HDF5-FastQuery:
Accelerating Complex Queries on HDF Datasets using Fast Bitmap Indices

Luke Gosink[1], John Shalf[2], Kurt Stockinger[2], Kesheng Wu[2], Wes Bethel[2]

[1]*Institute for Data Analysis and Visualization, University of California at Davis*
One Shields Ave, Davis, CA 95616, USA

[2]*Computational Research Division, Lawrence Berkeley National Laboratory*
One Cyclotron Road, Berkeley, CA 94720, USA

Abstract

Large scale scientific data is often stored in scientific data formats such as FITS, netCDF and HDF. These storage formats are of particular interest to the scientific user community since they provide multi-dimensional storage and retrieval. However, one of the drawbacks of these storage formats is that they do not support semantic indexing which is important for interactive data analysis where scientists look for features of interests such as "Find all supernova explosions where $energy > 10^5$ *and* $temperature > 10^6$*".*

In this paper we present a novel approach called HDF5-FastQuery to accelerate the data access of large HDF5 files by introducing multi-dimensional semantic indexing. Our implementation leverages an efficient indexing technology called bitmap indexing that has been widely used in the database community. Bitmap indices are especially well suited for interactive exploration of large-scale read-only data. Storing the bitmap indices into the HDF5 file has the following advantages: a) Significant performance speedup of accessing subsets of multi-dimensional data and b) portability of the indices across multiple computer platforms. We will present an API that simplifies the execution of queries on HDF5 files for general scientific applications and data analysis. The design is flexible enough to accommodate the use of arbitrary indexing technology for semantic range queries. We will also provide a detailed performance analysis of HDF5-FastQuery for both synthetic and scientific data. The results demonstrate that our proposed approach for multi-dimensional queries is up to a factor of 2 faster than HDF5.

1 Introduction

Large-scale scientific experiments often store data in scientific data formats such as FITS [5], netCDF [9] and HDF [11]. These data formats provide the ability to store and retrieve multi-dimensional arrays that are often regarded as the building blocks for scientific data exploration. The most recent implementations of these data formats, such as HDF5 and parallelNetCDF, have been extended to support parallel data access – a key requirement for the data output requirements for simulation codes on MPPs. However, one of the open problems that is common to all scientific data formats is that they to not have an interface to support semantic indexing. As pointed out by Jim Gray et al. [6] "Scientists need a way to use intelligent indices and data organizations to subset the search" .

In this paper we address this fundamental open problem of scientific data formats by providing an interface to support semantic indexing for HDF5 via a query API. We integrate an efficient searching technology named *FastBit* [19, 20] with HDF5. The integrated system named *HDF5-FastQuery* allows users to efficiently generate complex selections on HDF5 datasets using compound range queries such as $(enery > 10^5)$ *AND* $(70 < pressure < 90)$ and only retrieve the subset of data elements that meet the query conditions. FastBit technology generates compressed bitmap indices that accelerate searches on HDF5 datasets and can be stored together with those datasets in an HDF5 file. Compared with other indexing schemes, compressed bitmap indices are compact and very well suited for searching over multi-dimensional data – even for arbitrarily complex combinations of range conditions.

The main contributions of this paper are:

- We introduce HDF5-FastQuery, a novel approach for simplifying storage and retrieval of HDF5 data

sets. We describe the architectural layout and the API for creating and querying HDF5-files with multi-dimensional bitmap indices.

- We perform a detailed performance evaluation of our FastQuery enhancements to HDF5. The results demonstrate that our proposed approach for processing multi-dimensional queries is up to a factor of 2 faster than HDF5.

The remainder of the paper is organized as follows. Section 2 introduces the need for semantic indexing in HDF5 files and describes the related work on scientific data formats and indexing technologies that are relevant in this area. Section 3 outlines the architecture of HDF5-FastQuery. Section 4 gives a detailed performance evaluation. Concluding remarks and future work are presented in Section 5.

