Received: 18 December 2017 | Revised: 5 December 2018 | Accepted: 24 December 2018
DOI: 10.1002/cpe.5157

WILEY

RESEARCH ARTICLE

Parallel membership queries on very large scientific data sets using bitmap indexes

Beytullah Yildiz | Kesheng Wu | Suren Byna | Arie Shoshani

Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, California

Correspondence
Beytullah Yildiz, Computational Research Division, Lawrence Berkeley National Laboratory, Mail Stop 50B-3238, 1 Cyclotron Road, Berkeley, CA 94720.
Email: byildiz@lbl.gov

Funding information
TUBITAK BIDEB 2219; National Energy Research Scientific Computing Center (NERSC)

Summary

Many scientific applications produce very large amounts of data as advances in hardware fuel computing and experimental facilities. Managing and analyzing massive quantities of scientific data is challenging as data are often stored in specific formatted files, such as HDF5 and NetCDF, which do not offer appropriate search capabilities. In this research, we investigated a special class of search capability, called *membership query*, to identify whether queried elements of a set are members of an attribute. Attributes that naturally have classification values appear frequently in scientific domains such as category and object type as well as in daily life such as zip code and occupation. Because classification attribute values are discrete and require random data access, performing a membership query on a large scientific data set creates challenges. We applied bitmap indexing and parallelization to membership queries to overcome these challenges. Bitmap indexing provides high performance not only for low cardinality attributes but also for high cardinality attributes, such as floating-point variables, electric charge, or momentum in a particle physics data set, due to compression algorithms such as Word-Aligned Hybrid. We conducted experiments, in a highly parallelized environment, on data obtained from a particle accelerator model and a synthetic data set.

KEYWORDS

big data, bitmap index, data management, membership query, parallel query, scientific data

1 | INTRODUCTION

The volume of data produced by large-scale scientific experiments and simulations is growing massively. For example, experiments at the Large Hadron Collider (LHC)[1] at CERN in Switzerland, notably ATLAS[2] and CMS,[3] produce petabytes of high-energy collision data. The Square Kilometre Array (SKA),[4] which is a radio telescope consisting of several thousand antennas that operate as a single instrument, is designed to explore the earliest stages of the evolution of the universe. It will yield more than an exabyte of raw data per day when it becomes active. These are examples of scientific applications where the size of the data is increasingly getting larger and more complex over time. This brings many challenges for data analysis.

The volume and variety of data that must be analyzed and understood are the biggest challenges. Today, it is becoming commonplace for scientists to analyze multiple terabytes of data stored in thousands of files to generate a scientific result. This often involves weeks or even months using traditional analysis tools. Many of these tools used by the scientists were implemented decades ago when the volume and variety of data that scientists must now struggle with did not exist. Another major challenge today is data movement between computer components. Unfortunately, this challenge will become even more severe in the future. Performance improvements in data movement will continue to considerably lag the performance improvements in computing. Future systems are likely to continue a trend of noteworthy enhancements in total floating-point performance while the capability to move data both within a computing system and between a computing system and a storage subsystem will see much humbler developments.[5]

To address these challenges, it is necessary to improve query performance over these large data sets because the most common operation on scientific data is to select a subset of the data that contains valuable information for further analysis. The query improvement becomes more crucial for the values that commonly occur as discrete values, require random access, and have classifying characteristics. Attributes containing these characteristics such as zip code, location, occupation, and nationality commonly appear in commercial databases. Such attributes also occur frequently in scientific domains such as category of measured or calculated values and object types. Identifying whether a given set of values is a subset of attribute values, which is discrete in nature and has classification as its characteristics, is called a *membership query*. These queries have been mentioned in the literature for a couple of decades. Angluin defines a membership query to be a query that returns one bit of information: whether or not the queried element is a member of a known set.[6] Chan and Ioannidis state that a membership query is of the form $A \in \{v_1, v_2, \dots, v_k\}$, where A is a subset of the attribute values $\{v_1, v_2, \dots, v_k\}$.[7]

To improve query performance, indexing and parallelization are two key methods. For many years, indexing techniques have been used very effectively in database systems. However, indexing did not impact the area of scientific data management largely because scientific data are typically stored in formatted files rather than in database management systems.[6] Nevertheless, bitmap index technologies are very efficient when used on formatted files because the queries can be executed by using bitwise logical operations, which are very effective at the hardware level. In addition, since the queries can be performed in disjoint blocks of data, parallel execution can be applied successfully.

We experimented with membership queries using bitmap index technologies to analyze accelerated particles. Large-scale, high-resolution simulations of beam dynamics in an electron linear accelerator (LINAC) by IMPACT-T generated a very large data set of size 50 TB.[8] In the simulation, 1 billion particles were accelerated through 720 time steps. Some of the accelerated particles reached their final targets, although many particles were lost by escaping the beam. The scientist was interested in only a fraction of the particles, about 1000, that reached a high energy level at the final time step. The paths of these energetic particles required to be extracted from the original 1 billion particles in each of the time steps so that the reasons behind the acceleration to high energy were comprehended. Thus, for each time step, we searched the 1000 energetic particles out of 1 billion particles. If this were done in a naïve fashion by searching one by one, it would take a very long time. Our approach of performing a membership query using bitmap indexes accomplished the task of finding the desired particles much faster.

The first contribution of this research is the introduction of bitmap indexing to be used with membership queries to improve the query performance. Even though membership queries are not new, applying a membership query using a bitmap index to solve the problem very effectively is a novel approach. Our second contribution is to show how to scale indexing and querying in a highly parallelized environment. A membership query with bitmap indexing scales well to billions of elements by using our approach. Finally, the third contribution is to draw a roadmap for tuning membership queries; to the best of our knowledge, membership queries have not been evaluated by taking into account the features of data such as cardinality, order of data, and skewness.

The remainder of this paper is organized as follows. In Section 2, we provide background information on bitmap indexing techniques and discuss related work. The methodology of our membership query experiments is elaborated in Section 3. We present the experimental results in Section 4 and conclude the discussion in Section 5.

2 | BACKGROUND AND RELATED WORK

Database Management Systems (DBMS) are generally designed for transactions to provide quick access for business processes, known as Online Transaction Processing (OLTP). It is optimized to maximize the speed and efficiency in an environment in which data are updated frequently. Data warehouse systems are often used for reporting and data analysis.[9] They store large amounts of data that grow continuously but do not change frequently. Data warehouse systems enable query evaluation by using techniques referred to as Online Analytical Processing (OLAP), optimized to handle complex queries on aggregated large historical data sets. While searching and updating are performed with nearly the same frequencies in OLTP, the searching operations in OLAP are typically performed with a much higher frequency than that of updating operations.[10]

Database systems traditionally store data in a row-oriented manner, which is well suited for transactions. This method is not suitable for data warehouse applications where queries only touch a few columns but typically scan many records. Therefore, a column-oriented approach for data warehouses has been proposed as an alternative to the row-oriented approach. In a column-oriented database, all instances of a single data attribute are stored together. Therefore, column-oriented databases are more efficient for the analytical processing since queries must read all instances of a small number of data attributes.[11]

The column-oriented approach reduces the size of the data processed by a query because only the needed columns are retrieved. Moreover, the column-oriented data can often be compressed efficiently. Because of these advantages, SQL Server, a general-purpose database system, added a column-oriented approach to its system to improve performance for data warehouses.[12] MonetDB designed primarily for data warehouse applications achieves significant speedup compared to more traditional designs by a storage model using a column-oriented approach.[13]

In addition to the column-oriented approach, bitmap indexing is recommended to improve performance for data warehouses. Stockinger et al summarized the experience of implementing a series of bitmap index schemes on vertically partitioned data sets.[14] Vertical partitioning enables highly efficient compressed bitmap indexing that produces indexes smaller than traditional indexes. For a compressed bitmap index, the I/O cost does not dominate the total query processing time. When the number of attributes in a data set increases, the number of possible indexing combinations, often called the curse of dimensionality, increases. Bitmap indexing technology was shown to break the curse of dimensionality for data warehousing applications.[15] Moreover, the bitmap index has the best balance between searching and updating for OLAP operations.

Sorting can be applied to existing bitmap compression schemes based on run-length encoding to improve query performance in data storage and decision support systems. Sorting the attribute before a bitmap index is created enables reducing the space requirement for the bitmap index and the response time for queries. The advantage of this technique is that this can be accomplished without any additional workloads for the data warehouse.[16]

In addition to the column-oriented approach and bitmap indexing, new indexing techniques, such as the storage index, have been proposed to handle queries faster. The storage index is an in-memory approach that keeps information about the data within the specified regions of the physical storage space.[17] This index contains information on whether the disk does or does not contain the values for the query attributes of interest so that these attributes are not retrieved during the scan. This significantly reduces unnecessary I/O by excluding irrelevant database blocks in the storage cells.

