

# FastQuery: A Parallel Indexing 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.

# FastQuery: A Parallel Indexing System for Scientific Data

Jerry Chou, Kesheng Wu and Prabhat  
Lawrence Berkeley National Laboratory  
Email: {jchou, kwu, prabhat}@lbl.gov

July 29, 2011

## Abstract

Modern scientific datasets present numerous data management and analysis challenges. State-of-the-art index and query technologies such as FastBit can significantly improve accesses to these datasets by augmenting the user data with indexes and other secondary information. However, a challenge is that the indexes assume the relational data model but the scientific data generally follows the array data model. To match the two data models, we design a generic mapping mechanism and implement an efficient input and output interface for reading and writing the data and their corresponding indexes. To take advantage of the emerging many-core architectures, we also develop a parallel strategy for indexing using threading technology. This approach complements our on-going MPI-based parallelization efforts.

We demonstrate the flexibility of our software by applying it to two of the most commonly used scientific data formats, HDF5 and NetCDF. We present two case studies using data from a particle accelerator model and a global climate model. We also conducted a detailed performance study using these scientific datasets. The results show that FastQuery speeds up the query time by a factor of 2.5x to 50x, and it reduces the indexing time by a factor of 16 on 24 cores.

## 1 Introduction

Modern scientific applications produce and consume vast amounts of data [11, 23]. In many cases, the essential information is contained in a relatively small number of data records. For example, the Intergovernmental Panel on Climate Change’s (IPCC) is in the process of generating tens of petabytes ( $10^{15}$  bytes) for its fifth assessment report (AR5)<sup>1</sup> but critical information related to important events such as hurricanes might occupy no more than a few gigabytes ( $10^9$  bytes), an example of which is shown in Figure 1. The ability to directly access the necessary information instead of going through all data records can significantly accelerate analyses. This requirement for efficiently locating the interesting data records is indispensable to many data analysis procedures.

The best-known technology for locating selected data records from a large dataset is the database indexing [23, Ch. 6]. Typically, the database indexing techniques are available through large centralized database management systems (DBMS), however, scientists are often unwilling or unable to place the datasets under the control of DBMS [15]. For example, IPCC has settled on storing all AR5 data in the NetCDF [26] file format. Instead of requiring the users to place their data into DBMS, we propose to build indexes along with the existing data to accelerate the search operations. We implement this functionality in a system named FastQuery.

An essential challenge in this work is how to map the data model assumed by the index structures to the data model used in scientific data sets. More specifically, the indexing data structures assume a relational data model [18], while scientific data formats such as NetCDF [26] and HDF5 [25] typically use the array data model. The second challenge that we address is how to efficiently work with massive datasets. The main contributions of the paper are as follows:

---

<sup>1</sup>More information available at <http://www.ipcc.ch/>.

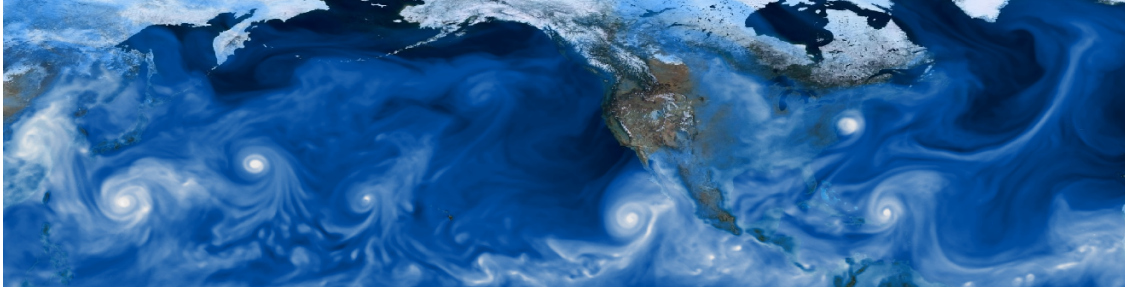


Figure 1: Visualization of a global climate simulation with multiple storms. The regions with storms cover only a small portion of the globe.

- We define a mapping between the relational model followed by the indexing software and the array model used by most scientific applications.
- We define a simple abstraction layer between FastQuery and the scientific format file libraries, so that our system can be applicable to common scientific data formats quickly and efficiently.
- We deploy a simple yet flexible naming scheme for users to specify datasets in an arbitrary arrays and subarrays in a scientific dataset.
- We implemented threading capability in FastQuery to take advantage of many-core architecture for processing large datasets.
- We extensively evaluate the performance of our new indexing software using realistic scientific dataset, and demonstrate the usability of our system through two application case studies.

The rest of the paper is structured as follows. We review the related work on scientific data formats and indexing in Section 2. We describe the design of FastQuery in Section 3, show an extensive performance evaluation of FastQuery in Section 4, and discuss two use cases in Section 5. We conclude with some thoughts for future directions in Section 6.

## 2 Related Work

To accelerate accesses to data, we bring database indexing techniques to scientific data. In this section, we briefly review related work on scientific data formats and database indexing, and point out the distinct features of our current work.

### 2.1 Scientific Data Formats

The database management systems generally follow the relation data model [18] that treats user data as tables with rows and columns. A typical scientific application views its data as arrays, often each physical quantity such as temperature and pressure is stored in its own array. In these cases, the columns in the relational data model can be easily mapped to variables in scientific applications. However, in many cases, an array may contain many physical quantities. This would make it necessary for us to work with subarrays. Earlier work on indexing scientific data have only considered indexing whole arrays [9]; we believe that we are the first ones to consider indexing subarrays. We will describe how to address the mapping of rows and tables in the next section.

Since scientific applications store their data as arrays, the commonly used scientific data formats are designed to store arrays efficiently. We use two of the most popular formats, NetCDF [26] and HDF5 [25], to demonstrate the flexibility of FastQuery design. Both NetCDF and HDF5 are designed to be portable and self-describing. With appropriate data access libraries, they can be transported across machines.

| RID | X | bitmaps     |             |             |             |             |
|-----|---|-------------|-------------|-------------|-------------|-------------|
|     |   | $b_0$<br>=0 | $b_1$<br>=1 | $b_2$<br>=2 | $b_3$<br>=3 | $b_4$<br>=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           |
| 7   | 1 | 0           | 1           | 0           | 0           | 0           |
| 8   | 4 | 0           | 0           | 0           | 0           | 1           |

Figure 2: The logical view of a sample bitmap index shown as eight columns on the right.

## 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 of scientific datasets, researchers are extending indexing techniques for scientific applications as well [23]. Traditional indexing techniques such as B-tree are designed primarily to accelerate access to individual data records, such as looking for a customer’s bank record [4]. In contrast, a query on scientific data typically returns 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 particles. Furthermore, scientific datasets are often produced or collected in bulk, and 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 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 first by O’Neil [17].

