AN ABSTRACT OF THE THESIS OF

Benjamin McCamish for the degree of Master of Science in Computer Science presented on June 11, 2015.

Title: Bitmap Indexing: Energy Applications and Improvements

Abstract approved: _____

Eduardo J. Cotilla-Sanchez

Large databases and data warehouses are becoming prevalent for the storage and management of energy data. Accelerating the rates at which data can be retrieved is beneficial not only to allow for more efficient search of the data, but also to be integrated with other energy system tools. In this work, a fast indexing and data retrieval method, known commonly as a *bitmap index*, is created to facilitate intelligent querying and indexing of data generated by phasor measurement units (PMU) at a rate of 60 Hz. In addition, compression of these bitmap indexes is parallelized, and the performance quality of bitmaps created over this dynamic database are tested. We find that bitmaps are amenable to managing efficient access to large amounts of PMU data. Furthermore, the bitmap-management process will provide decreased access time for data retrieval as well as decreased memory usage. From these experiments, we are able to improve compression times while discovering new research opportunities, decrease access times to the PMU database by 30 times the conventional method currently utilizes, and introduce a feasible method for bitmap operations over a dynamic database.

Bitmap Indexing: Energy Applications and Improvements

by

Benjamin McCamish

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 11, 2015
Commencement June 2015

Master of Science thesis of Benjamin McCamish presented on June 11, 2015.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Benjamin McCamish, Author

ACKNOWLEDGEMENTS

First I would like to acknowledge my wonderful wife. Understanding the amount of time I have to spend working late and away from home, keeping me fed and reminding me to get sleep on occasion, and encouraging me to continue my physical activities to stay healthy. Her support has been the keystone to the work that has been done.

I would also like to thank Dr. David Chiu for the crucial advice that he has provided for both academia and life. It was him who helped guide me through the initial phase of graduate school and set me on the path I am currently on. Without Dr. Chiu, I wouldn't be here today.

Dr. Eduardo Cotilla-Sanchez has helped me with my transition to Oregon State University. Through his leadership we have found new research opportunities. Everything I know to date about power systems is through him.

I would also like to acknowledge my peers and professors I have collaborated with over the years. They have presented with me with many opportunities to learn and humble myself. Working with them has been some of the best time I have spent.

This research was funded in part through grants provided by Oregon BEST Center Bonneville Power Administration. We thank the Bonneville Power Administration for providing the PMU data and support.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

Chapter 1: Introduction

With increasing need for real-time awareness in the grid, power supply companies are deploying what is known as Phasor Measurement Units (PMUs)[1]. These devices monitor the status of the grid by measuring electrical waveforms at their location at a given rate, typically very fast. Capable of being synchronized across multiple PMUs, they provide operators with a wide-area view of the grid's status [1–3].

PMUs today have become extremely sophisticated allowing up to 4000 measurements per second in recent commercial devices. Once the data has been collected by the PMU it is sent to what is known as a Phasor Data Concentrator (PDC). Once collected the PDC synchronizes the data points into a single tuple by joining the PMU measurements on their timestamps. This compensates for the latency in the network and the variable arrival time of packets. The tuples are then written to large, non-volatile storage, *e.g.* hard disks. With a fast rate of measurement coupled with a high volume of PMUs placed at various locations the database grows to enormous sizes. The size of the database and location it is stored prohibits certain techniques of data querying. To accommodate this issue a bitmap index has been built over the data.

Due to the variety of locations that PMUs can be located, a distinct feature of PMUs is that they make use of global positioning systems (GPS) time signal in order to gain extremely accurate time-stamping of the power system information. PMUs have several advantages, one of which is being able to monitor data at a high rate of speed. This allows for dynamic events to be detected and the operator to respond appropriately [4].

The amount of data produced by PMUs continually increases, presenting a problem from a database management viewpoint. For this data to be effective and notifying operators of the grids health it often needs to be queried in an efficient manner. A portion of this work will provide an efficient and scalable query system based upon bitmap indexing.

One data structure, known as a *bitmap index* [5], has become popular for managing large amounts of data in the context of scientific applications [6–8], network traffic monitoring [9], and data storage [10, 11]. A bitmap B is an $m \times n$ matrix where the n columns represent range-bins, and the rows correspond to the m records (*e.g.*, PMU measurements). To represent data as a bitmap, each attribute is first partitioned into bins that denote either a value or a range of values. A bit $b_{i,j} = 1$, if the ith record falls into the specified value/range of the jth bin, and $b_{i,j} = 0$, otherwise.

[1]Also known as *synchrophasors*, we refer to them as PMUs throughout this work.

Records	**Bins**						
	X				Y		
	x_1	x_2	...	x_{50}	y_1	y_2	y_3
t_1	0	1	...	0	0	0	1
t_2	0	0	...	0	0	1	0
t_3	0	0	...	1	0	0	1
...	...	...	...	...	...	...	...

Table 1.1: An Example Bitmap Index

Consider the bitmap in the Table 1.1. Suppose this example dataset has two attributes, X and Y, and the values of X are known to be integers in the range $(0, 50]$ and that the values of Y can be any real number. Due to its small cardinality[2], we can generate a bin x_j for each possible value of X. Because the values of Y are continuous and unbounded, we must discretize its values, *i.e.*, decide on an appropriate cardinality of bins to represent Y, and select the range of values associated with each bin. In our example, we chose to use only three bins, $y_1 = (-\infty, -5]$, $y_2 = (-5, 5)$, and $y_3 = [5, \infty)$.

Suppose we want to retrieve all records from disk where $X < 25$ and $Y = 0$. We can identify the candidate records by computing the following boolean expression,

$$v_R = (x_1 \vee ... \vee x_{24}) \wedge y_2 \tag{1.1}$$

v_R is a bit-vector, and the value of the ith bit denotes whether the ith record in the file is promising. Therefore, the bits with a value of 1 in v_R corresponds the set of candidate records on disk,

$$R = \{t \mid (t[X] < 25) \wedge (-5 < t[Y] < 5)\} \tag{1.2}$$

Intuitively, there could be false positives in R, which requires checking, but only the records $\{r_i\} \in R$ with a corresponding bit $v_R[i] = 1$ must be retrieved from disk and examined to ensure they meet the selection criteria. All records r_i with a corresponding bit $v_R[i] = 0$ are *pruned* immediately and do not require retrieval from disk. Because a well-designed bitmap is sparse and compressible, it can be stored in core-memory, which is orders of magnitude faster than disk.

As we can see, bitmaps can help reduce disk accesses when properly discretized, re-

[2]Domain of possible values within attribute.

sulting in a space/accuracy tradeoff. In this example, more precise pruning may have been possible had we split the attribute Y into even more, finer-grained, bins. However, each additional bin effectively adds an entire dimension, increasing the bitmap index's size, challenging its ability to fit in core-memory. Furthermore, appropriate range assignment of the bins will also affect accuracy.

The following experiments are the first time that bitmap indexes have been implemented and tested for the PMU dataset. All improvements to the bitmap index are made specifically for PMU datasets, but can be extended to other similar datasets containing frequently updated tuples. Parallel compression can be applied to any bitmap index, regardless of the data type it is built over.

1.1 Background

This work uses various methods and technologies to achieve its goals. This section introduces the compression methods commonly used, parallel computing algorithms, and hardware utilized. Word Aligned Hybrid and Variable Aligned Length are the two compression techniques that are used most often in this work. Work Sharing and Work Stealing are two parallel programming techniques commonly used to distribute work among simultaneous threads of execution. Phasor Measurement Units and Phasor Data Concentrators are receiving increased use throughout the world to monitor the status of the electrical grid by collecting information both at transmission and distribution levels. These are the technologies in the energy systems domain where we develop, analyze, and improve bitmap indexing methodologies.

1.1.1 Word Aligned Hybrid

Several specialized bitmap compression schemes have been introduced that allow for fast query execution directly over the compressed bitmap codes. The Word Aligned Hybrid code (WAH) [12] is one such efficient bitmap compression and querying scheme. In WAH sequences of bits in the bitmap are grouped into *words*, which are defined by the hardware (*e.g.*, $w = 32$ bits). Each size w word contains either a *fill* or *literal* encoding of $w-1$ bits. A fill-word is used to represent a run of either 1's or 0's. The structure of a fill-word is (*flag-bit*, *fill-bit*, *length*). The flag-bit indicates whether the word is a fill or a literal (1 or 0, respectively). The fill-bit represents the value of the run. The remaining bits represent the number of size $w-1$ runs of the fill-bit there are. In contrast, a literal-word is encoded with (*flag*, *exact encoding*). The flag-bit 0, signifies that the word is a literal. This is then followed by the exact encoding, which is the literal translation of the $(w-1)$ bits. In this work, we assume that word size $w = 32$.

Fig. 1.1 illustrates an example of compressing a bit vector using a 32-bit word. The example in Fig. 1.1 shows the compression of a bit vector that is 1155 bits long. Compressing with WAH using 32-bit words gives a compression ratio of approximately 18 times, resulting in a bit vector containing only 64 bits.