Similar to data warehouses, scientific data are usually written once and read several times and are often stored in certain file formats such as HDF5[18] and NetCDF.[19] In general, these specially formatted files do not provide an indexing mechanism. However, it has been shown that bitmap indexes can be used effectively to provide support for querying on these file formats without any modification to the files.[20]

Hardware-efficient bitwise logical operations are the key feature that bitmap indexes take advantage of. Additionally, the results of bitwise logical operations can be effectively combined. A bitmap vector can be defined as

$$\forall v_i \in \mathrm{M} : B_i = \begin{cases} 1, & \text{if } v_i \in \mathrm{D} \\ 0, & \text{if } v_i \notin \mathrm{D}, \end{cases}$$

where M is a set of query values and D is the data set. Since the bitwise logical operations on bitmaps create bitmap vectors, the results of the logical operations can be processed by applying bitwise logical operations as well.

The main deficiency of the bitmap index is that the size of the index grows linearly with the number of distinct values, referred to as the cardinality of the data. The bitmap index is very efficient for a low-cardinality data set. However, scientific data typically contain floating-point values, which may result in very high cardinality. For example, the temperature value from a combustion simulation would have very high cardinality because each instance of the value could be different from the next. Similarly, particles of a high-energy physics experiment contain multiple properties with floating-point values resulting in high cardinality. Therefore, especially for the high cardinality data, there are procedures that must be used to control the size of the index. Binning, encoding, and compression are the techniques that alleviate the growing bitmap index size issue.

2.1 | Techniques to create a bitmap index

2.1.1 | Binning

Binning produces a set of identifiers for bitmap index construction. In other words, an attribute value of data is usually divided into an arbitrary number of bins. The most basic binning strategy is no-binning, which has a single distinct value for each bin. However, a common binning strategy creates fewer bins than the number of distinct values. The main advantage of binning is that it is the most effective approach to reduce the index size. However, the index may not fully resolve all the queries when binning is used. Therefore, all values falling into a bin have to be scanned to answer a query correctly. This procedure is called a candidate check, which may dominate the total query time.[21] Therefore, applying binning requires a strategy with a balance between the size of the index and the overhead.

2.1.2 | Encoding

Bitmap encoding techniques create bitmaps in a way that reduces either the total number of bitmaps or the number of bitmaps needed to answer a query. That is to say, an encoding may target to control the size of an index or it may aim to reduce the necessary number of accesses to the bitmap index. There are three types of basic bitmap encoding methods: equality encoding, range encoding, and interval encoding.

Equality encoding is the most fundamental bitmap encoding. Usually, a bit for an attribute value is appended to a bitmap vector, which contains as many bits as the number of rows. Equality encoding is known as a very efficient method for equality queries because the query needs to look at only one bitmap. The second well-known encoding technique is range encoding, optimized for a one-sided query such as "$\mathrm{A} \le 1$." The size of the bitmap index with range encoding for an attribute is $\sum_{i=1}^{c}(n_i - 1)$, which is less than equality encoding. A range query takes, at most, one bitmap processing for a range query. The third most common technique is interval encoding, developed to process efficiently two-sided queries.[7] This encoding scheme is based on range encoding and is well suited for two-sided range queries such as "$2 \le \mathrm{A} \le 5$." Interval encoding guaranties performing any interval queries by accessing, at most, two bitmaps. Interval encoding is the most space efficient among the three basic schemes, which requires almost only half the number of bitmaps of the other two schemes. Chan and Ioannidis discuss the optimality of the encoding schemes.[7]

Each bitmap index is basically a representation of an attribute using some number of bits in a way that it is specific for an encoding scheme. This observation has resulted in new encoding schemes to reduce the index size and/or to improve the query performance. Almost all encoding methods proposed in the last decade can be classified as either multicomponent encoding or multilevel encoding. In multicomponent encoding, the attribute values are partitioned into several components. Each component is encoded by using one of the basic encoding schemes. In an earlier study by Chan and Ioannidis,[22] the optimal number of components is claimed to be 2. However, the optimal number may be different when the indexes are compressed.[23] Multilevel encoding is an encoding that progressively generates bins in a hierarchical manner.[24,25] Each level can be encoded by using one of the three basic encoding schemes. The finest level can have a distinct value for each bin. A higher level spans multiple values per bin and, therefore, has fewer bins. In contrast to conventional binning, multilevel encoding requires a candidate check when the query does not fall on the higher-level bin boundaries. Although the finest level can always answer any query, the coarse levels can be used to answer a query without accessing the finer levels when the query spans entire bins of the higher levels. This strategy results in reducing the amount of work for a query.

2.1.3 | Compression

The equality bitmap encoding scheme generates one bitmap for each distinct value of an attribute. This may generate a bitmap index whose size is larger than the size of the data itself. In other words, the size of the index may even surpass the data space in cases of high cardinality data. This would make searching of the index too expensive. This challenge is especially true for the scientific data because the cardinality of scientific attributes is usually very high. However, the bitmaps, which mostly contain zero bits, are very sparse and are therefore suitable for compression.

A number of algorithms have been proposed for bitmap compression. Johnson compares the well-known LZ text compression algorithm with variable bit-length codes and variable byte-length codes.[26] The text compression algorithms are effective in reducing the text size. Similarly, bitmap compression reduces the index size. However, performing logical operations on compressed bitmaps is usually slower than on uncompressed bitmaps because the compressed bitmaps have to be uncompressed before applying any operation on them. The logical operations using the specialized schemes, developed specifically for bitmap index compression, are usually faster than those using the text compression algorithms. One of the specialized algorithms for bitmap index compression is the Byte-aligned Bitmap Code (BBC).[27] BBC is known to be efficient and used in several commercial database systems. However, the logical operations on the compressed data can be still slower than the operations on the uncompressed data, especially for high cardinality attributes. To address this issue, Word-Aligned Hybrid (WAH) compression was proposed to improve the speed of logical operations at a cost of small increase in space.[28,29] It is designed to perform logical operations directly on the compressed bitmaps without decompressing them first. WAH compression supports faster logical operations and makes the bitmap index applicable to the data with high cardinality.[30] Contrary to the common perception, a smaller compressed bitmap index is not necessarily more efficient. WAH-compressed bitmaps are about 30% larger than BBC-compressed bitmaps in size, but logical operations on WAH-compressed bitmaps are significantly faster than on BBC-compressed bitmaps, as demonstrated in the work of Stockinger et al.[14] The reason is that WAH operates on words rather than bytes or bits, and it can perform logical operations directly on the compressed bitmaps. In addition to BBC and WAH, new compression schemes are proposed, such as the Position List Word-Aligned Hybrid (PLWAH),[31] the Enhanced Word-Aligned Hybrid (EWAH),[32] and the Partitioned Word-Aligned Hybrid (PWAH).[33] The common feature of these schemes is being a variant of WAH.[34]

2.2 | Related work

There are publications investigating parallel index and query operations. Byna et al presented parallel I/O, analysis, and visualization for the VPIC data, which were generated by a state-of-the-art plasma physics code that simulated trillion of particles running on 120 000 cores.[35] Hybrid parallel query using multicore CPUs on distributed memory hardware was applied to index and query the trillion-particle data set. In this study, range queries were explored by applying them to VPIC data. Chou et al also investigated range queries and range encoding schemes.[36] The queries were performed on a single attribute by utilizing 48 MPI processes. Kim et al explored parallel in situ indexing for intensive computing.[37] The in situ data processing approach is intended to bypass disk accesses whenever possible. The basic idea is to perform a series of predetermined operations while the data are in memory. In situ processing was adapted to generate the index in a parallel manner so that the necessity of reading data from disks was avoided. Consequently, it was shown that the generated index can improve the data access for 3 to 200 times depending on the fraction of the data selected. Su et al proposed another parallel approach for indexing and querying.[38] In addition to queries over multiple dimensions, the study explored queries over coordinate variables by using and optimizing multilevel bitmap indexing. Additionally, a parallel indexing architecture and an intelligent index partitioning strategy were proposed to improve the query performance. Workflow is one of the key features for better performance.[39-41]

In addition to parallel strategies, sorting has been investigated in several studies to improve efficiency. Lemire et al explored sorting for the bitmap index using WAH compression.[32] Pinar et al[42] and Sharma and Goyal[43] used row sorting to improve run-length encoding (RLE) and WAH compression. Although these studies showed an improvement for the bitmap compression, the largest bitmap index used in these research studies fits in a personal computer memory. The basic idea is that reordering rows of a compressed bitmap index can reduce the index size. The primary attribute is selected, and an uncompressed index is created. The rows are then rearranged so that the values are consecutive. This creates longer sequences of 1's and 0's. When compression is applied, the index size becomes smaller.[44] This may not be feasible for a large number of attributes and rows. A more practical approach is to reorder the base data and then to construct the compressed index directly.

