Design of FastQuery
How to Generalize Indexing and Querying System for Scientific Data

Jerry Chou, Kesheng Wu and Prabhat

Lawrence Berkeley National Laboratory
One Cyclotron Road
Berkeley, CA 94720

DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.

Design of FastQuery: How to Generalize Indexing and Querying System for Scientific Data

Jerry Chou, Kesheng Wu and Prabhat
Lawrence Berkeley National Laboratory
One Cyclotron Road
Berkeley, CA 94720

April 18, 2011

Abstract

Modern scientific datasets present numerous data management and analysis challenges. State-of-the-art index and query technologies such as FastBit are critical for facilitating interactive exploration of large datasets. These technologies rely on adding auxiliary information to existing datasets to accelerate query processing. To use these indices, we need to match the relational data model used by the indexing systems with the array data model used by most scientific data, and to provide an efficient input and output layer for reading and writing the indices. In this work, we present a flexible design that can be easily applied to most scientific data formats. We demonstrate this flexibility by applying it to two of the most commonly used scientific data formats, HDF5 and NetCDF. We present two case studies using simulation data from the particle accelerator and climate simulation communities. To demonstrate the effectiveness of the new design, we also present a detailed performance study using both synthetic and real scientific workloads.

1 Introduction

Modern scientific applications are producing vast amounts of data[5, 11]. In many cases, the essential information needed for understanding scientific processes is stored in a relatively small number of data records. In such cases, efficiently locating the interesting data records is indispensable to the overall analysis procedure. In database research, the data structures most effective at accelerating these search operations is known as indices [11]. This paper presents a flexible way of applying such an index to a range of different scientific data.

Scientific datasets are often large and shared by international collaborators. For example, the simulation data produced for Intergovernmental Panel on Climate Change's (IPCC) fifth assessment report (AR5) amounts to tens of petabytes (10^{15} bytes) and is shared by thousands of climate scientists around the word. Scientists are often unwilling or unable to place the datasets under the control of typical database management systems [6]. For example, IPCC has settled on storing all AR5 data in the NetCDF [13] file format.

Instead of requiring the users to place their data into centralized data management system (DBMS), we propose to build indices along with the existing data to accelerate the search operations. To demonstrate the effectiveness of this approach, we have chosen to use an indexing software called FastBit [15]. In a number of earlier studies, FastBit has been demonstrated to support scientific applications well. We have worked in the past on applying this indexing software to a small class of well-defined HDF5 files; the resulting system was called HDF5-FastQuery [4]. However, HDF5-FastQuery was focusing on the query functionality, and did not address other critical issues, such as the usability and flexibility of the API, the applicability to the complex scientific file structures, and the generalization of interface for different data formats. In this paper, we substantially extend our previous work to address all the concerns above, in terms of the system design, API and implementation. Specifically, the main contributions of the paper are summarized below.

- We match the relational model followed by FastBit indexing software and the array model used by most scientific applications.
- We deploy a simple yet flexible naming schema for users to specify datasets in an arbitrary and complex scientific data format file.
- We allow users to index and query on subarrays within a dataset which enables users to performance data analysis at a finer granularity level.
- We define a simple abstraction layer between FastQuery and the scientific format file libraries, so our system can be applicable to common scientific data formats quickly and efficiently.
- We extensively evaluate the performance of our new indexing software using both synthetic and simulation data, and demonstrate the usability of our system through two application case studies.

The rest of the paper is structured as follows. The related work on scientific data formats and indexing is in Section 2. Section 3 and Section 4 describe the design and implementation of FastQuery, respectively. The evaluations of FastQuery are shown in Section 5. We conclude with some thoughts for future directions in Section 6.

2 Related Work

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

2.1 Scientific Data Formats

Many scientific applications store their data as arrays. Thus the commonly used scientific data formats are designed to store arrays efficiently. In this work, we use two popular formats (NetCDF [13] and HDF5 [12]) to demonstrate the flexibility of our indexing interface. Both NetCDF and HDF5 are designed to be portable and self-describing. With appropriate data access libraries, they can be transported across machines and architectures. These data formats are widely used by scientific applications.

A key distinction between the HDF5 file format and the NetCDF file format is that HDF5 supports more comprehensive grouping structure. A typical NetCDF file might have all arrays exposed at the file level, while the top level objects in a HDF5 file are typically groups. Regardless of the file formats, a scientific data files often contains a single array for all physical quantities. This makes it necessary for an indexing software to not only address arrays but also subarrays. Earlier work on indexing scientific data have only considered indexing whole arrays [4]; we believe that we are the first ones to consider indexing subarrays.

2.2 Indexing for Scientific Data Formats

Most of the research work on indexing and searching techniques are designed for commercial database applications. However with the recent explosion in sizes of scientific datasets, researchers are starting to explore extending indexing techniques for scientific applications as well [11]. Traditional indexing techniques such as B-tree is designed primarily to accelerate access to individual data records, such as looking for a customer's bank record [2]. In contrast, a query on a set of scientific data typically result in a fairly large number of records, for example, a search for accelerated particles in a Laser Wakefield particle accelerator might result in thousands of data records corresponding to thousands of accelerated particles. Furthermore, scientific datasets are often produced or collected in bulk and once written to files are never modified. A class of indexing methods that can take full advantage of these characteristics is called the bitmap index.

2.3 Bitmap Indexing Technology

Bitmap indexing method logical contains the same information as a B-tree. 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 Patrick O'Neil in Model 204 [8].

The basic bitmap index uses one bitmap for each distinct key value. For scientific data where 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. In such a case, an index for 10^9 rows may require 10^{18} bits. Such an index is much larger than the raw data size and is not acceptable except for the smallest datasets.