Logical operations between bit vectors occur by performing a bitwise operator between them. Performing these operations on compressed bit vectors happens at the level

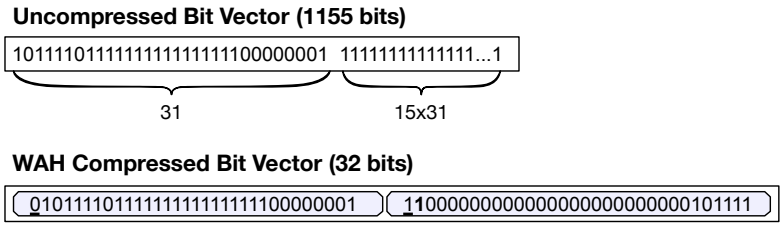

Figure 1.1: WAH compression example

of a single word. One important aspect during bitwise operations is the *alignment* of the words, *i.e.*, that each bit is operated on against its respective bit in another bit vector. This is already done through the design of WAH, since each word represents an equal sets of bits, $w - 1 = 31$ in our case, for words of size 32 bits. These words can either contain a single set of 31 bits or many, depending on whether they are literals or fills respectively.

When performing operations between each bit vector, a variable *active word* is created for each bit vector. The active word keeps track of the word that is being operated on in that bit vector, starting with the first. When the bitwise operation is applied to all the encoded bits in the active word, it then moves onto the next word in that bit vector. Three scenarios can occur when performing bitwise operations between two active words:

Fill Word vs. Fill Word: In this scenario, if both active word variables are encoding the same number bits, then the bitwise operation can be performed without extracting any additional sets of bits. That is, for a word of size 32, only that many bits need to be operated on, allowing for queries to take place without decompression. In the case where the active words do not encode the same number of bits, consider the following. Given two words, X and Y where X encodes more bits than Y, the equal number of bits are operated on, reducing the *length* of X to the difference and exhausting the bits encoded in Y. The active word that represented Y reads the next word while X remains with a reduced run length.

Literal Word vs. Fill Word: With a literal and a fill, a single set is again subtracted from the fill and compared with the literal set of bits encoded in the literal word. Then the active word moves to the next word in the respective bit vector. If there are no more bits encoded in the fill, then the active word moves to

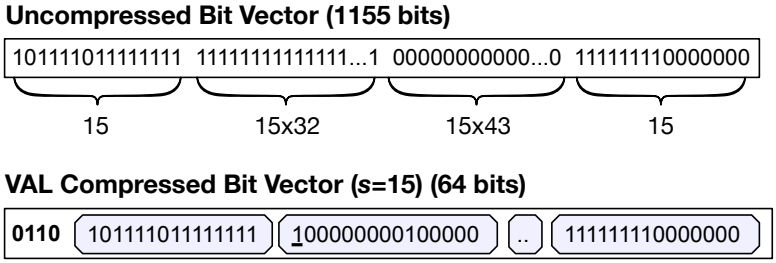

Figure 1.2: VAL Encoding Example

the next word in its respective bit vector.

Literal Word vs. Literal Word: The bitwise operations between two literals is performed normally on the encoded bits.

There is a direct correlation between how compressed a bit vector is with how fast queries can be performed. The more compressed a bit vector is, the faster the query will occur. This is because of how operations occur between fill words amortizes the cost of applying the logical operation between two bit vectors. This is rarely the case however as often *dirty bits* interrupt what would otherwise be a long run of bits. A dirty bit is one that is different from the surrounding bits and is in a place that prevents a longer run from occurring. Often the frequency of dirty bits depends on the domain the bitmap index is applied to.

1.1.2 Variable Aligned Length

In general, word-aligned bitmap compression schemes have a trade off between higher compression ratio and short query times. VAL minimizes this trade off by analyzing the bit vector to be compressed, looking at the distribution of bits and run lengths, then chooses an appropriate segment length and encoding algorithm. Like WAH, the bit vector is compressed by VAL into *words*, which are limited by hardware constraints. The significant difference is that VAL can specify a segment length, or how many segments the word is broken up into. These words are then compressed using the specified compression method, determined by VAL during the analysis of the bit vector [13].

Consider the following example where WAH will be used as the compression method

and the segment length will be set to 157 with a word size of 6432, represented in Figure 1.2. The uncompressed bit vector consists of a literal, a run of ones, a run of zeros, and another literal. Each word in the VAL compressed bit vector will have a header of 4 bits representing whether the respective segment is a fill or a literal, reducing some of the overhead from decompression during queries. The first and last segments are literals and are encoded as such. The next two are fills and are encoded such that the first bit is their fill bit and the remaining bitsnext six are the number of segments contained in that fill. When VAL has the same segment and word length as WAH, then the compression ratio is equivalent. VAL benefits from being able to change the segment length for each bit vector if necessary.

Since the bit vectors are compressed into words, the more words that are compressed the longer it takes to perform a query due to the decoding that takes place. If bit vectors are compressed efficiently, then it is possible to reduce these times.

Due to the variable segment length that can occur across all bit vectors, it is necessary to align the bits appropriately before performing logical operations on them. Given $V_{1.s}$ and $V_{1.s}$ where V is a compressed bit vector and s is the segment length used on that vector, there are two scenarios: the segment length of each word is the same or they are different. If the segment length is the same, $V_{1.15}$ and $V_{2.15}$ for example, then the bit vectors are already aligned and the logical operation can proceed normally. When the segment length differs, $V_{1.15}$ and $V_{2.30}$ for example, then the segment lengths need to be changed so that the bits are aligned. This happens on the fly during query execution time and these changes to segment length are not reflected in the bitmap.

There are two methods that VAL uses to adjust the segment length of a bit vector, *decode down* and *decode up* this consists in, increasing or decreasing the segment length of one vector to match that of the other. When decoding down there are two cases that can happen, either the block that is being decoded is a fill or a literal. If the block is a fill, then a segment with the same fill value is added to the word with the same value representing the number of segments in that fill. Figure 1.3 shows an example where the bits representing the number of segments represented can not translate into a single segment when decoding down from 60 to 15. Thus two segments are created such that when summed together they represent the same number of zeros in the bit vector. A literal segment is decoded down by dividing it into the smaller segment length and appending it onto the word. In this case a literal with segment 60 would be broken down

with segment 15 into four literals.

Decode up consists of more cases than decode down. One needs an additional data structure called the alignedBlock which stores intermediate data such as remainder bits or partial words. There are four scenarios when decoding down:

The activeBlock is a fill and the alignedBlock is empty The activeBlock is added to the currentWord with the number of segments being a fraction of the new segment length with any remaining bits being appended to the aligned block as a literal.

The activeBlock is a literal and the alignedBlock is empty The literal is appended to the alighedBlock.

The alignedBlock is not empty and activeBlock is a literal. The aligned-Block is then filled with bits from the literal in activeBlock until it is full, that is the size of the new segment length.

The alignedBlock is not empty and activeBlock is a fill This is the example that is presented in Figure 1.4. The fill is expanded to fill the alignedBlock, in this case it happens to be the same size as the word, and then adds the alignedBlock to the word. If there are any remaining bits in the activeBlock, they are added to activeWord as a fill.

1.1.3 Caching

Caching is a process that is used to keep data that will be used frequently closer to the CPU. All of the data that gets pulled from memory is first read into different levels of the cache, as to increase access times. As storage gets closer to the CPU, there is less of it at the given level and access times are much faster. Typically the CPU checks each level in cache, memory, and non-volatile storage, in that order, with the last having the highest capacity and slowest access times. In most CPU architectures there are three levels of cache. Each core on the CPU has 2 independent levels of cache, with the second level being larger and slower than the first. These are only available to that specific core. The last level of cache or the lower level cache is shared by all of the cores, ergo

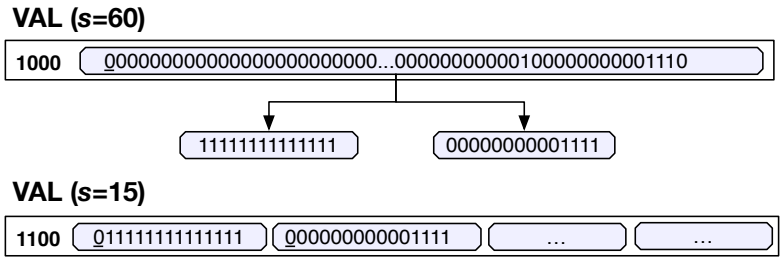

Figure 1.3: Decode Down

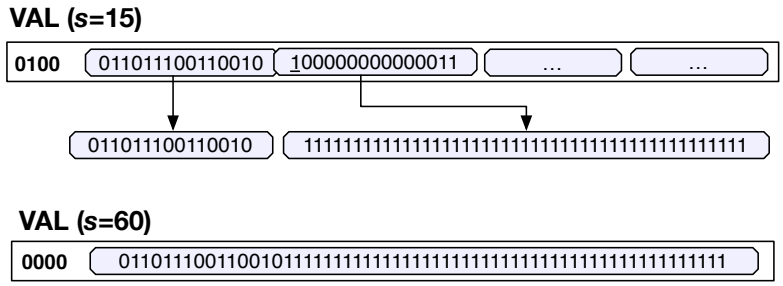

Figure 1.4: Decode Up

all of the threads. Since all cores have access to this level they are allowed to access and manipulate data that another core has read in.

