash memory disks are reaching the mass market. These flash memory disks use electronic circuits to store information, which provides lower access latency and faster data transfer rate. They have the potential to significantly improve the efficiency of database applications [11].

In this paper, we concentrate on a specific data analysis application: answering queries on large data sets. The straightforward solution called *full scan* examines all the data records to evaluate which ones satisfy the query conditions. A common acceleration strategy is to develop an auxiliary data structure called an *index*. The full scan requires all the records and is usually implemented with sequential data disk accesses.

*The authors gratefully acknowledge Dr. Douglas Olson for the use of his storage allocation. We also like to express our gratitude to David Paul and Matthew Andrews for their help in working with the Fusion I/O devices. This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy.

In contrast, answering a query with indexes typically accesses a lower data volume than the full scan, but may read the required data in a number of I/O operations. On magnetic disks, because the access latency is high, most indexing methods aggregate their read operations to reduce the number of disk accesses. On flash memory devices, with their significantly lower access latency, we may potentially perform a larger number of smaller disk reads to answer queries faster. The multi-level bitmap indexes are designed in this spirit.

The main contributions of this paper includes the following.

- Design a set of multi-level bitmap indexes, and prove that the total number of bytes read in order to answer a query is reduced with these methods. These multi-level indexes also require less computation to answer a query than commonly used bitmap indexes.
- Evaluate the performance of a new flash memory-based system, and compare it with other high-performance storage systems, e.g., RAID and parallel file systems.
- Present a brief analysis of the query processing cost that takes into account all the operations on metadata and files. Earlier analyses have typically concentrated only on the sizes of the bitmaps involved.
- Conduct an empirical evaluation of the new indexes using an established benchmark suite called the *Set Query Benchmark*. Thus, our conclusions are more representative of the performance a typical user may observe.

This paper is organized as follows. We describe some related work in Section 2. We present the design of multi-level bitmap indexes and give a brief account of practical operational costs in Section 4. We discuss the four test systems in Section 5, and present the measurements in Section 6. Finally, we conclude with a brief summary of our contributions in Section 7.

2 Related Work

In this section, we briefly describe the technologies related to our study and give some justifications of the choices made. More specifically, we review I/O technologies, indexing methods, and the benchmark used.

I/O technologies. As of 2010, a typical computer has one disk attached through a serial ATA (SATA) interface[1]. The SATA interface is capable of supporting 3 Gb/s data transfer rate, and the peak speed of a disk is about 300 MB/s. The sustained read speed is usually less than 100 MB/s, and the write speed is about the same as the read speed. With a spinning disk, the access latency is primarily determined by the rotation rate. For a disk rotating at 7200 revolutions per minute (RPM), it takes about 8.3 ms for the disk to turn a round. On an average, one needs to wait about half a revolution before the correct data can be read by the disk head. The disk access latency is thus 4.2 ms in this case.

Common ways to improve I/O performance include caching the content in memory and aggregating multiple disks to increase throughput. Caching takes advantage of the fact that many data records are accessed repeatedly, and stores these frequently accessed records in a memory cache. The main limitation is that there must be a good amount of reuse in the application.

There are several different ways of aggregate multiple disks. One commonly used strategy is called redundant arrays of inexpensive disks (RAID) [18]. A simple version of this is known as RAID0, which

[1] More information about SATA can be found at `http://www.serial-ata.org/`.

splits a logical file into fixed-size blocks and stores the blocks on the disks one after another. This option allows multiple disks to be used at the same time, which increases the I/O throughput. Other forms of RAID systems can additionally provide fault tolerance, error correction, and so on. Though a RAID system can provide faster read and write speeds, it may also increases the access latency, because of the need to coordinate the accesses to multiple disks. A RAID can be controlled by a dedicated hardware controller or through the system software. Typically, a RAID with a dedicated hardware controller has a lower access latency. However this access latency is still longer than that of a simple disk.

Other common ways of aggregating multiple disks include network attached storage (NAS) [7] and storage area network (SAN) [5]. These systems employ dedicated fiber channels to carry data traffic. There are a number of different parallel file systems such as GPFS [19], PVFS [2], and Google File System [10], that can provide a coherent file structure over disks attached to different computers. As with RAID systems, these technologies provide a higher aggregate I/O speed and improve fault tolerance. In addition, some of the popular parallel file systems can provide superb analytical capability by coordinating the storage resources and computational resources.

In our study, we compare three common I/O systems against the flash memory storage technology. They are a single serial ATA (SATA) disk, a software RAID of five SATA disks, and a GPFS parallel file system.

Flash storage. Solid-state devices (SSDs) are rapidly emerging as a viable alternative to magnetic disks. Due to their popularity in mobile consumer electronics such as digital cameras and cellular phones, the prices of flash devices are falling quickly. The primary advantage of flash devices is that the access latency is significantly lower due to lack of mechanical parts. Unlike mechanical storage devices, where a single I/O seek needs to be amortized by a read/write of a large sequential block of data, I/O accesses are much cheaper on flash storage, and even random I/O incurs a substantially lower cost.

The database research community has proposed a number of new algorithms and modifications of existing algorithms to take advantage of this emerging storage media [11, 12]. Here are some examples. Shah et al. proposed the RARE-join algorithm for SSDs [20]. Myers studied how several common database algorithms are affected by IDE-based flash memory devices [13]. Do and Patel revisited the I/O costs of common ad-hoc join algorithms and evaluated how their parameters need to be tuned to achieve optimal performance on flash devices [8]. Tsirogiannis et al. introduced the FlashJoin algorithm for binary joins, and studied how data structures such as PAX speed up query processing on flash devices [25].

Most the studies point out the relatively-slow write throughput capabilities of flash devices. However, for typical *data warehousing* applications, the amount of write operations can be minimized with techniques such as bitmap indexes [15, 27]. In this work, we present a set of multi-level indexes that reduce the number of bytes needed to answer a query, by accessing the bytes in smaller chunks. We anticipate that this trade-off will reduce the overall query response time on flash devices.

Indexing methods. The best-known indexing method in the database community is the *B-tree* [6]. There are many variants of B-trees used in database management systems. They are designed to balance the need to accelerate queries and to update themselves after modifications to data records. This is essential in transactional applications such as banking and Internet commerce. However, for most scientific and statistical databases, there is not need to support transactions. In many cases, the datasets are either read-only or updated through bulk append operations. Most operations on these datasets are queries. Such datasets are commonly referred to as *data warehouses*.