2 Related Work

2.1 Scientific Data Formats

Scientific applications have used a variety of ad-hoc I/O methods, including ASCII, raw binary, and Fortran unformatted binary. In order to support more transparent sharing of data, various scientific communities have developed their own file formats (or format conventions) and associated APIs. For instance NetCDF has been engineered primarily to support the climate modeling community; Plot3D format supports aeronautics and FITS was developed as the storage format for sharing astronomy and astrophysics data. Efforts such as OpenDAP have attempted to separate the high level data model from the underlying implementation. This separation of concerns has enabled a unified interface for managing data in files, or in directory servers, remote data retrieval, and even support for data query operations.

While high level data interfaces, such as OpenDAP have successfully separated underlying data layout issues from the higher level data schemas, file formats like HDF5 are directly addressing the concerns of low-level file organization issues. HDF5 offers a hierarchical data model packed in a self-describing binary, platform-independent file format. HDF5's hierarchical data model is flexible enough to accommodate the requirements of numerous higher-level data schemas. The data organization is very similar to an object database, but unlike most object database implementations HDF5 is portable, non-proprietary, and supports concurrent access to individual records (the kind of parallel I/O that is essential for HPC applications). So, for example, Version 4 of NetCDF will jettison its own low-level file format and implement its data schema on top of HDF5. HDF-EOS is another example of a complex high-level data schema that is implemented on top of HDF5 as a substrate.

Our work extends HDF5 in two dimensions. First, we develop a high-level data schema that is appropriate for time-series block-structured and particle data that is typical of a number of applications that we are interested in supporting. We developed the high-level schema in order to provide a testbed for our work on HDF5 indexing technology (HDF's data schema would otherwise be too low-level to use sensibly in scientific applications). Next, we extend HDF5's low-level dataset selection mechanisms to incorporate our accelerated bitmap indexing technology. Our work differs from the HDF5 Storage Resource Broker (SRB) work at SDSC in the granularity of our query/selection mechanism. Whereas SRB focuses on queries and selections at the file and full dataset granularity (object-level access), our selection mechanism focuses on queries and selections of data elements **within** the dataset. (http://hdf.ncsa.uiuc.edu/hdf-srb-html/)

2.2 Indexing for Scientific Data Formats

HDF5 has several parallel I/O optimization techniques based on caching and prefetching. HDF5 uses B-tree indices internally but does not expose them to the end-user. However, semantic indexing for improving the performance of range queries got very little attention.

PyTables [1] manages persistent collections of data objects for improved I/O speeds. The collections can be efficiently accessed with B-trees.

Nam and Sussman [8] have designed an indexing library that supports R*-trees for HDF4 and HDF5 datasets. This type of index is particularly well-suited for querying spatial data.

The above described solutions work well for low-dimensional queries. Our approach focuses on improving the performance of high-dimensional queries with 5, 10 or even more query dimensions. In order to achieve this goal, we use bitmap indices for querying high-dimensional HDF5-files.

2.3 Bitmap Indexing Technology

A bitmap index uses a set of bitmaps to mark whether or not each record (or row) of a dataset has a particular property, e.g., whether the value of an attribute (or variable) is a particular value or falls in a particular bin. Because most CPUs support efficient operations between bitmaps, bitmap indices can efficiently answer range queries [14, 4, 13, 18]. They are particularly well suited for data warehousing type of applications where the experts often submit complex, multi-dimensional ad-hoc queries on read-only data. They have been introduced into major commercial database systems by vendors such as Sybase, IBM and Oracle.

RID	I	bitmap index					
		=0	=1	=2	=3	=4	=5
1	0	1	0	0	0	0	0
2	1	0	1	0	0	0	0
3	3	0	0	0	1	0	0
4	2	0	0	1	0	0	0
5	3	0	0	0	1	0	0
6	5	0	0	0	0	0	1
7	5	0	0	0	0	0	1
8	2	0	0	1	0	0	0
		b_1	b_2	b_3	b_4	b_5	b_6

Figure 1. A sample bitmap index where RID is the record ID and I is the integer attribute with values in the range of 0 to 5.