1.1.4 Work Sharing

Work sharing is a commonly used method for distributing work. Work sharing stems from the concept that there are tasks that need to be done and workers to do them. The workers are either multiple threads on the same machine, nodes in a distributed cloud, or both. To organize this work sharing introduces the concept of a queue of tasks, also commonly known as a bag of tasks. When a task is ready to compute, it will be inserted into the queue. Workers then grab jobs from the queue when they become available. Since all the workers share the same queue, the queue has to become mutually exclusive, only allowing a single worker to access it at a time. This method is often paired with the Manager-Worker paradigm. The manager would create or gather tasks that need to be done then insert them into the queue while the workers continue working on the tasks. This centralizes the whole process of adding tasks to the queue [14, 15].

1.1.5 Work Stealing

Work stealing is a method for parallelizing a set of tasks. Similar to Work Sharing, jobs are all grouped together into a queue also commonly called a "bag of jobs", that workers grab from. The main difference from Work Sharing is that each worker gets its own queue of tasks that it has to work on. Once a workers' queue is empty, then it will begin to steal jobs from other worker's queues, hence the name "Work Stealing". The reason why this is sometimes more beneficial for some applications compared to the Work Sharing method is because the queues are all atomic, meaning that only a single worker can access it at a time. If there are many workers all accessing the same queue very quickly, then the overhead and waiting to get into the queue will slow things down. By giving each worker its own queue of tasks, then these workers can get jobs without needing to compete with other workers. The stealing from other queues is to ensure that even if one worker finishes before another, it won't be sitting idle when there are tasks still waiting to be done. The amount of workers accessing the same queue is therefore reduced. It can be reduced even more by algorithms that choose specific queues to steal from, or a different distribution of tasks into the worker queues [14, 16, 17].

1.2 Related Works

1.2.1 Bitmap Reordering

One way to improve how fast and small a bitmap index can be compressed is by leveraging reordering techniques. Either rows or columns can be reordered. Reordering one effects how well the other is compressed. Row reordering is most common since a lot of the compression techniques are based upon column compression. When reordering, the entire column or row must be moved to maintain the integrity of the data, as specific tuple are usually associated with an entire row. Grey Code is a common reordering method that is used to reorder a bit vector to increase the amount of 1's and 0's that occur in sequence. Often an expensive operation, this method often helps improve query times and the overhead an index can have. This is especially helpful for compression methods that perform exceptionally well on long sequences of 1's or 0's, for example WAH. Of the experiments that use reordering techniques, Grey Code is the method chosen [18, 19].

1.2.2 Binning Methods

Goyal, *et al.* present a method for binning where the ranges are completely dependent on the frequency of queries [20]. The bins are chosen such that the more a specific attribute is queried, the smaller range the bin that contains it will have. This continues to the point where if a specific attribute is queried enough, it will be in an exact bin. This method also works for any type of query, one sided, two sided or equality. The goal of this is to reduce the amount of candidate checks. The overlapping of bins can lead to queries being answered using more than one set of bins, a fact that can be exploited to further optimize the amount of candidate checks. At the end of this paper an additional research question was asked, can this method be done dynamically. That is, can the range of bins be optimized as more data on the frequencies of different queries comes in [20]?

Rotem, *et al.* present an algorithm for optimizing the number of bins, with emphasis on minimizing candidate checks [21]. Their approach is to change the range the bins cover so that the fewest amount of edge queries happen. Edge queries are the case when a specific query crosses over two bins, having at most two edges. The query can cover an entire bin and then cross over one or both edges. This means that for the other bins

where the query only needs a portion of the values represented by that bin, candidate checks have to be performed. To minimize the amount of edge queries, they propose an optimization of bins, thereby modifying the ranges that the bins cover. According to their experiments this approach leads to much faster query times when compared with bins that use equi-width and equi-depth binning strategies [21].

Sinha, *et al.* propose a multiresolution bitmap index [22]. This method greatly improves query performance, at the cost of additional space required to store the bitmap index. It not only reduces the amount of time it takes to process a query, but it also reduces the portion of the data that has to be loaded into memory. This is done by creating bitmaps using generic binning strategies, such as equi-width. Sinha, *et al.* describes each level as being structured with different resolutions. Resolution in this case means how many attributes each bin contains. The higher the resolution the less attributes the bins will contain down to the highest resolution containing a bin for each attribute possible. The bins follow normal binning protocols for handling floating point numbers, that is at the highest resolution bitmap containing variables with attributes being floating point numbers, only have bins with accuracy matching the queries that will be used on this bitmap. This provides a bound on what would otherwise be unbounded bins for floating point attributes [22].

Sinha, *et al.* also proposed an innovative way of accessing the bitmap index using a caching method [23]. Using different levels of bitmaps coupled with the multilevel and caching system they are able to reduce the amount of candidate checks needed for bitmap indexes with ranges. They also propose using a projection index or PI that consists of `<objectID, attributeValue>`. This will also be partially cached in memory for fast look up. The list of PI is sorted on attribute value allowing for easy checks to see if a specific objectID has the attribute of the query without doing a candidate check [23].

1.2.3 Compression Methods

Fusco, *et al.* propose to compress bitmaps in a way that is unique this data set [24]. There are two different areas where compression takes place, the incoming flow of data and the index. The related topic is with regards to the index, *i.e.*, bitmap. The method proposed is dubbed COMPAX. It is similar in nature to the common compression method called WAH, in fact builds upon it. In addition to the literal and fill word types that

WAH uses, they propose two additional word types called LFL and FLF. These two are an additional compression of 3 words, Literal-0-Fill-Literal and 0-Fill-Literal-0-Fill respectively. The compression performed takes a Literal or a Fill with only one dirty bit, the dirty bit being in position 0 for the 0-fill word and compresses 3 words into 1. The COMPAX compression method compressed the bitmaps up to half of what the traditional WAH method could. The query times also decreased by more than 40 percent. This is related in that they customized the compression to be specifically tailored to the dataset. They did this through observation and began to notice patterns of words that could easily be compressed even more without effecting the performance or functionality of the compressed index. The compression method also deals with very fast data streams [24].

Deliege, *et al.* devise a method that builds upon WAH [25]. Compression takes place according to WAH's algorithm. Once compressed PLWAH, or Position List Word Aligned Hybrid, attempts to merge fills with literals. This is done by looking at words adjacent to fills. If these adjacent words are literals and are *nearly identical*, then the two words are merged into one. Nearly identical is defined in the work as the maximum number of bits differing set by some threshold, which is defined in the work. To merge the two words, a portion of the fill word will be used to represent the literal word. This only occurs if there are enough unset bits in the fill word. The number of bits available in the fill word is determined by how many fills it is currently representing. This leads to higher initial compression values, but on average 20% faster query times [25].

1.2.4 Visualization

Cuff, *et al.* introduce a novel way of visualizing empiric power systems [25]. They propose a 2-D model using electrical distance and colors to define the electrical connectivity and structure. To assist in visualization matrices are also created containing various measurements from the power system and are color coded to match the associated graphs. Their goal is to allow operators to easily determine the systems connectivity by improving upon the single line graphs commonly used today [26].

Chapter 2: Bitmap Indexing Applied to PMU Datasets

2.1 System Design

Given a user query that selects a subset of records from the PMU data archive, the naïve approach to respond to the query would be to perform a linear scan of the database, comparing each record for selection, and then returning the matching records. For a real-time application that depends on the power operators' situation awareness, this operation would be too expensive because disk I/O operations are slow. Our PMU data management system, depicted in Fig. 2.1, has multiple software components allowing the user to build a bitmap index over raw data, and to efficiently query records that match specifications.

The *Bitmap Creator* inputs the raw PMU data and generates a bitmap using the binning strategy specified in Section 2.2. When new files are added to the database, these records will simply be appended onto the index. Once the bitmap is created, the *Compressor* will compress the index using WAH. After compression, the system is ready to receive queries from the user. These queries will give selection conditions on values of particular attributes the user is interested in. The *Query Engine* then translates the query into boolean operations over the specified bins in the compressed index. This will produce a *Result Bit Vector* v_R, which contains information on which records we need to retrieve from disk.