Another very commonly used indexing method, particularly suited for data warehouses, is the *bitmap index* [15]. Major commercial database management systems, such as ORACLE and IBM DB2, have some

variations of the bitmap index. One key advantage of bitmap indexes is that they can typically answer queries much more efficiently than B-trees. In particular, partial solutions from multiple bitmap indexes can be easily and efficiently combined to answer a more complex query, while it is much harder to combine answers from B-trees.

For our software implementation, we choose to work with FastBit[2] due to its advanced compression method [27]. Furthermore, it also stores the user data in a column-based organization, which is known to be efficient for data warehousing applications [14, 24]. In FastBit, each column is also its own projection index [17]. Using such a projection index to resolve a query requires only a single read operation, whereas using other common bitmap index schemes generally requires more read operations.

Benchmark to use. In many published studies mentioned above, measurements are obtained using a small number of randomly-chosen queries. In this study, we decide to use an established benchmark to better represent a realistic query load. We select the Set Query benchmark, which is designed to mimic common business analysis queries [16]. Therefore, the test results should be more representative than random queries.

Other common benchmarks such as TPC-H were considered, but not chosen, because the test software is not able to answer some of the queries, such as those involving nested queries and some forms of multi-table joins. Furthermore, in answering queries from these benchmarks, the majority of the processing time goes into accessing the raw data rather than the indexes.

To evaluate the effectiveness of the new indexes, we focus on measuring time taken for queries that can be directly answered by the indexes. For this reason, we modified all queries from the Set Query benchmark to count the number of hits, i.e., the number of rows satisfying the query conditions, without ever actually retrieving the selected values. This allows us to concentrate on the index operations, particularly the trade-offs among the bitmap indexes.

3 Set Query Benchmark

The Set Query benchmark is a test suite based on common business analysis queries [16]. It consists of one data table with thirteen columns and 1 million rows. In our tests, we scale the dataset size to contain 100 millions rows. The original benchmark contains six queries. To emphasis the performance of index operations, we only use the first four, and modify the queries to count the number of hits only.

Most columns in the test data are named after their *column cardinalities*, i.e., the number of distinct values in the columns. Their names are: K2, K4, K5, K10, K25, K100, K1K, K10K, K40K, K100K, K250K and K500K. They have a common prefix K, followed by their column cardinalities. For example, K2 contains two possible values, 1 and 2; K500K contains 500,000 possible values from 1 to 500,000. It uses a simple pseudo-random number generator to produce a uniformly distributed values[3].

The remaining column is named KSEQ, which contains a sequence of numbers starting from 1 and is used as the identifier of the records in some of the queries. With 100 million rows, the value of this column simply goes from 1 to 100 million. Since each row of this column has a different value, we build a bitmap index with the new binning technique described in the next section.

The test data set is replicated 5 times to make it easier to gather timing statistics. For each run of our tests, we exercise all five replicas of the test data. The timing measurements reported later are the median values of these measurements.

[2] `http://sdm.lbl.gov/fastbit/`.

[3] The core random number generator uses a 32-bit seed. The seed is initialized to 1 and updates before each use by multiplying it with 16,807.

No.	condition	# bitmaps	No.	condition	# bitmaps
1	`K2 = 1`	1	6	`K4 = 3`	1
2	`K100 > 80`	20	7	`K100 < 41`	40
3	`K10K between 2000 and 3000`	1001	8	`K1K between 850 and 950`	101
4	`K5 = 3`	1	9	`K10 = 7`	1
5	`K25 = 11 or K25 = 19`	2	10	`K25 between 3 and 4`	2

Figure 1: List of range conditions used for Q4 of the Set Query benchmark. The column labeled **# bitmaps** shows the number of equality encoded bitmaps touched by the condition.

Next we briefly explain the queries used in our tests. Note that the data table is named BENCH. The SQL statement of the query Q1 is

```
select count(*) from BENCH where KN = 2;
```

where `KN` can be any of the thirteen column names.

The query Q2 has two variants as expressed in the following two SQL statements,

```
select count(*) from BENCH where K2 = 2 and KN = 3;
select count(*) from BENCH where K2 = 2 and not KN = 3;
```

where `KN` can be any of the twelve column names other than K2.

The query Q3 has two variants as follows,

```
select count(*) from BENCH where
  KSEQ between 400000 and 500000 and KN = 3;
select count(*) 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;
```

where `KN` can be any of the eleven columns other than K2 and KSEQ.

The query Q4 has two variants that select three or five conditions from the list given in Figure 1. For example, the first and eighth instances of Q4 with three conditions are

```
select count(*) from BENCH where K2 = 1 and K100 > 80
  and K10K between 2000 and 3000;
select count(*) from BENCH where K1K between 850 and 950
  and K10 = 7 and K25 between 3 and 4;
```

We construct eight instances of the variants with five range conditions by reusing the first two conditions after exhausting the ten listed in Figure 1.

		binary		equality				range			interval		
		$\&2$	$\&1$	$=0$	$=1$	$=2$	$=3$	≤ 0	≤ 1	≤ 2	$[0,1]$	$[1,2]$	$[2,3]$
RID	**X**	b_1	b_0	e_0	e_1	e_2	e_3	r_0	r_1	r_2	i_0	i_1	i_2
1	1	0	1	0	1	0	0	0	1	1	1	1	0
2	3	1	1	0	0	0	1	0	0	0	0	0	1
3	2	1	0	0	0	1	0	0	0	1	0	1	1
4	0	0	0	1	0	0	0	1	1	1	1	0	0
$\vdots$	$\vdots$												

Figure 2: An illustration of the basic bitmap encoding methods for bitmap indexes: binary, equality, range, and interval encodings.

4 Bitmap indexes

In this section, we explain the multi-level bitmap indexes. To simplify the discussion, we only consider indexing one column at a time. We give a brief analysis of the expected query processing cost in our current implementation. As a brief aside, we also describe a low-precision binning strategy that turns out to be very effective for a column of the Set Query Benchmark data.

4.1 Common bitmap encoding methods

Following the established terminology of describing bitmap indexes [22], the techniques for composing a bitmap index can be divided into three categories: encoding, binning, and compression. The multi-level methods fall into the category of encoding. They are composed from the more basic bitmap encoding methods that are summarized in the next paragraph.

Figure 2 contains an illustration of the basic bitmap encoding methods in literature [15, 3, 4]. This illustration uses an integer column **X** as an example, which can be one of four values 0, 1, 2, and 3. Each row of the data is denoted by an identifier called Row ID (RID in short). Each of the twelve columns on the right side of the table in Figure 2 is a *bitmap*, and each bitmap has as many bits as the number of rows in the data table. The binary encoding concatenates a binary digit from each row into a bitmap. The equality encoding defines a set of bitmaps where the value of a bit is determined by an equality test indicated on the top of the bitmap. Similarly, the range encoding uses a set of one-sided range conditions [3], e.g., $\mathbf{X} \leq 2$, and the interval encoding uses a set of overlapping two-sided range conditions [4], e.g., $2 \leq \mathbf{X} \leq 3$.