The basic bitmap index uses one bitmap for each distinct key value as illustrated in Figure 2. 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 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 [23, Ch. 6]. FastBit [28] is an open-source software package that implements many of these methods. In this work, we choose to use FastBit as a representative of general indexing methods. FastBit has been shown to perform well in a number of different scientific applications [28]. In addition, there are also a series of theoretic computation complexity studies to further establish its soundness [29, 30].

## 2.4 Distributed Indexing

The benefits of performing data analyses in parallel have been recognized since the dawn of parallel computing [10, 19]. Unlike many other high-performance computing applications, database operations are dominated by data accesses rather than computations performed by the CPUs. Optimal I/O performance is, hence, the most critical consideration when designing parallel database systems. A number of parallel database vendors have opted for custom hardware to achieve this objective. For example, Netezza uses an active

storage approach where the disk controllers are modified to carry out certain database operations [5]. Teradata employs a specialized interconnect, called BYNET, to increase the bandwidth among data access modules [2, 7]. Alternatively, a number of researchers proposed to store the data in main memory [6, 14]. Such hardware-based solutions are typically not available to scientific users, or are unable to handle a large volume of data. To maximize the impact of our indexing system, we use a purely software-based approach with threading to take full advantage of commodity hardware.

Among the approaches that use commodity hardware, the “shared-nothing” approach has been demonstrated to be the most effective for data management applications [24]. Our work follows this strategy by partitioning an array provided by the user into a number of disjoint sub-arrays. We build and use the index for each sub-array independently. This approach minimizes the coordination required among the parallel tasks. Since tasks can read (and process) data in parallel, the underlying file system has an opportunity to maximize data access throughput using this approach.

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 and have implemented our parallel indexing system based on the sequential software FastBit [28].

In this work, we explore thread parallelism for the emerging many-core architecture. This threading approach complements the on-going effort of MPI-based parallel approach [3]. Issues such as how threading is affected by the I/O system and how to integrate thread parallelism with MPI parallelism are to be explored in the future.

### 3 FastQuery

Our design goal for FastQuery was to perform searching operations on scientific data efficiently with database indexing technologies. To achieve this, we need to address two challenges: (1) mapping the array data model used by common scientific data format libraries to the relational data model assumed by the indexing data structure, and (2) parallelizing indexing functions to take full advantage of many-core architectures.

We map the arrays into the relational data model using the following three assumptions: (I) each array element is mapped to a row in a relational table, (II) elements of a multidimensional array are linearized following C/C++ convention, that is the first dimension is the slowest varying dimension while the last dimension is the fastest varying dimension, and (III) an arbitrary number of arrays or subarrays can be placed into a relational table as long as they have the same number of elements. A key piece of information most indexing data structures rely on the relational data model is the the row identifiers. Assumption (II) defines a row identifier from any array, and assumption (III) allows us to use the row identifiers defined for different arrays to refer to the same rows. In many simulations, the physical quantities such as temperature and pressure computed are stored in separate arrays, an element in one of these arrays refer to values associated with a point in space or a cell in space. In such a case, it is natural to select the pressure values in regions of space where temperate is greater than 1000. Under the above assumptions, this selection operation can be expressed as a SQL statement as follows “SELECT pressure FROM simulation-data WHERE temperature > 1000”. In general, our approach defines a mechanism for expressing SQL-style selections on arrays. It is up to users to ensure that the selection is actually meaningful.

To produce a concrete design for a software package, FastQuery addresses the following specific issues: (1) how to handle variables organized in arbitrary file layout, (2) how to support array-based specification and interface, (3) how to parallelize the indexing operations, and (4) how to work with different array-based data formats in a unified way. In the remainder of this section, we give an overview of the FastQuery design and then describe each of these four issues in turn.

#### 3.1 System Design

FastQuery consists of a few key components as shown in Figure 3. The main purpose and responsibility of each component is briefly described below:

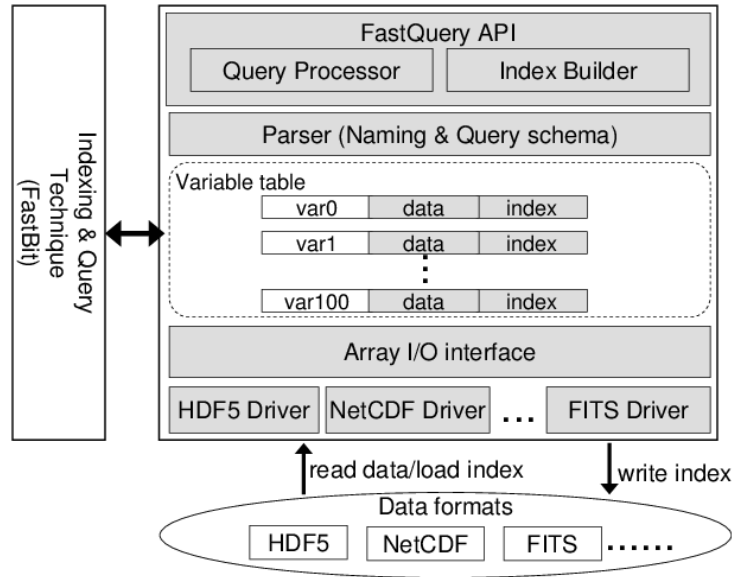


Figure 3: FastQuery architecture

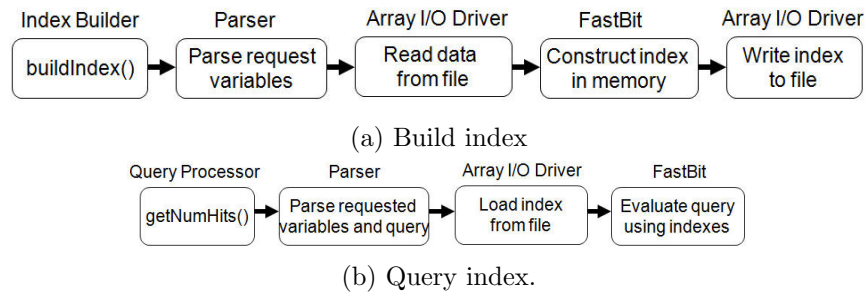


Figure 4: Processing steps in FastQuery.

- FastQuery API defines the main functions for indexing and querying. The two set of functions are implemented in two C++ classes named IndexBuilder and QueryProcessor.
- Parser defines the naming scheme for variables and syntax for query expressions. The naming scheme will be explained later in this section. In the current implementation, a query consists of a list of range conditions on arithmetic operations such as `"/time1/x[1, :] > 10*2 && /time2/x[1, :] > 7+5"`.
- FastBit provides the indexing and querying functions used in FastQuery. More information about FastBit can be found in Section 2.3.
- Array I/O interface defines a unified interface for the underline array-based file formats. At this time, FastQuery can support NetCDF and HDF5 files.
- Variable table represents the relational data model used by FastQuery and other indexing and query technology [28, 4]. The array data read from file is interpreted and temporarily stored in tables dynamically created for each query or index operation.

