Simplifying Index File Structure to Improve I/O Performance of Parallel Indexing

Hsuan-Te, Chiu*, Jerry Chou*, Venkat Vishwanath†, Surendra Byna‡, and Kesheng Wu‡

*National Tsing Hua University, Taiwan. Email: {jchou, albertchiu}@lsalab.cs.nthu.edu.tw
†Argonne National Laboratory, USA. Email: venkat@anl.gov
‡Lawrence Berkeley National Laboratory, USA. Email: {sbyna, kwu}@lbl.gov

***Abstract*—Complex indexing techniques are needed to reduce the time of analyzing massive scientific datasets, but generating these indexing data structures can be very time consuming. In this work, we propose a set of strategies to simplify the index file structure and to improve the I/O performance during index construction using FastQuery, which is a parallel indexing and querying system for scientific data. FastQuery has been used to analyze data from various scientific applications, including a trillion plasma particles simulation. To accelerate query process, FastQuery uses FastBit to build indexes, and then stores the indexes into file system through parallel scientific data format libraries, such as HDF5. Although these data format libraries are designed to support more complex multi-dimensional arrays, we observed that it still takes considerable work to map the indexing data structures into arrays, especially on parallel machines. To address this problem, in this paper, we attempt to minimize the I/O time by storing indexes into our self-defined binary data format. By fully controlling the data structure, we can minimize the I/O synchronization overhead and explore more efficient I/O strategy for storing indexes. Our experiments of indexing a trillion particle dataset using 20,000 cores of a supercomputer show that the proposed binary I/O driver can reach 85% of the peak I/O bandwidth on the system, and achieves a speedup of up to 4X in terms of the total execution time comparing to the previous FastQuery implementation with HDF5 I/O driver.**

***Keywords*—Parallel I/O, Storage system, Bitmap indexing**

I. Introduction

Scientific applications are known to produce and consume massive amounts of data from their large-scale experimental facilities or from simulations on peta-scale computing systems. For instance, the Intergovernmental Panel on Climate Change (IPCC) multi-model CMIP-3 archive is about 35 TB in size [2], and the LHC experiment is capable of producing 1 TB of data in a second. However, in many cases, the essential information is contained in a relatively small number of data records. For example, in the IPCC data, the critical information related to important events, such as hurricanes, is no more than a few gigabytes (10^9 bytes) out of the total data size in petabytes (10^{15} bytes). Therefore, the capabilities for accessing only the necessary information-rich data records, instead of sifting through all of them, can significantly accelerate scientific discoveries. This requirement for efficiently locating interesting data records is indispensable to many data analysis procedures.

Instead of asking the scientists to load data into commercial database systems, we advocate an approach of using an indexing library with common scientific data format libraries, so that scientists can stay with their existing data management and analysis software. For that purpose, we have developed FastQuery [6], which is built on top of the FastBit [19] bitmap indexing software to accelerate data selection based on arbitrary range conditions defined on the available data values, e.g., "energy $> 10^5$ and temperature $> 10^6$". To ensure applicability, a common data access layer is defined in FastQuery for working with many popular array-based scientific data formats, including HDF5 [17], NetCDF [18], pNetCDF [8], and ADIOS-BP [10]. The flexibility of this data layer and the scalability of the tool has also been demonstrated for analyzing large datasets from various scientific applications [4], [14], [5].

In our previous work [9], [6], we assumed indexes are written to files using the same data format as the user datasets, and we showed that the I/O time of writing indexes can be significantly reduced by carefully tuning the setting of file system and data format library. However, we also observed that the scientific data format may not be suitable for storing indexes for the following reasons. First, the indexes such as those built by FastBit includes a number of different components with a variety of organizations, while a file can only hold a single sequence of bytes. Although the scientific data format libraries are designed to support more complex multi-dimensional arrays, it still takes considerable work to map the indexing data structures into arrays, especially on parallel computing systems. Second, scientific data format often introduces an additional layer of data management, so that users can access and manipulate their data in a more flexible and complex manner. For instance, HDF5 maintains its own internal fixed-size data chunking to allow users control the data layout of a multi-dimensional array. But in order to extend a dataset by adding new chunks, it requires global synchronization across all the processes and causes expensive overhead. Third, scientific data format libraries often implement many complicated performance optimization techniques, and require users to carefully tune the controlling parameters in order to achieve good I/O performance. It is often unclear to a user if all the tunable options are optimal for a specific application. To tackle these issues, a redesign of bitmap index generation in FastQuery is necessary.

In this work, we propose a new index file structure to bypass the multiple layers of data format libraries and design our own custom binary file format. The key challenge in this work is to create a minimalist file structure that captures the index data structures and that is flexible enough to maximize I/O performance for a variety of problems. This simplified new file structure shortens the I/O path by directly writing indexes

RID	X	bitmaps				
		b_0 =0	b_1 =1	b_2 =2	b_3 =3	b_4 =4
1	1	0	1	0	0	0
2	0	1	0	0	0	0
3	4	0	0	0	0	1
4	2	0	0	1	0	0
5	3	0	0	0	1	0
6	3	0	0	0	1	0

Fig. 1. The logical view of a sample bitmap index.

into a binary file using low-level I/O operations from file system and from the MPI-IO library. This new approach prevents the dataset extension overhead of HDF5 by simply appending data into the index files rather than creating data chunks. The indexes are stored into multiple files for minimizing file locking and synchronization overhead caused by parallel I/O from a large number of MPI processes. Our experiments of indexing a trillion particle dataset using 20,000 cores of the Edison supercomputer show that our new approach reaches 85% of the peak I/O bandwidth on the system, and achieves a speedup of 2X to 4X over the previous implementation of FastQuery.