There are many ways of combining the above bitmap encoding methods into more complex bitmap indexes [3, 21]. These indexes fall into two broad categories: multi-component encodings and multi-level encodings [28]. The multi-component encodings break an input value into multiple components similar to the manner in which a number is broken into decimal digits. The main difference is that the bases for each component (i.e., digit) can be different. A common example of this scheme is to use base size 2 for each component, which produces two possible values for each component. Since the bitmaps representing these two values are complements of each other, it is possible to store only one of them for each component. This creates the binary encoding or the bit-sliced index, which is one of the most effective multi-component methods [17]. In later tests, we denote this method as BN. Another widely used multi-component method is the one-component equality encoded index (E1), which is highly compressible.

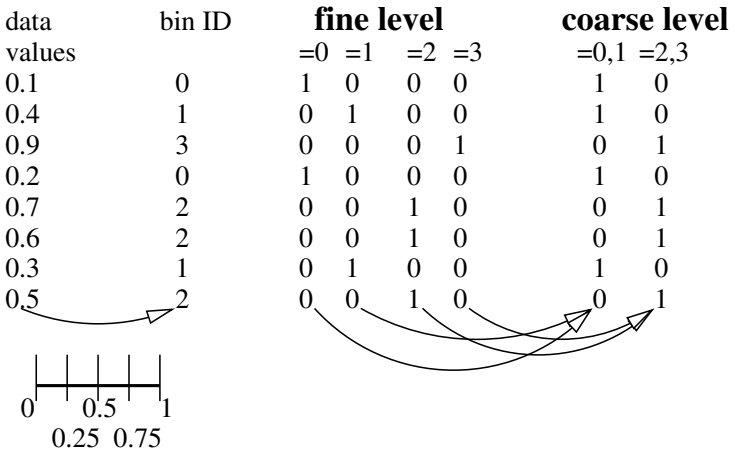

Figure 3: An illustration of the two-level equality-equality bitmap encoding method.

4.2 Multi-level encoding methods

A multi-level bitmap encoding employs a hierarchy of bins and encodes each level of the bins separately, as illustrated in Figure 3. Note that the finest level can have each distinct value in its own bin, which produces a precise bitmap index. In our tests, we use the precise indexes for all columns except the one column named KSEQ to be discussed later in this section.

Unlike the multi-component encoding where all components are needed together to answer a query, the multi-level bitmap encoding methods have "levels" that can be used independently. In particular, the finest level can always answer any query the multi-level index could answer. Thus, the main reason for having the coarse levels is to reduce the amount of work needed to answer the query. Let **A** denote the column shown in Figure 3. To resolve the range condition "$\mathbf{A} < 0.5$", we need the first two bitmaps from the fine level or the first bitmap from the coarse level. Clearly, the coarse level is helpful in this case.

Instead of giving a precise definition of a multi-level index, we next give another more realistic example. Instead of **X** taking on values between 0 and 3 as in Figure 2, let's assume the values can go up to 999. We build a three-level index and denote the three levels as level-1, level-10 and level-100. The level-1 is the finest level, where each distinct value has its own bin. This level uses 1000 bins. The level-10 combines every 10 bins from level-1 and produces 100 bins. This corresponds to using 0, 10, 20, ... as bin boundaries. Using this level alone, we can accurately answer queries of the form "$\mathbf{X} \geq 150$" and "$250 \leq \mathbf{X} < 360$." The level-100 again combines every 10 bins from level-10 and produce 10 bins. This corresponds to using 0, 100, 200, ... as bin boundaries, and this level alone can accurately answer queries of the form "$\mathbf{X} \geq 100$" and "$500 \leq \mathbf{X} < 800$." To resolve a query condition such as "$678 \leq \mathbf{X}$" we can use a number of different options. The one that involves all three levels may proceed like this: use level-100 to resolve the condition "$700 \leq \mathbf{X}$," use level-10 to resolve the condition "$680 \leq \mathbf{X} < 700$," and use level-1 to resolve the condition "$678 \leq \mathbf{X} \geq 680$." Regardless what encoding is used for each level, these decomposition of the user-specified condition can be done. The key question is whether using multiple levels can answer the user query faster than using the finest level alone.

In order for the multi-level index to be faster than the finest level alone, the work required of the individual levels must be less than that using the finest level alone. From the previous example, we see that a user-specified range condition is broken into narrower ranges suited for different levels of binning. Typi-

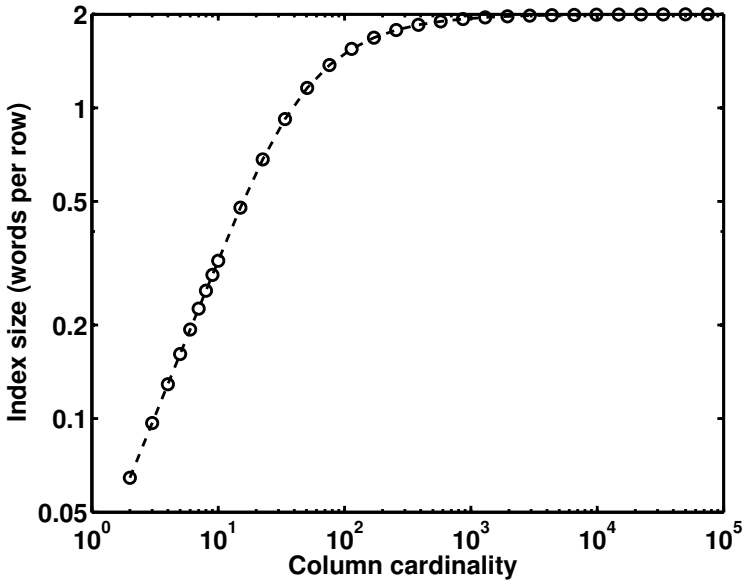

Figure 4: Sizes of equality-encoded indexes on uniform random data, scaled by the number of rows.

cally, the widest range is given to the coarsest bins and the narrowest range is given to the finest bins. Since options of using all the levels or using the finest level alone both require the finest level, the difference is that using all the levels, the finest level is used to resolve a narrower range condition. In order for the multi-level option to be faster, the finest level must be able to resolve a range condition faster. Among the four bitmap encoding methods shown in Figure 2, only the equality encoding has this property. The other three essentially require the same amount of work, no matter what is the query condition [3, 4, 17].