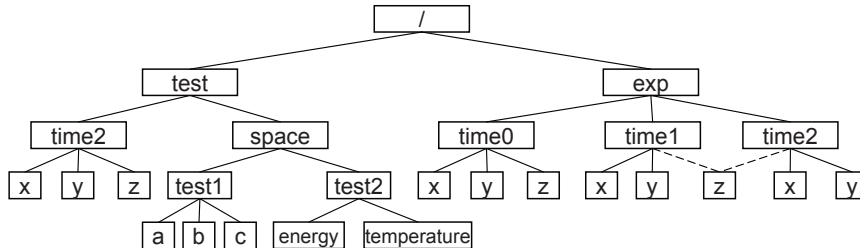


Figure 5: An example of scientific data format file layout. In FastQuery, variables are identified by their full-paths. So, variable with path “\exp\time1\z” can also be recognized by FastQuery with path “\exp\time2\z”. But FastQuery allows users to specify variable with a naming tuple (*varName*, *varPath*), which will match to variables whose path satisfies to regular expression “\\*\varPath\\*\varName”. Therefore, FastQuery can answer a user-friendly query such as “temperature > 3” without asking for additional path information, because there is only one variable in the file that matches our regular expression.

Figure 4 (a) and (b) further illustrate the process of indexing and querying in FastQuery. Upon receiving the buildIndex call from users, FastQuery first parses the variable name specified by the user to identify the corresponding dataset and array range. Then FastQuery creates a variable table that provides the relational table view needed by the underlying indexing building software FastBit. Finally, the bitmap indexes are stored as arrays in a user specified location.

Similarly, during query processing, FastQuery starts with parsing the user query string to determine the query conditions and variables. Then, a variable table is created from variables involved in the query, and the indexes of the variables are loaded from file into the table. Finally, FastQuery uses FastBit to resolve the query and return the number of hits or coordinates of the selected data from query.

### 3.2 Variable naming scheme

As shown in Figure 5, scientific formats commonly store data in a hierarchical layout. This layout provides users the flexibility to organize data in the most convenient way for their applications. However, it also introduces additional complexity and difficulty when the data has to be used by different applications. To overcome the issue, one approach that has been adapted in several scientific data analysis tools is to enforce an internal file layout, such as H5hut [13]. While such approach could simplify the issue, it limits the usability of a file, and the restricted layout may not be applicable to all the science datasets or future applications. In contrast, we design FastQuery to work with arbitrary file layout.

Our basic approach is to require a full-path for each variable. Any path that refer to a unique variable can be used, and every variable can be identified by any of its paths. For instance, in the file layout shown in Figure 5, for a full-path “\exp\time1\z”, FastQuery identifies an unique variable in the file, and the variable can also be referred by its alternate path “\exp\time2\z”.

FastQuery also provides a regular expression like naming scheme to simplify specification of variable names. In particular, the current implementation lets users specify variables by a naming tuple (*varName*, *varPath*). Then FastQuery then internally parses and matches a naming tuple to variables whose full-path satisfying the regular expression “\\*\varPath\\*\varName”. For instance, in the file shown in Figure 5, using the *varName* “temperature” is sufficient for FastQuery to identify the variable “\test\space\test2\temperature”. Therefore, the FastQuery interface improves the flexibility and usability of the query system without exposing unnecessary details to users.

### 3.3 Array model and Subarray specification

Scientific data is commonly represented and stored in the form of a set of multi-dimensional arrays because scientists tend to generate and manipulate data in the array model. Hence, a key objective of FastQuery



|   |    |    |    |    |    |    |    |    |    |    |
|---|----|----|----|----|----|----|----|----|----|----|
|   | 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 6: Examples of FastQuery array specification for multi-dimensional data variable. In FastQuery API, “A[4, :]” refers to the data in the last row of the dataset, “A[0:2, 1:4]” refers to the continuous subarray region at the top-left corner, and “A[0:3:2, 5:10:2]” refers to the striped subarray region at the top-right. FastQuery considers these user specified arrays as individual variables. Thus, FastQuery can process a query like “A[0:2, 1:4] > 0 && A[0:3:2, 5:10:2] > 0”. The query will result in 1 hit as circled in the figure, and the resulting coordinate of the selected element will be [0, 1], which is the relative coordinate position in the two user specified arrays.

is to present a array model interface for users to manipulate and query their data, so that FastQuery can more seamlessly connect with the scientific applications and use cases. For example, instead of storing each variable in a separate dataset, scientists normally just dump their data to a single 2-dimensional dataset where each column represents one variable. As a result, users cannot access the values of a particular variable without specifying an array region of the dataset. Furthermore, some scientific applications might only be interested in one particular region of the data. For instance, climate data is commonly stored in a 2-dimensional dataset, and data points are placed in the dataset according to their latitude and longitude geographical positions. But for detecting atmospheric rivers [16, 21], scientists are only interested in the water vapor values in the area of tropical regions, which corresponds to a particular data array region in the dataset. Thus, the scientist would like the query system be able answer a query of the form “vapor[:, 452:600] > threshold”. Otherwise, users would have to send queries like “vapor > threshold”, then filter out the resulting data by the associated coordinates. As shown by our experimental evaluations in Section 4.2, without the support of array specification (i.e. such as using HDF5-FastQuery [9]), the post-processing overhead could easily dominate the overall query response time and limits the performance benefit from using an indexing/query system.

The approach of FastQuery is to allow users explicitly specify data with an array description through the FastQuery API. Specifically, FastQuery uses the array specification, *var[start : end : stride]*, which is similar to the syntax used by Fortran and other programming languages used by scientists. As shown by the examples in Figure 6, users can use “A[4, :]”, “A[0:2, 1:4]” or “A[0:3:2, 5:10:2]” to refer to different regions of a two-dimensional variable, or they could simply use the variable name, “A”, to refer the whole dataset. In fact, the array specification defined in FastQuery is relatively flexible because the specification is recognized by a language syntax implemented in the FastQuery parser. Once the array specification is recognized, FastQuery internally creates a temporary variable in the variable table. The temporary variable only corresponds to the data from the specified array region, so that FastQuery can perform all its functions on the array, just like a regular variable from file.

### 3.4 Parallel indexing

It has been demonstrated that indexing can accelerate data query in earlier research work [17, 28]. From the summary of our experiments in Table 1, we also found that using indexes could achieve a significant speed-up over a factor of 50. Even in the worst case, when 50% of data is selected by a query, we still observed a speed-up factor around 1.5. However, one shortcoming of using an index is that we have to build the index in advance, and this pre-processing step can be a time consuming process for large datasets. As shown in Table 2, the time of building index can increase linearly with the size of the data, and it can take more than 6 hours for a dataset of size 90GB. Therefore, it is critical to minimize the time for building indexes, and prevent this step from becoming a bottleneck in our index/query system.