A number of different strategies have been proposed to reduce the sizes of bitmap indices and improve their overall effectiveness. Common methods include compressing individual bitmaps, encode the bitmaps in different ways, and binning the original data [11]. FastBit [1] is an open-source software that implements many of these methods. In this work, we choose to use the FastBit software as a representative of general indexing methods. FastBit has been shown to perform well in a number of different scientific applications [15]. In addition, there are also a series of theoretic computation complexity studies to further establish its effectiveness [16, 17].

3 FastQuery Architecture

In this section, we describe the design of FastQuery as well as its user API and file interface for supporting general query selection mechanism for scientific data. Using the FastQuery API, one can quickly select subset of data from a scientific data format file using text-string queries, such as `getNumHits`("energy $> 10^5$ && temperature $> 10^6$"). In contrast to our previous work [4], the FastQuery design is general and will work on multiple data formats; our current implementation is for the widely popular HDF5 and NetCDF data formats. Our design also supports arbitrary hierarchical and multi-dimensional dataset files.

3.1 System Design

The goal of our design is not just to support query and indexing functionality, but more importantly to address the usability, applicability and flexibility issue for scientific data formats. In particular, our design attempts to address the following key challenges: (1) The mismatch between the array data model of users and the relational data model of FastBit. (2) The complex and arbitrary multi-dimensional and hierarchical file structure. (3) The diversity of common scientific data formats. Figure 1 illustrates the system architecture of FastQuery. The role and design of each component in the system is briefly described below.

3.1.1 Query Processor and Index Builder

These are initiated by users to perform the functionality of FastQuery. queryProcessor opens a file with read permissions and processes queries by reading data and indices from the file. indexBuilder opens a file with write permissions and creates indices by writing bitmap indices to the file. Details of the corresponding function calls are available in Section 3.2.

3.1.2 FastQuery Parser

The parser plays the role of interpreting information from users into a format that FastQuery can process. On one hand, it needs to support the array data model and request description from users. On the other hand, it must match the user information to the underlying FastBit relational data model and the hierarchical multi-dimensional scientific format. FastQuery achieves this by defining a simple yet flexible naming convention and query syntax for users to describe their requested variables and query constraints through the API. FastQuery currently supports a query syntax that is similar to the SQL-like range condition query used by FastBit. Details of the naming schema are presented in Section 4.1.

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

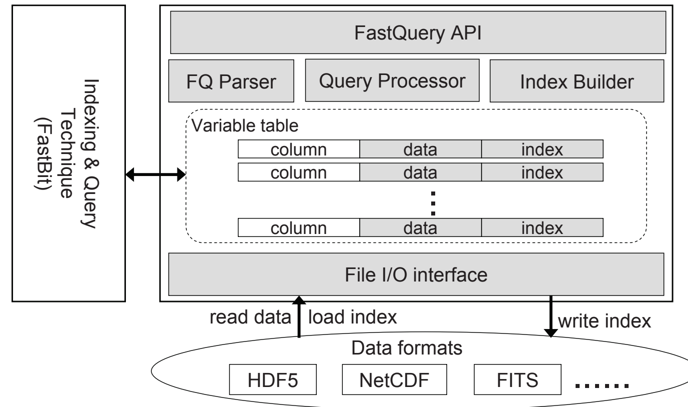

Figure 1: FastQuery architecture

3.1.3 File I/O Interface

This defines a set of unified I/O functions for FastQuery to access data and index from files. As shown in Figure 1, the primary function of the interface is to read data, load indices and write indices. This functionality is implemented by the user API of the underlying data formats, such as HDF5 and NetCDF. Once the interface for the data format has been implemented, FastQuery can provide the data query and indexing functionality for the data format. The design of the interface is crucial to the applicability and efficiency of FastQuery. In Section 3.3, we discuss our design decision in more detail.

3.1.4 Variable Table

This represents the relational data model used by FastQuery and other indexing and query technology [15, 2]. A variable table contains a list of columns where each column maps to a variable (i.e. a dataset in HDF5 terminology), and each row maps to a record of a variable (i.e. data at a mesh point). FastQuery applies FastBit to build and query indices by reading the data and bitmaps associated with each record in the table. In order to resolve queries, all variables in the table must have the same number of rows or records. Since variables in a file may not have the same dimension and size, FastQuery dynamically creates tables according to the user API calls. Hence, tables only contain the columns of variables (or subarray of variables) that users attempt to build index or query on. In contrast, HDF5-FastQuery can only handle files or queries that have the same dimension for all variables.

3.2 FastQuery User API

FastQuery provides an API for users to create indices, query indices and retrieve data using the query results. The function calls related to creating indices are handled by the indexBuilder, while the calls for querying indices and retrieving data are handled by the queryProcessor.

Both indexBuilder and queryProcessor are initiated by users with the arguments of a data format (i.e. HDF5, NetCDF, etc) and two file names, one for data and the other one for index. If the file for storing indices is not specified, FastQuery assumes that the index will be stored on the same file as the data. By specifying different names for the data file and index file, users can load or write indices to a file separate from the original data. This improves the usability of the API since most data files are read-only and stored in a shared repository. A brief description of the main FastQuery function calls is given below.

3.2.1 Creating Index

The main function for creating indices is `buildIndices`. It requires a user-specified binning operation to create indices for a set of variables, and writes the bitmap indices and associated metadata to the index file. If the binning

operation is not provided, the default binning operation is to use one bin for each distinct value of a variable. This function returns the number of indices successfully created and stored.