While v_R holds the selected record information (all bits with a value of 1), it is the actual data on disk that must be returned. An intermediate data structure, the *File Map*, was created to facilitate this role. The File Map is an intermediate data structure that holds metadata on the files and how many tuples[1] they each contain. There are two values per File Map entry: *totalRowCount* and *filePointer*. The *totalRowCount* contains the total number of tuples up to and including that particular file. The *filePointer* holds a pointer to the corresponding file on disk that contains the next set of tuples. To retrieve files with this method, the result bit vector is first scanned and a *count* is kept for the number of bits that have been read. For each hit, the count is hashed to its corresponding index in the File Map. This is an upper-bound hash, meaning that the count value is hashed to the closest *totalRowCount* value, without being greater than it. This will give the corresponding file that is desired. This allows us to scan the result bit-vector and only access disk when a record needs to be retrieved, significantly reducing the amount

[1] An entry (or record) in the database

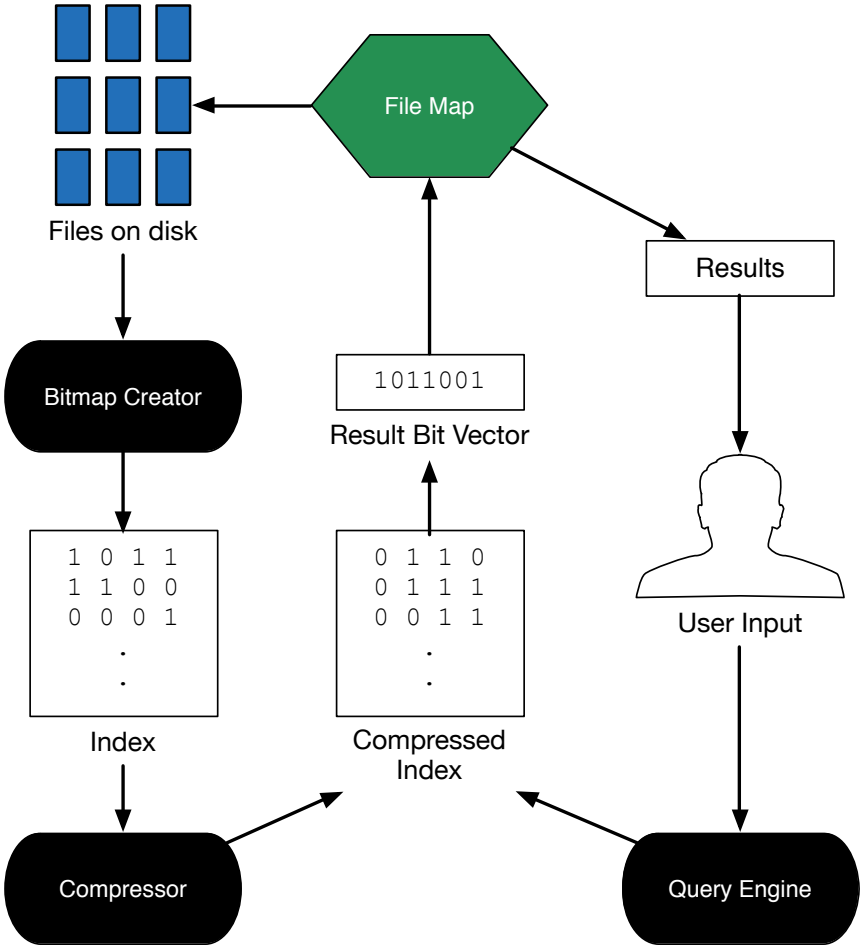

Figure 2.1: PMU Data Management System Architecture

of I/O required.

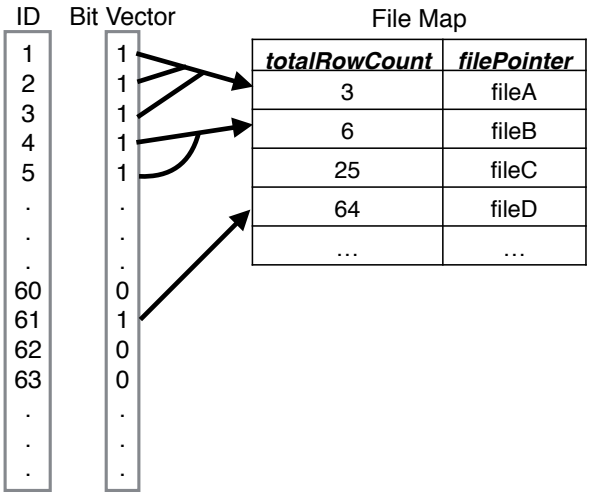

Figure 2.2: File Map Structure

Fig. 2.2 illustrates a small example of a bit vector and where the bits hash to the filemap. Bits one through three are hashed to the first row in the File Map structure. Bits 4 and 5 are hashed to the second row since these bits represent tuples 4 and 5, which are stored in *fileB*. With the upper bound hash, bits 4 and 5 hash to *totalRowCount* 6, since they are both greater than 3 but less than or equal to 6. Bits 60, 62, and 63 are not hashed since they are not hits. Only bits in the bit vector that have value one will be hashed. This leads to improved performance when there are long stretches of zeroes in the bit vector.

2.2 Characterizing PMU Data for Bitmap Indexing

We obtained 950 GB of data from a number of PMUs within the Bonneville Power Administration's (BPA) operating region from August 2012 to August 2013. At each PMU, a *phasor* measurement is sampled every 1/60 sec. Each measurement is represented by a *date-time* and a *phasor*, which is a pair of values: the phase angle ϕ and the voltage magnitude V. The phasors from 20 PMUs are combined, resulting in 2×20 PMUs $= 40$ attributes. The phase angle ϕ is a time-varying real number that oscillates within the range of $[-180, 180]$. The voltage, on the other hand, is a non-negative real number. In order to define the bitmap ranges, we examined ϕ and V's distributions. We analyzed the distribution of ϕ and V over a sample size of 30 days ($155,520,000$ measurements).

To optimize for speed, the design of the bitmap must be informed by the queries that will be frequently executed. For frequently queried values in bitmap structures, a crippling factor in response time is the candidacy checks to identify true positives, which require disk access. Due to imperfect discretization, bins will often contain bits that indicate more than one value. It is therefore necessary to check whether that bit is an indication of the correct value. For example, if a bin has the range of five possible values then that means each bit in that bin is one of five different values. Performing this check, called a *candidacy check*, ensures that the *tuple* contains the desired value for the query. Choosing the correct binning strategy can therefore potentially improve our query times by reducing candidacy checks among values that were expected to be queried.

From discussions with power systems experts at BPA, queries typically comprise a specific range of dates, voltage magnitude V, phase angle ϕ, or any combination of these attributes. When generating the bitmap, the binning (discretization) strategy can minimize candidate record checks and provide fast query response times. Due to the low cardinality of the *date-time* attribute, the generation of bins is simple: 60 bins each for second and minute, 24 bins for hour, 31 bins for day, etc. with the exception of the year. In this case we used 11 bins for the year, ranging from year 2010 to 2020. Since there were no range bins, no candidacy checks were necessary when performing queries on the dates. Because ϕ and V are real values, we discretize based on their distribution. In order to find the distributions of both ϕ and V, the cumulative distribution function (CDF) plots are constructed. These distributions determined what binning strategies were used.

Fig. 2.3 illustrates the phase angle ϕ distribution. From this graph we can see that ϕ follows a uniform distribution. Because ϕ is also bounded, we apply an *equal-width* binning strategy over ϕ, meaning the range of each bin is equivalent. We designed the *bitmap creator* in such a way that this range can be assigned by the user before creation of the bitmap. For our experiments we set this value to 10, leaving 36 bins for each PMU attribute. Fig. 2.4 represents the phase angle values that were assigned to each bin.

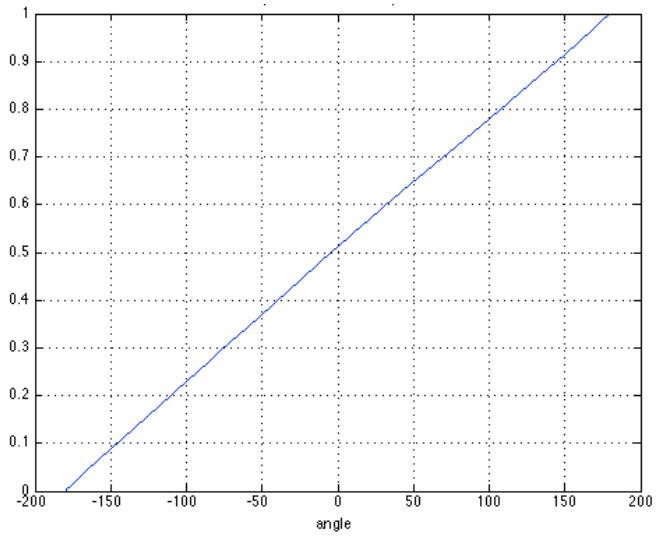

Figure 2.3: Normal Phase Angle CDF

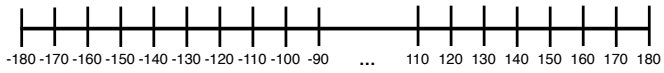

Figure 2.4: Phase Angle Bins

Fig. 2.5 illustrates the distribution for normal operation of a PMU's voltage magnitude. The majority of the values occur between [535...545]. For this attribute, we used a binning strategy which attempts to minimize candidacy checks for the values that are most likely to be queried. We assume the majority of queries from the user will pertain to some anomaly, that is values that are not apart of normal operations. Therefore, a bin with range [535...545] can be created to contain the regularly occurring values. Since

the range of the bin is quite large, and it spans the values which occur most frequently, then the majority of tuples which fall into this category will require candidacy checks. However, our assumption is that queries will occur for values outside the normal range. This leads to a specific strategy for binning: There are ten bins on either side of the central bin representing the normal operational range. Each of these outer bins is capable of containing a value with a range of one. Fig. 2.6 represents the binning distribution for voltage magnitude. There is an additional bin for the value zero, since this is an indication of a data event at a PMU site. This strategy generates bins of small ranges for values of V that will be queried frequently and very large bins for those that aren't.