The above argument regarding the finest level can be recursively applied to the higher levels as long as there are more than one level. Using the example mentioned earlier, the above argument essential states that we should use the equality encoding for level-1. We can then consider treating level-10 and level-100 as a two-level index. To resolve the range condition "$680 \leq \mathbf{X}$" faster with both levels than with level-10 alone, we again use the equality encoding for level-10. In general, other than the coarsest level, all finer levels should use the equality encoding. Since the finer levels typically have many bitmaps, and the equality encoding produces bitmaps that are easier to compress than others. This design leads to relatively compact indexes.

For the coarsest level, the range encoding and interval encoding are preferred over the binary encoding and the equality encoding because the former require only one or two bitmaps to resolve any range condition, which is the smallest number of bitmaps possible [4]. In short, the coarsest level of a multi-level encoding should either be the range encoding or the interval encoding, and the remaining levels should use the equality encoding. To determine the number of levels to use, we only need to consider how many levels of equality-encoded indexes to use.

In this paper, we give a qualitative argument for the number of levels to use in a multi-level bitmap encoding. A more quantitative analysis can be found elsewhere [28]. Figure 4 shows the worst-case index size as a function of the number of distinct values in a column (also known as the column cardinality). Note that the Word-Aligned Hybrid (WAH) compression is used [27]. We observe that the index size curve mostly consists of two straight lines connected together with a bend. When the column cardinality is small, say, less than 100, the index size grows linearly with the column cardinality because each bitmap is incompressible

and takes the same amount of space. When the column cardinality is high, say, more than 100, the index size is basically two words per row. From the published analyses on the equality encoding, we know that the time to answer a query is essentially proportional to the compressed size of the bitmaps involved [26, 27]. Figure 4 shows that as long as more than 100 bins are used, the worst-case size of a WAH-compressed index is about two words per row, no matter what the bin count is, which means that the number of bytes needed to answer a query using an equality-encoded index are same regardless of the number of bins. In short, using multiple levels of equality encoded indexes does not reduce the cost of query processing if the coarse levels have more than 100 bins[4].

If the coarse level has less than 100 bins, then the bitmaps are nearly incompressible, and each bitmap has the same size, no matter what encoding is used. In such a case, the range encoding and the interval encoding are better than the equality encoding scheme because they require less bitmaps to answer a query. Since the bitmaps produced by the range encoding and the interval encoding are about the same size in most cases, having multiple levels of the range encoding or the interval encoding does not reduce the query processing cost. Therefore, we shall use only one level of the range encoding or the interval encoding. For the fine level with the equality encoding, we shall always use more than 100 bins; if there is no way to produce more than 100 bins, we shall use one-level with the range encoding or the interval encoding. In any case, we need to use no more than one level of equality encoding as the fine level.

The preceding analysis suggests two multi-level methods, which we denote as the *range-equality encoding* and the *interval-equality encoding*. They use either the range encoding or the interval encoding on the coarse level, and the equality encoding for the fine level. As Figure 3 suggests, we typically produce the fine level first and then decide how to combine the fine level bitmaps to produce the coarse level index. To minimize the worst-case time needed to answer a query, we ensure that the total size of the fine-level bitmaps in each coarse bin is the same, which ensures that the average cost of using the bitmaps in each coarse bin is about the same. The number of coarse bins is chosen such that the average cost of operating on the coarse bitmaps and the average cost of operating on the fine bitmaps are about the same.

Since the fine level of a multi-level index can answer any query the whole index can, the main reason for using the coarse levels is to reduce the query processing time. The multi-level index is designed to reduce the number of bitmaps and the number of bytes needed to answer a query. By reducing the number of bitmaps needed to answer a query, we can also reduce the amount of computation needed. From literature, we know that the computation time could dominate the query response time in some cases [23]. On systems with fast I/O operations, the computation time can potentially become a even larger portion of the total time. Therefore, reducing the computation time can also be an important consideration for the system with flash memory storage devices.

4.3 I/O cost

With WAH compression, the bitwise logical operations usually take lesser time than the I/O operations when answering a query. For this reason, we next briefly examine the I/O cost for the simple cases.

Among the four bitmap indexes, we need all bitmaps in a binary index for answering most queries, and therefore the test software always reads all bitmaps into memory in one read operation. Typically, the binary

[4]Note that using multiple levels can reduce the number of bitmaps needed to answer a query, which is beneficial in some cases [21], for example, when the per-bitmap overhead is high due to the need to open a file for each bitmap, or to allocate a temporary bitmap in memory for each input bitmap. Furthermore, the index sizes plotted in Figure 4 are for the worst-case scenario; in actual applications, the index sizes can be much smaller. For example the values of column KSEQ from the Set Query Benchmark are all distinct and ordered; it produces very compact binned indexes but a very large precise index with the equality encoding. This gives a unusual advantage to the multi-level indexes as illustrated later in this paper.

index of a column requires less storage space than the projection of the column, and therefore, the binary index should require less time to read. However, using the binary index to answer a query may require more CPU time than using the projection.

When using an equality index to resolve a range condition such as "$\mathbf{X} \leq 1$" or "**X** between 2 and 3", a group of bitmaps corresponding to neighboring values are needed. FastBit organizes such bitmaps near each other so that they can be read into memory with one read operation. When bitmaps from both levels of a two-level index are needed, they will be read from different locations in the index file. This increases the number of read operations needed, which may increase the cost of answering a query if the access latency is high. In Figure 1, we show the number of fine bins spanned by each query condition from Q4 of the Set Query Benchmark. When a larger number of fine bins are involved, the coarse level of a two-level index may become useful in reducing the query processing cost. From the table in Figure 1, we see that a majority of the test cases need only one or two fine bins. In such cases, the interval-equality index and the range-equality index behave as the equality indexes. Because the two-level indexes read more metadata than the equality index, they may actually take slightly longer than the equality index to answer the same query.

In later tests, the time measurements include the time to read the metadata; we next count their sizes in FastBit. In general, each index file contains information about the number of rows (N) represented and the number of bitmaps used, along with other information depending the type of index. For a precise index, where each distinct value is indexed, the current FastBit implementation records the C distinct values as C 8-byte floating-point values. For a binned index, FastBit stores the bin boundaries as well as the minimum and maximum values within each bin. Thus, each bin requires three 8-byte floating-point numbers. In addition, FastBit packs all bitmaps of an index tightly in an index file so it can use $B + 1$ integers to mark the starting and ending positions of B bitmaps. For the test data studied in this paper, these positions are recorded as 4-byte integers.