For equality and membership queries, Vanichayobon et al proposed the Scatter Bitmap Index, which used less space than the other indexes while maintaining a competitive query processing time.[45] In this indexing method, each attribute value was represented by using only two bitmap vectors, but each bitmap vector symbolized many attribute values. Weahama presented an improvement for membership queries of the Virtual Classroom System by using the Dual Bitmap Index, also called the Scatter Bitmap Index, and applying data clustering.[46] The method was based on clustering the data before indexing by using some defined distance measure, which determined the similarity or proximity of the data elements. In these studies, the data sets were small enough to fit into a single desktop computer memory. Therefore, the proposed methods may not be appropriate for the high volume data. Moreover, the data features are not easily adaptable to speed the query up, such as cardinality, skewness, and order of the data. In addition to these studies that use an index, solutions using *bloom filter* have been proposed for the membership queries, where *false positive* matches are possible, but *false negatives* are not.[47,48] A query returns either *possibly in a set* or *definitely not in the set*. In other words, the bloom filter approach provides an approximate answer for a membership query with a reasonable performance.

3 | EXPERIMENTAL METHODOLOGY

In the experiments, we compared predicate queries with membership queries using the bitmap index. Additionally, we investigated how to get the best outcome from the features of data, such as cardinality, order of data, and skewness. Moreover, scalability was studied by using massive data sets and utilizing a large number of MPI processes.

3.1 | Testbed

We conducted our experiments on the NERSC* Cray XE6 supercomputing system, named Hopper. Hopper is a petaflop system, with a peak performance of 1.28 petaflops/s, 153 216 compute cores, 212 TB of memory, and 2 PB of the disk. The system has 6500 compute nodes, with 24 cores per node and 32-GB memory per node.

The file system used by Hopper is the Lustre parallel file system that consists of 13 LSI 7900 disk controllers. Each disk controller has two I/O servers, called Object Storage Servers (OSS), and each OSS hosts six Object Storage Targets (OSTs). Hence, there is a total of 156 OSTs, which can be considered as a software abstraction of physical disks. Lustre provides I/O parallelism by striping data across the multiple disks (OSTs). A striping is organized by parameters such as strip count and stripe size. Stripe count is the number of OSTs to use to store a file, and strip size is the number of bytes written on an OST. The larger number of stripe count offers better parallelism because the data can be read/written from more disks at the same time. However, when the larger number of stripe count is used, the likelihood of I/O contention increases with other applications.

Hopper uses the Cray Message Passing Toolkit (currently MPT/3.3), which has a Lustre-aware implementation of the MPI-IO collective buffering algorithm. It enables data to be buffered on aggregator nodes and to be stripped into Lustre stripe-sized chunks to reduce the number of I/O writers or readers. Hence, all reads and writes to the Lustre are automatically stripe size aligned. Because of the way that Lustre is designed, alignment is a vital element in ensuring good I/O performance.

3.2 | Data sets

In our experiments, we used a data set generated by large-scale, high-resolution simulations of beam dynamics in an electron LINAC by IMPACT-T.[8] The data set had 720 time steps with 1 billion particles per time step. Each time step was stored in a single HDF5 file containing the properties of each particle in nine variables. Each variable was kept in a one-dimensional array. The properties were the identification number (ID), charge-to-mass ratio of each particle (qom), the ratio of the total charge (in Coulomb) of a type of particle to the total number of macroparticles (chgpt), and the coordinates of the particles. The size of each file was approximately 68 GB, and the total size of the whole data set was approximately 50 TB.

To look at the behavior of membership queries in more detail, we also experimented on synthetic data, which were created to fill the gap of the missing features in the LINAC data set. The synthetic data with different cardinalities were formed to investigate the effect of the proportion of distinct values in a data set to the index size and the query performance. We also performed our experiments for highly skewed data, which is typical for many quantitative attributes in scientific applications. We also used a huge data set containing 10 billion rows to investigate the scalability of parallel membership queries.

3.3 | Software components

The data sets were stored in HDF5 files. We used FastQuery to create the bitmap indexes and accelerate membership queries. We describe both next.

3.3.1 | HDF5 data format

The Hierarchical Data Format v5 (HDF5)[18] implements a model to manage and store data. It includes an abstract data model, an abstract storage model, and a library to implement the abstract model. The HDF5 library stores data in machine-independent and self-describing binary files organized for high-performance access. The HDF5 library generates portable files by using encapsulation so that the data and metadata are combined together in a container. With the HDF5 model, the layout of bytes in a file is not a concern for users; instead, the users can deal with the high-level concepts of relationships between the data objects.

* NERSC is the National Energy Research Scientific Computing Center at Lawrence Berkeley National Laboratory, California.

In the experiments, we utilized a parallel HDF5 library, which is designed to work on supercomputers. The parallel implementation of HDF5 relies on Message Passing Interface (MPI) implementation for I/O, called MPI-IO. HDF5 especially relies on communication, synchronization, and collective I/O operations of MPI. HDF5 can use either the MPI-IO routines for collective or independent I/O operations or a mixture of POSIX file I/O operations and MPI communications.

3.3.2 | FastQuery

FastQuery is a parallel querying library that uses FastBit bitmap indexing and supports querying on multiple scientific data formats such as NetCDF and HDF5.[20,49] FastBit, developed at Lawrence Berkeley National Laboratory (LBNL), is an open-source software that implements many of the encoding scheme, binning, and compression strategies. Basic encoding schemes like equality, range, and interval encoding are supported. It also uses multilevel encoding methods that improve the query efficiency while maintaining theoretical space-time optimality. FastBit uses the WAH compression method that makes it possible for bitmap indexes to answer queries in optimal computational complexity. Moreover, a set of binning strategies that can suit diverse needs is also used to enhance the efficiency of query processing.[50] Its successful use in several scientific applications was presented in the work of Wu et al.[51]

FastQuery benefits from parallelism for both computation and I/O. As illustrated in Figure 1, it divides a data set into various fixed-size subarrays in order to use the distributed memory nodes and multiple cores of a computing system. The indexes are iteratively built from the subarrays by assigning each subarray to an MPI process running on a CPU core. The generated indexes for each subarray are collected together and stored into a single index file. Similarly, the queries are performed in a parallel manner. An index file is evaluated chunk by chunk. Each chunk is handled by an individual MPI process.

FastQuery performs I/O operations by using the HDF5 library that handles the logical view of a file. FastQuery's parallel I/O relies on the parallel HDF5 framework. The HDF5 library delivers I/O requests through the MPI-IO layer that enables multiple MPI processes to run on a single file in a parallel manner. At the lowest level, the Lustre parallel file system controls the I/O requests and performs the actual data read/write operations in parallel.

3.4 | Membership queries and measurements

We applied membership queries to find specific energetic particles in the IMPACT-T simulation data set and synthetic data sets. Moreover, 1000 particles were queried for each time step. The number of particles is reasonable to come to a conclusion for the behavior of the particles that reach the final target. A sample script executing 48 MPI processes to query the accelerated particles is shown in the snippet below. This script format was used for all queries on the IMPACT-T simulation data set and the synthetic data set. The executable batch file requires the path of