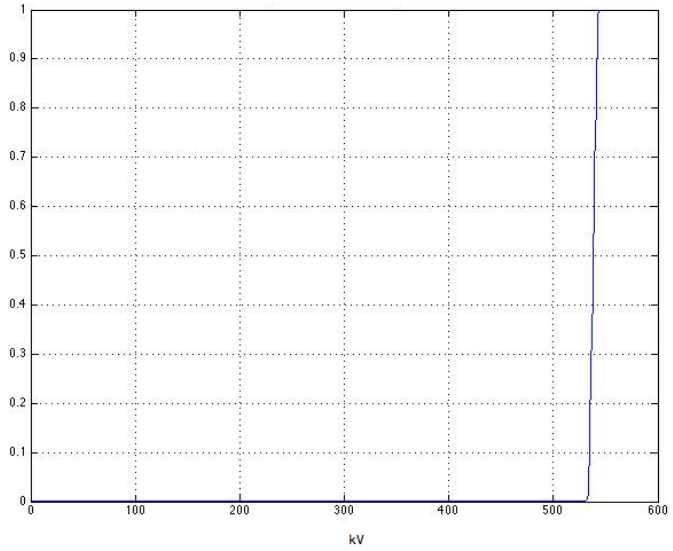

Figure 2.5: Normal Voltage Magnitude CDF

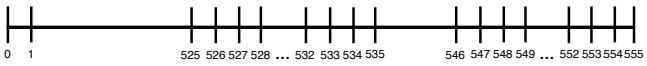

Figure 2.6: Voltage Magnitude Binning

In addition to the aforementioned attributes, we also introduced an attribute Δ, which represents the displacement between phase angles from the previous time-stamp, *i.e.*, $\Delta_t = |\phi_t - \phi_{t-1}|$. We bin Δ with smaller ranges, reducing the number of candidacy

ID	Selection Criteria	Linear Scan (sec)	Bitmap (sec)	Tuples Retrieved
1	Find all tuples where PMU1 has a magnitude Voltage Magnitude of 533.	25.859666	0.379387	160
2	Find all tuples that happened on exactly June 24, 2013 at 21:05 hours.	25.350993	0.854952	7204
3	Find all tuples that happened on exactly June 24, 2013 at 21:06 hours.	28.001001	0.922941	7204
4	Find all tuples that happened on exactly June 24, 2013 at 21:07 hours.	26.133607	0.785588	7204
5	Find all tuples that happened on exactly June 24, 2013 at 21:06 hours with PMU having a Voltage Magnitude of 533.	28.019449	0.001772	0
6	Find all tuples in 2012.	26.720291	0.0000601	0

Table 2.1: Query Performance

checks. Listed below are the bins that we used for each attribute. The total is 9,768 bins for each row in the bitmap index.

- Year: 11
- Month: 12
- Day: 31
- Hour: 24
- Minute: 60
- Second: 60
- Millisecond: 10
- ϕ (23 for each PMU): 40×36
- V (36 for each PMU): 40×23
- Δ (180 for each PMU): 40×180

2.3 Results

Queries were ran over the database to demonstrate the performance gains from analyzing and creating a bitmap index over the data. For these experiments, 4 million rows from the database were queried. File Map retrieved the tuples from the database once a query has been serviced. The bitmap results are compared against the common linear scan that is performed when searching a database.

Table 2.1 shows results from six queries that were run. Query ID 1 is an example of a query where the user wishes to find when a specific PMU had a voltage magnitude of 533. An example of when this might happen is if the Correlation Visualization indicates there is an event occurring when that PMU has a voltage magnitude of 533. The exact same query to the bitmap engine provide a 68× speed up on retrieval. Query IDs 2-4 demonstrate examples of requests for tuples at specific dates. These demonstrate that performing multiple queries with small adjustments does not require much additional time. Query IDs 5 and 6 shows queries for tuples that do not exist in this data set. Since the bitmap engine able to look at the bit vector results without every going to disk to see if the desired tuples are in the database, the speedup is many orders of magnitude greater. The bitmap query ID 5 takes slightly more time than ID 6 because ID 5 has to perform bitwise ANDs between each column, while ID 6 is simply checking a single column. There is very little time difference between the linear scan in ID 5 and 6.

The linear scan times are so similar because no matter the query given, it is necessary to scan the entire data set to ensure accuracy. Bitmap index query times can vary and primarily depend on how many columns need to be compared and how many tuples need to be pulled from disk. In fact the majority of the time spent for the bitmap index queries is simply retrieving the tuples from disk, making I/O the limiting factor.

2.4 Future Work

The methods presented produced results that prove promising when retrieving this data. Presented below is the direction we plan to take the methods we currently have, making them scalable and more efficient.

Given large data sets, it is necessary to add additional methods of indexing for faster navigation and for queries to be returned in reasonable amounts of time. One such method that could be applied is sampling. This adds tiers of bitmaps, *i.e.*, bitmap indices for progressively more precise bitmaps, each one at a lower resolution of the data. For small amounts of data this is simply wasted space and too much overhead. When bit data such as this is introduced the sampling overhead begins to diminish as access times to the data doesn't scale up with the amount of data as quickly.

2.5 Conclusions

In conclusion, we have shown that our bitmapping database minimizes data driven bottlenecks that are typically associated with data sets of this size. Specifically, compression of the data minimizes the space overhead required for indexing, allowing it to be operated on within memory. Query response times are also minimized due to the utilization of indexing coupled with the FileMap structure. This results in the ability to perform frequent queries leading to better analysis of the data.

This system is ideal to be coupled with other PMU analysis tools in order to better understand the data. One such tool that this system has already been coupled with is a correlation algorithm that is used to detect events occurring within the power system [27]. The combination of these two systems allow for fast retrieval of data in order identify anomalies among other events.

Chapter 3: Parallel Bitmap Compression

Data collection and generation devices such as sensors have become incredibly popular and capable of storing large amounts of data. Due to this massive rise in data collection rates, bitmap indexes can quickly grow out-of-core without compression. Machines have continued to gain computational power, more recently with the addition of parallel architectures (*e.g.*, multi-core CPUs), enabling users to use symmetric multiprocessing (SMP) for computational tasks. Parallelizing bitmap compression is the logical next step to improve run times. The two different parallel methods are compared against each other initially. What occurs is neither perform as expected, leading to further investigation into possible causes for the poor performance.

3.1 Work Distribution

Bitmap compression takes place by columns, and each column is compressed independently from each other. Therefore, we have n independent tasks (for the n columns of the bitmap), distributed to a number of threads using the Work Sharing and Work Stealing methods.

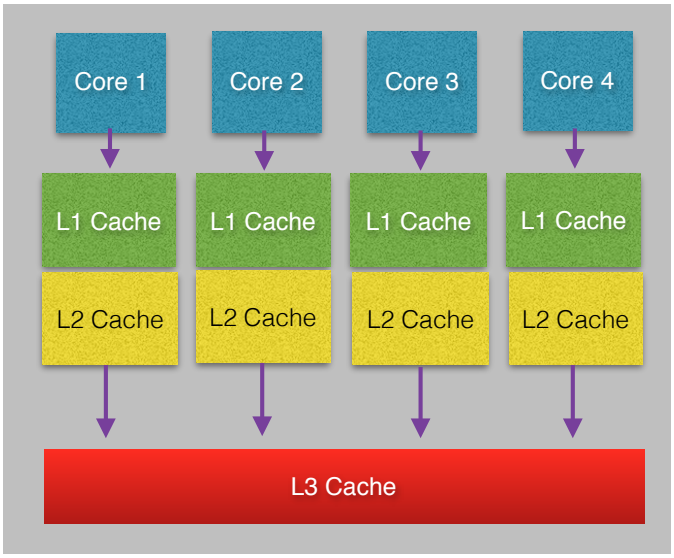

Figure 3.1: CPU cache for test implemented

The following experiments were run in the Java environment. *Workers* refer to an individual thread and a *task* represents a column that needs to be compressed. Two

different schemes are used when parallelizing the compression, Work Sharing and Work Stealing, introduced earlier in this work.

These experiments were tested on a machine that had an architecture with four cores. Figure 3.1 illustrates a simple example of the different levels of caching that this specific CPU implements.

3.2 Results

The experiments comparing work sharing and work stealing were run on two different bitmaps, each with different properties: (1) in the uniform bitmap, the 1's are distributed out evenly. It consists of only 100 columns and 10000 rows, which means the tasks are small. (2) The second bitmap used for evaluation is known as HEP, which consists of 122 columns and 2.2 million rows. The tasks, which consists of compressing a single column, will be much larger. We run our experiments on an OSX Mavericks machine running Intel Core i5 Quad Core (2.5GHz) with 8 GB memory. Table 3.1 contains the specifications on the machine these tests were performed. The performance graphs are based upon how much of an improvement the parallelization made when compared to the sequential execution. The Y axis is times speedup based upon 1 being the sequential version of the compression.