3.2.2 Processing Query

Three functions are available for users to select data by querying the data. `getNumHits` and `executeQuery` select data based on the constraints of a given query string. `getNumHits` returns the number of records that match the query, while `executeQuery` returns the coordinates of the matching records. In contrast, `executeEqualitySelectionQuery` takes an argument of a list of values, and selects the data of a variable that matches to the given values instead of a query. Both the number and coordinates of the selected data are returned by the function. The returned coordinates can be in the form of either a list of points or bounding boxes. A bounding box is essentially a block of data, and it is represented by the diagonal coordinates of the box. The coordinates returned from the query functions can then be used to retrieve the data at the selected locations from any variable using the function call, `getSelectedData`.

3.3 File I/O Interface

The goal of the FastQuery file I/O interface is to allow FastQuery to access data stored in various scientific formats. Hence, the design of the interface must be simple and general enough to be implemented for different data formats, yet it must be specialized for the purpose of performance, especially in terms of data retrieval. We now discuss our design decisions based on the study of existing data formats.

As shown in Figure 1, FastQuery requires the file I/O interface to read/write index and data to files. Ideally, index data should be distinguished from the user data. Indices are generated by the indexing tool and should not be arbitrarily modified or perhaps even made visible to the users. Therefore, in the file I/O interface, we explicitly separate the functions for handling data from the functions for handling indices to allow different implementations of the two sets of functions.

For data retrieval, we define three functions. `readAllData` retrieves all data of a variable. `readArrayData` retrieves a subarray of variable data specified by a list of starting offsets, counts and strides along each data dimension. Finally, `readPointData` retrieves a list of arbitrary data points specified by a list of coordinates. On one hand, these three functions represent the three types of data selection available in FastQuery API, that is `getAllData`, `getSelectedData` with bounding boxes option, and `getSelectedData` with points option. On the other hand, these three functions are commonly supported by existing data file formats, such as HDF5. Therefore, data could be more efficiently retrieved from file by a single file library call, and the interface can be easily implemented for a file format. Taking HDF5 as an example, `getPointData` is implemented by the HDF5-function `H5Sselect_elements`, and `getArrayData` is implemented by the HDF5-function `H5Sselect_hyperslab`. Hence, the selected data is retrieved with a single file library call.

However, a file format may not always have the functionality that directly match the FastQuery interface functions. For example, NetCDF does not retrieve a list of point data in a single function call. As a result, the `getPointData` for NetCDF is implemented with other un-optimized NetCDF calls. Specifically, when a higher percentage of data is selected, NetCDF reads all data into memory using the call `nc_get_var` rather than extracting the point data. Otherwise, it reads the point data one by one from file using the NetCDF-call `nc_get_var1`. Therefore, the performance of data retrieval could be largely determined by the underlying file library. We have designed the FastQuery I/O API to be flexible in this regard.

For index read/write, the file I/O interface includes functions to read/write a bitmap as well as the information associated with a bitmap, such as the bitmap keys and offsets. `getBitmapKeys/Offsets` and `setBitmapKeys/Offsets` let FastQuery read and write the bitmapKeys/Offsets in a single function call. In contrast, `readBitmap`/`writeBitmap` let FastQuery to read/write a subset of bitmap specified by a starting and an end offset for I/O performance optimization. This is because the bitmap size could over the physical memory space, but FastBit only reads the necessary subset of bitmap into memory to perform query search. Since the existing file libraries do not support data indices, the current implementations of the interface simply store bitmap in a new dataset separated from its original data. However, the developers of HDF5 and NetCDF are in the process of developing indexing interface. Once their design is finalized, the FastQuery interface can be easily implemented to support it.

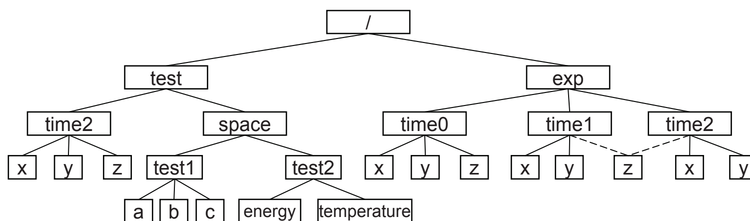

Figure 2: An example of scientific data format file that is organized as an arbitrary hierarchical structure. The file structure is not necessary a tree, because a node could have multiple ancestor links as shown by the dash line in the figure. For this example, we could build indices using the FastQuery API `buildIndices`($varPathStr$, $varNameStr$, $binning$) as follows. `buildIndices` ("/exp/time0", "", "") builds all datasets under group "/exp/-time0". `buildIndices`("", "x", "") builds the four datasets named 'x" in the file, while `buildIndices`("test", "x", "") builds only the variable with path "/exp/test/time2/x". Additionally, `getNumHits`("energy>10") is a valid query to FastQuery in this example, because only one dataset named "energy" in the file, so the variable still can be identified by FastQuery without being specified in its full-path.

4 FastQuery Implementation

In this section, we describe the two key features and implementation details of FastQuery which enable us to support more generalized scientific data formats and flexible data analysis. First, we present the simple yet flexible variable naming schema deployed in the FastQuery API. The naming scheme allows users to specify and identify their querying or indexing variables from arbitrary and complex scientific data format file structure. Secondly, we introduce the subarray feature in FastQuery to provide finer-grained data analysis by allowing users to index and/or query on a subarray in a dataset.

4.1 Variable Naming Schema

In scientific data formats, data can be stored and organized as an arbitrary hierarchical structure in a file as shown in Figure 2. The data format library maintains the structure using a object model of nodes and links, and users access these objects by their unique object ID. However, from the user view of FastQuery, object IDs are hidden from users. Users only have the knowledge of the name of groups or dataset in the file. Therefore, FastQuery needs to provide a variable naming schema or convention that allows users to describe the set of variables that they want to build index or query on. In the following subsection, we describe the naming schema of FastQuery, and how the naming schema is exposed to users through the FastQuery API.