For example, the integer attribute **I** shown in Figure 1 can be one of 6 distinct values, 0, 1, 2, 3, 4 and 5. For each value one bitmap is generated. Since the value in record 5 is 3, the fifth bit in b_4 is set to 1 and the same bits in other bitmaps are 0. Assume we wish to answer the following range query $I < 3$. We know that bin b_1 represents records with the value 0, bin b_2 represents records with the value 1, and bin b_3 represents records with the value 2. In order to retrieve all records that fulfill the query constraint < 3, the bins b_1, b_2 and b_3 are ORed together.

FastBit [12] is a specialized bitmap indexing software for scientific data that uses a bitmap compression method designed to be more compute-efficient than the best available commercial implementations [19, 20]. In the worst case, the FastBit index size can be twice as large as the user data which compares favorably against some commercial B-tree implementations. In many tests on application data sets, the size of the compressed indices is typically about a third of the data size.

It was further proven through formal analysis that the time required to answer a one-dimensional range query using the compressed bitmap index used in FastBit scales linear with the number of hits. In terms of computational complexity theory, this is optimal. Some of the well-known indexing methods, such as B^*-trees and B^+-trees, have this same optimality property. However, the bitmap index has a unique advantage that answers to one-dimensional queries can be efficiently combined to answer multi-dimensional range queries.

For most data analysis tasks, the procedure of searching for interesting data records is one step in a long chain of activities. In this process, the user data often needs to be in a particular order to make most of the steps efficient. The compressed bitmap index is much easier to accommodate this requirement than similar indexing methods because it does not require one to sort the data in any particular way. This leaves the users the freedom to choose the way to organize their data to reduce the total data analysis time. For data produced on uniform grids, FastBit has been demonstrated to find regions of interest in time that is proportional to the size of boundaries of the regions [16, 21]. Since finding regions of interest is a common task in many visualization and data analysis tasks, FastBit is clearly a useful tool.

3 Architecture of HDF5-FastQuery

HDF5 supports slab and hyper-slab selections of N-dimensional datasets. *HDF5-FastQuery* extends the HDF5 selection mechanism to allow arbitrary range conditions on the data values contained in the datasets using the bitmap indices. This allows the HDF5-FastQuery technology to support a fast execution of results for compound queries that span multiple datasets. The API also allows us to seamlessly integrate the FastBit query mechanism for data selection with HDF5's standard hyper-slab selection mechanism. Using the HDF5-FastQuery API, one can quickly select subsets of data from a HDF5 file using text-string queries.

The bitmap indices are created and stored through a single call to the HDF5-FastQuery API. The storage of these indices uses separate arrays in the same file as the datasets they refer to and are opaque to the general HDF5 functions. It is important to note that all such indices must be built before any queries are posed to the API. Once the bitmap indices have been built and stored in the data file, queries are posed to the API as a text-string such as "$(temperature > 1000)$ *AND* $(70 < pressure < 90)$", where the names specified in the range query correspond to the names of the datasets in the HDF5 file. The HDF5-FastQuery interface uses the stored bitmap indices that correspond to the specified dataset to accelerate the selection of elements in the datasets that meet the search criteria. An accelerated query on the contents of a dataset requires only small portions of the compressed bitmap indices to be read into memory, so extremely large datasets can be searched with little memory overhead. The query engine then generates an HDF5 selection that can be used to only read the elements from the dataset that are specified by the query string.

The FastBit technology is amenable to handling datasets and selections that are far larger than system memory. In recent experiments[17] with data of 241 GB in size, a search that consumed 2467 seconds using sequential scan was reduced to only 22.8 seconds using the bitmap indices. This same ability to handle out-of-core data selections will be available in the HDF5-FastQuery implementation.

3.1 Design

In this section, we present a high-level view of the HDF5-FastQuery architectural layout. We begin by defining relevant terms used throughout the architectural layout as well as the HDF5-FastQuery API.