The rest of paper is organized as follows. In Section II, we introduce the background of FastQuery. Section III describes our new binary data format for storing indexes. The experimental setup and results are presented in Section IV and Section V, respectively. We summarize related work in Section VI and conclude the paper in Section VII.

II. Background

A. Bitmap indexing technology and FastBit

A bitmap index logically contains the same information as a B-tree index. A B-tree consists of a set of pairs of key value and row identifiers; however a bitmap index replaces the row identifiers associated with each key value with a bitmap. Because the bitmaps can be operated efficiently, this index can answer queries efficiently as demonstrated by Patric O'Neil [12]. The basic bitmap index uses one bitmap for each distinct key value, an example of which is shown in Figure 1. For scientific data whose the number of distinct values can be as large as the number of rows (i.e., every value is distinct). The number of bits required to represent an index may scale quadratically with the number of rows. Therefore, a number of different strategies have been proposed to reduce the bitmap index sizes and improve their overall effectiveness. Common methods include compressing individual bitmaps, encode the bitmaps in different ways, and binning the original data [15, Ch. 6]. FastBit [19] is an open-source software package that implements many of these methods, and it has been shown to perform well in a number of different scientific applications [19]. In addition, there are also a series of theoretic computation complexity studies to further establish its soundness [20]. However, FastBit is designed to run on a single computer, and it only supports a self-defined file format for user data. Therefore, it relies on FastQuery to achieve parallelism on multiple nodes and to support generic data formats.

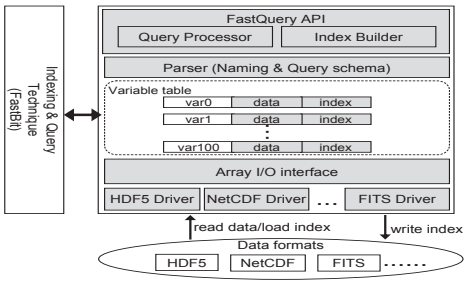


Fig. 2. FastQuery system architecture.

B. FastQuery overview

FastQuery is a parallel querying and indexing system for scientific data. It uses the FastBit to support data selection on scientific data based on arbitrary range conditions defined on the available data values, e.g., "energy $> 10^5$ and temperature $> 10^6$". As illustrated in Figure 2, to allow users to query and to access datasets in their original formats, FastQuery implements a unified array interface for various data formats including HDF5 [17], NetCDF [18], pNetCDF [8] and ADIOS-BP [10]. During data analysis process, users first build indexes on their dataset using the indexing API. The indexes can then be repeatedly used to accelerate queries applied on the indexed dataset.

The indexing operation contains: (1) reading the original data values from the file, (2) constructing bitmap indexes data structure in memory, and (3) writing the bitmaps and associated metadata, such as bitmap keys and offests, to the original file. On the other hand, the querying function evaluates different queries by accessing both the data and indexes. If the necessary indexes have not been built, FastQuery simply scans through the data values for evaluation. When the necessary indexes are available, the querying process involves: (1) loading the bitmaps keys to identify the required bitmaps for evaluating the range, (2) loading the required bitmaps from index file, and (3) evaluating the indexes by the query constraint.

Furthermore, in order to process massive datasets, FastQuery exploits parallelism at both computation and I/O levels. To take advantage of distributed memory nodes and multiple CPU cores systems, FastQuery divides a full dataset into multiple fixed size subarrays, and builds or queries the indexes of those subarrays iteratively as described in Section III-A. If the datasets are too large to fit into the memory of compute nodes, a smaller subarray size can be used to index the entire dataset in multiple iterations. Our previous work [9] has also shown that the I/O bandwidth of writing or reading indexes can be optimized by carefully tuning the I/O stack, such as the stripe size and the stripe count of a Lustre parallel file system, the collective buffer in the MPI-IO library, and the dataset chunk size in HDF5 library.

III. FastQuery with a New Binary I/O Driver

A. Motivation

Toward motivating the need for designing a new index file structure and a custom data format for storing indexes, we first explain the approach and limitations from our previous implementation using HDF5 library in this subsection. As illustrated by Figure 3, FastQuery partitions user data into fixed-size subarrays for creating indexes using parallelism. The indexes

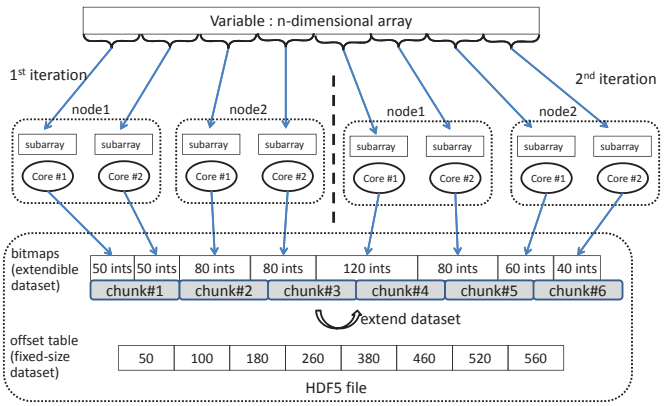


Fig. 3. An example of indexing in FastQuery using the HDF5 library. The user data is divided into fixed-size subarrays and iteratively built by multiple cores in parallel. The generated indexes are then stored to two separate datasets in a HDF5 file for query evaluation.

of those subarrays can be built and stored independently, so that they are assigned to MPI tasks iteratively in a round-robin fashion. For instance, in Figure 3, user data is divided into eight subarrays and built by two nodes using four cores in two iterations. Note that FastBit actually generates three sets of data from processing each subarray (i.e., bitmaps, bitmaps offsets and bitmaps keys). We show only the bitmaps dataset in the example for simplicity as the bitmaps offsets and keys are stored in the same way.