In short, FastQuery uses paths as the naming convention for variables. As shown in Figure 2, because scientific data format is a hierarchical structure consisted of groups and datasets, each path refers to a unique variable. However, a variable may not have a unique path, because an object is allowed to have multiple ancestor links. For example, the variable referred by path "/time/exp1/c" can also be referred by path "/time/exp/c". Fortunately, this one-to-many relation between variable and path does not affect the semantics of FastQuery functions since both paths represent a valid route to reach the variable. In other words, users can index and query the variable using either the path of "/time/exp1/c" or "/time/exp/c". Once a variable path is given to FastQuery, it is parsed by the FastQuery parser into the object Id of the variable. Hence, no duplicate indices would be load or written to file. For performance efficiency, the mapping between individual variable and the associated path is explored by traversing the file structure and cached by FastQuery once the file is opened.

To expose our naming conversion to users, we could have opted to maintain a list of full-path strings of variables as an argument in the FastQuery API. However, such an approach has several drawbacks and limitations. (1) First and foremost, it requires users to have the full-knowledge of the file structure to explicitly specify the full-path of each variable in the argument. This is proven to be difficult, because scientific data files could contain multiple datasets generated from various experiments and tests, and the datasets are normally populated by scripts or programs. As a result, groups could be created arbitrarily for programming purposes and the file structure could get complicated.

Users would ideally like to write queries like “energy>10” without knowing where the variable “energy” is located in Figure 2. (2) Even if users are familiar with the file structure, it is inconvenient for users to specify the paths one by one in their program. Hence, it would require another set of functions to retrieve variable paths with certain condition, such as finding variable paths under a specific group “/exp/time0” in Figure 2, for example. (3) Finally, and perhaps most importantly, using variable full-path prevents users from re-using their query in the file. As showing in Figure 2, data collected from time-varying experiments is commonly stored with the same file structure but under different groups, such as the variables under group “time0”, “time1” and “time2”. During data analysis, the same query like “x>10” is likely be issued by scientists for each of the timesteps or groups. If variable is specified by its full-path, each time users have to construct a new query, like “/exp/time0/x>10” and “/exp/time1/x>10”.

To overcome all the above drawbacks and provide a simple yet flexible naming schema for users, FastQuery API uses a naming tuple of ($varPathStr$, $varNameStr$) to identify and specify variables. Specifically, $varPathStr$ and $varNameStr$ refer to the substring and the suffix-string of a variable path, respectively. Given a naming tuple, FastQuery matches it to a set of variables whose full-path string satisfying the descriptions of the naming tuple. For instance, a naming tuple of (“exp”, “y”) in Figure 2 would match to the three variables with path “/exp/time0/y”, “/exp/time1/y” and “/exp/time2/y”.

If $varPathStr$ or $varNameStr$ is not specified, its corresponding matching requirement is simply not considered by the FastQuery parser. Hence, all variables in a file can be simply referred by an empty naming tuple or by not specifying the argument of $varPathStr$ and $varNameStr$ in the FastQuery API. Similarly, users can create indices for all variables under a group by only specifying the API argument $varPathStr$ as the name of the group, or create indices for a specific variable collected from different experiments by only specifying the $varNameStr$ API argument as the name of the variable.

With our naming tuple schema, query re-use is also naturally supported by having a $varPathStr$ argument with the query API. The variables associated with the query is again identified by the $varNameStr$ specified in the query, and the $varPathStr$ argument. For instance in Figure 2, `getNumHits`(“x>0 && y>0”, “time0”) returns the number of record larger than 0 in both variables “/exp/time0/x” and “/exp/time0/y”. Clearly, users can re-use the query “x>0” to different groups by replacing the $varPathStr$ argument to “time1” or “time2”.

In summary, our naming schema allows variables to be specified by the names of the datasets or groups that users are interested in. If needed, users can further filter out unwanted variables by adding more information into the naming tuple. Query re-use is also naturally supported by the naming schema in the API. Finally, whether a naming tuple specified by user is valid for a FastQuery is determined the semantics of the API. For example, `getSelectedData` is supposed to retrieve data for a specific variable, so the specified naming tuple must match to a unique variable in the file, otherwise the input is considered as invalid and is rejected by the FastQuery API.

4.2 Subarray Specification

FastQuery deploys subarray as a means to provide finer-grained data analysis by allowing users to index and/or query on a subarray in a dataset. As showing by the climate study in Section 5.3 where each mesh point in the dataset is corresponding to the measurement at a latitude and longitude location, scientists can often be interested in only a subregion of the dataset rather than the whole extent. Without the support for selecting subregions (or what we call subarrays), users must perform their queries at the per-dataset level, then filter out the unnecessary results outside the subarray region afterwards. As showing in our evaluation results in Section 5.1.2, these actions could complicate the query process and cause significant performance overhead.

FastQuery adopts a similar subarray specification used by Fortran and other programming languages where a subarray is specified by the general form of “$[lower : upper : stride]$”. The $lower$ and $upper$ indicate the first and last record position of the subarray in a dataset. The $stride$ is the step size between $lower$ and $upper$. For a n-dimensional dataset, the subarray range of each dimension must also be specified and separated by comma in the bracket. Figure 3 shows three subarrays examples specified as “data[0:1,1:3]”, “data[0:2:2,5:9:2]” and “data[4,:]” in a 5 by 10 two-dimensional dataset of variable “data”.