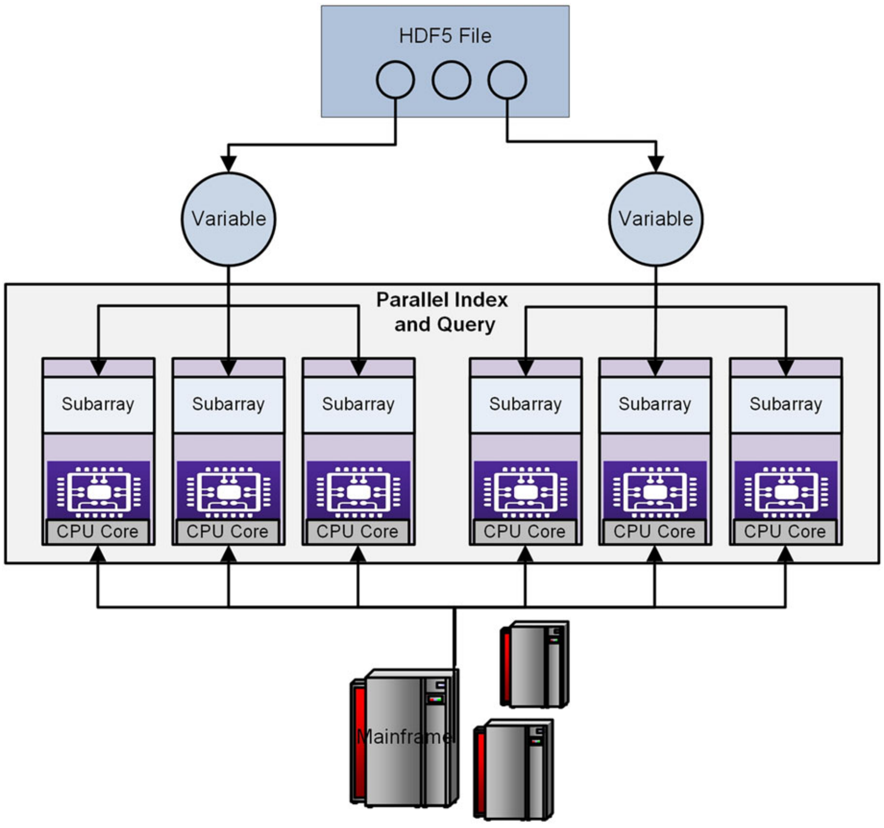

FIGURE 1 Parallel indexing and querying using FastQuery

a base data file, the path of an index file, and the query containing a thousand values as parameters. An index file path is optional, whereas the remaining two parameters are necessary for query execution.

```
#PBS -q regular
#PBS -l mppwidth = 96
#PBS -l walltime = 00:30:00
#PBS -N my_job
#PBS -e my_job.$PBS_JOBID.err
#PBS -o my_job.$PBS_JOBID.out
cd $PBS_O_WORKDIR
aprun -n 48 -N 12 -S 3./QueryIndex -f./basedata.h5 -i./index.h5 -n ID -q "ID IN
(490027,3368690,2520059,7513926,5180540,4089172,3455736,5005211,1595368,
8413784,6898537,4575198,3594324,8664370,9566413,1759956, .........................)"
```

The execution of a query starts with reading the index file in chunks, called subarrays. Each subarray is assigned to an MPI process using its own CPU core. A query using the index returns the coordinates of matched values. Instead of scanning the whole data, these coordinates are used to retrieve the values from the base data. In other words, only the matched values in the base data are accessed. The base data are not scanned except for the queries that use an index created with a binning strategy. A query may require the scanning part of the base data in case that the query does not fall on a bin boundary. When the queried values are not found within the bin boundaries, each value in the base data needs to be compared with the queried values to check whether there is a match. In contrast, the file containing the base data has to be scanned if an index is not present. The values are read from the base data in subarrays. Each row in a subarray is scanned to check whether there is a match. The results from each MPI process are gathered to provide the final collective output.

We preferred equality encoding to create indexes. It is the optimal encoding for membership queries, as discussed in the work of Chan and Ioannidis.[7] As mentioned previously, equality encoding contains many zeros that can be effectively compressed with the WAH compression method.[28]

To efficiently answer queries involving multiple attributes, one option is to select a combination of attributes to generate a multidimensional index tailored for a specific query. For additional queries, other combinations may be necessary. Thus, this approach is only practical if we are only interested in a specific query. Another option is to create a separate index for each attribute. To answer a query that includes conditions with multiple attributes, we first resolve the conditions on each attribute, and we get a solution as a bitmap for each condition. Then, we combine these bitmaps and get the answer to the overall query.[15] For instance, to handle the query "A = 3 and B = 10," a bitmap index of attribute A and a bitmap index of attribute B are used separately to create two bitmaps that satisfy the conditions. The final answer is the result of the bitwise logical AND operation on these two bitmaps. The logical operations on bitmaps are well supported by computer hardware; hence, bitmaps can be combined easily and efficiently. The total query response time scales linearly with the number of attributes involved in the query, not with the number of attributes of the data set. Moreover, the selectivity and attribute access order make no difference for multidimensional queries when using bitmap indexes.

Sorting does not complicate multidimensional queries while providing a performance gain. Sorting is applied to the bitmap indexes to reduce the index size and to improve query performance. There are several approaches for sorting. The first approach is to reorder the compressed bitmap itself. However, this requires very intensive computation. The second approach is to rearrange the rows of an uncompressed index so that longer sequences of 1's and 0's are created. Then, compression is applied to create a smaller index.[42,43] This may not be suitable for a large number of dimensions. The practical approach we apply is to reorder the base data and then create the compressed index. Only the attributes for which bitmaps will be created are sorted. These extra sorting tasks on attributes can be easily done before indexing.

To ensure that the query response time contained the full disk I/O cost and that our measurements were not affected by previous caching, we executed each experiment using a new copy of a file. Consequently, the operating system had no information about the specific directory containing the files or any information about the files. Otherwise, the operating system would have cached the directory information about the files and the IO time might have been significantly reduced if the files had been accessed previously.

Tuning several parameters for the underlying parallel file system provides better I/O performance.[52] Therefore, we used 1 MB stripe size in the Lustre parallel file system and set the stripe count to be 48. In other words, the file was distributed on 48 OSTs in 1-MB–sized chunks. For the processing of queries, we used 48 MPI processes to handle a 1-billion–row data set from the LINAC simulations and the synthetic data. To investigate scalability, the number of MPI processes was increased step by step from 24 to 3072 for the 10-billion–row data set. In each step, the number of MPI processes was doubled to collect the scalability measure. We used 24 for the scaling factor because each Hopper computing node has 24 CPU cores.

4 | EXPERIMENTAL RESULTS

4.1 | Naïve query processing versus queries with bitmap indexes

The naïve query processing we used is the conventional way of searching values from a data set without using an index. The whole IMPACT-T simulation data set was scanned for each energetic particle that reached the final target. Even though the data were loaded once, the performance result of the naïve query processing was not a good strategy because it required sequentially scanning the whole data set as many times as the number of energetic particles. This was the motivation to apply membership queries using bitmap indexes to overcome the problem. Figure 2 shows the log-scale performance result comparing the naïve query processing with parallel queries using bitmap indexes in a single time step. Using a bitmap index expedites the query by more than 300 times. When the data are sorted, the improvement becomes even better. While sorting

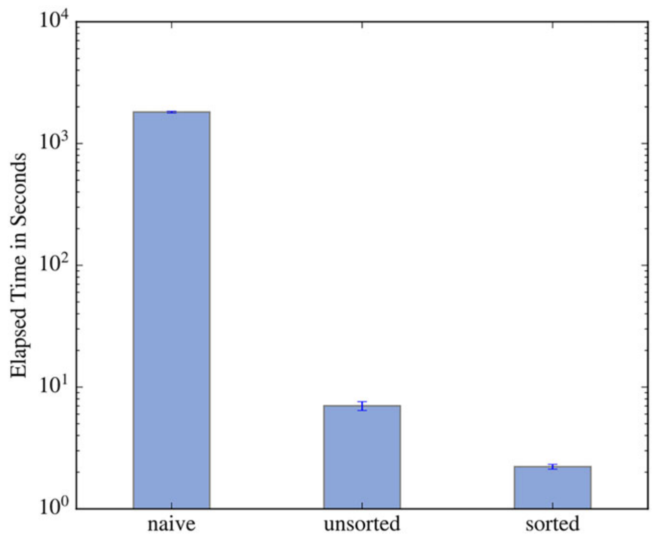

FIGURE 2 Query performance result comparing the naïve query processing with membership queries using bitmap indexes. One thousand particles that reached the final target are searched in the 1-billion–row data set of IMPACT-T simulation