Table 1: Elapsed time for processing query with a single core.

| dataset                  | query                        |                               |                               |
|--------------------------|------------------------------|-------------------------------|-------------------------------|
|                          | 1% selectivity<br>(speed-up) | 10% selectivity<br>(speed-up) | 50% selectivity<br>(speed-up) |
| small dataset<br>(9GB)   | 1.43s<br>(44.23x)            | 9.58s<br>(6.60x)              | 40.11s<br>(1.58x)             |
| medium dataset<br>(45GB) | 6.38s<br>(52.2x)             | 46.10s<br>(7.23x)             | 200.0s<br>(1.67x)             |
| large dataset<br>(90GB)  | 12.17s<br>(48.91x)           | 90.70s<br>(6.56x)             | 409.5s<br>(1.45x)             |

Table 2: Elapsed time for building index with a single core.

| dataset                  | read data      | build index        | write index    | total   |
|--------------------------|----------------|--------------------|----------------|---------|
| small dataset<br>(9GB)   | 72s<br>(3.2%)  | 2,113s<br>(86.3%)  | 62s<br>(2.5%)  | ~40mins |
| medium dataset<br>(45GB) | 274s<br>(2.4%) | 10,751s<br>(95.7%) | 206s<br>(1.8%) | ~3hours |
| large dataset<br>(90GB)  | 405s<br>(1.8%) | 21,683s<br>(96.3%) | 422s<br>(1.9%) | ~6hours |

In this paper, we attempt to reduce the index building time by exploring the parallel processing power offered by the modern many-core processors. Our approach is to implement the FastQuery system using multi-threaded programming, so that the index of each variable in a dataset can be built independently on a single core of a compute node in a parallel fashion. In our experiments, we will demonstrate that our approach reduces the time to build indexes by a factor of 16 when using 24 cores. Furthermore, we analyze the parallel efficiency with respect to I/O and CPU to show the potential issues and bottlenecks of our parallel approach.

### 3.5 Unified array-based I/O interface

Various array-based data file formats, such as NetCDF and HDF5, are widely used by the scientific community. Scientists often store their data in the format that best fits to the needs of their applications. Although the functionality and design of these file formats are similar, the raw data can only be accessed by their respective APIs. As the size of a scientific datasets increase, it could be costly or even infeasible to convert datasets between different file formats. Hence, we do not want to limit our system to a specific file format.

In order to make FastQuery applicable to any scientific data file format, we define a unified I/O interface for FastQuery to interact with the underlying file format libraries. Hence, once the interface for the specific file format is implemented, FastQuery is able to perform indexing and querying functions on the files conforming to the format. To demonstrate the applicability of our I/O interface, we have implemented the interface for the two most common scientific data formats, HDF5 and NetCDF.

As shown in Figure 3, the main I/O operations from FastQuery are reading/writing index and data. According to the data access pattern of FastQuery, our interface includes a set of functions that perform I/O on the whole region of a dataset, a subarray region of a dataset, or a list of coordinates of a dataset. Most of our interface functions can be directly mapped to the I/O functions provided by the HDF5 and NetCDF file libraries. We note that NetCDF cannot retrieve data by coordinates, thus results have to be read from file one-by-one in special case. Overall, the underlying file format libraries do have a significant impact on the I/O performance of FastQuery. But, we do expect to leverage various performance improvement efforts currently underway by major library developers.

## 4 Experiments

In this section, we extensively evaluate the performance of FastQuery by using a real scientific dataset and a 24-core compute node from the NERSC Cray XE6 Hopper II supercomputer [12]. We describe our experimental testbed in Section 4.1. Then we present the query and indexing performance in Section 4.2 and Section 4.3, respectively. All the results reported in our evaluation correspond to the median value over at least 5 experimental runs.

### 4.1 Testbed

We evaluate FastQuery by processing queries on a real scientific dataset collected from a large-scale, high resolution physics simulation generated by the IMPACT-T [20] code. The original dataset is over 50TB and consists of 720 time steps. Each time step has 9 variables, and each variable has 1 billion records/particles. For our evaluation, we use three different subsets of the original dataset. Each subset has 120 time steps, and each time step has one variable. We adjust the size of the subset by choosing different number of particles per variable. The “Large” dataset has 100 million particles per variable, the “Medium” dataset has 50 million particles per variable, and the “Small” dataset has 10 million particles per variable. Accordingly, the total size of the three datasets are 90GB, 45GB and 9GB. To perform query on the datasets, indexes are built for each of the datasets using the FastBit binning option “precision=3” [23]. The built indexes are stored in file separate from the data, and the size of each index file is roughly the same as the size of its data file. For these experiments, all the data is stored in the HDF5 file format, but we expect to obtain similar performance on NetCDF files because the latest NetCDF-4 implementation uses HDF5 as its storage layer.

The experiments was ran on a single compute node from NERSC Cray XE6 supercomputing system Hopper II [12]. A hopper compute node has 2 twelve-core AMD ‘MagnyCours’2.1 GHz processors, and each processor consists of 2 NUMA nodes with 6 cores each. Hence, each compute node has a total of 24 cores and 32GB memory. Each core has its own L1 and L2 cache, but a 6MB L3 cache is shared by the 6 cores from a NUMA node. Under our multi-threaded execution environment, threads are placed next to the cores in the order of NUMA node. In other words, the first 6 threads runs on the six cores of the first NUMA node, then the next 6 threads runs on the second NUMA node, and so on. Although the placement of threads could affect performance, we will not explore this issue further in the present work.

Hopper uses Lustre as its file system. The current Lustre hardware consists of 156 OSTs (Object Storage Target). Our datasets are stored on the file system with the default stripe size of 1MB and stripe count of 13. According to a system profile report from NERSC [12], the peak aggregated I/O rate is 25 GB/s. But the maximum I/O rate between a single compute node to the file system is around 1.5~2 GB/s.

### 4.2 Query Evaluation

Our evaluations compare the performance of three different query techniques: FastQuery, HDF5-FastQuery and Scan. Scan simply reads all the data values from file and scans through them to answer the query. On the other hand, HDF5-FastQuery uses index like FastQuery, but it doesn’t support array specification. As a result, given a query like  $var[0 : 50] < 0.5$ , users first have to submit HDF5-FastQuery the query  $var < 0.5$ , then filter out the results by comparing the coordinates of each selected data with the query array region. Hence, HDF5-FastQuery needs to both index and scan the data. In contrast, the query can be directly answered by the FastQuery system using indexes without reading data, because FastQuery supports array specifications. We now explain our methodology for generating queries for the evaluation, and compare the performance among the three query techniques.