In our previous implementation using the HDF5 format, the indexes from all the subarrays are concatenated and stored into a single dataset for two main reasons. First, it reduces the number of datasets and the associated metadata overhead. Second, it benefits from the collective I/O method when the bitmap size written by each MPI task is small. Although subarray size is fixed, the size of indexes from each subarray is variable depending on the data values in those subarrays. Furthermore, because indexes could be built in multiple iterations to use the parallelism available on supercomputing systems, the total length of bitmaps from all the subarrays cannot be predetermined at the beginning of the building process. Therefore, the HDF5 dataset for storing bitmaps must be extendible to accommodate data from each iteration. Finally, an offset table is also created to store the location of the bitmaps from each subarray, and therefore given a subarray index, we can locate and load its corresponding indexes from the bitmaps dataset based on the information in the offset table. Different from the bitmaps dataset, the length of the offset table can be predetermined by the total number of subarrays.

There are three major issues we found in current implementation of FastQuery. (1) the function in HDF5 to extend dataset (i.e., H5D_extend) is a collective call that causes global synchronization among all processes. As shown by the experimental result in Section V-B, this overhead can grow significantly as the number of building iterations increases. The issue also cannot be resolved by storing the indexes from each subarray in a separate datasets, because dataset creation function in HDF5 (i.e., H5D_create) is also a collective call and suffers from the same problem. (2) in HDF5, the extendible dataset has to be managed and stored by a set of fixed-size chunks. For instance, if the chunk size is 100 integers, the bitmaps dataset in Figure 3 is divided into 6

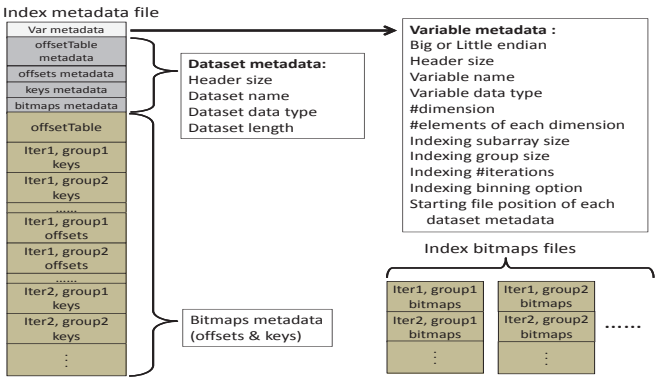


Fig. 4. The file structure of binary driver shows the indexes are stored in an index metadata file, and a set of index bitmaps files. All metadata information is compacted and serialized into a binary string. The length of bitmaps files are different and depending on the bitmaps length.

chunks and stored separately by HDF5. To obtain better I/O performance, the chunks of the dataset needs to be split along the logical boundary of the dataset array. However, because the sizes of bitmaps from every subarray are all different, it is difficult to find the best setting of the chunk size. As a result, some bitmaps could spread across multiple chunks and cause more I/O overhead. For instance, the bitmaps from the fourth subarray is split between the second and third chunks in our example, and causing two separate I/O operations instead of one. (3) Finally, since all the processes are performing I/O on the same dataset in the same file at the same time in each iteration, higher I/O contention could be expected. On the other hand, storing indexes into multiple files or datasets could cause more metadata overhead. But this trade-off factor was not investigated in our previous implementation. These issues motivate the need for a simple and flexible file structure.

B. Binary Index File Structure

To overcome the limitation of FastQuery implementation and to shorten the I/O path, in this work we introduce a self-defined binary data format to store indexes. In addition, we introduce a new FastQuery option called the *group size* to control the number of processes written to the same index file. Hence as shown in Figure 4, the indexes built from a variable of user dataset is written to an index metadata file and a set of index bitmaps files. All the metadata information for indexes built from different iterations and groups are written in the same file, because the size of the metadata is negligible compared to the size of the indexes. On the other hand, depending on the binning option used by FastBit, the size of bitmaps could even grow larger than the size of user data. Therefore, by adjusting the group size parameter, we allow MPI processes to write indexes in parallel on multiple independent files, and limit the total number of index files. As observed in our experiments in Section V-C, the I/O performance can be improved by setting the group size according to the number OSTs (object storage targets) of Lustre file system. Physical separation of the index metadata and bitmaps into different files also allows us to choose different I/O strategies and settings for performance optimization. For instance, according to our previous study [9], the smaller metadata file should choose a smaller stripe size setting of Lustre, and use collective I/O method of MPI library, while the larger bitmaps files should use a larger strip size setting in file system, and use independent MPI-I/O method.

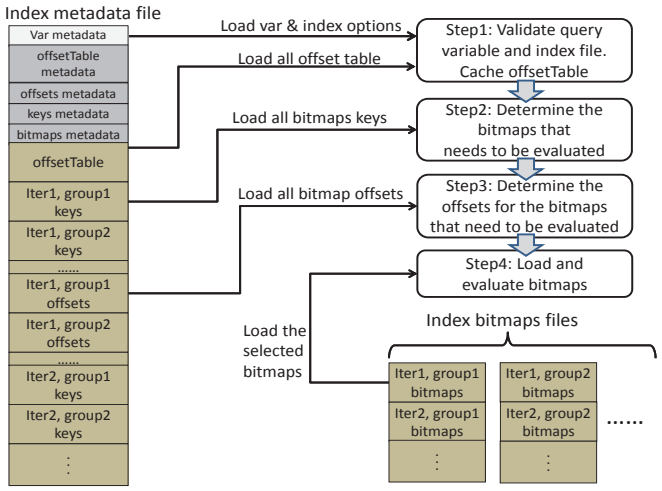


Fig. 5. An example of the query process performed on the first subarray built by the first MPI task in the first iteration.