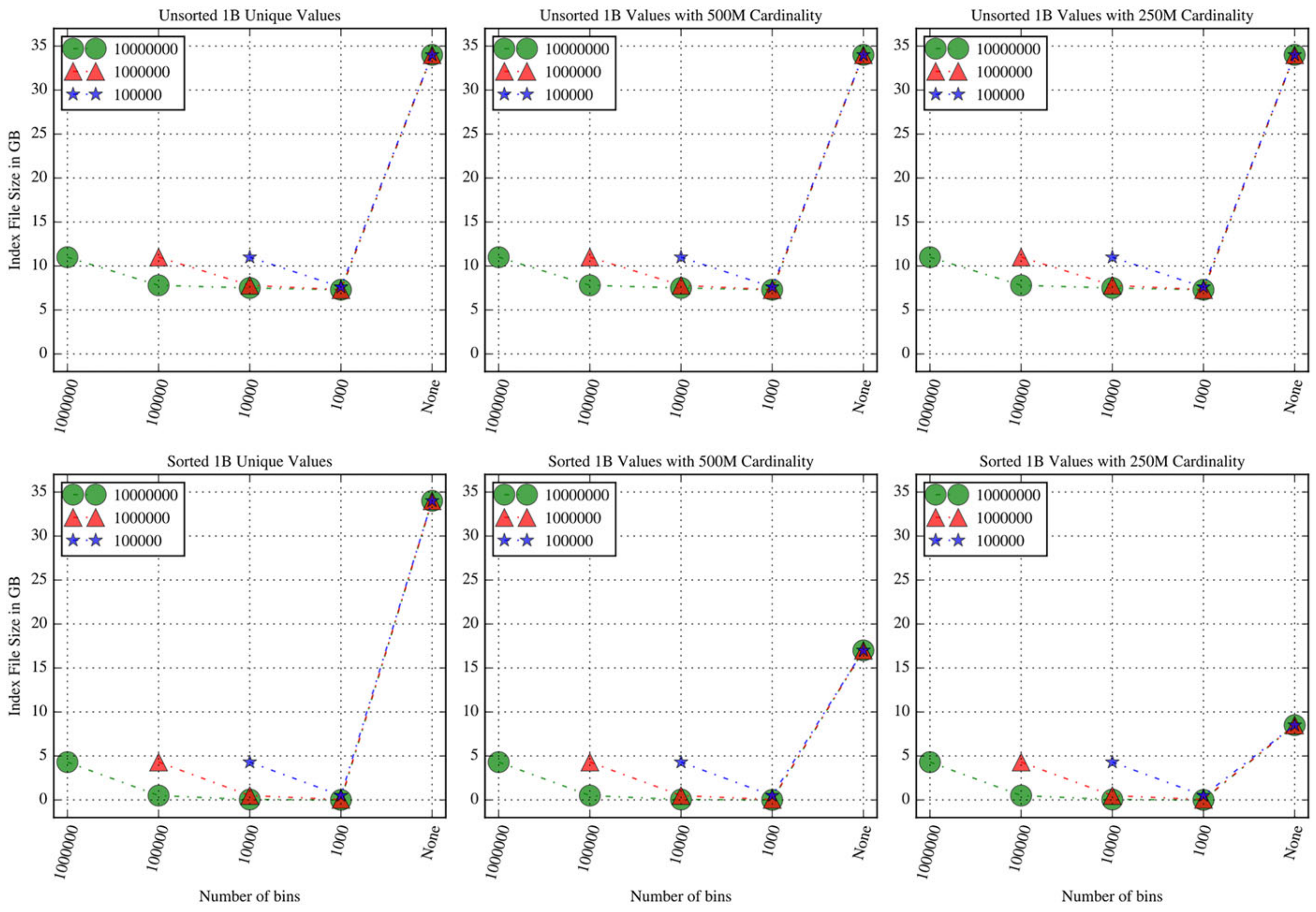

FIGURE 3 The size of index files for varying subarray size, bin size, and cardinality. The colored values are for the subarray sizes (green circles for 10 million, red triangles for 1 million, and blue stars for 100 000)

expedites the query execution for membership queries using the bitmap index, the naïve query processing does not benefit from sorting because of the necessity of repeatedly scanning the whole data. Hence, we show a single bar for the naïve fashion queries of the sorted and unsorted data.

The execution of the naïve query processing took more than 2 weeks in order to discover the behavior of 1000 particles for 720 time steps. Moreover, increasing the number of particles that reach the final target causes a linear increase in the execution time. This is an important deficiency comparing with the queries using the bitmap indexes. In contrast, the number of queried particles does not have a real impact on the execution time when a bitmap index is used for the membership query.

4.2 | Optimizing indexed query via data features

We investigated the data features, such as cardinality, the order of the data, and skewness, to determine their effect on query processing and to use them for performance improvement. First, because of the I/O cost, we looked at the index size for varying data features, shown in Figure 3. The top three plots show the index size for unsorted data, whereas the bottom three plots show the index size for sorted data. The Figure shows how the index size changes for varying values of binning and subarray size as well as for various data cardinalities.

We notice that sorting reduces the index size except in the case of unique values. Also, one can see that the index size significantly shrinks while the cardinality of the sorted data is getting smaller. Conversely, cardinality has no effect on the index size when the data are not sorted. This is reasonable because the values in unsorted data may not be sufficiently clustered to affect the index size when bitmap compression is applied. Binning has one of the greatest impacts on the index size. On the contrary, the subarray size has a relatively small effect on the index size. The index size is larger for the same binning option if the subarray size is smaller.

Figure 4 illustrates the effect of data skewness on the index size while keeping varying the other data features. The data set has 1 billion rows and 100 million unique values. The left-hand side plots have uniformly distributed data values, whereas the right-hand side plots have skewed data values. The uniformly distributed data display the same pattern as the plots shown in Figure 3. The skewed data show a better space performance than the uniformly distributed data. We realize that the index size shrinks well when the data are skewed even if the data are not sorted. However, sorting skewed data is still more appealing because the index size either with binning or without binning decreases dramatically.

Additional attribute values almost double the size of the bitmap index when introducing parallel indexing. To keep track of the values in different subarrays, the attribute values such as *bitmap offsets* and *bitmap keys* need to be stored in the index file. This is an acceptable space

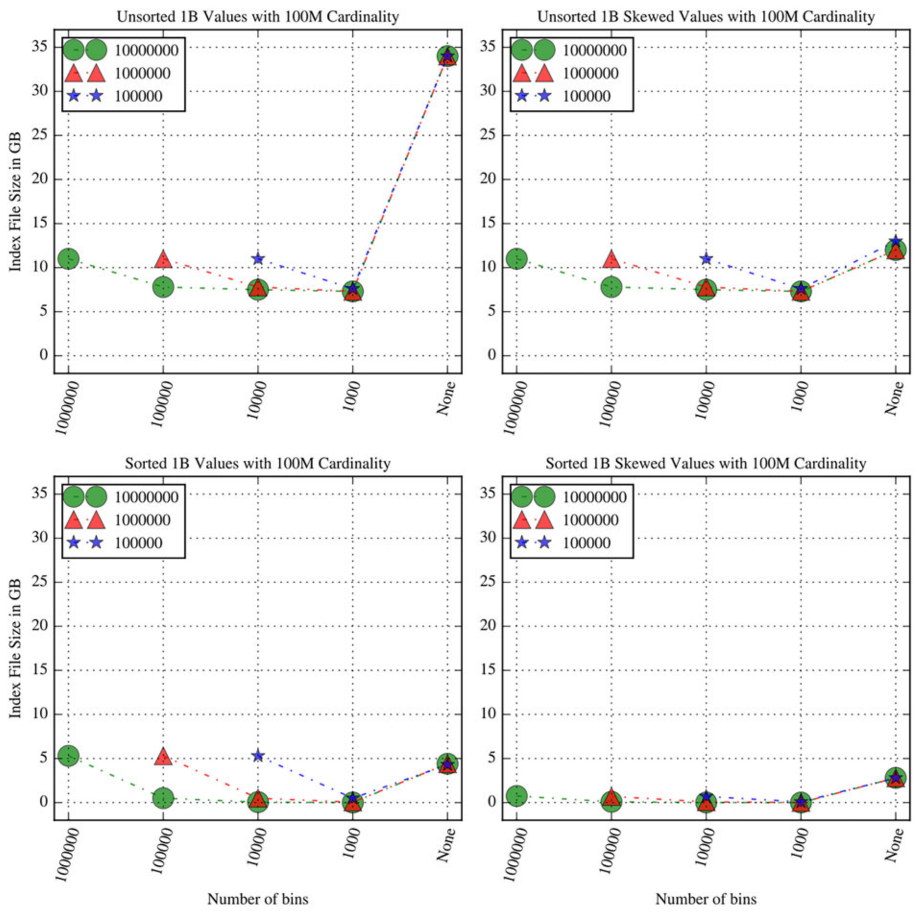

FIGURE 4 The size of index files when data are skewed for low cardinality. The colored values are for the subarray sizes (green circles for 10 million, red triangles for 1 million, and blue stars for 100 000)