#### 4.2.1 Query generation

The queries used by our experiments are generated in the form of “ $var[0 : region] < threshold$ ”, where the value of “*region*” is determined by the query parameter “coverage”, and the value of “*threshold*” is determined by the query parameter “selectivity”. A 100% coverage query means the array region of a query

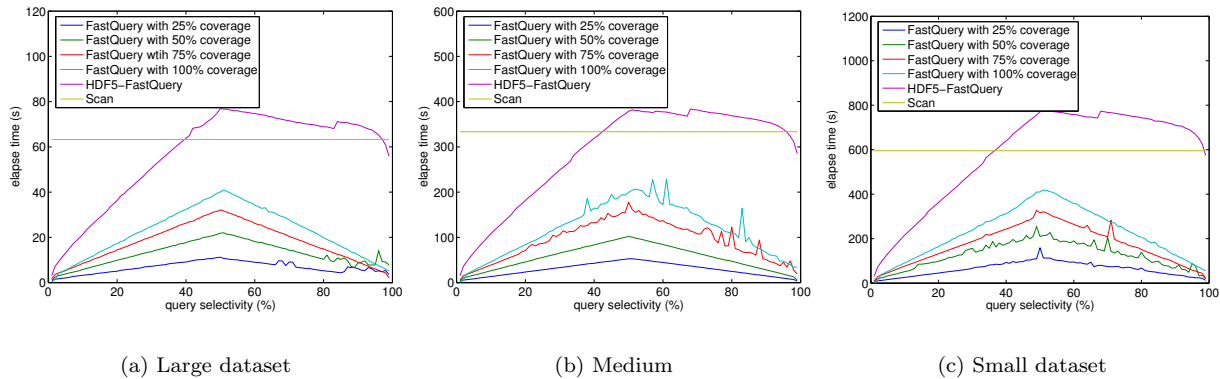


Figure 7: Elapsed time for processing queries with various selectivity and coverage settings.

covers the whole variable dataset, and a  $x\%$  coverage query means the array region of a query covers  $x\%$  of the region of the variable dataset. On the other hand, the selectivity represents the percentage of elements would be selected from the query array region. Let us consider a one-dimensional variable with 100 elements, and the values of its elements as being uniformly distributed between 0 and 1. An example of query with 100% coverage and 50% selectivity would be  $var[0 : 100] < 0.5$  (or simply  $var < 0.5$ ). This is because the query is applied to the whole array region of the variable, and 50% of the elements from the query region is expected to be selected due to the uniform data value distribution. Similarly, a query  $var[0 : 50] < 0.5$  has 50% coverage and 50% selectivity because it queries on an array region with the size of 50% of the total number of data elements, and only 50% of the elements from the array region would be selected. Hence, by reducing the coverage or the selectivity of a query could both result in fewer number of selected elements.

#### 4.2.2 Performance comparison

In the experiments, each query is iteratively processed on each of the 120 time steps, and the total elapsed time is shown in Figure 7. The selectivity of queries is varied from 0% to 100% in steps of 1%, and the coverage is varied from 25% to 100% in steps of 25%. As shown from the figure, Scan always scans through all data values for any query regardless its selectivity and coverage. Therefore, its response time is a constant, and plotted as a flat line in the figures.

For FastQuery, under a fixed query coverage, its response time changes along with the selectivity. This behavior has been observed and explained in the previously study in the HDF5-FastQuery paper [9]. The reason is that as the selectivity increases, more bitmaps need to be read from file and processed by FastBit. In addition, the results are symmetric after 50% selectivity because queries are evaluated by negating the query expression. For instance, a query “ $A > 0.8$ ” is evaluated as “ $NOT(A \leq 0.8)$ ”. Thus, the selectivity of the query is actually 20% as oppose to 80%. Lastly, we found the time also reduces as the query coverage decreases. This is because the size of bitmaps associated with an array region has positive co-relation to the number of elements in the array. Since a smaller coverage query corresponds to a smaller array region, the size of bitmap indexes read by FastQuery is also reduced. From the results under various query selectivities and coverages, we show that FastQuery scales well to the number of selected data, and it can have significant performance advantages for queries with smaller selectivity or coverage.

The query performance of HDF5-FastQuery is different from the results of Scan and FastQuery because it needs both scanning and indexing. Since HDF5-FastQuery does not support array specification, all queries must be converted to 100% coverage queries by ignoring the array specification from the original queries. Hence, the total query time of HDF5-FastQuery is at least as long as the total query time of FastQuery. After getting the query results from HDF5-FastQuery, users still have to scan through the coordinates of selected data and filter out the data outside the array region. Therefore, the time of scanning is proportional to the

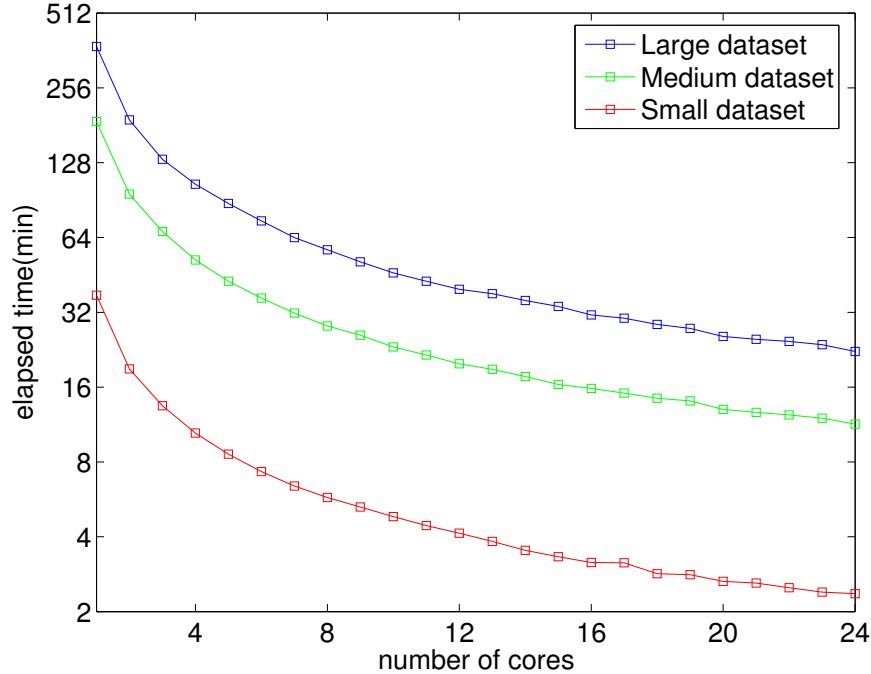


Figure 8: Elapsed time for building indexes with increasing number of cores/threads.

number of selected data (i.e. query selectivity). As a result, the total response time of HDF5-FastQuery further grows as the selectivity increases. In fact, we found the scanning time starts to dominate the total time after 50% selectivity, and it would be essentially more efficient to use Scan instead of HDF5-FastQuery when selectivity is over 40%.

### 4.2.3 Summary

Our evaluation shows that using indexing can significantly improve query performance by a factor of 1.5 to over 50 depending on the selectivity of queries. We also demonstrate the importance of supporting array specification for queries. Our results show that the post-processing time for HDF5-FastQuery, which does not support array specifications, can dominate the query response time and this can potentially degrade the performance of query resolution to below that of Scan. Finally, we demonstrate that FastQuery scales well with the number of selected data from query regardless of the data size. Therefore, as dataset sizes continue to grow, the performance benefits from using FastQuery for query resolution will become even more significant.