As shown in Figure 4, the data structure of index metadata file consists of three parts: variable metadata, dataset metadata, and actual datasets for storing the bitmaps keys and offsets. The variable metadata stores two types of information: the user variable information, such as the variable name, data type, and data dimension, and the indexing parameters, such as the the subarray size, group size, binning options, and the total number of building iterations. Therefore, based on those information we can identify the indexes are for which user variable and how they were built by FastQuery. To search the dataset metadata information in the next level, we also reserve the space to record the starting file position of each dataset metadata entry at the end.

The dataset metadata is used to describe the datasets for storing the bitmaps data. In our file structure, four datasets are created: offset table, bitmaps offsets, bitmaps keys, and bitmaps. For each dataset, we record its dataset name, data type and length, so that FastQuery can load the values from the dataset in the correct format and within the valid data region. Finally, the offset table, bitmaps offsets and bitmaps keys are written iteratively into the index metadata file similar to the steps described in Section III-A. Again, because the total size of offset table is fixed, its space is reserved at the beginning, and its content is updated after each iteration. However, unlike the HDF5 file format, our index files are simply binary string, so the new written data from each iteration can be directly appended to the end of the files rather than calling the expensive API (i.e. H5D_extend or H5D_create) from the HDF5 library. Furthermore, the bitmaps are written to separate files, so the MPI processes from each group are operating on their own independent file separately to prevent I/O contention.

C. Index Storing and Loading Process

Based on the file structure defined above, we briefly describe the process of storing and loading the indexes from those index files below. The indexes are stored to files in the following four steps. Step 1: the index metadata file is created, and the variable information and indexing parameters are written to the file by the master MPI process. According to the number of groups, the index bitmaps files are also created. Step 2: the dataset metadata is written to the index metadata file, and the dataset length is initialized to 0. Step 3: the space of fixed-size offset table is reserved and initialized. The number of table entry is the same as the total number of subarrays which can be computed by the total size of variable divided by the size of subarray. In fact, three equal size offset tables are created to record the file position of each subarray for the bitmaps, bitmaps offsets, and bitmaps keys, respectively. Since the indexes are written in multiple iterations, the offset table is updated after each iteration. Finally, after each iteration Step 4 is performed to gather the indexing results from the MPI tasks. The bitmaps offsets and keys from different groups are concatenated and written together to the index metadata file. But the bitmaps are only aggregated by the per-group basis, and written to different index files for each group.

During query evaluation, the indexes are loaded from files in the following steps as illustrated in Figure 5. Step 1: load the variable and dataset metadata into memory to verify the indexes are built for the query variable. The offset table is also loaded and cached at compute nodes for quick lookup in the future steps. Step 2: the hit count of query is evaluated on each of the subarrays in parallel and independently by multiple MPI processes. More specifically, each MPI process first loads the bitmaps keys of its subarray to determine which bitmap needs to be loaded for evaluation. The location of its bitmaps keys in the index file can be looked up from the offset table according the iteration number, group number and MPI rank. Step 3: if a bitmap needs to be loaded for evaluating the query, a MPI process determines the offset of this bitmap from the compressed bitmaps string using the bitmaps offsets data. Finally, Step 4: a MPI process first use the bitmaps metadata stored in the index metadata file to locate the bitmaps built for its subarrary. Then based on the offset value returned by Step 3, the process loads the bitmap from the index bitmaps file, and evaluate the hit count using the FastBit function.

IV. Experimental Methodology

A. Research Questions

In evaluating the performance of the new binary I/O driver, we address the following questions:

- What is the scalability and overall performance improvement?
- What is the impact of FastQuery subarray size?
- How to set the group size (processes per file)?

We performed a number of detailed performance studies that are listed in Section V. In the reminder of this section, we briefly discuss the hardware platform and the datasets used in the study.

B. Hardware Platform

We have conducted all our experiments on the Cray XC30 system, called Edison, at the National Energy Research Scientific Computing Center (NERSC). Edison has 5576 compute nodes, where each node has two 12-core Intel Ivy Bridge 2.4GHz CPU and 64GB of memory. Edison has three Lustre parallel file system storage spaces. The input data file is stored by the owner of the data in a file system (named /scratch1) that has a peak I/O bandwidth 48 GB/s with 96 OSTs. We write the

bitmap index generated by FastQuery to a file system (named /scratch3) that has a peak performance at 72 GB/s with 144 OSTs.

For each experiment, we launch FastQuery as a set of MPI tasks with one core per task. To prevent memory contention between the MPI tasks on the same node, we have limited the memory usage of each FastQuery process to 2 GB, using 24 cores per node; the rest of memory space is reserved for system usage, such as initiating MPI tasks, etc. Based on our previous study [9], we tune the I/O setting of our experiments as follows unless specified otherwise. We have set the Lustre stripe size as 64 MB, and the FastQuery subarray size as 9.25 million elements. The Lustre stripe count of the HDF5 driver output index file is 144 (equaling to the number of OSTs on Edison /scratch3). On the other hand, our binary driver writes indexes into 144 files, and the Lustre stripe count of each file is 1 according to our tuning result of the group size presented in Section V-C. Both binary and HDF5 drivers use independent MPI-IO calls to write bitmaps because the data size written in each call (i.e., 44 MB) is large and may not benefit from using collective I/O method as shown in Section V-B.