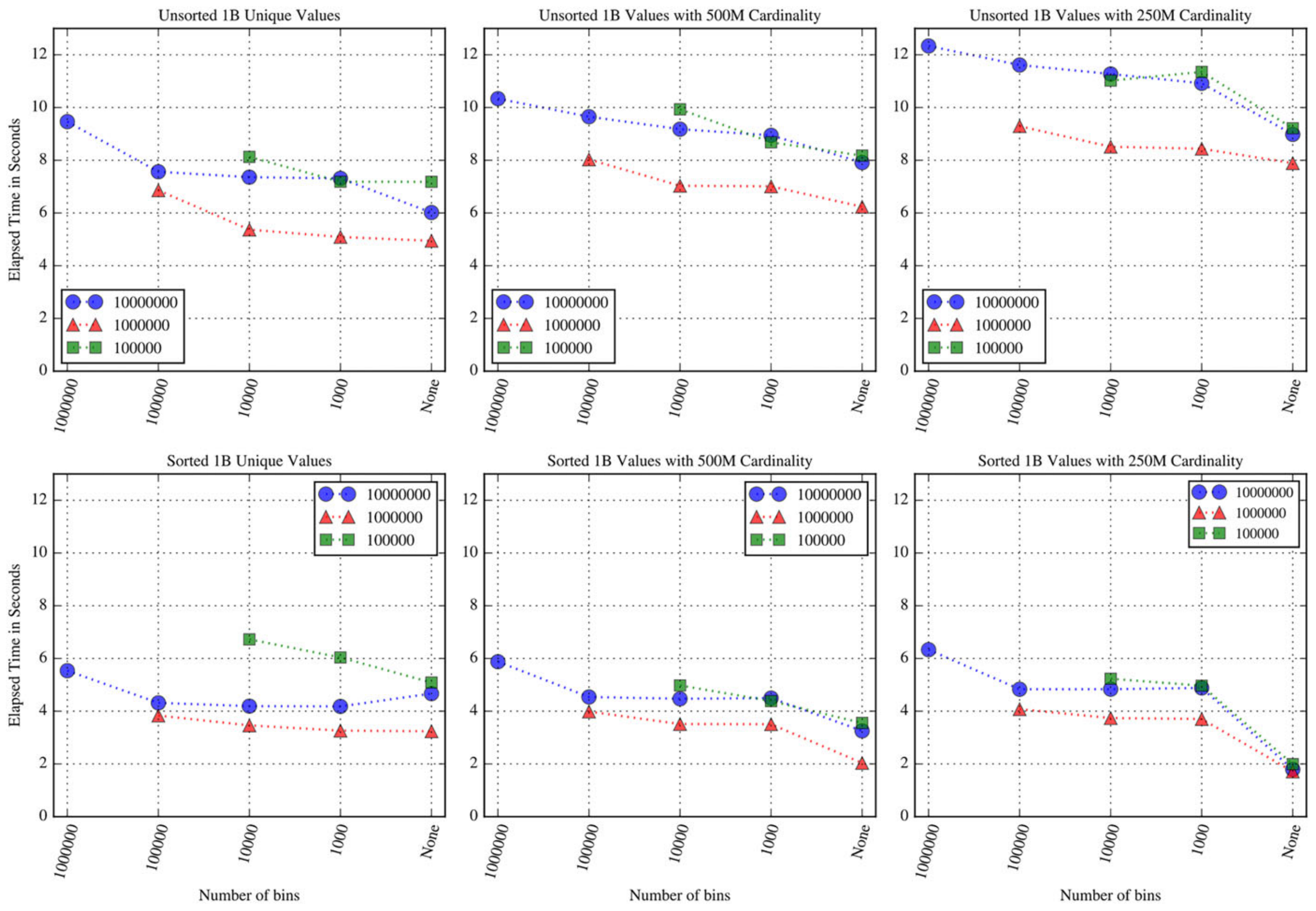

FIGURE 5 Query performance by using bitmap indexes created with varying subarray sizes, binning options, and cardinality. The colored values are for the subarray sizes (blue circles for 10 million, red triangles for 1 million, and green squares for 100 000)

and time trade-off in order to introduce parallelism. Otherwise, we would not have exploited the performance improvement by using parallel execution on the disjoint data blocks (subarrays).

After collecting the results for the index size, we explored how to optimize the query processing by tuning the data features, such as the order of the data, cardinality, and skewness. The index size is relevant to the I/O operation since it dominates the overall processing time in many cases. Hence, the index files in the Figures above were utilized to examine the effect of the I/O cost. Figure 5 shows the query performance result while using bitmap indexes.

We realize that, on the one hand, choosing a small value for the subarray size does not contribute to an efficient parallel execution. On the other hand, the performance degrades when the subarray size exceeds a threshold value. The optimal subarray size is 1 million for our experimental settings. Using the subarray size of 10 million causes inefficient memory usage and I/O utilization.

Sorting base data reduces the query time more than 3 folds comparing with unsorted data. The gain is even higher for smaller cardinality. A smaller cardinality degrades the query performance for unsorted data because the number of the retrieved values increases; fetching more values, actually more subarrays, from base data takes more time. On the other hand, we do not see this behavior in the sorted data because the retrieved values are stored within a smaller number of subarrays. Either for a single value or multiple values, the necessary subarrays are retrieved once. Therefore, with smaller cardinality, the query takes less time for sorted data.

We finally measured the effect of skewness on query performance because many data sets are naturally skewed in scientific applications. Figure 6 demonstrates the impact of data skewness on the performance. The queries using an index with various binning options perform closely to each other. Hence, binning does not contribute to the performance of skewed data. Furthermore, we see an almost 2-fold gain when using an unbinned index for the skewed data. The gain originates from the clustering of the queried attribute values. The best query performance in the 1-billion–row data set is observed when data are skewed, sorted, and have low cardinality; the execution is completed in less than a second. Skewed sorted data perform better because each subarray is optimally aligned with the queried attribute values. The index size is also smaller because the skewed and sorted data are well compressed.

The experimental results clearly illustrate that a membership query does not benefit from binning. Indexes without binning perform better, and using a binned index does not improve the query performance even though it reduces the space requirement. This observation reveals that the cost of a candidate check is high for a membership query. For different queries and settings, there are a number of approaches to reduce the

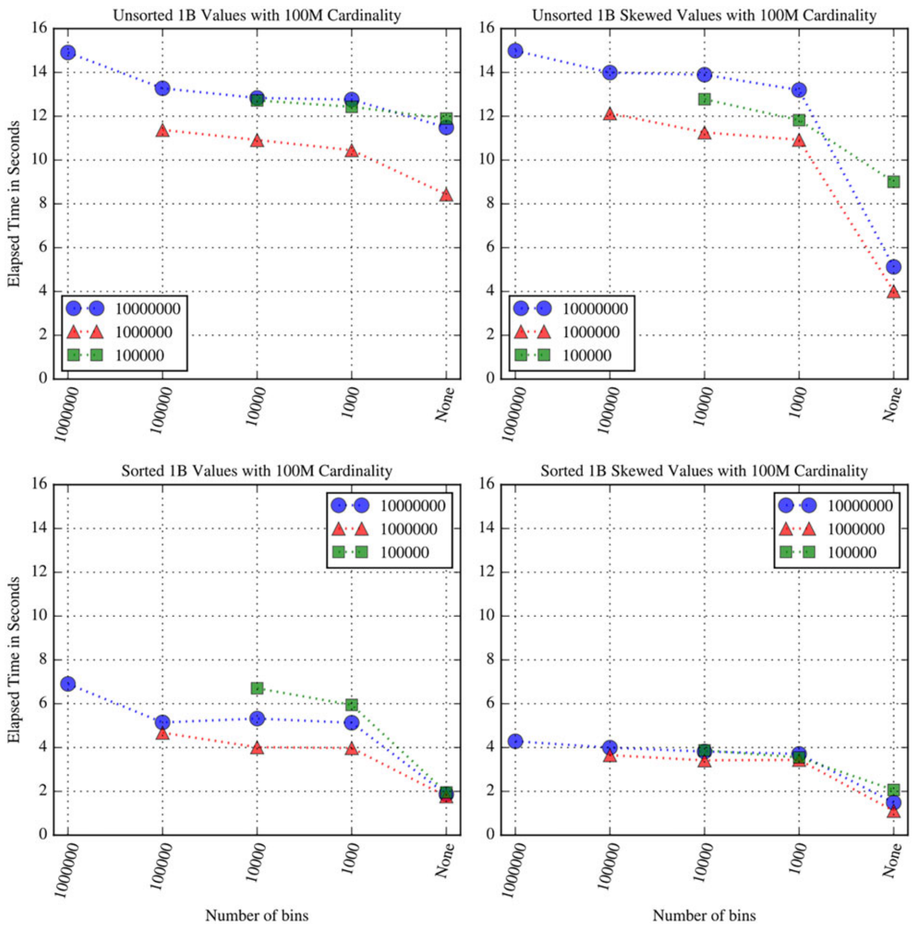

FIGURE 6 Query performance for skewed data and low cardinality. The colored values are for the subarray sizes (blue circles for 10 million, red triangles for 1 million, and green squares for 100 000)

time required for the candidate check. Rotem et al discussed the challenges of finding the optimal binning for range queries.[21] Koudas explored the optimal binning for a given set of equality queries and introduced a space-efficient indexing for the equality query by jointly encoding the attribute values using the frequency of access and occurrence of the attribute values.[53] Stockinger et al investigated the optimization of multidimensional range queries for relatively small sets of known queries by using additional operations on bitmaps to reduce the number of checks for the candidates.[54] We did not benefit from these approaches because the data in our study had comparatively high cardinality.

The conventional wisdom regarding the index size appears to be wrong for compressed bitmap indexes; the smaller index may not necessarily perform better. We observe that the index size for the 1 million subarray size without binning is larger than the 10 million subarray size without binning. However, the query using the index with 1 million subarray size performs better. Hence, the index size is not the main contributor to the query processing time. Additionally, the largest index file, which is slightly less than 35 GB, can be theoretically retrieved in a second because the data transfer rate of the underlying Lustre parallel file system is 35 GB/s. Consequently, we clearly have to take into account CPU efficiency as well as I/O efficiency.