The specified subarray range along with its associated variable name is passed into FastQuery through the $varNameStr$ argument of user API. The subarray information is then extracted from the argument string by the FastQuery parser, and a FastQuery column object is created to perform indexing and querying for the subarray. Therefore, a subarray

	0	1	2	3	4	5	6	7	8	9
0	1	1	2	4	1	-5	1	5	4	-1
1	-1	-2	-3	-4	-5	7	-5	7	3	2
2	-1	-1	-1	-1	-1	-1	3	5	3	-2
3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
4	-1	-4	2	5	2	7	-1	2	1	3

Figure 3: An example of *subarray* in a 5*10 two dimensional dataset for variable "data". The subarray at the top left is "data[0:1,1:3]", the one at the top right is "data[0:2:2,5:9:2]", and the one at the bottom is "data[4,:]". FastQuery allows users to build indices for each of the subarrays, or to query subarrays with the same dimensions. For instance, `getNumHits`("data[0:1,1:3]>0 && data[0:2:2,5:9:2]>0") returns 1, because there is only one record satisfying the query constraints in both subarrays as circled in table at the relative coordinates (1,2).

is treated just like a dataset inside FastQuery, except the data coordinate is relative to the region of the subarray. For example, in Figure 3, users can call `buildIndices`("data[0:1,1:3]") to create a bitmap index for the specified subarray. Note that the bitmap index created for a subarray can only be used to process the query for the exactly same subarray, such as getNumHits("data[0:1,1:3]>0"), because bitmaps are compressed and encoded according to data. When a query involves multiple variables, the subarray of each variable can be specified individually, but the dimension of the subarrays in a query must be the same. Take Figure 3 as an example, `getNumHits`("data[0:1,1:3]>0 && data[0:2:2,5:9:2]>0") is a valid query since both subarrays have the dimension 2*3, and the query produces only 1 hit which is at the relative coordinate (1,2) of both subarrays.

5 Experiments

In this section, we evaluate FastQuery performance for both synthetic and simulation datasets. We also apply FastQuery to two scientific analysis tasks. All our tests were performed on a workstation with 2.67 GHz Intel processor with 4 GB of memory and a commodity SATA disk.

5.1 Synthetic Data

In our first set of experiments, we generate synthetic data with 100 million records and store the data in an one-dimensional HDF5 dataset. The values of the records are uniformly random distributed in the range of [0:99]. The size of the original dataset is 382MB. We use the dataset to evaluate FastQuery by reporting the response time of processing two sets of queries. The first set of queries is basic range query, such as "energy>100". The second set of queries is subarray query which attempts to search on a subarray of a dataset. For instance, an subarray query that only searches the first 1000 records in dataset "energy" would be "energy[0:1000]>100". In the basic range query case, we show FastQuery utilizes the FastBit indexing technology to accelerate the query response time by at least a factor of 2.5. In the subarray query case, we show FastQuery further reduces the query response time comparing to HDF5-FastQuery, and the improvement increases as the size of subarray decreases.

5.1.1 Basic Range Query

We generate queries that cover the entire domain range of our synthetic data. To process the queries, FastQuery builds the equality-encoded [1] bitmap indices for all the data in the dataset. The size of the bitmap indices is 571MB and they are created in 13.48 seconds. As shown in the previous analysis [4], the size of compressed bitmap indices is larger when variable values are uniform random distributed.

Figure 4 plots the response time of queries versus the query selectivity, which is the percentage of records selected by a query from the dataset. For instance, a query selectivity of 50% means that 50 million records satisfy the query constraint, and these records are called *hits*. In the figure, we use HDF5 to represent the traditional data analysis where all data values are retrieved from file using the HDF5-calls(`H5Dread`) and sequentially compared with the query constraints.

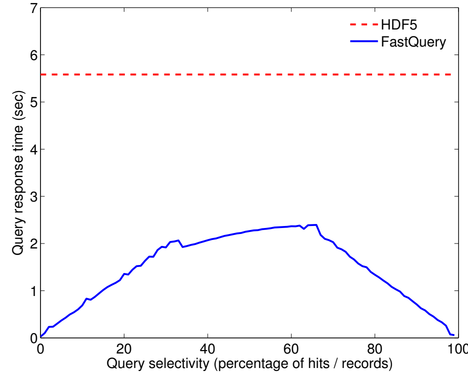

Figure 4: Response time of one-dimensional queries on synthetic dataset.

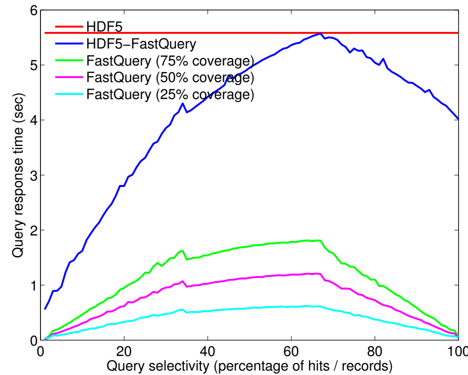

Figure 5: Response time of one-dimensional subarray queries on synthetic dataset.

FastQuery processes queries by only reading the necessary compressed bitmaps from file to answer each query. Hence, the query response time of FastQuery is increasing up to 50% selectivity (50 million hits), because for queries with higher selectivity, more bitmaps must be read from disk and OR-ed together. However, the response time after 50% selectivity is decreasing, because these queries are evaluated by negating the query expression. For instance, a query "A>80" is evaluated as "NOT(A $\leq$ 80)". Thus, the selectivity of the query is 20% as oppose to 80%. In contrast, the query response time of HDF5 stays the same because it simply scans through all data for every query. As a result, FastQuery significantly reduces query response time. As shown in the figure, even at the worst case of selectivity 50%, FastQuery is almost 2.5 times faster than HDF5. In average, FastQuery is about 10 times faster than HDF5.