C. Dataset

All our evaluations use a test dataset produced by running a plasma physics simulation called VPIC [4]. VPIC writes a significant amount of data at a user-prescribed interval. In this study, we use the data files from a simulation of 1 trillion electrons. Each particle is represented by eight properties and the total size of the particle data varies between 32 TB and 43 TB. We generate indexes for the Energy property of the particles and execute queries based on the Energy, such as "$Energy > 1.8$" etc. The size of the bitmap indexes generated by FastQuery is ≈1.5 TB.

D. Performance Metrics

As mentioned earlier, the process of building indexes includes four main steps: reading data, computing indexes, writing bitmap metadata, and writing indexes. The time for writing bitmap metadata of HDF5 includes the time for creating and for extending the HDF5 datasets. In our experiments, we record the completion time of each step over all the index building iterations, and report their aggregated numbers. Because the file system used in our testbed is a shared resource among hundreds of users, the file system experiences contention from other jobs running on the system. In addition, as observed in a previous study [11], many other issues, such as the complexity of the I/O software stack and the contention of non-IO MPI communication, could also contribute to the variability of performance results. Therefore, we repeat each experiment at least 5 times over the course of a week, and report the medium value over the experiment runs.

V. Performance Evaluation

In this section, we present the performance results from our experiments to answer the three research questions mentioned earlier. We demonstrate the performance and scalability of newly developed binary driver by comparing with the results of the previous HDF5 driver that writes indexes to HDF5 files. We also investigate the impact of the subarray size and the group size of the binary driver on the performance of FastQuery.

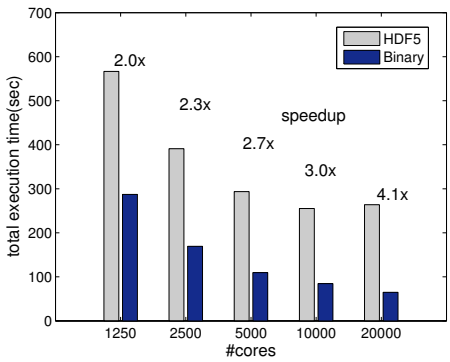


Fig. 6. Comparison of the total execution time between HDF5 and binary drivers using CPU cores increasing from 1,250 to 50,000. The speedup of binary driver over HDF5 driver from each run is listed above the bar.

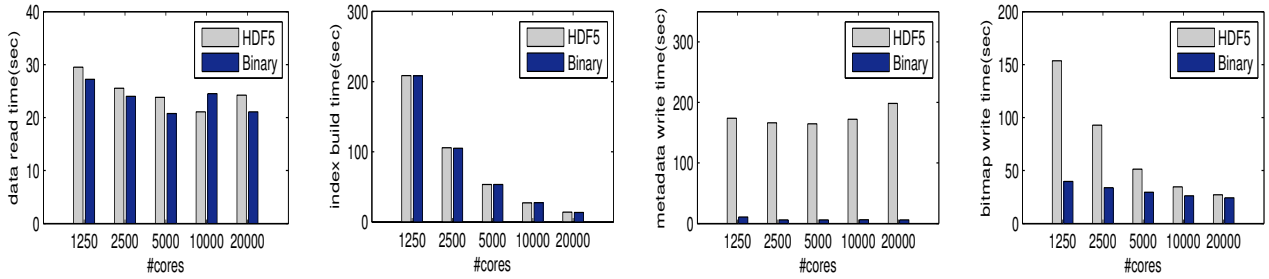
A. Scalability and overall improvement

We first compare the total execution time of the binary driver and the HDF5 driver as the number of cores increases from 1,250 to 20,000 as shown in Figure 6. We found that the execution time of both drivers decreases with more cores until the I/O bandwidth is saturated. However, the new binary driver achieves a much shorter execution time than the HDF5 driver. The speedup of the binary driver over the HDF5 driver increases from a factor of 2 to a factor of 4.1. To further analyze this result, we present a breakdown the execution time into 4 steps as shown in Figure 7.

Figure 7 (a) and (b) show the data read time and index compute time. The time of those two steps are expected to be the same for both drivers, because our binary driver only affects the time of writing index files. The data read time did not improve further after 5,000 cores because the input file is stored on /scratch1 which only has 48 GB/s peak performance, and our result with 5,000 cores achieves 40 GB/s. On the other hand, the index compute time is pure CPU computation, so it follows the ideal linear speedup to the number of CPU cores.