Operating System	Mac OSX Mavericks
Processor	Intel Core i5 Quad Core (2.5GHz)
Memory	8 GB, 1333 MHz, DDR3

Table 3.1: System Specifications

We first focus on the results for the uniform bitmap, shown in Figure 3.3. Both Work Stealing and Work Sharing have similar speedup times. At three threads Work Stealing appears to start gaining some performance above Work Sharing, but then it tapers off. Figure 3.2 shows how the Work Stealing and Work Sharing methods perform on the Hep bitmap index. Again the two are relatively similar in speedup time relative to the sequential version. In both results the Work Stealing model does not perform better than Work Sharing. This is because threads are completing their tasks at relatively the same speed with little to no stealing occurring.

As we saw in our results, the expected speedup is not achieved. We suspect the cause of the bottleneck being attributed to the last-level (L3) cache misses. In the CPU used in our experimental study, the L3 cache is shared among all the processors. We used Intel VTune Amplifier XE to analyze cache misses for each thread count. Figure 3.4 illustrates the cache misses on increasing number of threads. As we can see, there is indeed an increase in L3 misses when four threads are used. We believe these are due to conflict misses, *i.e.*, prematurely evicting necessary data from L3 that another

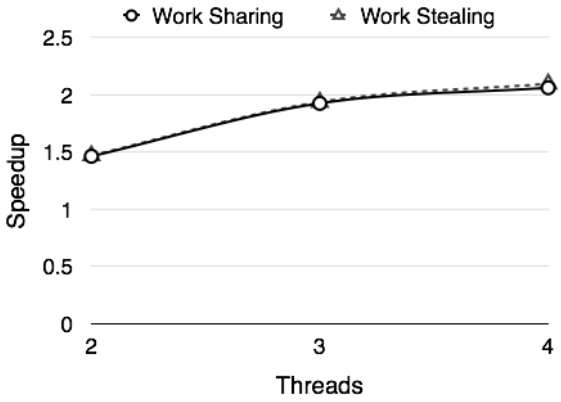

Figure 3.2: HEP Bitmap Index

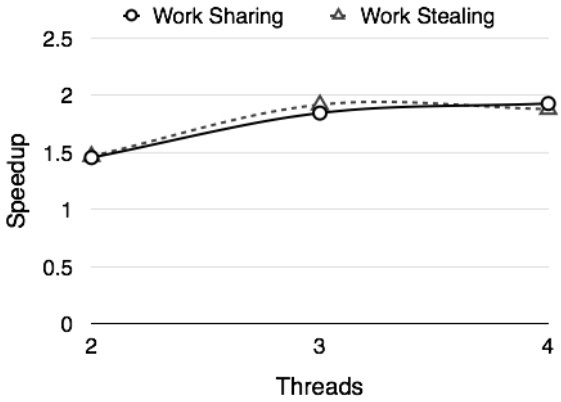

Figure 3.3: Uniform Bitmap Index

thread eventually requires. When space is needed by another core, other data it deems unnecessary will be evicted out of L3.

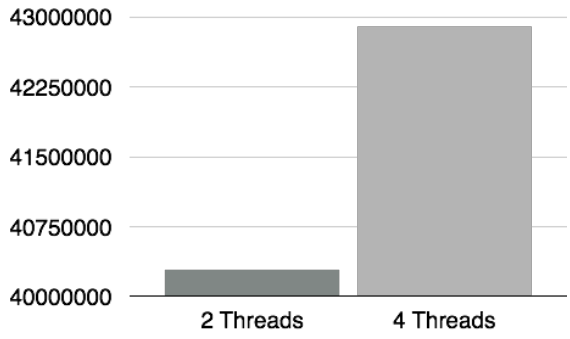

Figure 3.4: Cache Misses per Thread Count

3.3 Future Work

There are still other ways that compression could be made parallel. Instead of having the threads write the compressed bit vector to a file themselves, they could assign this to a new set of threads. This new set would only have the purpose of writing to files and won't actually do any computation. Whether this would actually increase performance or not would still have to be tested, but it may decrease the amount of workers needed to do the computation since none of them will be performing I/O. The compression would probably happen faster, but the computer will still be performing I/O which will affect scheduling and therefore performance.

Tests should also be run on bitmaps of much larger size. This could be tested on a distributed system where instead of using threads the different parallel methods would use nodes in a cloud. This could drastically improve the performance of compression because then each node would have its own memory, CPU, hard disk, etc. and wouldn't have to compete with the other workers for resources. Communication between nodes for Work Stealing may slow things down slightly, due to the need to talk to other workers in the system and steal tasks from their queues. This option would negate the drawbacks of parallization on a single machine. There should be very minimal L3 cache misses, leading to a speedup closer to what is expected.

One possible way to improve parallel operation on a single machine would be specifying the blocks that each thread should read into the CPU cache. In this case, each thread could read in a portion of data that each thread needs, leaving its block containing data for all the threads. This could minimize evictions since threads first have to check to see if the data the data is present in the L3 cache before retrieving it from memory, essentially allowing threads to share blocks of data and help with the retrieval.

3.4 Conclusion

Our preliminary results show there was not a salient difference in performance between work stealing and work sharing in parallel. Interestingly, we also showed that neither achieved the expected speedup over our simple SMP, and we believe that this is due to the amount of conflict misses that occur in the L3 cache with an increasing number of threads. This causes additional overhead that inhibits the performance increases that can be gained from parallelization.

For future work, we will design and evaluate a cache-friendly compression framework. In this design, we will interleave data elements that each thread needs such that when a single block is read into the cache it will contain data elements for all threads. Cache blocks will contain data elements for all threads instead of a single thread. With this design, the number of conflict misses should be reduced as data that other threads need would already be in the L3 cache since it was brought in by another thread, leading to reduced reads from memory. Since the threads are completing tasks at relatively the same speed, when blocks are evicted from cache the data contained therein will have already been operated on where before blocks could be evicted that contained data another thread needed causing it to be read in again.

Chapter 4: Dynamic Bitmap Operations

4.1 Methodology

This section is separated into two subsections and illustrates the experiments that will be run for this research. The first type of experiment will be performing various queries on the bitmaps that have been compressed using the methods described next. Query performance in a bitmap index depends on the compression ratio of said index. For most word aligned hybrid models, better compression means faster queries since the queries can take place without decompression. The second types of experiments that will be run will be a comparison of size. Bitmaps usually compress better with long vectors of bits. There is a trade off between how often we compress and how good our compression ratio is. These experiments utilize two different compression methods, WAH and PLWAH. WAH uses both 32 and 64 bit word sizes while PLWAH uses only 64 bit word sizes.

4.1.1 Dynamic Compression

As is mentioned in Section 1.1, PMUs update their datasets frequently and the data they produce can grow large rather quickly. Our PMUs are gathering data at a rate of 60 measurements per second, still capable of generating large amounts of data in a short period of time.

Normally bitmap indexes are applied to large static databases due to their inability to inherently update. Traditional means of updating a bitmap consist of a complete decompression, adding new bitmap index data, and then performing compression over the entire index again. This can be slow and time consuming given the large datasets indexes are usually built over. We propose that the bitmap can be updated at certain intervals allowing for recent data to be queried. The updates will be fast since they will not require the entire bitmap index to be decompressed, only that the new data is compressed and appended onto the existing index.

Figure 4.1 depicts an example of how the files are broken up to simulate the time intervals at which the bitmap is compressed. The bitmap index is created over a large PMU dataset. For the initial experiments we took a 100MB subsection of that index to obtain some preliminary results and construct the experiments. Additional experiments were run on a bitmap index of size 3GB. Each file is broken up the same way in the same intervals. As depicted in the figure, 25 percent is relative to the size of the bitmap index

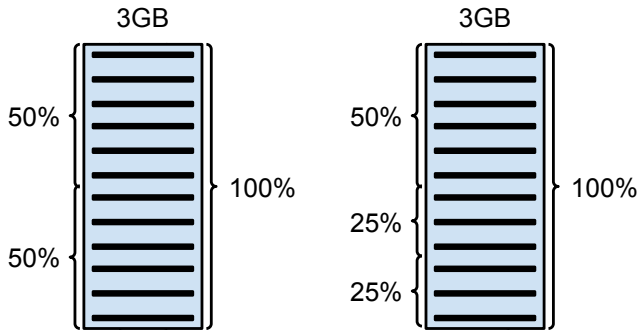

Figure 4.1: Bitmap separation example

being broken up. In a real system implementation this value would be some threshold that when met, the new data would be compressed and appended. For this example, the threshold is 0.75GB, meaning that when the new bitmap index built over the live data meets this size, it will be compressed and appended onto the already compressed bitmap index. The compression ratio of the dynamic compression will be compared to what is considered to be the optimal compression for this bitmap index, the unbroken bitmap of either 100MB or 3GB.