4.3 | Scalability

We conducted experiments to increase further scalability for membership queries by using a very large data set, which had 10 billion rows. The same membership queries containing 1000 values as the queries of the previous experiments were applied. Only 1 million and 10 million subarray sizes were used to generate the indexes because the 100 000 subarray size had shown its inefficiency in previous experiments.

Figure 7 shows the index file sizes for the data containing 10 billion unique values and 10 billion values with 2.43 billion cardinality. For the data containing only unique values, sorting the base data does not have a real impact on the index size when binning is not applied. However, sorting reduces the index size when binning is used. For fewer distinct values in the data set, the index file becomes smaller. The index size reduces further when the data are sorted and have smaller cardinality.

We encountered a sparse file problem while creating the indexes for this very large data set. The index file without binning for the unsorted data became huge, especially for 1 million subarray size. The problem occurred because the subarray size was not ideal for the underlying chunking method of the HDF5 file creation framework. The size was almost reduced to half after repacking the HDF5 files by using the *h5repack* command, provided by the HDF5 library. For example, the size of the index file of 10 billion values with 2.43 billion cardinality for 1 million

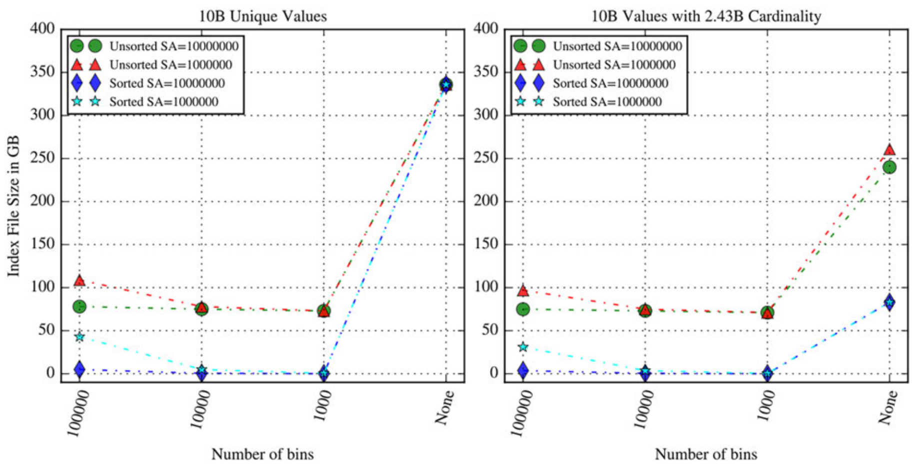

FIGURE 7 The size of index files for the 10-billion–row data set. The colored values are for the subarray sizes (green circles for the 10 million subarray size and unsorted data, red triangles for the 1 million subarray size and unsorted data, blue diamonds for the 10 million subarray size and sorted data, and cyan stars for the 1 million subarray size and sorted data)

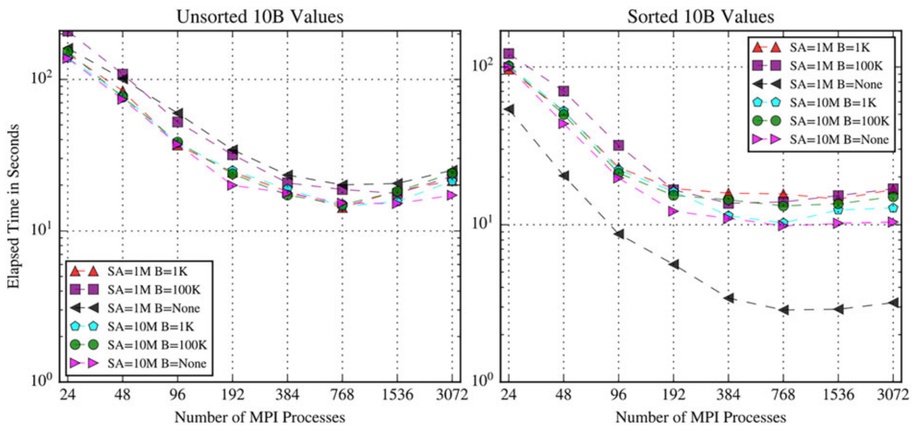

FIGURE 8 Query performance for varying number of MPI processes

subarray was 635 GB, and the size of the index file for 10 million subarray was 262 GB. After applying the repacking, the sizes were reduced to 262 and 240 GB, respectively. This problem seemed more common for the smaller subarray and the larger data set.

We conducted a series of experiments to show how our system was scaled up in a highly parallelized environment. The results were gathered by using the index files for varying subarray sizes and binning options. Figure 8 shows the query performance when we introduce an increasing number of MPI processes. The number of MPI processes ranges between 24 and 3072. For unsorted data, 1536 MPI processes generally perform the best. Using more MPI processes does not contribute to the query performance. Messaging between the high numbers of MPI processes hinders the gain from parallel execution. The index of 10 billion subarray size without binning provides the best timing results, whereas the index of 1 billion subarray size without binning performs the worst. Although we are expecting the best performance from the 1 billion subarray size, the impact of the sparse file problem seems to affect the performance. Reducing the size of the sparse index file by applying the repacking does not improve the performance even though the file size reduces almost by half.

For sorted data, the performance of the query becomes much better. As we increase the number of MPI processes, the query time improves until we reach 1536 processes. Even 768 MPI processes offer enough computing power for most cases. However, the query performance does not improve much for more MPI processes than 1536. We attribute the communication cost between the MPI processes beyond 768 MPI processes as causing the slowdown. The query results using binned indexes are not impressive because of the additional cost required for the candidate check.

Membership query performance using a bitmap index scales very well for up to billions of elements in a highly parallelized environment. A very significant increase in query performance is achieved so that the query on the sorted data having 10 billion rows with 2.43 billion cardinality is accomplished in 2 seconds. This noteworthy performance is repeated when the bitmap index with 1 million subarray size is used.

5 | CONCLUSION

Membership queries on very large data sets have presented serious challenges because of the need for repeated random access to the data. We have applied a bitmap index to membership queries and showed high-performance improvement over the queries using naïve query processing. Hence, the main contribution of this study is showing that using bitmap indexing for membership queries is very effective. While the concept of membership query is not new, using bitmap indexes to solve the performance problem of membership queries very effectively is novel.

Our second contribution is to show scalable indexing and querying in a highly parallelized environment. Showing that performing a parallel membership query using bitmap index scales well to billions of elements is an important result. The indexing and querying become "pleasingly parallel" because the data can be arranged in disjoint blocks. The parallelism happens at several levels. The performance of membership queries benefits from using a parallel file system in addition to many CPU cores. The scientific file format API such as HDF5 also helps in easily specifying the desired level of parallelism.

Lastly, the third contribution is to tune membership queries; to the best of our knowledge, membership queries were not evaluated by taking into account the features of the data, such as cardinality, the order of the data, and skewness. We explored the row ordering over large data sets. Sorting contributes greatly to improve query performance. For low cardinality, the performance becomes even better. The query processing becomes more efficient when the data are skewed. We observed excellent performance results for skewed data, especially when the data were sorted and had low cardinality. Tuning cardinality, the order of the data, and skewness contributes to reducing the index size as well. Furthermore, we observed that I/O cost does not necessarily dominate the query processing time when using a compressed bitmap index. The conventional wisdom is that the smaller the index size, the lesser the query processing time. However, it was observed that the query using a larger index occasionally outperformed the query using a smaller index.

ACKNOWLEDGMENTS

This research was supported by the TUBITAK BIDEB 2219 program and used resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility supported by the Office of Science of the US Department of Energy under Contract No. DE-AC02-05CH11231.