To use a bitmap index, the FastBit software needs to load the metadata into memory. For an equality-encoded precise index, the metadata includes the values of N and C, C floating-point values, and $C + 1$ integers. (Note that the number of bitmaps B equals C in this case.) The total number of bytes is given by the following equation,

$$M_{E1} = 12C + 12. \tag{1}$$

To answer a query, such as an instance of Q1 from the Set Query Benchmark, using an equality-encoded precise index, we need to find the bitmap corresponding to the value 2 and read the bitmap into memory. The Set Query Benchmark uses only uniform random values, and therefore, the number of bytes in a bitmap is

$$m = \frac{4N}{31}\left(1 - (1 - 1/C)^{62} - 1/C^{62}\right), \tag{2}$$

where C is the column cardinality. The total I/O cost captured in our test for an instance of Q1 is

$$s = M_{E1} + m. \tag{3}$$

FastBit constructs a bitmap index for each column separately. To answer a query, we need to identify the column involved, and load the metadata of the corresponding index into memory. The current software implementation may read the metadata in a number of operations, because the size information has to be read before key values and the bitmap positions. Furthermore, a two-level index has two sets of metadata, one for each level. Thus, a two-level index requires about twice as many read operations to prepare the necessary metadata compared to the equality index and the binary index.

name	disk	latency	speed	CPU
data3	SATA	4.2 ms	300 MB/s	2.4 GHz Intel Core 2 Quad
data1	RAID	4.2 ms	300 MB/s	2.4 GHz Intel Core 2 Quad
davinci	GPFS		5 GB/s	1.4 GHz Intel Itanium 2
turing	Fusion-io	26 μs	750 MB/s	3.0 GHz Dual-Core AMD Opteron

Figure 5: Information about the test systems. The GPFS system attached to **davinci** consists of 72 logical units connected with fiber channels.

4.4 Low-precision binning

For columns with extremely high cardinalities, such as column KSEQ of the Set Query Benchmark, where every value is distinct, binning is an effective way to control the index sizes. Next, we describe a new binning technique based on rounding raw data to lower precision numbers, and show that it not only produces compact indexes, but also answers queries efficiently.

The precision of a value refers to the number of significant digits used in the scientific notation. For example, $2 \times 10^4 (= 20,000)$ has one-digit precision, and $4.6 \times 10^3 (= 4,600)$ has two-digit precision. To use this binning technique, the user supplies a number indicating the desired precision. In the case of KSEQ, a two-digit precision bin is sufficient to answer all queries involving KSEQ in the Set Query Benchmark. This is common in many applications, because the query conditions specified by users typically include relatively low precision constants. For example, Q3 includes conditions of the form "KSEQ between 400000 and 500000" and "KSEQ between 420000 and 430000." The constants in the first expression have one-digit precision and those in the second have two-digit precision.

To determine which bin a value belongs to, we reduce the precision of the incoming value to the specified number of digits. For example, the value 101 expressed in two-digit precision is $1.0 \times 10^2 = 100$. We call the value 100 the representative of the bin. Based on whether the actual value is less than, equal to, or greater than the representative, we record the row in three separate compressed bitmaps. For each representative value, we effectively build three bins. For the value 101, it is associated with the representative 100, and falls in the bin for values greater than 100.

Among three bins for each representative, the middle one is reserved for rows with values equal to the representative, while the two bins on the left and right represent ranges of possible values. The choice to leave the representative value in its own bin allows the index to answer arbitrary conditions involve the representative values, e.g., "KSEQ $= 2$" and "KSEQ between 420000 and 430000." This ensures all query conditions involving low precision constants can be answered with this index only.

5 Test systems

Our tests are performed on four different systems listed in the table in Figure 5. These systems are chosen because they have different I/O devices. The system named **data3** holds the test data in a simple SATA drive, **data1** holds the data in a software RAID consisting of 4 SATA disks, **davinci** is attached to a large GPFS system with its own storage area network, and **turing** contains a flash memory drive from Fusion-io [9].

GPFS is a shared disk POSIX based parallel file system from IBM that allows hundreds or thousands of clients to concurrently access a single file system [19]. The file system can be accessed through a storage area network, or via a network to a network storage device. GPFS ensures consistency by maintaining global locks on file regions and distributes blocks of data from a file across multiple storage units. The file system

used for this test has been demonstrated to provide over 5 GB/s of aggregate bandwidth using many clients and over 600 MB/s using a single client.

Our flash device tests were performed on a 160 GB single-level cell (SLC) ioDrive from Fusion-io. It is a PCI-Express (PCIe) card that directly connects to the bus of a computer, which allows it to have much lower access latency and higher speed (see Figure 5). Flash memory storage typically comes in SLC and multi-level cell (MLC) forms. SLC devices provide better write performance and endurance compared to MLC devices (used in most commodity storage devices such as USB keys and digital camera storage). The Fusion-io cards have excess storage cells and have specially designed wear-level algorithms to extend the life of the card and work around worn cells. The cards utilize a RAID-like algorithm and implement checksums to improve data integrity and protect against failures. They also have special logic that attempts to hide the erase cycle from applications. Furthermore, a grooming processor runs periodically to erase unused cells. However, these methods have their limits. For example, sustained random write workloads will eventually exhaust the pool of pre-erased cells. Once this happens, the additional delay to erase cells for updated data significantly impacts performance. Specialized benchmarks such as uFLIP [1] can be used to evaluate these performance characteristics.

6 Timing measurements

In this section, we report the key results from the timing measurements. The measurements are primarily collected through two commands, `gettimeofday` to measure the elapsed time of answering a query, and `vmstat` to collect the performance statistics of the I/O systems.

To ensure all I/O costs are captured in our measurements, before answering each query, we clear all file caches by unmounting the file systems on **data1**, **data3**, and **turing**. On **turing**, we also remove the Fusion-io device driver to clear its cache. The system **davinci** is a large shared facility and hence we were unable to clear the GPFS caches. The I/O traffic reported by `vmstat` on **davinci** is not necessarily all due to our jobs. Fortunately, the command `vmstat` also reports the amount of time used by each I/O device. This allows us to evaluate the performance of I/O devices without distinguishing who initiated the I/O traffic. We start our discussion on performance measurements with the performance of I/O devices, then present the index sizes, and finally discuss the overall query response time.

6.1 Read speed

Figure 6 shows a sample of reading time against the number of bytes read. These values are reported by `vmstat`, which gives the time as an integer measured in milliseconds. Therefore, the minimum non-zero value reported in Figure 6 is 10^{-3} seconds. The reported values are the medians of actual measurements.

The case involving the least number of bytes has the query condition "KSEQ = 2" because the index involved is much smaller than others as shown in Figure 8. On all four systems, the number of disk sectors read is 176, which amounts to 90 KB. Tracing the execution of FastBit showed that it opened two files[5]: the metadata file about the data set and the index file for KSEQ. The metadata file is 1 KB and the index file for KSEQ is 89 KB. The total number of bytes is exactly as reported by `vmstat`.