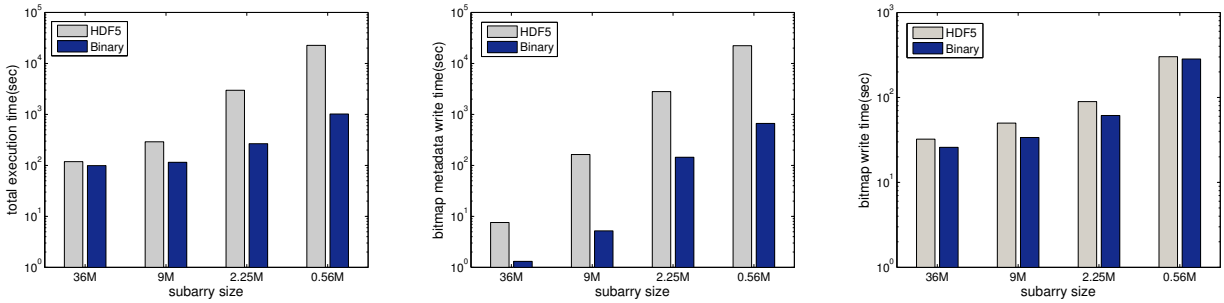
As shown in Figure 7 (c) and (d), the binary driver significantly reduces the bitmap metadata write time and bitmap write time. Especially, the bitmap metadata write time of HDF5 driver is much longer than binary driver, because of the synchronization time of the HDF5 dataset creation and extension functions in this step. In comparison, binary driver does not have to manage the datasets or the data chunks like HDF5, so its bitmaps metadata write time only depends on the size of bitmaps metadata (i.e., bitmaps keys and offsets), which is relatively small (a few GBs) compared to that of the bitmaps (TBs). In terms of the bitmaps write time, binary driver also achieves a faster write time because binary driver directly writes indexes into the file system without the additional layer of HDF5. By writing indexes into multiple independent files also reduces the I/O contention from multiple MPI processes of index generation task.

The corresponding I/O bandwidth from writing the bitmaps step is shown in Figure 8. The figure shows that the binary driver can achieve about 60 GB/s I/O bandwidth (almost 85% of the peak value of the file system, i.e., 72 GB/s). The achieved bandwidth of both the drivers gradually reduces as they approach to this peak performance. But when the number of cores is fewer than 5,000 cores, the binary driver shows much greater improvement over the HDF5 driver.



(a) Data read time. (b) Index compute time. (c) Bitmaps metadata write time. (d) Bitmaps write time.

Fig. 7. A comparison of time spent in each of the index generation step as the number of cores increases from 1,250 to 20,000.



(a) Total execution time. (b) Bitmap metadata write time. (c) Bitmaps write time.

Fig. 9. Performance comparison with varying subarray size, from 36 million elements to 562.5 thousand elements, by a factor of 4 in each step. Under the same total data size of each experiment, the number of iterations for building the indexes also increases from 1 to 64 by a factor of 4 in each step accordingly.

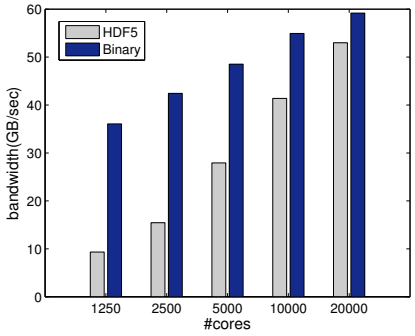


Fig. 8. I/O bandwidth comparison of the bitmaps writing step. The binary driver achieves higher bandwidth than the HDF5 driver, and obtains almost 85% of the peak I/O rate of 72 GB/s.

In summary, our experiment demonstrates that the new binary driver of FastQuery achieves better scalability as the number of cores increase and higher I/O bandwidth than the previous HDF5 driver implementation because of the simpler and more controllable file structure for storing indexes. Our experiments show that the binary I/O driver can reach 85% of the peak I/O bandwidth of the system, and successfully achieves a speedup factor of 2 to 4 in terms of the total execution time over the previous HDF5 data format driver.

B. Impact of subarray size

One of the most critical parameters in FastQuery is the subarray size that controls the amount of data to be indexed by each MPI task at a time. A larger subarray size means the indexes can be built and written in bigger I/O size and fewer iterations, so we often prefer to use a larger subarray size. However, the subarray size is also bounded by two important factors. The first is the number of MPI processes for building indexes, because its maximum number is equal to the total data size divided by the subarray size. The second factor is the memory space limit per core, because the input data and the generated indexes of a subarray must reside in memory Therefore, we evaluate the impact of subarray size on performance by fixing the number of cores at 5088, and reducing the subarray size from around 36 million elements to 562.5 thousand elements by a factor of 4 in each step. Since the total data size is the same, the number of iterations for building the indexes also increases from 1 to 64 accordingly.

As shown in Figure 9 (a), the total execution time increases as the subarray size reduces because of the slower I/O bandwidth caused by the smaller I/O size. Because we show the $y - axis$ is in the log-scale, the performance degradation for the HDF5 driver is much more significant than the binary driver. To further analyze the performance degradation, we compare the bitmap write time and bitmap metadata write time in Figures 9 (b) and (c), respectively. We observe that the bitmap write time for both drivers are similar, but the binary driver clearly outperforms the HDF5 driver as the subarray size reduces. This is mainly because the number of iterations increases as the subarray size reduces. Since the HDF5 driver has to call HDF5_extend() after each iteration, its synchronization overhead grows further with smaller subarray size. More surprisingly, we found that the subarray size has even greater impact to the bitmaps metadata write time. As seen from the two figures, when the subarray size reduces from 36M to 0.56M, the bitmap write time increases by a factor of 10, but the bitmap metadata write time increases by a factor over 100. In other words, when the subarray size is small, the bitmap metadata write time could dominate the bitmap write time even though most actual I/O occurs during writing

	64K	256K	1M	4M	16M
collective I/O	70.45s	17.49s	9.15s	6.05s	4.90s
independent I/O	89.29s	21.32s	10.36s	5.33s	3.89s
normalization	-26.74%	-21.89%	-13.22%	11.85%	20.65%

TABLE I. Comparison of bitmap write time (in seconds) between using collective and independent MPI-IO methods in the binary driver. The value of normalization is computed by the amount of time reduction of independent I/O as a percentage of the collective I/O time. Positive normalization value implies that the independent I/O is faster than the collective I/O.

	stripe count=1	stripe count=2	stripe count=144
#files=1	944.45s	390.21s	82.41s
#files=72	221.28s	75.65s	88.68s
#files=144	49.08s	55.67s	63.22s

TABLE II. Bitmap write time (in seconds) in different Lustre stripe count and index files. The number of index file is the number of processes divided by the group size. Based on this result, we decided to use 144 index files with Lustre stripe count 1 as the setting throughout the paper.

bitmaps. Therefore, the improvement from the binary driver can increase as the subarray size reduces or the number of building iteration increases.