ORCID

Beytullah Yildiz https://orcid.org/0000-0001-7664-5145

REFERENCES

1. The large hadron collider. http://home.web.cern.ch/topics/large-hadron-collider. Accessed November 20, 2018.
2. ATLAS. http://home.web.cern.ch/about/experiments/atlas. Accessed November 20, 2018.
3. CMS. http://home.web.cern.ch/about/experiments/cms. Accessed November 20, 2018.
4. Dewdney PE, Hall PJ, Schilizzi RT, Lazio TJL. The square kilometre array. *Proc IEEE*. 2009;97(8):1482-1496.
5. Shoshani A, Rotem D. *Scientific Data Management: Challenges, Technology, and Deployment*. Boca Raton, FL: Chapman & Hall/CRC Press; 2010.
6. Angluin D. Queries and concept learning. *Mach Learn*. 1988;2(4):319-342.
7. Chan C-Y, Ioannidis YE. An efficient bitmap encoding scheme for selection queries. In: Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data; 1999; Philadelphia, PA.
8. Qiang J, Ryne RD, Venturini M, Zholents AA, Pogorelov IV. High resolution simulation of beam dynamics in electron linacs for x-ray free electron lasers. *Phys Rev ST Accel Beams*. 2009;12(10). https://doi.org/10.1103/PhysRevSTAB.12.100702
9. Chaudhuri S, Dayal U, Ganti V. Database technology for decision support systems. *Computer*. 2001;34(12):48-55.
10. Chaudhuri S, Dayal U. An overview of data warehousing and OLAP technology. *ACM SIGMOD Rec*. 1997;26(1):65-74.
11. Stonebraker M, Abadi DJ, Batkin A, et al. C-store: a column-oriented DBMS. In: Proceedings of the 31st International Conference on Very Large Data Bases; 2005; Trondheim, Norway.
12. Larson P-A, Clinciu C, Hanson EN, et al. SQL server column store indexes. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data; 2011; Athens, Greece.
13. Idreos S, Groffen F, Nes N, Manegold S, Mullender S, Kersten M. MonetDB: two decades of research in column-oriented database architectures. *IEEE Data Engineering Bulletin*. 2012;35(1):40-45.
14. Stockinger K, Wu K, Shoshani A. Strategies for processing ad hoc queries on large data warehouses. In: Proceedings of the 5th ACM International Workshop on Data Warehousing and OLAP; 2002; McLean, VA.
15. Stockinger K, Wu K. Bitmap indices for data warehouses. In: *Data Warehouses and OLAP: Concepts, Architectures and Solutions*. Hershey, PA: IGI Global; 2007:157-178.
16. Goyal N, Zaveri SK, Sharma Y. Improved bitmap indexing strategy for data warehouses. Paper presented at: 9th International Conference on Information Technology; 2006; Bhubaneswar, India.
17. Clarke J. Storage indexes. In: Clarke J, ed. *Oracle Exadata Recipes: A Problem-Solution Approach*. Berkeley, CA: Apress; 2013:553-576.

18. Hierarchical data format version 5 (HDF5). http://www.hdfgroup.org/HDF5/. Accessed November 20, 2018.
19. Network common data form (NetCDF). https://www.unidata.ucar.edu/software/netcdf/. Accessed November 20, 2018.
20. Gosink L, Shalf J, Stockinger K, Wu K, Bethel W. HDF5-FastQuery: accelerating complex queries on HDF datasets using fast bitmap indices. Paper presented at: 18th International Conference on Scientific and Statistical Database Management; 2006; Vienna, Austria.
21. Rotem D, Stockinger K, Wu K. Optimizing candidate check costs for bitmap indices. In: Proceedings of the 14th ACM International Conference on Information and Knowledge Management; 2005; Bremen, Germany.
22. Chan C-Y, Ioannidis YE. Bitmap index design and evaluation. In: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data; 1998; Seattle, WA.
23. Wu K, Shoshani A, Stockinger K. Analyses of multi-level and multi-component compressed bitmap indexes. *ACM Trans Database Syst*. 2010;35(1):1-52.
24. Wu K, Madduri K, Canon S. Multi-level bitmap indexes for flash memory storage. Paper presented at: 14th International Database Engineering & Applications Symposium; 2010; Montreal, Canada.
25. Sinha RR, Winslett M. Multi-resolution bitmap indexes for scientific data. *ACM Trans Database Syst*. 2007;32(3).
26. Johnson T. Performance measurements of compressed bitmap indices. In: Proceedings of the 25th International Conference on Very Large Data Bases; 1999; Edinburgh, Scotland.
27. Antoshenkov G. Byte-aligned bitmap compression. In: Proceedings of the IEEE Data Compression Conference; 1995; Snowbird, UT.
28. Wu K, Otoo EJ, Shoshani A. Compressing bitmap indexes for faster search operations. In: Proceedings of the 14th International Conference on Scientific and Statistical Database Management; 2002; Edinburgh, Scotland.
29. Wu K, Otoo EJ, Shoshani A. Optimizing bitmap indices with efficient compression. *ACM Trans Database Syst*. 2006;31(1):1-38.
30. Wu K, Otoo EJ, Shoshani A. On the performance of bitmap indices for high cardinality attributes. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases; 2004; Toronto, Canada.
31. Deliège F, Pedersen TB. Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In: Proceedings of the 13th International Conference on Extending Database Technology; 2010; Lausanne, Switzerland.
32. Lemire D, Kaser O, Aouiche K. Sorting improves word-aligned bitmap indexes. *Data Knowl Eng*. 2010;69(1):3-28.
33. van Schaik SJ, de Moor O. A memory efficient reachability data structure through bit vector compression. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data; 2011; Athens, Greece.
34. Chen Z, Wen Y, Cao J, et al. A survey of bitmap index compression algorithms for big data. *Tsinghua Sci Technol*. 2015;20(1):100-115.
35. Byna S, Chou J, Rübel O, et al. Parallel I/O, analysis, and visualization of a trillion particle simulation. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis; 2012; Salt Lake City, UT.
36. Chou J, Howison M, Austin B, et al. Parallel index and query for large scale data analysis. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis; 2011; Seattle, WA.
37. Kim J, Abbasi H, Chacon L, et al. Parallel in situ indexing for data-intensive computing. Paper presented at: IEEE Symposium on Large Data Analysis and Visualization; 2011; Providence, RI.
38. Su Y, Agrawal G, Woodring J. Indexing and parallel query processing support for visualizing climate datasets. Paper presented at: 41st International Conference on Parallel Processing; 2012; Pittsburgh, PA.
39. Yildiz B, Fox G, Pallickara S. An orchestration for distributed web service handlers. Paper presented at: 3rd International Conference on Internet and Web Applications and Services; 2008; Athens, Greece.
40. Yildiz B, Fox GC. Toward a modular and efficient distribution for web service handlers. *Concurr Comput Pract Exp*. 2013;25(3):410-426.
41. Aktas MS, Plale B, Leake D, Mukhi NK. Unmanaged workflows: their provenance and use. In: *Data Provenance and Data Management in eScience*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2013:59-81.
42. Pinar A, Tao T, Ferhatosmanoglu H. Compressing bitmap indices by data reorganization. Paper presented at: 21st International Conference on Data Engineering; 2005; Tokyo, Japan.
43. Sharma Y, Goyal N. An efficient multi-component indexing embedded bitmap compression for data reorganization. *Inf Technol J*. 2008;7(1):160-164.
44. Apaydin T, Tosun AŞ, Ferhatosmanoglu H. Analysis of basic data reordering techniques. In: *Scientific and Statistical Database Management: 20th International Conference, SSDBM 2008, Hong Kong, China, July 9-11, 2008 Proceedings*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2008.
45. Vanichayobon S, Manfuekphan J, Gruenwald L. Scatter bitmap: space-time efficient bitmap indexing for equality and membership queries. Paper presented at: IEEE Conference on Cybernetics and Intelligent Systems; 2006; Bangkok, Thailand.
46. Weahama W. Strategies to improve query processing time in searching membership queries of virtual classroom by using DBIC. *Int J Comput Internet Manag*. 2012;20(1):51-56.
47. Crainiceanu A, Lemire D. Bloofi: multidimensional bloom filters. *Inf Syst*. 2015;54:311-324.
48. Hua Y, Xiao B, Veeravalli B, Feng D. Locality-sensitive bloom filter for approximate membership query. *IEEE Trans Comput*. 2012;61(6):817-830.
49. Chou J, Wu K, Prabhat. FastQuery: a general indexing and querying system for scientific data. In: *Scientific and Statistical Database Management: 23rd International Conference, SSDBM 2011, Portland, OR, USA, July 20-22, 2011. Proceedings*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2011.
50. Wu K, Stockinger K, Shoshani A. Breaking the curse of cardinality on bitmap indexes. In: *Scientific and Statistical Database Management: 20th International Conference, SSDBM 2008, Hong Kong, China, July 9-11, 2008 Proceedings*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2008.
51. Wu K, Ahern S, Bethel EW, et al. FastBit: interactively searching massive data. *J Phys Conf Ser*. 2009;180(1):012053.
52. Lin K-W, Byna S, Chou J, Wu K. Optimizing fastquery performance on lustre file system. In: Proceedings of the 25th International Conference on Scientific and Statistical Database Management; 2013; Baltimore, MD.

53. Koudas N. Space efficient bitmap indexing. In: Proceedings of the 9th International Conference on Information and Knowledge Management; 2000; McLean, VA.

54. Stockinger K, Wu K, Shoshani A. Evaluation strategies for bitmap indices with binning. In: *Database and Expert Systems Applications: 15th International Conference, DEXA 2004, Zaragoza, Spain, August 30-September 3, 2004. Proceedings.* Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2004.

How to cite this article: Yildiz B, Wu K, Byna S, Shoshani A. Parallel membership queries on very large scientific data sets using bitmap indexes. *Concurrency Computat Pract Exper*. 2019;e5157. https://doi.org/10.1002/cpe.5157