## 4.3 Indexing Parallelism

Now, we will evaluate our parallel implementation for building indexes. In our approach, FastQuery utilizes multiple cores on a node using multi-thread programming. In these experiments, each thread runs on a single core, and we report parallel performance and efficiency as the number cores increases from 1 to 24.

### 4.3.1 Performance comparison

Figure 8 shows the elapsed time for building indexes for each of the three datasets. As shown from the figure, the elapsed time gradually decreases as the number of cores increases from 1 to 24. When using a single core, the time to build index for Small, Medium and Large datasets, are 2,243s (37mins), 11,204s (3hours)

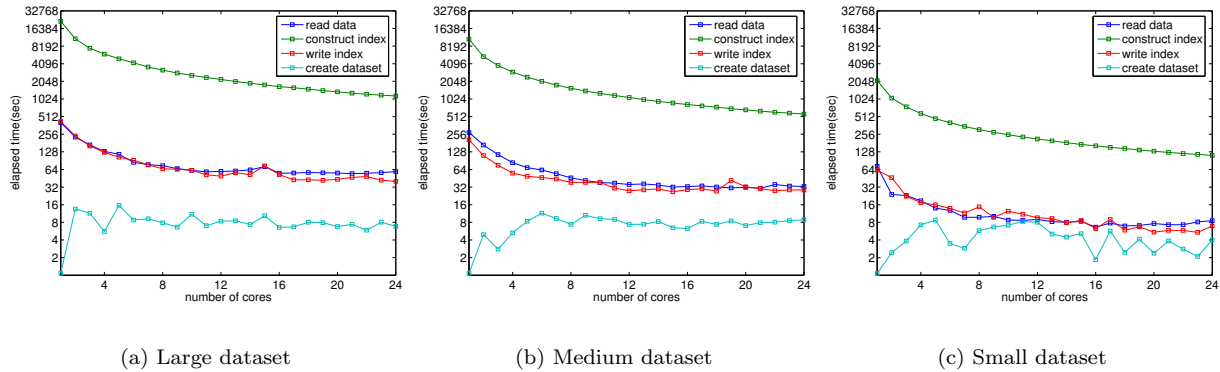


Figure 9: Time breakdown for building index versus number of cores/threads.

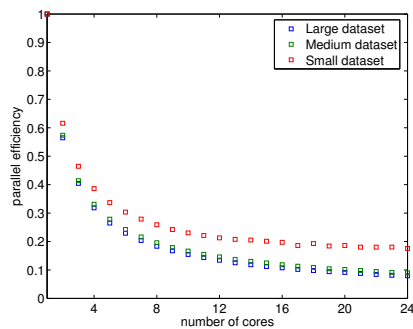


Figure 10: Overall efficiency.

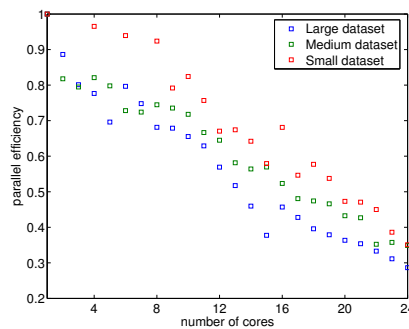


Figure 11: I/O efficiency.

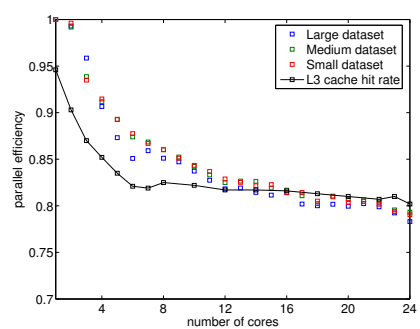


Figure 12: CPU efficiency.

and 22,535s (6hours), respectively. In comparison, with 24 cores, we achieve a speed-up factor of 16 for all three datasets by reducing their indexing time to 143s, 674s and 1350s.

To further explore this result, we plot Figure 9 which shows the time taken by various steps in the index building process. As described in Section 3.1, there are four steps for building indexes: a). reading data from file, b). constructing index in memory, c). creating a new dataset in file for storing the index, and finally d). writing index to the file. As expected, the elapsed time for I/O and computing decreases as more cores are used. In particular, the aggregate I/O rate increases from 200MB/s to 1.5GB/s which is around the peak single node I/O rate reported on this NERSC platform [12]. However, the dataset creation time significantly increases from less than 1 second to more than 10 seconds when multiple threads are used. The dataset creation step involves modifications to the file metadata, and requires synchronization among threads. Therefore, the increasing time is likely due to the synchronization cost from the HDF5 library. We also observe that the synchronization cost roughly stays constant regardless the number of threads. Hence, the synchronization cost could have greater impact on a smaller datasets. As shown in Figure 9 (c), the data creation time is almost the same as the I/O time for the Small dataset. We cannot do much about this synchronization cost at the FastQuery layer, we do note that efforts are currently underway to make performance improvements for collective metadata modifications in HDF5 files.

#### 4.3.2 Efficiency analysis

Finally, we study the efficiency of our parallel approach by plotting the efficiency of the overall elapsed time in Figure 10. As shown from the figure, the overall efficiency decreases rapidly for 6 cores, then it asymptotically reaches 0.1. To gain a better understanding of results, we analyzed the efficiency with respect to both I/O and CPU. The I/O efficiency corresponds to the time for reading data and writing indexes. Since we observe

similar results for both I/O operations, we plot the efficiency of reading data to represent the I/O efficiency in Figure 11. As shown from the figure, I/O causes contentions among threads, because all cores on a compute node share the same network bandwidth to the file system. As a result, in spite of the increasing aggregate I/O rate previously observed in Figure 9, the I/O efficiency decreases linearly from 1 to 0.3. We also observe that the efficiency is higher for the Small dataset possibly due to smaller data size and less bandwidth contention. We plot results for the CPU efficiency which corresponds to the time for constructing indexes in Figure 12. Although FastQuery constructs index independently across threads, surprisingly we still observed the CPU efficiency drops from 1 to 0.8. Upon further investigation, we found the drop is caused by the L3 cache being shared amongst 6 cores. We profiled and plotted the L3 cache hit rate in Figure 12. The cache hit rate appears to be strongly correlated with the CPU efficiency. Therefore, for the overall efficiency plot shown in Figure 10, we can attribute the steep drop within 6 cores to the CPU efficiency, then slower drop afterwards to the I/O efficiency. Finally, the overhead of dataset creation causes the large efficiency drop from 1 core to 2 cores, and the overall efficiency could be even slightly worse than the I/O efficiency.