Groups: Groups are the logical way in a HDF5 file format to organize data. In this paper we will use the term *group* or *grouping* to refer to this logical structuring. These groups act as a container of various metadata which in our approach is specific to a given dataset. Note that these groups may be assigned *type information* (float, int, string etc.) to uniquely describe these datasets.

Variables vs. Attributes: The properties assigned to a specific group (i.e. group metadata) are called *attributes* or *group attributes*. For all datasets, the specific physical property that the dataset quantizes (density, pressure, helicity etc.) will be referred to as dataset *variables*.

To organize a given multivariate dataset consisting of a discrete range of time steps, a division is made between the raw data and the attributes that describe the data. This division is represented in the architectural layout by the separation and formation of two classes of groups: the *TimeStep* groups for the raw data, and the *VariableDescriptor* groups for the metadata used to describe the dataset variables.

For the dataset variables, one *VariableDescriptor* group is created for each variable (pressure, velocity etc.). The metadata saved under these groups usually includes:

- The size of the data set
- The name of the dataset variable
- The coordinate system used in the dataset (spherical, Cartesian etc.)
- The schema (structured, unstructured, AMR [3])
- Centering (cell centered, vertex centered, edge centered etc.)
- The number of coordinates which must exist per centering element (each vertex, each face etc.)

The various *VariableDescriptor* groups are then organized under one TOC (table of contents) group that retains common global information about the file's variables (the names of all variables, bitmap indices metadata information). For the raw datasets, a unique *TimeStep* group is created for each time step in the discrete time range. Under each *TimeStep* group exists one HDF5 dataset that contains the raw data for a given variable at that time step. At this group too will also exist a variable bitmap dataset for the corresponding variable dataset. That is to say variable dataset data, for both raw and bitmapped data, will exist logically under the same *TimeStep* group. Additionally, all bitmap-key and bitmap-offset datasets for a given variable at a given time step are also recorded and saved here.

This division between data and metadata is essential for the primary reason that variable metadata for a given dataset will be relevant and accurate across all time steps for that dataset variable (there is no need to store redundant metadata). Figure 2 illustrates the HDF5-FastQuery architectural layout.

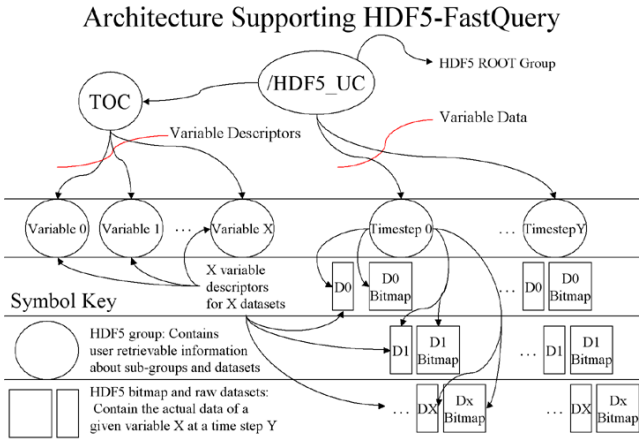


Figure 2. Architectural layout of HDF5-FastQuery.

3.2 API for Indices

From the user's perspective, the HDF5-FastQuery API provides a way to store and retrieve subsets of their dataset variables. The API's basic design maintains many of the current design principles of the index interface from the HDF5 developers [10]. However, HDF5-FastQuery requires extra parameters to address queries. This subsection briefly outlines the top level interface related to *index creation* and data subset *selection*.

Conceptually, FastBit views all user data as relational tables where each variable (dataset in HDF5 terminology) maps to a column and each record (e.g., variables associated with a mesh point) maps to a row. In HDF5-FastQuery, each time step described above is a FastBit table and users only need to know about the time steps rather than tables. The operations of creating indices, creating selections and using selections are based on specified time steps. This design can be easily changed to match that of HDF5 indexing interface once the HDF5 developers have finalized their design.