Since both files are smaller than the read-ahead buffer size, they are read into memory in two read operations. The total time to complete these operations is dominated by the access latency on systems with spinning disks. Our measurements show that the SATA disk took 10 ms, the RAID of SATA disks took 20

[5]It also opened about 50 system files and run-time libraries. Presumably, these files are frequently used by many programs and are cached by the OS.

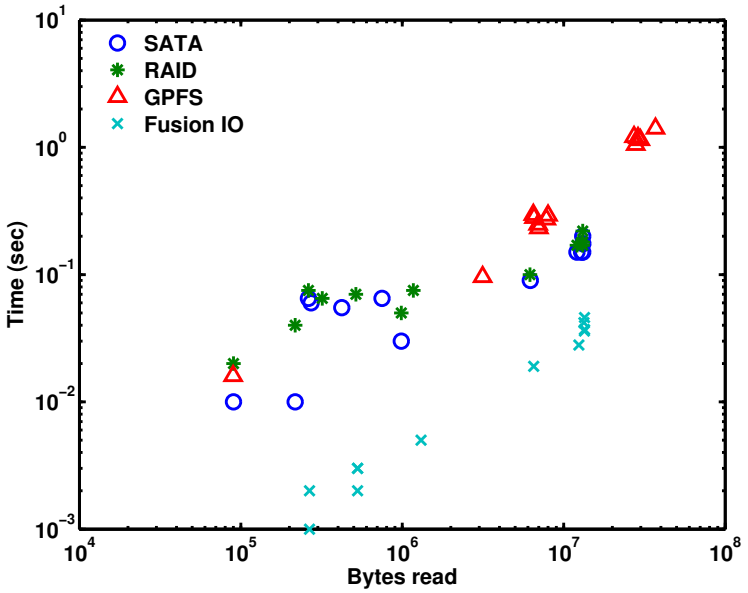

Figure 6: The read time (seconds) reported by `vmstat` plotted against the number of bytes read to answer 13 instances of Q1 using index E1.

	β	γ	Speed (MB/s)
SATA	0	1.79×10^{-8}	55.9
RAID	0	1.47×10^{-8}	68.0
GPFS	7.45×10^{-4}	3.30×10^{-8}	30.3
Fusion-io	0	2.84×10^{-9}	352.4

Figure 7: Performance of test systems computed with a non-negative least-squares fitting. The reported read speed is simply the inverse of γ and is the observed speed per I/O device. There are 72 IO devices for the GPFS on **davinci**.

ms, and the GPFS system took 16 ms. On the system with a single SATA disk, the two read operations require a minimum of 8.4 ms due to the access latency. The observed time is close to this minimum. On systems involving more disks, such as a RAID or a parallel file system, the access latencies are longer because of the need to coordinate multiple disk accesses. On the flash memory storage, `vmstat` reported 0 ms because the actual time is less than 1 ms.

Assume that the observed time is given by the following linear equation

$$t = \beta r + \gamma s,$$

where r is the number of read operations and s is the number of bytes read. From the r, s, and t values reported by `vmstat`, we can perform a nonnegative least-squares fitting to determine the parameters β and γ. The table in Figure 7 shows the results of this fitting. In most cases, the number of read operations requested by OS (and reported by `vmstat`) has no direct relationship with the actual number of disk reads due to various caching mechanisms such as read-ahead, therefore, the coefficient β is assigned 0 by the non-negative fitting algorithm.

column	cardinality	# bitmaps	**Proj**	**BN**	**E1**	**IE**	**RE**
K2	2	2	400	12.9	25.8	25.8	25.8
K4	4	4	400	25.8	51.6	51.6	51.6
K5	5	5	400	38.7	64.5	64.5	64.5
K10	10	10	400	51.6	128.8	129.0	129.0
K25	25	25	400	64.5	296.9	322.6	322.6
K100	100	100	400	90.3	598.3	701.6	765.8
K1K	1000	1000	400	129.0	776.1	1060.0	1146.7
K10K	10000	10000	400	180.8	797.7	1081.7	1181.1
K40K	40000	40000	400	206.8	800.0	1084.2	1183.6
K100K	100000	100000	400	220.6	801.4	1098.9	1185.4
K250K	250000	250000	400	235.3	803.9	1102.7	1189.2
K500K	500000	500000	400	251.1	808.0	1108.7	1207.8
KSEQ	100000000	1720	400	0.066	0.089	0.089	0.089

Figure 8: Sizes information about the bitmap indexes tested. The index sizes are in millions of bytes (MB). Note that all columns except KSEQ are index precisely and KSEQ is indexed to two-digit precision. The total sizes are: Proj 5.2 GB, BN 1.5 GB, E1 6.0 GB, IE 7.8 GB, and RE 8.5 GB.

The coefficient γ is the average amount of time (in seconds) needed to read a byte, the inverse of which is the read speed. Since the command `vmstat` reports performance per I/O device, the read speeds given in Figure 7 are the average values per device. There are 72 devices for the GPFS on **davinci**, and hence the aggregated read speed is 2,182 MB/s.

6.2 Index sizes

The table in Figure 8 shows the index sizes. Overall, the binary index (BN) is the most compact, which is less than 30% of the raw data size. The index E1 takes about 15% more space than the raw data. The indexes IE and RE are about 30% and 40% larger than E1 respectively. All of these indexes are more compact than the commonly used B-Tree implementations, which may take 20 GB to index all columns.

By indexing KSEQ to 2-digit precision, all four versions of bitmap indexes are less than 1 MB, much more compact than any other bitmap indexes used. The binned index is very compact because each bin contains only a consecutive set of values in adjacent rows.

6.3 I/O cost predictions

In Section 4, we give a formula for the number of bytes needed to answer Q1 with index E1. We plot the predictions along with the observations in Figure 9. The horizontal axis in this figure is the number of bitmaps used in an index. The exact number of bitmaps used for each column is given in Figure 8.

On the three machines where we cleared the file system caches before each query, the observed read sizes fall right on top of the plus symbols representing the predictions when there are less than 1000 bitmaps. In such cases, the size of data accessed is dominated by the bitmap sizes. Because Equation (2) accurately predict the bitmap sizes, the predictions and observations agree.