We also compare the performance of using collective and independent MPI-IO methods in the binary driver as the subarray size reduces. As shown in Table I, because the size of indexes built from a subarray has positive correlation to the size of subarray, smaller subarray implies smaller I/O size. When the subarray size is small, I/O performance can be improved by using collective MPI-IO method to aggregate multiple small I/O requests from MPI processes into a single I/O request. On the other hand, when the subarray size is large, using collective MPI-IO could introduce additional overhead without reducing the number of I/O requests, and therefore independent MPI-IO will provide better I/O performance for this case as we have observed in our previous study [9] for the HDF5 driver as well.

Overall, we found subarray size has significant impact on the I/O performance of FastQuery, because smaller subarray size can cause overhead from synchronization call and file metadata management. In comparison to the HDF5 driver of FastQuery, we show that the binary driver is much more resilient to the subarray size because it simplifies file structure and prevents synchronization call. Depending on the subarray size, both drivers should choose the proper setting of independent or collective MPI-IO method accordingly.

C. Impact of group size (number of processes per file)

By setting the group size of the binary driver, we can control the number of MPI processes that write bitmap indexes to the same file. The smaller group size means less I/O contention between processes during the actual I/O operations because data locking can be prevented at the file system level. However, smaller group size also generates more index files and could cause higher contention to the file system to manage the metadata of multiples files at the same time. Hence, the group size clearly presents a trade-off for tuning the performance of the binary driver. Furthermore, in Lustre file system, because files could be striped on to multiple OSTs (object storage targets), even though processes are not accessing the same file, they could still request files from the same OST and cause I/O contention. Therefore, in this subsection, we tune the I/O performance of binary driver by adjusting the group size and Lustre stripe count together, so that we could find the best setting of binary driver and understand the correlation of those two tuning parameters.

As summarized in Table II, we report the bitmap write time as the Lustre stripe count is set to 1, 2, and 144 (the maximum number of OSTs) and the group size is adjusted to 1, 72 and 144, respectively. Our key observations are summarized as follows. First, when we use a single index file (i.e., the group size is equal to the number of total MPI processes), we can gain significant performance improvement by using a lager stripe count. This is because higher stripe count can increase the I/O parallelism of a single file by using more OSTs in Lustre. Second, if the number of index files is set equal to 72, we can achieve the peak performance with stripe count equal to 2. Higher or lower stripe count could result in lower I/O performance because the number of OSTs on Edison /scratch3 is 144. If the stripe count equals to 1, the binary driver only makes use of half of the OSTs' I/O bandwidth. On the other hand, if stripe count larger than 2, files will be striped onto the same OST and cause I/O contention between MPI processes. Finally, when we set the number of index files equal to 144. This case can have the best performance when the stripe count is equal to 1 due to the same reason. However, comparing to the 72 files case, the performance of 144 files decreases slightly as the stripe count increases. This is likely because the MPI processes can also be more evenly distributed across OSTs with more files.

In summary, we found that the group size and Lustre stripe count will determine the number of OSTs used during indexing. Best I/O performance is likely to be achieved when the multiplication of those two parameters is the same as the number of OSTs in the system. When using fewer number of OSTs, I/O performance could drop significantly due to less available I/O bandwidth. When using higher number of OSTs, performance could be slightly degraded due to I/O contention. The level of I/O contention will be dependent on the load balancing of OSTs, so with more files the load is more likely to be balanced by the Lustre system, and less impact is observed. Therefore, based on this result, we decided to use 144 index files with the Lustre stripe count at 1 as the setting of our binary driver throughout all the experiments in this paper.

VI. Related Work

In this section, we briefly review related work and point out the distinct features of our current work.

Scientific applications generally store their data in application-specific data formats. Over the years, a consensus has gradually emerged that arrays can be used to capture the main data structures required for scientific data. Thus, the commonly used scientific data formats are designed to store arrays efficiently [17], [18]. For this reason, we designed FastQuery to work with arbitrary and multi-dimensional array data structures.

A number of database systems, such as, SciDB [16] and MonetDB [3], are based on a similar array data models. However, our approach does not require the user to load their data into the database system, avoiding the need for additional data

copies. This is a significant benefit, in particular considering the massive volume of many scientific data, loading into these systems is cumbersome, time-consuming and possibly error-prone. In addition, our approach makes it possible to integrate the indexing capability directly with the scientific data formats themselves.

A variety of indexing techniques are available in popular database systems [13], many of which are variations of the B-Tree [7]. These indexing methods are designed for transaction-type applications, exemplified by interactions between a bank and its customers. Typical interactions with scientific data, however, are significantly different from operations on transaction-type data. A typical search operation in transactional data retrieves very few data records, such as a look-up of a single customer's banking account information. In contrast, search operations on scientific data commonly retrieve many more data records. For example, a scientist might be interested in studying how the ignition progresses in a combustion simulation from a spark into a flame engulfing the whole combustion chamber. In this case, resolving the query of interest might result in a few records in the beginning of the simulation, but might expand to include the majority of records towards the end. Furthermore, transactional data is frequently modified, one record at a time, whereas scientific data typically stays as is after it has been generated. The B-Tree data structure is designed to update quickly as the underlying data records are modified. This feature is unnecessary in indexes for the majority of scientific data sets. For such scientific data sets, the bitmap index is a more appropriate indexing structure [12][15].