4.1.2 Queries

All queries performed on bitmap indexes are bitwise operations between at least two compressed bit vectors. Neither compression method used requires any alignment, since the word size is static throughout all the bit vectors. Currently only point queries between two columns are implemented. In the future range queries and multiple column queries will be available.

Two different variations will be used when constructing which queries should be run. The first variation is to change which columns are queried. This is important because we can estimate the length of runs in a given column based upon our knowledge of the data and how that column is binned. For example, the voltage magnitude bin of [335...346] consists of all the data points that happen during normal operation for our PMUs. That is to say that the majority of the tuples will have a hit in this column resulting in long

sequences of 1's which compress very nicely. We suspect these bins will be affected the most by dynamic compression. On the other hand the millisecond bin will be an even mixture of 1's and 0's resulting in very poor compression. Knowing this we can perform queries on this bin knowing that even if given the entire dataset to compress, it would be no better then compressing in real time and adding to the index.

The second query variation we can construct would be to perform different queries that contain different bit operations between columns that are full of runs and those full of literals. We expect that the more frequently the bitmap is compressed in real time, the worse these results will be since the runs will get smaller and increase the chance for literals to occur. The more bits in a bit vector increase the amount of time it takes to solve a query due to the increased number of bit operations that need to be applied.

When measuring the query response times, the amount of time it takes to retrieve the tuple from disk will be neglected. The queries will all have the same result, regardless of how well they are compressed, leaving this factor to be considered a constant time necessary for any query. Instead the determining metric when measuring how long a query takes when compared to another would be the difference it bit operations between the two. The query itself that is resolved through bitwise operations will be faster the more compressed the bit vector is.

Query ID	**Description**
1	Find all tuples that occur in 2014 on the 22nd of each month
2	Find all tuples in 2014 that have voltage magnitude of 0-1
3	Find all tuples with voltage 0-1 on the 22nd of each month
4	Find all tuples in 2014 with voltage magnitude 525-526

Table 4.1: List of Queries

Table 4.1 lists the queries that will be run on the dataset. Query 1 is a simple selection of dates. The year column will be evenly populated while the day column will be sparsely populated. Queries 2 and 3 look for a particular voltage measurement at specific times. The voltage value will be even more sparsely populated than the 22nd column, as it isn't regular. This will lead to more dirty bits and a worse compression ratio. Query 4 seeks another voltage reading, this one more popular as it is closer to the nominal operating range of a PMU. This column should have the most even ratio of 0

to 1 bits in the selected columns.

These queries were selected to highlight one of the benefits of bitmapping that dynamic updating may impair, compression ratio. When a column such as 2014 is compressed, its compression ratio will be very near the maximum a particular column can be compressed, since it will either a small run of 1s with the remaining tuples containing all 0s, leaving longs runs to be compressed. Dynamically compressing columns like this will impair the long runs, interrupting them requiring the use of another word. Dirty columns such as the voltage magnitude of [525...526] won't be affected by the dynamic compression as much, since there won't be many runs being broken up.

4.2 Results

Bitmap indexes of both 100MB and 3GB in size are used for the compression experiments. The query results use the 3GB bitmap, as the 100MB bitmap was too small to show any significant results and the query times were hardly measurable. In these experiments we assume the optimal compression ratio is a compression of the entire dataset. The benefit of compressing dynamically is that you can update the bitmap. In practice the bitmap would not be updated like this, but would instead be decompressed and compressed again with the additional data. We propose that this is more time consuming than doing it dynamically, to a point. The speed benefit isn't worth the time to decompress and recompress the index when the additional data reaches a critical point in which query times would be far faster with a stand recompression of the entire index.

4.2.1 Compression Results

Tables 4.2 and 4.3 illustrate the compression results from our initial experiments. The *Type* column indicates how the file is broken up. Columns two and three represent the size of the resulting bitmap after compression using 32 and 64 bit words respectively. We start with the original file which is 100MB in size. This is assumed to be the optimal compression of the dataset when compressed as a whole. Once compressed you can see that the final bitmap only takes up approximately one quarter of a megabyte.

The next step is breaking the file in half. In this scenario the initial bitmap would be half the size of the entire dataset. To simulate the dynamic compression we set our compress point to 50 percent. This means that when the live datastream reaches a certain size, in this case 50 percent of the entire test dataset, we compress it and add it to the initial compressed bitmap.

For the second compression experiment we did a similar thing, except we set the threshold for compression to be 25 percent of the original dataset. In this case that is when the new bitmap is 25MB large it is compressed. This continues for the rest of the test data. The 50-25-25 case compresses almost as well as the 50-50 case. This result is probably the most telling. It may be that compressing a larger chunk as the initial bitmap then compressing live data when it reaches a reasonable size. This will become clearer with the future experiments.

Similar steps are taken for the remaining experiments. It is interesting to note that Wah 64 performs worse than Wah 32 in all cases. This is due to the fact that when a compression method is forced to use larger words, it is more likely that a dirty bit will be included in the set, therefor increasing the number of literals in the compressed bitmap. Larger words are best used to compress bit vectors containing mostly runs. As Table 4.2 and 4.3 depicts, when the entire bitmap is compressed it is smaller in size. Most of the time this also carries the correlation that queries will be faster as well.

Type	**Wah 32 (MB)**	**Wah 64(MB)**
Original	0.267239	0.294951
50-50	0.272918	0.300886
25-25-25-25	0.28398	0.312926
50-10-10-10-10-10	0.29557	0.34722
50-25-25	0.278525	0.306997

Table 4.2: Compression Results - 100MB

Type	**Wah32 (MB)**	**Wah64 (MB)**
Original	0	0
50-50	.021% Larger	.020% Larger
50-25-25	.042% Larger	.041% Larger
50-10-10-10-10-10	.106% Larger	.177% Larger
25-25-25-25	.063% Larger	.061% Larger

Table 4.3: Compression Results - 100MB Percentage

Tables 4.4 and 4.5 depict the compression results when the same file thresholds were used on the 3GB bitmap. When the experiments were first setup we expected the size difference to be even more pronounced then the 100MB file, but the opposite is true. WAH32 hardly changes with PLWAH being the least effected by the dynamic compression strategies employed. The initial size of PLWAH is much larger than that of WAH, but it shows promise for not being impacted much by the dynamic compression strategies.

Both of the compression experiments indicate that no matter the threshold, compression will indeed be worse then optimal. We propose that this fact doesn't outweigh the benefits of being able to update the bitmap without the traditional method. PLWAH32

Type	Wah32 (MB)	Wah64 (MB)	PLWAH32 (MB)
Original	8.635359	9.728087	23.084176
50-50	8.64011	9.733774	23.089032
50-25-25	8.645861	9.739549	23.093984
50-10-10-10-10-10	8.663218	9.758658	23.10972
25-25-25-25	8.651532	9.755668	23.092744

Table 4.4: Compression Results - 3GB

Type	Wah32 (MB)	Wah64 (MB)	PLWAH32 (MB)
Original	0	0	0
50-50	.0010% Larger	.0010% Larger	.0002% Larger
50-25-25	.0012% Larger	.0012% Larger	.0004% Larger
50-10-10-10-10-10	.0012% Larger	.0031% Larger	.0011% Larger
25-25-25-25	.0012% Larger	.0028% Larger	.0004% Larger

Table 4.5: Compression Results - 3GB Percentage

particularly showed the least impact by this. The difference in size between the optimally compressed bitmap and the dynamic compression at thresholds is negligible. We further minimized the difference when the same experiments were tested on a larger bitmap than the initial index. This may not hold with further testing into larger bitmaps.

4.2.2 Query Results

The queries discussed in Section 4.1 were all run together and timed. The group of 4 queries was run 4 times after an initial run to load into memory and load some into CPU cache with the average taken of all 4 runs. The times shown in Table 4.6 are in milliseconds. Since the compressed bitmaps that were appended to each other use the same word size for each compression method they are always aligned.

As expected, PLWAH has a lower query time when compare to WAH when the bitmap is compressed optimally. As the file gets broken up even more we can see that the query times for WAH32 increase quickly. WAH64 starts off slower than PLWAH, but eventually catches up. They are both still competitive and do not differ much, but WAH64 seems to overtake PLWAH and perform better when the bitmap is compressed

Type	**Wah32 (ms)**	**Wah64 (ms)**	**PLWAH32 (ms)**
Original	0.401408	0.402005	0.328192
50-50	0.507563	0.414464	0.379136
50-25-25	0.583766	0.397568	0.45056
50-10-10-10-10-10	0.926891	0.471381	0.49314
25-25-25-25	0.674133	0.414123	0.447403

Table 4.6: Query Results - 3GB

dynamically.

4.3 Future Work

There are multiple experiments that can be run in addition to the work that has already been performed. One such experiment would be to test on a larger bitmap index. The compression differences became less apparent between the 100MB and 3GB bitmaps indicating that running similar tests on a larger bitmap could produce different results as well.