### 4.3.3 Summary

We were able to achieve substantial performance speed-up by a factor of 16 when using 24 cores. The time for building index was reduced from over several hours to less than half hour. The aggregated I/O rate also increased from 200MB/s to 1.5GB/s. However, we did observe that the CPU efficiency reduces, this is most likely due the L3 cache being shared among cores. The I/O efficiency also gradually decreases from 1 to 0.3. Finally, the synchronization overhead from creating dataset during the index building process could be an issue for smaller datasets. As a result, the overall efficiency decreases from 1 to 0.2. Our experiments not only show the performance and efficiency of our parallel approach, but also identify some of the bottlenecks and issues for the future improvement.

## 5 Case Studies

In this section, we highlight two case studies to show how FastQuery can be applied to real life 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.

### 5.1 Atmospheric Rivers

An atmospheric river is a weather phenomenon where a significant amount of tropical moisture in the air is transported far away into extratropical regions [16], an example is shown in Figure 13. This phenomena produces unusually heavy amounts of rainfall and could cause catastrophic damage to the local communities [21], such as the western coast of north America. Therefore, 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 [27]. The particular dataset used is on a fixed latitude-longitude mesh with quarter degree spacing, and many variables are stored in 2D arrays of size  $1440 \times 720$ . Based on earlier work on detecting atmospheric rivers in the observational data [16], 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 ( $60^\circ > \text{latitude} > 23^\circ$  or  $-60^\circ < \text{latitude} < -23^\circ$ )”.

Since data is located in a dataset according to its latitude and longitude position, the condition of above query “ $60^\circ > \text{latitude} > 23^\circ$ ” is converted to the mesh points between 452 and 600 along the latitude and

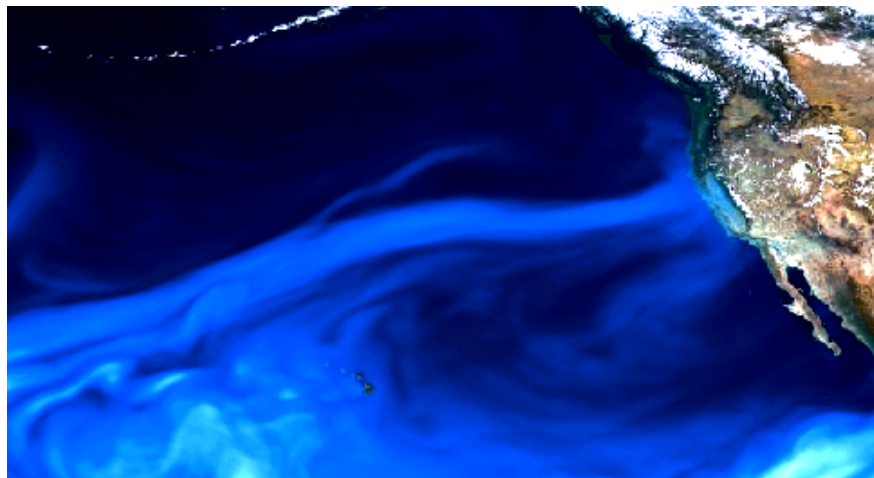


Figure 13: A plot of the water vapor concentration with an atmospheric river reaching from Hawaii to California.

similarly “ $-60^\circ < \text{latitude} < -23^\circ$ ” 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 part of the data, “vapor[:,120:268]” and “vapor[:,452:600]”, then search data from the subarray with conditions “vapor[:,120:268] > threshold” and “vapor[:,452:600] > threshold”.

## 5.2 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 that needs tens of kilometers for accelerating particles, Laser Wakefield accelerators can achieve the same acceleration in meters. 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 program to model physical experiments, gain deeper understanding of the physical observations and optimize methodology for future experiments [8, 22].

VORPAL simulations currently generate its output in HDF5. The sizes of the dataset are proportional to the number of particles used in the simulation and the discretization of the electromagnetic fields. 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 use FastQuery to 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 them back in time. Tracing these particles back in time reveals acceleration dynamics and potentially provides insights into how to better design experiments.

To help scientists make meaningful selections, we use a parallel coordinate display as illustrated in Figure 14. After the user selected particles of interest, we read the particle identifiers, extract the particles based on the identifiers from other timesteps, and then produce a trace for each particle.

## 6 Conclusions

In this paper, we present a design that maps array data to a relational data model in order to make use of database indexing techniques. The resulting software, named FastQuery, is a general indexing and querying system for scientific data. In addition to supporting semantic queries on NetCDF and HDF5, FastQuery is able to take full advantage of the emerging many-core computers through threaded parallelism. We believe FastQuery is an important step towards a general infrastructure for scientific data analysis by bridging



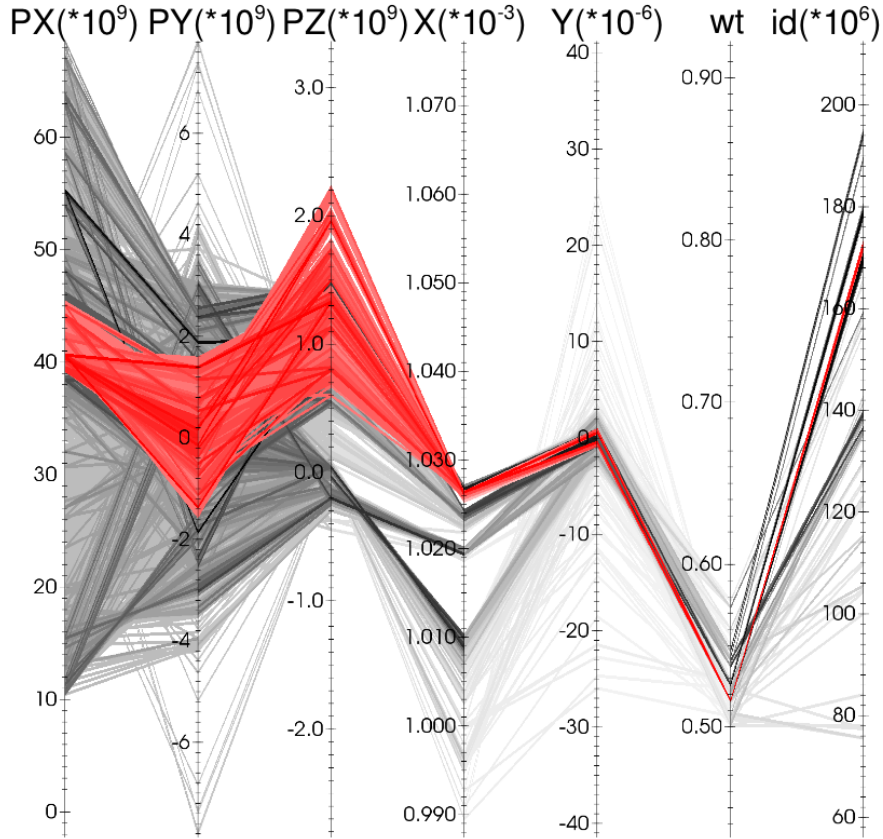


Figure 14: Example parallel coordinate display for selecting particles from Laser Wakefield Particle Accelerator simulations.

the gap between the array model used by the scientific data and relational model used by the database technology.