On the right side of the graph, the predicted sizes grow linearly with the numbers of bitmaps because the metadata sizes shown in Equation (1) dominate the predictions. The metadata consists of a number of

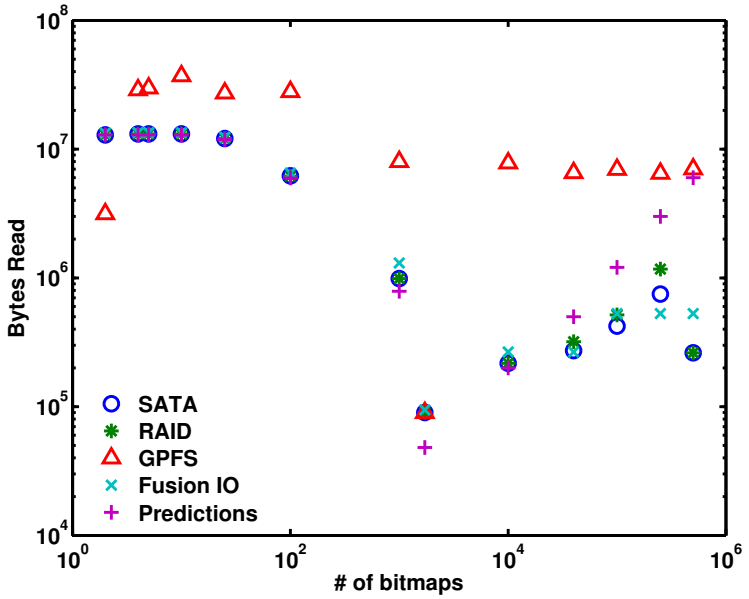

Figure 9: The observed I/O volumes plotted along with the predictions for instances of Q1.

	Elapsed time (sec)					Read time (sec)					Read size (MB)				
	Proj	**BN**	**E1**	**IE**	**RE**	**Proj**	**BN**	**E1**	**IE**	**RE**	**Proj**	**BN**	**E1**	**IE**	**RE**
data3	581.4	109.9	24.4	33.4	34.3	581.9	65.5	12.9	26.4	24.2	37103	6226	947	1939	1651
data1	481.8	92.0	23.7	29.3	29.6	487.4	97.9	16.0	30.9	26.7	38859	8953	953	2037	1706
davinci	454.4	118.8	42.9	52.6	59.8	1588.5	130.9	85.9	129.7	120.6	44482	4351	2237	3485	3331
turing	259.0	44.3	17.1	16.1	19.0	126.2	3.8	2.6	4.8	4.2	61718	1819	1255	2335	2006

Figure 10: The overall time and I/O costs to complete the Set Query Benchmark.

arrays, but only a small number of elements of these arrays are used while answering query Q1. Since we access the data through memory maps instead of actually reading all the bytes into memory, a portion of the data pages containing the accessed elements are read into memory by the run-time system.

On **davinci**, where GPFS is used, because the system is busy, the observed read sizes do not agree with the predictions. In most cases, we see that the observed read sizes are larger than the predictions because of other users also performing read operations. In one case, the observed size is less due to caching.

6.4 Overall performance

The table in Figure 10 shows a summary of all measurements. The values are divided into three groups: elapsed time, read time, and read size. Each number in the table is the sum of all 75 queries. Overall, the test took the the least amount of time on **turing** because it has the fastest I/O system and the fastest CPU.

Among the five indexing methods, index E1 required the least amount of time in three out of four cases. The minimum time was achieved with index IE on **turing**. Index IE requires no more bitmaps to answer a query than E1. In general, it takes less time to combine these bitmaps than IE as well.

The projection index needed the largest number of bytes for reading 169 projections totaling 67.6 GB[6].

[6]Each of the 13 instances of Q1 needs 1 projection index, each of the 24 instances of Q2 needs 2 projection indexes, each of the 22 instances of Q3 needs 2 projection indexes, 8 instances of Q4 need 3 projection indexes each, and 8 other instances of Q4 need

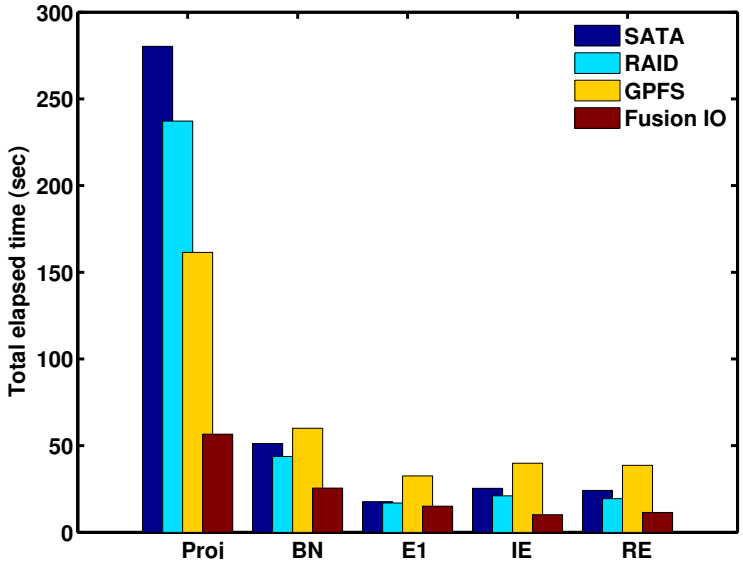

Figure 11: Total time needed to answer different instances of Q4.

By using memory maps, we are able to skip some data pages and the actual number of bytes reported by `vmstat` is slightly less.

The index that required the least amount of data was E1. The theoretical analysis predicted IE and RE to use even less bytes. However, this analysis neglected the overhead of loading metadata to initialize the indexes; our measurements included these initialization costs. Additionally, most queries need to access only 1 bitmap, in which cases indexes IE and RE have no advantage over E1. Since E1 read the least number of bytes, it also takes the least amount of time. On **turing**, the I/O time reported by `vmstat` was 2.6 seconds. During which time, 1255 MB was read at a speed of 483 MB/s.

When the index E1 requires a larger number of bitmaps, such as to answer the first instance of Q4, using the two-level indexes has a significant performance benefit. To answer the first instance of Q4, using E1 took 1.8 sec on **data3** and 1.7 sec on **turing**, using IE took 1.5 sec on **data3** and 0.6 sec on **turing**, and using RE took 1.1 sec on **data3** and 0.5 sec on **turing**. With E1, the CPU time dominates the query response time, and therefore using a faster I/O system did not significantly decrease the query response time. Switching to the two-level indexes on **data3** reduced the total query response time somewhat, while further switching to a faster I/O system reduced the query response time by a factor of 3 compared with E1 on **data3**.

Figure 11 shows the total time used to answer all instances of Q4. Since there are more range conditions involved in this set of queries, we see a much larger advantage of using the two-level indexes. In this case, the average speedup of IE and RE over E1 is about 2. The speedups over BN and Proj are much larger.