5.1.2 Subarray Query

Next, we evaluate FastQuery with subarray queries to show our performance improvement over HDF5-FastQuery. The subarray queries are generated to search data on three specific subarrays which cover 25%, 50% and 75% of the dataset, respectively. For each of subarrays, we generate queries that has 0% to 100% selectivity from the data of subarray, and we report their response time.

For FastQuery, bitmap indices for each of the three subarray are created, and the bitmap sizes are 428MB, 286MB and 143MB, respectively. The bitmap size decreases as the size of subarray is reduced because less data records are built in the bitmap. As shown, the bitmap size built for the 25% coverage subarray (i.e. 143MB) is 4 times smaller than the bitmap size built for the whole dataset (i.e. 571MB). With the built indices, a query generated in an subarray, say [0:25,000,000], is also processed by using the indices corresponding to that particular subarray. In addition, the hit count of each query is retrieved by a single FastQuery-call, such as `getNumHits`("data[0:25,000,000]>50"). On the other hand, because HDF5-FastQuery does not support subarrays, it can only create indices for the whole dataset, then

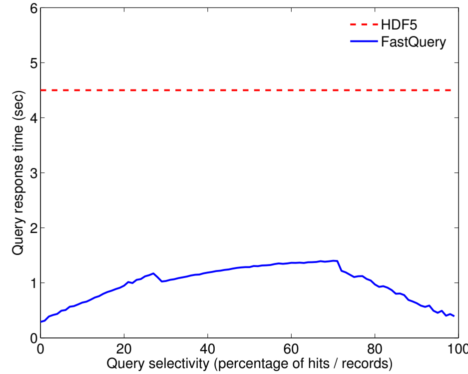

Figure 6: Response time of one-dimensional queries on simulation dataset.

process the subarray queries in two steps. First, the queries are processed using the dataset index, and the coordinates of the selected data are returned. Then the selected data must be scanned through one-by-one to discard the data outside the subarray of queries.

Figure 5 plots the response time of processing subarray queries using FastQuery, HDF5-FastQuery and HDF5 (i.e. `H5Dread`). For HDF5-FastQuery and HDF5, the size of subarrays has limited impact to the response time, because the size is only used as filter conditions to discard data outside the subarray. Hence, the figure only shows their results of the subarray that covers 75% of the dataset, while the results of FastQuery are shown for each of the three subarrays.

For the results of HDF5-FastQuery, again we observe the response time is increasing before 50% selectivity, then decreasing afterwards for the same reason explained for Figure 4. However, because HDF5-FastQuery has to scan the query results to filter out the data outside the query subarray, its response time suffers from the overhead penalty. The overhead penalty grows as the selectivity increases, since more coordinates must be retrieved and scanned. As a result, the response time decreases at a flatter rate as the selectivity is getting close to 100%. The fact that HDF5-FastQuery has to scan through the selected data also greatly reduces the advantage of using indexing. As shown from the figure, at the worst case, HDF5-FastQuery has the same response time as HDF5.

On the other hand, FastQuery achieves much faster response time than the others approaches, because it is still able to take full-advantage of the indexing to process queries without scanning through data. Furthermore, the response time of FastQuery decreases linearly with the size of subarray, because the bitmaps size corresponding to the subarray is also smaller and takes less time to read from file. Under 75% coverage subarray, FastQuery accelerates the response time of HDF5-FastQuery by a factor of 2.5 in the worst case, and by a factor of 5.5 in average. Under 25% coverage subarray, FastQuery accelerates the response time of HDF5-FastQuery by a factor of 7.5 in the worst case, and by a factor of 13 in average. Therefore, by supporting subarrays, FastQuery significantly improves its query performance and usability.

5.2 Scientific Simulation Data

In our second set of experiments, we evaluate FastQuery on a Laser Wakefield data [3, 10]. The data size is about 3GB, and it has 7 variables. Each variable is one-dimensional with 50 million records. The data of all 7 variables is stored in a single 50,000,000 by 7 two-dimensional dataset of a HDF5 file where each column represents a single variable. Although this data storage format does not use the best organization and metadata scheme, we do commonly observe simulation datasets dump data in this interleaved format. In order to use the HDF5-FastQuery tools in the past, we had to resort to converting the dataset into an HDF5 file conforming to the H5Part data model. The resulting duplicate file would have separate datasets for each variable. While the scheme was desirable from an organization point of view, the conversion process would take a few minutes to hours depending on the data size, and the resulting duplicate file had to be maintained and stored. This caused data management and resource usage problems. However, with FastQuery, we can use the file as it is, and build indices for each of the variables by using the *subarray* feature introduced in Section 4.2.

For the aforementioned dataset, we created the bitmap indices in 22 minutes with the size of 857MB. Additionally, the indices were written in a separated index file, and stored locally to users disk space, while the original data file was not modified and keep in a shared disk space. All of these features were simply unavailable in the previous implementation of HDF5-FastQuery.

For the simulation data, we demonstrate our performance with a set of randomly generated queries. The queries are generated by randomly selecting the variable and the query range. The variable name is identified by its column index. For instance, "data[:,0]>0" means the value of the first variable must be larger than 0. We perform 100 such queries and reports results in Figure 6. Again, we observe similar results as the synthetic data in Figure 6. FastQuery still significantly improves the response time of HDF5 by a at least a factor of 3.2 and a factor of 5 on average.

5.3 Case Studies

In this section, we highlight two case studies to show how FastQuery can be used to complete scientific data analysis tasks. The first study comes from the field of climate modeling: we process climate simulation data stored in NetCDF files and identify an extreme weather pattern known as an atmospheric river. In the second study, we look at accelerator modeling data stored in HDF5 files and study energetic particle bunches.