3.2.1 Creating Indices

The main function for creating indices in HDF5-FastQuery is

```
int createIndex
(const std::vector<const char*>& variable_names,
 const char* binning_options);
```

which is a member function of the class `timestep`. It creates a compressed bitmap index using the named variables and stores the result in HDF5 format back in the file that contains the original user data. This function takes two arguments. The first argument specifies a list of variables to be indexed. The second argument specifies the binning operation that will be used to generate the indices. If the name list is empty, the default behavior is to index every variable across all time steps. If the binning option is not specified, the default binning option is to not bin or use one bin for each distinct value. This function returns the number of indices successfully created and stored.

3.2.2 Querying

The functions for creating a selection, computing the number of records selected, and retrieving the respective values along with the coordinates of the records selected are called `createSelection`, `evaluateSelection`, `getSelectedData` and `getSelectedCoordinates`. The function `createSelection` has two versions depending on the specification of selection conditions. The first version takes the user specified selection conditions in a string form. The second version is based on the HDF5 indexing function `H5INquery`, and takes three arrays as function input. In both cases this function returns a token as a string. Functions calling `createSelection` use this token to identify the selection made.

`evaluateSelection` computes the number of hits. It will attempt to use the indices in the data file if there are any. Alternatively, it will read the values into memory and evaluate the selection conditions in memory.

The functions `getSelectedData` and `getSelectedCoordinates` are intended for retrieving the records selected by the user. The first function retrieves the selected values one variable at a time. The second function retrieves the coordinates of the records selected.

4 Experiments

In this section we compare the performance of HDF5-FastQuery against the R*-tree implementation on top of HDF5[8]. We call this software HDF5-R*-tree. To the best of our knowledge, this is currently the only semantic indexing software for HDF5 available. We show that HDF5-FastQuery is significantly faster. Next, we perform a detailed performance analysis of HDF5-FastQuery on both synthetic and real data sets. All experiments are executed on a 2.8 GHz Pentium 4 with 2 GB of main memory and a RAID-5 disk.

4.1 Synthetic Data

In our first set of tests we performed a comparison of HDF5-FastQuery against HDF5-R*-tree. Following the example code that is part of the HDF5-R*-tree software distribution, we generated a one-dimensional HDF5 data set that consists of 1,000,000 records. The values of the records are uniformly random distributed in the range of [0; 99]. First we built a R*-tree and then we built an equality-encoded [4] bitmap index for the same data set. The size of the original data set was 4 MB. The size of HDF5-R*-tree is 6.8 MB, i.e. the size of R*-tree is 2.8 MB. The size of the HDF5-FastQuery file is 10 MB, i.e. the size of the bitmap index is 6 MB. The time to create the bitmap index is 0.33 seconds.

From previous analyzes [7], we know that the compressed bitmap index sizes of uniform random variables are larger than other variables with skewed distributions, but the R*-tree sizes for uniform random data are generally smaller than those with skewed distributions.

With both systems we performed 100 range queries that cover that entire domain range, i.e. the query selectivity is in the range of 0 and 100%. A query selectivity of 50% means that 500,000 records fulfill the query constraint. These records are called hits. Figure 3 shows the performance comparison of HDF5-R*-tree and HDF5-FastQuery. The graph also shows the time to access the respective data set with HDF5-calls (H5Dread). The results show that, on average, HDF5-FastQuery is a factor of 3 faster than HDF5-R*-tree and HDF5.

Let us interpret the shape of the performance graph for HDF5-FastQuery. We observe that for queries with up to 50% selectivity (500,000 hits), the query response time is increasing. This is due to the fact that for higher selectivities, more bitmaps have to read from disk and need to be ORed together. In the specific case of 50% selectitivity, 50 bitmaps out of 100 need to be scanned. We also notice that for queries above a selectivity of 50%, the query response time decreases. This is because queries with selectivities above 50% are evaluated by negating the query expression. For instance, a query $A < 80$ is evaluated as $NOT(A >= 80)$. Assuming that the attribute range is between 0 and 100, then a query with 80% selectivity is first evaluated as a query with 20% selectivity which means that only 20 bitmaps need to be scan as apposed to 80. Finally, the result bitmap is negated.