Another aspect that could be researched is running the experiments with a much smaller threshold over more data. For example, the threshold could be set at 100MB for a large bitmap and then receive data for a few hundred gigabytes. This method could be extended by keeping the threshold that was used in the experiments, but running it over more data. This might make the results that were presented in this project more apparent.

A final extension would be to use different compression methods. PLWAH with 64 bit words for example would be the logical next step. A requirement of any additional compression method used would be that the word size would have to be static, removing the possibility of dynamic word size compression methods such as VAL.

4.4 Conclusion

In summary, we have tested multiple thresholds for when compression should occur with the live data. These tests were compared on two levels, the compression ratio and the query times, with the optimal method respectively.

The compression results indicate that PLWAH would be a good candidate for dynamic compression of live data. The initial size of the compressed bitmap is larger than WAH, but PLWAH shows little to no impact when the bitmap is dynamically updated with respect to the compression ratio. WAH shows little change as well, but the change it does show is larger than that of PLWAH indicating that if the compression were to continue in the respective manner it would pass PLWAH in size and perform worse.

The query results lead us to conclude that WAH32 performs worse then both the other two methods of compression. WAH64 is comparable with PLWAH in many of the threshold configurations leaving us to choose which would be better.

Given the conclusions above and the experiments that were run we recommend that PLWAH be used for this type of dynamic compression of live data. Due to the minimal impact that the dynamic compression has to both the initial size of the bitmap and the comparable query times to other methods, this is the best fit.

The dynamic compression eventually will eventually lose its benefit, at which point it would be more beneficial to simply decompress all the bitmap parts and recompress to the optimal size. With this approach the bitmap index can be updated frequently allowing for recent data to be queried with the use of traditional means which can be time consuming.

Chapter 5: Conclusion

5.1 Conclusion

In summary, bitmap indexes are an efficient way to query large databases. This research has shown that when applied to the datasets generated by PMUs, they perform exceedingly well. The bottlenecks associated with large databases are minimized by performing query operations and storing the required overhead in memory. The FileMap structure that was created minimized disk access, increasing query times even further by reducing the I/O required to return tuples to the user.

Testing on a single machine suggested that parallelization will provide a small performance increase. Often when parallelization is implemented, performance gains are typically higher then were seen in these experiments. Further investigation led to the conclusion that the L3 cache is restricting performance increases that could otherwise be obtained by the parallel architectures. All threads using the same cache can cause data to be evicted when the thread using it wasn't finished, causing performance to slow down.

Bitmap indexes are known to be used on large static databases. The PMU database is indeed extremely large, but it is dynamic since it is continuously being updated. We defined a possible solution to creating bitmaps over dynamic databases while minimizing the performance drawbacks that are usually associated. There is little impact to the performance of both queries and compression when PLWAH is used. Overall this minimizes the amount of time that needs to be spent decompressing the entire bitmap index, updating it, and recompressing.

The contribution of this research is to demonstrate a framework for the PMU database system using bitmap indexing. When built upon this database, special considerations need to be realized, such as the frequently updated data that PMUs produce. To accommodate this, an model has been built and tested that shows minor impact on performance. Improvements have also been made to bitmap indexing to decrease compression times.

5.2 Future Work

Further analysis of the bitmap index built over the PMU dataset will allow for further study into its performance. One promising method is bitmap sampling, where bitmaps

are built upon other bitmaps. Even with additional overhead, this method might bring further improvements to the access times of the dataset as well as increasing scalability.

The study of parallelization over bitmap compression has suggested additional research into reducing cache misses in the L3 cache. Building the same algorithm on a computer cluster may reduce theses misses. Manually controlling what each thread reads into the CPU cache may provide additional benefits, as threads can assist one another in operations, reducing the need to evict data to make room.

Extending the research of building bitmaps over dynamic databases by testing the algorithm on larger datasets will allow for further feasibility testing. Testing different compression algorithms will allow users to have a variety of options when updating bitmap indexes.

Bibliography

[1] T.-C. Lin, P.-Y. Lin, and C.-W. Liu, "An Algorithm for Locating Faults in Three-Terminal Multisection Nonhomogeneous Transmission Lines Using Synchrophasor Measurements," 2014.

[2] M. Gol and A. Abur, "LAV Based Robust State Estimation for Systems Measured by PMUs,"

[3] V. Salehi, A. Mohamed, A. Mazloomzadeh, and O. A. Mohammed, "Laboratory-based smart power system, part II: Control, monitoring, and protection," *IEEE Transactions on Smart Grid*, vol. 3, no. 3, pp. 1405–1417, 2012.

[4] D. G. Hart, D. Uy, V. Gharpure, D. Novosel, D. Karlsson, and M. Kaba, "Pmus- a new approach to power network monitoring," *ABB Review*, vol. 1, pp. 58–61, 2001.

[5] P. E. O'Neil, "Model 204 architecture and performance," in *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, (London, UK), pp. 40–59, Springer-Verlag, 1989.

[6] R. R. Sinha and M. Winslett, "Multi-resolution bitmap indexes for scientific data," *ACM Transactions on Database Systems*, vol. 32, p. 2007.

[7] A. Romosan, A. Shoshani, K. Wu, V. M. Markowitz, and K. Mavrommatis, "Accelerating gene context analysis using bitmaps," in *SSDBM*, p. 26, 2013.

[8] Y. S. *et al.*, "Taming massive distributed datasets: data sampling using bitmap indices," in *HPDC*, pp. 13–24, 2013.

[9] F. Fusco, M. P. Stoecklin, and M. Vlachos, "Net-fli: On-the-fly compression, archiving and indexing of streaming network traffic," *PVLDB*, vol. 3, no. 2, pp. 1382–1393, 2010.

[10] K. Stockinger and K. Wu, "Bitmap indices for data warehouses," in *In Data Warehouses and OLAP. 2007. IRM*, Press, 2006.

[11] "Apache Hive Project, http://hive.apache.org."

[12] K. Wu, E. J. Otoo, and A. Shoshani, "An efficient compression scheme for bitmap indices," 2004.

[13] R. Slechta, J. Sawin, B. McCamish, D. Chiu, and G. Canahuate, "Optimizing query execution for variable-aligned length compression of bitmap indices," in *Proceedings of the 18th International Database Engineering & Applications Symposium*, pp. 217–226, ACM, 2014.

[14] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "A comparison of receiver-initiated and sender-initiated adaptive load sharing," *Performance Evaluation*, vol. 6, no. 1, 1986.

[15] D. Neill and A. Wierman, "On the benefits of work stealing in shared-memory multiprocessors,"

[16] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal, "A simple load balancing scheme for task allocation in parallel machines," in *SPAA '91*, pp. 237–245, ACM, 1991.

[17] P. Berenbrink, T. Friedetzky, and L. Goldberg, "The natural work-stealing algorithm is stable," *SIAM Journal on Computing*, vol. 32, no. 5, pp. 1260–1279, 2003.

[18] D. Lemire, O. Kaser, and K. Aouiche, "Sorting improves word-aligned bitmap indexes," *Data and Knowledge Engineering*, vol. 69, no. 1, pp. 3 – 28, 2010. Including Special Section: 11th {ACM} International Workshop on Data Warehousing and {OLAP} (DOLAP08) - Five selected and extended papers.

[19] T. Apaydin, A. Ş. Tosun, and H. Ferhatosmanoglu, "Analysis of basic data reordering techniques," in *Scientific and Statistical Database Management*, pp. 517–524, Springer, 2008.

[20] N. Goyal and Y. Sharma, "New binning strategy for bitmap indices on high cardinality attributes," in *Proceedings of the 2nd Bangalore Annual Compute Conference*, COMPUTE '09, (New York, NY, USA), pp. 22:1–22:5, ACM, 2009.

[21] D. Rotem, K. Stockinger, and K. Wu, "Efficient binning for bitmap indices on high-cardinality attributes," 2004.

[22] R. R. Sinha and M. Winslett, "Multi-resolution bitmap indexes for scientific data," *ACM Transactions on Database Systems (TODS)*, vol. 32, no. 3, p. 16, 2007.

[23] R. R. Sinha, M. Winslett, K. Wu, K. Stockinger, and A. Shoshani, "Adaptive bitmap indexes for space-constrained systems," in *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pp. 1418–1420, 2008.

[24] F. Fusco, M. P. Stoecklin, and M. Vlachos, "Net-fli: on-the-fly compression, archiving and indexing of streaming network traffic," *Proc. VLDB Endow.*, vol. 3, pp. 1382–1393, Sept. 2010.

[25] F. Deliège and T. B. Pedersen, "Position list word aligned hybrid: optimizing space and performance for compressed bitmaps," in *Proceedings of the 13th International Conference on Extending Database Technology*, pp. 228–239, ACM, 2010.

[26] P. Cuffe and A. Keane, "Visualizing the electrical structure of power systems,"

[27] R. Meier, E. Cotilla-Sanchez, B. McCamish, D. Chiu, B. Bass, J. Landford, and M. Histand, "Power system data management and analysis using synchrophasor data," in *IEEE Conference on Technologies for Sustainability*, (Portland, OR.), 2014.