5.4 Atmospheric Rivers

Atmospheric rivers is a weather phenomenon where a significant amount of tropical moisture in the air is transported far away into extratropical regions [7], an example is shown in Figure 7. This phenomena produces unusually heavy amounts of rainfall and could cause catastrophic damage to the local communities [9], such as the western coast of north America. Because of their impact, it is desirable that global climate model simulations faithfully capture atmospheric rivers. We describe the basic query operations needed to detect and characterize atmospheric rivers in global climate simulation output.

The simulation data we use is produced by fvCAM, a version of the Community Climate Model adapted for high-performance computing [14]. The data produced from fvCAM is on a fixed latitude-longitude mesh with quarter degree spacing between neighboring mesh points, and the data of each measurement variable from the simulation is stored on a 1440*720 two-dimensional NetCDF dataset. Based on earlier work on detecting atmospheric rivers in the observational data [7], the key simulated quantity associated with the phenomenon is the concentration of water vapor in the atmosphere. In particular, an atmospheric river is basically a region with high vapor concentration outside of the tropical zone. Hence, it can be identified by expressed a range query such as "select * from simulation data where vapor concentration $>$ threshold and ($60^\circ >$ latitude $> 23^\circ$ or $-60^\circ <$ latitude $< -23^\circ$".

Since data is located in a dataset according to its latitude and longitude position, the condition of above query "$60^\circ >$ latitude $> 23^\circ$" is converted to the mesh points between 452 and 600 along the latitude and similarly "$-60^\circ <$ latitude $< -23^\circ$" is converted to the mesh points between 120 and 268. Let "vapor" denote the vapor concentration stored in the data file. Instead of indexing "vapor" directly, our software allows us to index only the relevant port of the data, "vapor[:,120:268]" and "vapor[:,452:600]", then search data from the subarray with queries "vapor[:,120:268]$>$ threshold" and "vapor[:,452:600]$>$ threshold". We created the subarray index using the current FastQuery implementation and performed the analysis task.

5.5 Laser Wakefield Particle Accelerator

Particle acceleration using plasma laser wakefields is a promising new development in the area of particle physics. In contrast to conventional electromagnetic accelerators which needs tens to thousands of meters for accelerating particles, Laser Wakefield accelerators can achieve much higher energy gradients in a very short distance. Laser wakefield simulations model the behavior of individual particles, and the electric and magnetic fields associated with the plasma. Scientists at the LBNL LOASIS facility are currently using the VORPAL simulation code to model physical experiments, gain deeper understanding of the physical observations and optimize methodology for future experiments [3, 10].

VORPAL simulations currently generate vast quantities of data in the HDF5 file format. The sizes of the dataset are proportional to the number of particles used in the simulation and the discretization of the electromagnetic fields.

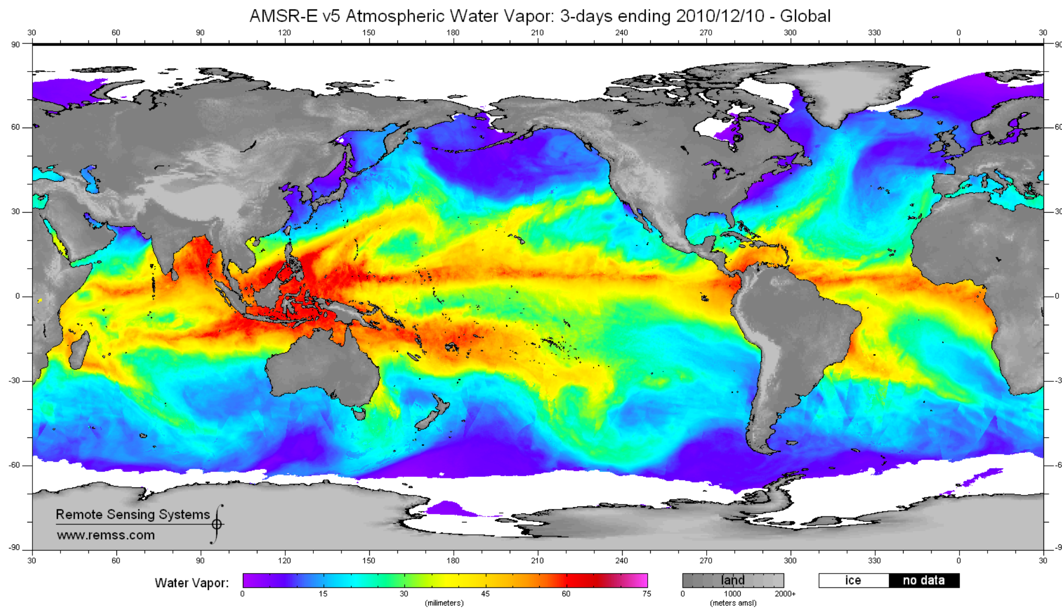

Figure 7: A plot of the water vapor concentration with an atmospheric river reaching from Hawaii to California (Image from Remote Sensing Systems).

Typical simulations with 100 million particles can produce datasets of the order of 10 GB per timestep, and dozens of timesteps. For this case study, we accelerate a common analysis operation that is of importance to the particle physicists: find all particles that have undergone significant wakefield acceleration, and then trace then back virtually through time. It is typical to look a few hundreds to thousands of particles in the accelerated bunch; tracing these particles back in time reveals acceleration dynamics and potentially provides insights into how to better design experiments (to in turn produce particles with high energy).

In terms of queries, we perform a query of the form "px$> 1e10$" at one of the later timesteps in the simulations to isolate the highly-accelerated particles. After we obtain the particles, we read their unique particle identifier information, and perform an equality selection query at all of the earlier timesteps. These traces can then be stitched together (as shown in Figure 8). The data is also used to evaluate our system performance in Section 5.2.