From Figure 11, we see the gain from using the flash memory storage devices decreases as the indexing methods become more efficient. For example, the projection index is the least efficient among the five shown in Figure 11, it has the best gain going from SATA to Fusion IO. Using the more efficient binary index (BN) and the basic bitmap index (E1), the gain of using flash memory decreases. In fact, using E1 on different I/O systems require nearly the same amount of time because a significant of amount of CPU time is needed to answer a query with this index. In contrast, using one of the two-level indexes, the time is quite different on

5 projection indexes each. Altogether, the 75 queries needs 169 projection indexes. Since we cleared the file caches, we need to perform 169 reads of 400 MB each.

different I/O systems. In particular, the time needed on the SATA disk system using IE or RE is more than using E1, however, the time needed on Fusion IO system using IE or RE is less than using E1. This clearly indicates that the two-level index is able to take advantage of the flash memory system much better than the one-level index.

7 Summary and Future Work

In this paper, we developed a set of multi-level bitmap indexing method and examined their performance on different I/O systems including a new flash memory storage system from Fusion-io. This storage device is connected directly to the PCIe bus, which results in an extremely fast read speed and a very low access latency. The new indexes are designed to reduce the volume of data read into memory in order to answer a query, by breaking the read operations into smaller chunks. This design choice takes full advantage of the reduced access latency of the flash memory devices. However, since the new indexes reduce the volume of data as well as the amount of computation needed to answer a query, they are potentially useful on any storage system.

We presented the key design considerations for the multi-level bitmap indexes and concluded that the two-level interval-equality encoding and the range-equality encoding are likely to perform better than other combinations. We compared the performance of these two-level bitmap indexes against two well-known bitmap indexes and the projection index because they have been shown to perform well in the literature. To answer a query, the binary index and the projection index can utilize sequential read operations, which reduce the number disk accesses and minimizes the I/O time. The other indexing methods require a varying number of read operations in order to answer a query, and therefore have different performance characteristics.

The Fusion-io storage system significantly reduces the time needed for reading the necessary data, and makes the compute time a much larger portion of the overall query response time. On such storage systems, reducing the computation can more directly reduce the overall time. From our tests, the two-level interval-equality index used the least amount of time on the flash memory storage system. On other storage systems, the basic bitmap index (labeled E1) required the least amount of time overall because it read the least amount of data to answer the test queries.

In terms of the total query response time, the interval-equality index was only 6% faster than the basic bitmap index with the flash memory storage device. This is largely because most of the queries in the Set Query Benchmark involves only equality conditions that are particularly well suited for the basic bitmap index. However, in cases involving range conditions, the two-level indexes were observed to be 3 times faster than the basic bitmap index.

We also introduced a new binning strategy for bitmap indexes based on reducing the precision of the input values. This binning strategy takes advantage of the fact that most user queries involve constants with a relatively low precision. In our tests, the binned index was very compact and could answer the suite of queries very efficiently.

Furthermore, we used the memory mapping mechanism for reading the index files in our experiments. The tests showed that we could avoid reading some of the data pages, which reduced the overall query response time. However, it also caused varying data sizes to be read on different systems. For the purpose of evaluating I/O systems, a better control on how many bytes to read may be useful. Our test programs were run with a single thread of execution. For a more extensive exercise of the I/O systems, it might be interesting to answer queries with multiple threads as well. We plan to investigate these options in the future.

References

[1] L. Bouganim, B. Jónsson, and P. Bonnet. uFLIP: Understanding Flash IO patterns. In *Proc. CIDR*, 2009.

[2] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: a parallel file system for Linux clusters. In *Proc. ALS*, pages 317–327, 2000.

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

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

[5] T. Clark. *Designing storage area networks: a practical reference for implementing Fibre Channel SANs*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[6] D. Comer. The ubiquitous B-tree. *Computing Surveys*, 11(2):121–137, 1979.

[7] Y. Deng. Deconstructing network attached storage systems. *J. Netw. Comput. Appl.*, 32(5):1064–1072, 2009.

[8] J. Do and J. M. Patel. Join processing for flash SSDs: remembering past lessons. In *Proc. DaMoN*, 2009.

[9] Fusion-io. `http://www.fusionio.com/`.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. SOSP*, pages 29–43, 2003.

[11] G. Graefe. The five-minute rule twenty years later, and how flash memory changes the rules. In *Proc. DaMoN*, pages 1–9, 2007.

[12] J. Gray and B. Fizgerald. Flash disk opportunity for server-applications. *ACM Queue*, 6(4):18–23, 2008.

[13] D. Myers. High-performance relational databases. Master's thesis, MIT, 2008.

[14] E. O'Neil, P. O'Neil, and K. Wu. Bitmap index design choices and their performance implications. In *Proc. IDEAS*, pages 72–84, 2007.

[15] P. O'Neil. Model 204 architecture and performance. In *Proc. HPTS*, volume 359 of *Lecture Notes in Computer Science*, pages 40–59, September 1987.

[16] P. O'Neil and E. O'Neil. *Database: principles, programming, and performance*. Morgan Kaufmann, second edition, 2000.

[17] P. O'Neil and D. Quass. Improved query performance with variant indices. In *Proc. SIGMOD*, pages 38–49, 1997.

[18] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. SIGMOD*, pages 109–116, 1988.

[19] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proc. FAST*, page 19, 2002.

[20] M. A. Shah, S. Harizopoulos, J. L. Wiener, and G. Graefe. Fast scans and joins using flash drives. In *Proc. DaMoN*, pages 17–24, 2008.

[21] R. R. Sinha and M. Winslett. Multi-resolution bitmap indexes for scientific data. *ACM TODS*, 32(3):16, 2007.

[22] K. Stockinger and K. Wu. Bitmap indices for data warehouses. In *Data Warehouses and OLAP: Concepts, Architectures and Solutions*, pages 179–202. Idea Group, Inc., 2006.

[23] K. Stockinger, K. Wu, and A. Shoshani. Strategies for processing ad hoc queries on large data warehouses. In *Proc. DOLAP*, pages 72–79, 2002.

[24] 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. In *Proc. VLDB*, pages 553–564, 2005.

[25] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *Proc. SIGMOD*, pages 59–72, 2009.

[26] K. Wu, E. Otoo, and A. Shoshani. On the performance of bitmap indices for high cardinality attributes. In *Proc. VLDB*, pages 24–35, 2004.

[27] K. Wu, E. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM TODS*, 31(1):1–38, 2006.

[28] K. Wu, A. Shoshani, and K. Stockinger. Analyses of multi-level and multi-component compressed bitmap indexes. *ACM TODS*, 35(1):1–52, 2010.