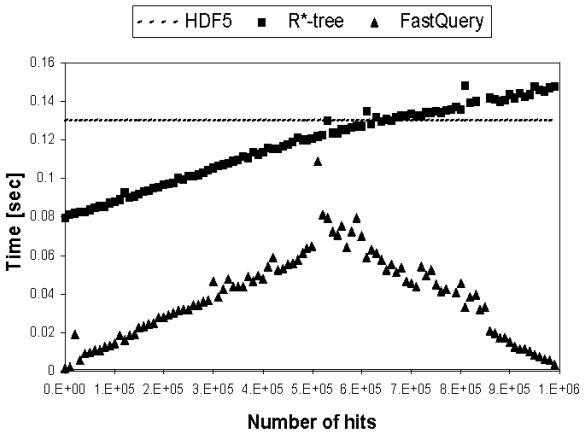


Figure 3. Response time of 1-dimensional queries with HDF5, HDF5-R*-tee and HDF5-FastQuery.

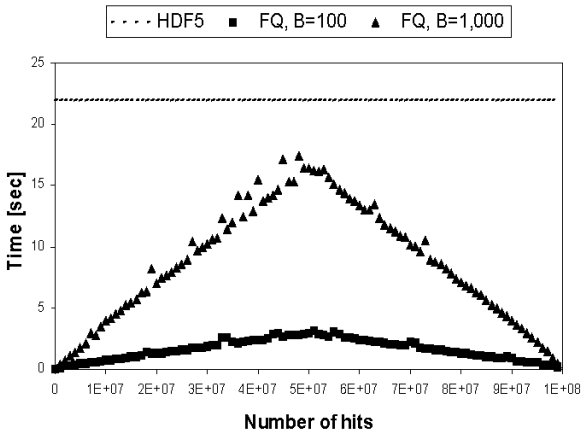


Figure 4. Response time of 1-dimensional queries with HDF5 and HDF5-FastQuery. FQ, B=100 and FQ=1000 correspond to HDF5-FastQuery with 100 and 1000 bins respectively.

Next we generated two data sets with 100 million records with different attribute cardinalities. The values of the first data set are uniformly random distributed in the range of [0; 99]. This corresponds to an attribute cardinality of 100. The second data set has values in the range of [0; 999]. In other words, the second data sets has an attribute cardinality of 1,000. For the first data set, we generated a bitmap index with 100 bins (1 bin per attribute cardinality). For the second data set we generated a bitmap index with 1,000 bins. The size of the HDF5 file is 400 MB. The sizes of the bitmap indices are 600 MB and 770 MB, respectively. The time to create these two indices was 36.8 and 260 seconds. The goal of this experiment was to show the impact of the attribute cardinality on the performance of the bitmap index. Due to stability issues with the HDF5-R*-Tree implementation, we do not include these measurements.

Figure 4 shows the query response time for 100 one-dimensional queries with selectivities of 0 to 100%. The graph also shows the time it takes to perform the same query with traditional HDF5-calls. The bitmap index with 100 bins is, on average, 13 times faster than HDF5. With 1,000 bins the performance of HDF5-FastQuery is about a factor of 2 better than HDF5. The reason is that with 1,000 bins, in the worst case, 500 bins have to be read which corresponds to 385 MB. Note that this is about the same size as the raw data.

4.2 Scientific Simulation Data

In our next set of experiments we used a large combustion data set that is a time-dependent simulation of a laboratory-scale rod-stabilized premixed turbulent V-flame [2]. The surface of the flame is shown in Figure 5.