6 Conclusions

We have designed and implemented FastQuery, a general indexing and querying system for scientific data. FastQuery utilizes the FastBit bitmap indexing technology to support semantic query on common data format, such as HDF5 and NetCDF. We significantly extended our previous work, HDF5-FastQuery, by addressing the usability, applicability and flexibility issues of the indexing and querying system. Through the evaluations and case studies, we demonstrated that the FastQuery implementation can be used on multiple data formats, arbitrary file structure, and various scientific applications. Our experiments also show that FastQuery substantially simplifies the data analysis process and improves query performance for subarray queries.

We would like to pursue three directions in our future work. First, we are working with HDF5-developers on a tighter integration between indexing and the HDF5 data format. The lessons learned from the design and implementation of FastQuery will guide this endeavor. Second, we would like to support more common data formats besides HDF5 and NetCDF. Finally, we would like to further improve the performance of the system. For example, we plan to investigate building indices and resolving queries on distributed multi-core systems.

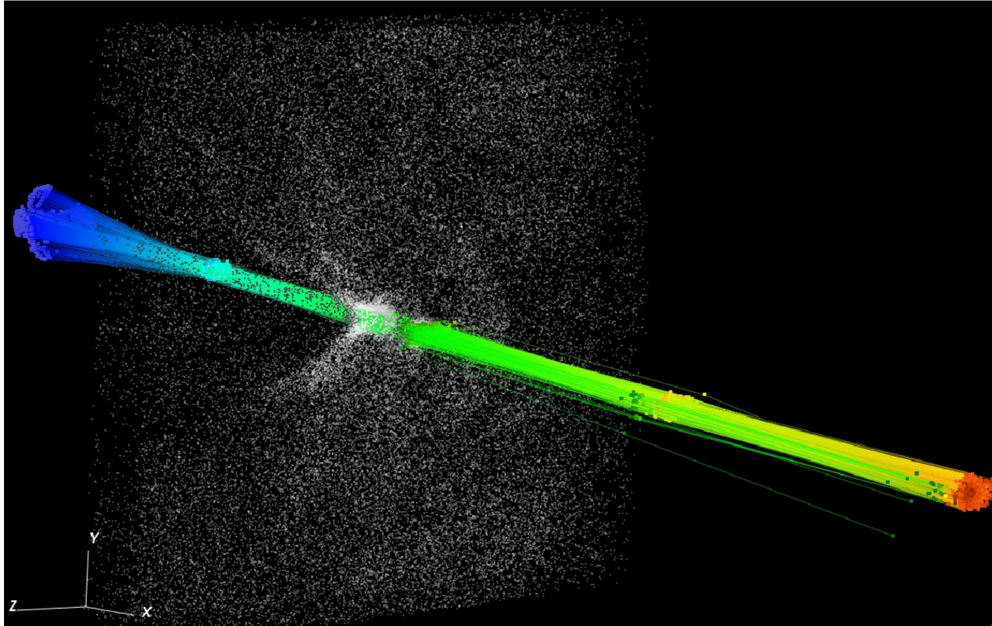

Figure 8: Trace of energetic particles from a Laser Wakefiled simulation.

Acknowledgments

The authors wish to thank John Shalf, Quincey Koziol and E. Wes Bethel for their helpful discussions leading up to the design and specification of FastQuery software. This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, and used resources of the National Energy Research Scientific Computing Center, which is also supported by the above contract.

References

[1] C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, SIGMOD '98, pages 355–366, New York, NY, USA, 1998. ACM.

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

[3] C. G. R. Geddes, C. Toth, J. van Tilborg, E. Esarey, C. B. Schroeder, D. Bruhwiler, C. Nieter, J. Cary, and W. P. Leemans. High-quality electron beams from a laser wakefield accelerator using plasma-channel guiding. *Nature*, 431:538–541, 2004.

[4] L. Gosink, J. Shalf, K. Stockinger, K. Wu, and W. Bethel. HDF5-FastQuery: Accelerating complex queries on HDF datasets using fast bitmap indices. In *SSDBM*, pages 149–158, 2006.

[5] T. Hey, S. Tansley, and K. Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft, Oct. 2009.

[6] R. Musick and T. Critchlow. Practical lessons in supporting large-scale computational science. *SIGMOD Rec.*, 28(4):49–57, 1999.

[7] P. J. Neiman, F. M. Ralph, G. A. Wick, Y. Kuo, T. Wee, Z. Ma, G. H. Taylor, and M. D. Dettinger. Diagnosis of an intense atmospheric river impacting the pacific northwest: Storm summary and offshore vertical structure observed with COSMIC satellite retrievals. *Monthly Weather Review*, 136(11):4398–4420, 2008.

[8] 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.

[9] F. M. Ralph, P. J. Neiman, G. A. Wick, S. I. Gutman, M. D. Dettinger, D. R. Cayan, and A. B. White. Flooding on california's russian river: Role of atmospheric rivers. *Geophysical Research Letters*, 33:L13801, 2006.

[10] 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*, pages 1–12, Nov. 2008.

[11] A. Shoshani and D. Rotem, editors. *Scientific Data Management: Challenges, Technology, and Deployment*, chapter 6. Chapman & Hall/CRC Press, 2010.

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

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

[14] M. F. Wehner, L. Oliker, and J. Shalf. Towards ultra-high resolution models of climate and weather. *IJHPCA*, 22(2):149–165, 2008.

[15] 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.

[16] K. Wu, E. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems*, 31:1–38, 2006.

[17] K. Wu, A. Shoshani, and K. Stockinger. Analyses of multi-level and multi-component compressed bitmap indexes. *ACM Trans. Database Syst.*, pages 1–52, 2010.