Many of the parallel and distributed indexing techniques are derived from the B-Tree [1]. These parallel trees support only limited amounts of concurrency in both index construction and use and have been shown to not perform as well as bitmap indexes. In general, we see bitmap indexes as more appropriate for scientific data applications and have implemented our parallel indexing system based on the sequential bitmap index software FastBit [19].

VII. Conclusion

In this work, we propose a set of strategies to simplify the index file structure and improve the I/O performance during index construction of FastQuery. To better understand how the index file structure affects the I/O performance, we propose to bypass multiple layers of the scientific data format libraries such as HDF5 and design a custom binary file format. Our design shortens the I/O path by directly writing indexes into a binary file using low-level I/O operations from file system and the MPI-IO library. The dataset extension overhead from HDF5 can also be prevented by simply appending data into the index files rather than creating data chunks. Finally, we allow indexes to be stored into multiple files for minimizing file locking and synchronization overhead caused by the parallel I/O from all processes. Our experiments of indexing a trillion particle dataset using 20,000 cores on Edison supercomputer at NERSC shows that the newly proposed binary I/O driver can reach 85% of the peak I/O bandwidth on the system, and successfully achieve a speedup of 2X to 4X in terms of the total execution time compared to the previous HDF5 driver of FastQuery. We plan to evaluate the performance of our new proposed driver on Mira, a BlueGen/Q supercomputer with GPFS file system from Argonne National Lab. With more control over the I/O path of FastQuery using the binary driver, we will explore different I/O strategies on different types of computing and storage systems.

References

[1] M. K. Aguilera, W. Golab, and M. A. Shah. A practical scalable distributed b-tree. *Proc. VLDB Endow.*, 1:598–609, August 2008.

[2] IPCC Fifth Assessment Report. http://en.wikipedia.org/wiki/IPCC_Fifth_Assessment_Report.

[3] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[4] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan. Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation. *Physics of Plasmas*, 15(5):7, 2008.

[5] S. Byna, J. Chou, O. Rübel, Prabhat, H. Karimabadi, W. S. Daughton, V. Roytershteyn, E. W. Bethel, M. Howison, K.-J. Hsu, K.-W. Lin, A. Shoshani, A. Uselton, and K. Wu. Parallel i/o, analysis, and visualization of a trillion particle simulation. In *SC*, page 59, 2012.

[6] J. Chou, M. Howison, B. Austin, K. Wu, J. Qiang, E. W. Bethel, A. Shoshani, O. Rübel, Prabhat, and R. D. Ryne. Parallel index and query for large scale data analysis. In *SC*, page 30, 2011.

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

[8] J. Li, W. keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *SC'03*, page 39, New York, NY, USA, 2003. ACM.

[9] K.-W. Lin, J. Chou, S. Byna, and K. Wu. Optimizing fastquery performance on lustre file system. In *SSDBM*, page 29, 2013.

[10] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *CLADE'08*, pages 15–24, 2008.

[11] J. Mache, V. Lo, and S. Garg. The impact of spatial layout of jobs on I/O hotspots in mesh networks. *JPDC*, 65(10):1190–1203, Oct. 2005.

[12] P. O'Neil. Model 204 architecture and performance. In *2nd International Workshop in High Performance Transaction Systems, Asilomar, CA*, volume 359 of *Lecture Notes in Computer Science*, pages 40–59. Springer-Verlag, Sept. 1987.

[13] P. O'Neil and E. O'Neil. *Database: principles, programming, and performance*. Morgan Kaugmann, 2nd edition, 2000.

[14] O. Rübel, Prabhat, K. Wu, H. Childs, J. Meredith, C. G. R. Geddes, E. Cormier-Michel, S. Ahern, G. H. weber, P. Messmer, H. Hagen, B. Hamann, and E. W. Bethel. High Performance Multivariate Visual Data Exploration for Extemely Large Data. In *SuperComputing 2008 (SC08)*, Austin, Texas, USA, November 2008.

[15] A. Shoshani and D. Rotem. *Scientific Data Management: Challenges, Technology, and Deployment*. Chapman & Hall/CRC Press, 2010.

[16] M. Stonebraker, J. Becla, D. Dewitt, K. tat Lim, D. Maier, O. Ratzesberger, and S. B. Zdonik. Requirements for science data bases and scidb. In *Conference on Innovative Data Systems Research*, 2009.

[17] The HDF Group. HDF5 user guide. http://hdf.ncsa.uiuc.edu/HDF5/doc/H5.user.html, 2010.

[18] Unidata. The NetCDF users' guide. http://www.unidata.ucar.edu/software/netcdf/docs/netcdf/, 2010.

[19] K. Wu, S. Ahern, E. W. Bethel, J. Chen, H. Childs, E. Cormier-Michel, C. Geddes, J. Gu, H. Hagen, B. Hamann, W. Koegler, J. Lauret, J. Meredith, P. Messmer, E. Otoo, V. Perevoztchikov, A. Poskanzer, Prabhat, O. Rubel, A. Shoshani, A. Sim, K. Stockinger, G. Weber, and W.-M. Zhang. FastBit: Interactively searching massive data. In *SciDAC*, 2009.

[20] K. Wu, A. Shoshani, and K. Stockinger. Analyses of multi-level and multi-component compressed bitmap indexes. *ACM Transactions on Database Systems*, pages 1–52, 2010.