Specifically, FastQuery achieves its goal with the following approaches. (a) FastQuery accelerates query performance using the start-of-art indexing technology, FastBit. (b) FastQuery integrates the array data model into its interface and allow users to query data using a general array specification, such as “A[1:10] > 15”. The array model interface also provide more seamlessly connection between FastQuery and the scientific applications. (c) Instead of enforcing data to be stored in a specific layout, such as in H5hut [13], FastQuery deploys a flexible yet simple variable naming scheme based on regular expressions to handle arbitrary file layout. (d) FastQuery defines a unified array I/O interface to interact with the array-based scientific data formats. The applicability of the interface is demonstrated by the use of HDF5 and NetCDF in our implementation. (e) FastQuery exploits multi-core architectures to minimize the indexing time.

Finally, the performance and the efficiency of FastQuery were extensively evaluated and analyzed in our experiments. Using real scientific datasets, we showed indexing can improve query response time by a factor of 2.5 to over a factor of 50. We also showed the time for building indexes can be reduced by a factor of 16 when 24 cores are used. We verified that I/O contention contributes to the loss in efficiency, and that CPU efficiency degrades because the L3 cache is shared among cores. We made an interesting observation in that creating new dataset for storing indexes could cause a constant synchronization overhead. Therefore, to further improve parallel efficiency, we plan to minimize the I/O contention and reduce thread synchronization in the future. We also plan to developing a hybrid implementation of FastQuery to deploy it on large scale distributed multi-core platforms.

## 7 Acknowledgments

The authors wish to thank Allen Sanderson, John Shalf, Quincey Koziol, E. Wes Bethel and Arie Shoshani 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.

## References

- [1] Marcos K. Aguilera, Wojciech Golab, and Mehul A. Shah. A practical scalable distributed b-tree. *Proc. VLDB Endow.*, 1:598–609, August 2008.
- [2] Carrie Ballinger and Ron Fryer. Born to be parallel: Why parallel origins give teradata an enduring performance edge. *IEEE Data Eng. Bull.*, 20(2):3–12, 1997.
- [3] J. Chou, K. Wu, O. Rübel, M. Howison, J. Qiang, Prabhat, B. Austin, E. W. Bethel, R. D. Ryne, and A. Shoshani. Parallel index and query for large scale data analysis. In *SC11*, 2011.
- [4] Douglas Comer. The ubiquitous B-tree. *Computing Surveys*, 11(2):121–137, 1979.
- [5] G. S. Davidson, K. W. Boyack, R. A. Zacharski, S. C. Helmerich, and J. R. Cowie. Data-centric computing with the netezza architecture. Technical Report SAND2006-3640, Sandia National Laboratory, 2006.
- [6] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. In *SIGMOD Conference*, pages 1–8, 1984. <http://doi.acm.org/10.1145/602259.602261>.
- [7] David J. DeWitt, Marc Smith, and Haran Boral. A single-user performance evaluation of the teradata database machine. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, pages 244–276, London, UK, 1989. Springer-Verlag.
- [8] 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.
- [9] Luke Gosink, John Shalf, Kurt Stockinger, Kesheng Wu, and Wes Bethel. HDF5-FastQuery: Accelerating complex queries on HDF datasets using fast bitmap indices. In *SSDBM*, pages 149–158, 2006.
- [10] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25:73–169, June 1993.
- [11] T. Hey, S. Tansley, and K. Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft, October 2009.
- [12] Hopper II: Cray XE6. <http://www.nersc.gov/nusers/systems/hopper2>.
- [13] Mark Howison, Andreas Adelman, E. Wes Bethel, Achim Gsell, Benedikt Oswald, and Prabhat. H5hut: A High-Performance I/O Library for Particle-Based Simulations. In *Proceedings of Workshop on Interfaces and Abstractions for Scientific Data Storage*, pages 1–8, 2010.
- [14] Hongjun Lu, Yuet Yeung Ng, and Zengping Tian. T-tree or b-tree: Main memory database index structure revisited. In *Australasian Database Conference*, pages 65–73, 2000.

- [15] R. Musick and T. Critchlow. Practical lessons in supporting large-scale computational science. *SIGMOD Rec.*, 28(4):49–57, 1999.
- [16] 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.
- [17] 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, September 1987.
- [18] Patrick O’Neil and Elizabeth O’Neil. *Database: principles, programming, and performance*. Morgan Kaufmann, 2nd edition, 2000.
- [19] M. Tamer Ozsu. *Principles of Distributed Database Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.
- [20] J. Qiang, R. D. Ryne, M. Venturini, and A. A. Zholents. High resolution simulation of beam dynamics in electron linacs for x-ray free electron lasers. *Physical Review*, 12:100702–1 – 100702–11, 2009.
- [21] 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.
- [22] Oliver Rübél, Prabhat, Kesheng Wu, Hank Childs, Jeremy Meredith, Cameron G. R. Geddes, Estelle Cormier-Michel, Sean Ahern, Gunther H. weber, Peter Messmer, Hans Hagen, Bernd Hamann, and E. Wes Bethel. High performance multivariate visual data exploration for extremely large data. In *SuperComputing*, pages 1–12, November 2008.
- [23] Arie Shoshani and Doron Rotem, editors. *Scientific Data Management: Challenges, Technology, and Deployment*, chapter 6. Chapman & Hall/CRC Press, 2010.
- [24] Michael Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986. <http://sites.computer.org/debull/86MAR-CD.pdf>.
- [25] The HDF Group. HDF5 user guide. <http://hdf.ncsa.uiuc.edu/HDF5/doc/H5.user.html>, 2010.
- [26] Unidata. The NetCDF users’ guide. <http://www.unidata.ucar.edu/software/netcdf/docs/netcdf/>, 2010.
- [27] Michael F. Wehner, Leonid Oliker, and John Shalf. Towards ultra-high resolution models of climate and weather. *IJHPCA*, 22(2):149–165, 2008.
- [28] Kesheng Wu, Sean Ahern, E. Wes Bethel, Jacqueline Chen, Hank Childs, Estelle Cormier-Michel, Cameron Geddes, Junmin Gu, Hans Hagen, Bernd Hamann, Wendy Koegler, Jerome Lauret, Jeremy Meredith, Peter Messmer, Ekow Otoo, Victor Perevoztchikov, Arthur Poskanzer, Prabhat, Oliver Rubel, Arie Shoshani, Alexander Sim, Kurt Stockinger, Gunther Weber, and Wei-Ming Zhang. FastBit: Interactively searching massive data. In *SciDAC*, 2009. <http://sdm.lbl.gov/fastbit/>.
- [29] Kesheng Wu, Ekow Otoo, and Arie Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems*, 31:1–38, 2006.
- [30] Kesheng Wu, Arie Shoshani, and Kurt Stockinger. Analyses of multi-level and multi-component compressed bitmap indexes. *ACM Trans. Database Syst.*, pages 1–52, 2010.