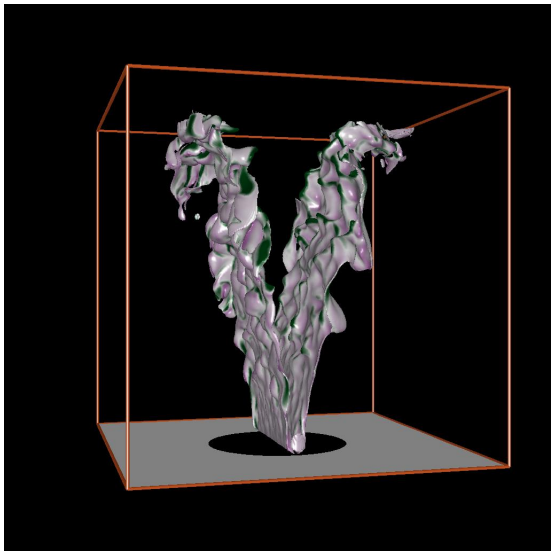


Figure 5. Surface of a V-flame for a time-dependent combustion simulation.

The data set contains 5 variables over 8 time steps. Each variable consists of some 56 million records. For each variable and each time step we built an equality-encoded bitmap index with 100 bins. The total size of the data set with 5 variables and 8 time steps is about 18GB. The size of the

bitmap indices is 4GB. The time to create the indices for all the data took about 2 hours. The average time to create a bitmap index per attribute consisting of 56 million records was 2 minutes. Note that this data set including the indices is about 10 times larger than the main memory of our test machine.

For each of the 5 variables we calculated the minimum and the maximum value. Next we generated 100 one-dimensional queries where we randomly selected the query range within the minimum and the maximum value of the chosen attribute. In order to cover the widest possible query performance matrix, we also made a random choice on (1) the data variable, (2) the time step and (3) the query range.

Since bitmap indices with bins represent attribute ranges rather than attribute values, the query results contain also values that might not exactly match the query. In order to return only qualifying values, some of the data records must be fetched from disk to verify them against the query constraint. This additional step is also known as the *candidate check* [18]. We will first show the query response time without candidate check and next present results including the candidate check.

The number of values that do not fulfill the query constraint due to binning is indirect proportional to the number of bins. In general, the maximal error in % introduced with equi-depth bins is $100/B$ where B is the number of bins. In our experiments we use 100 and 1000 bins which means that the maximal error rate of the results is 1% and 0.1%, respectively.

Figure 6 shows the query response time for one-dimensional queries as a function of the number hits. The average response time for a one-dimensional query over 56 million records with 100 bins is 0.4 seconds. For 1000 bins, the query response time is on average 1.12 seconds. Performing the same query with HDF5 takes some 12 seconds which shows that HDF5-FastQuery is about a factor of 10 faster than HDF5. Again we notice the characteristic *A-shape* of the graph. However, we see no value with selectivities around 50%. This is due to the highly skewed data values and the random selection of the query range.

Next we measured the performance of multi-dimensional queries. Similar to our previous experiments, we randomly selected the query range between the minimum and the maximum value of the respective attribute. Figures 7 and 8 show the response times of 3 and 5-dimensional queries. The average response time for these queries with 100 bins is 1.2 and 2 seconds, respectively. With 1000 bins the average query response time is 3.5 and 5.6 seconds. Answering these queries with HDF5 takes 36 and 60 seconds. Since HDF5-FastQuery scales linear with the number of query dimensions, the performance speedup over HDF5 is again a factor of 10.

In the last set of tests we measured the time for queries with exact answers, i.e. the query evaluations include the candidate check. To best minimize the overhead incurred in the disk access performed for each candidate check, HDF5 *scatter-gather* techniques are used via the $H5Sselectelements()$ function call. $H5Sselectelements()$ allows a list of defined elements to be included in the selection of a given dataspace which is then accessed for candidate verification. The performance advantage to this approach lies in submitting multiple candidates as apposed to individually defining and accessing hyper-slabs for each candidate check.

Note that as the number of bins increases, the bitmap

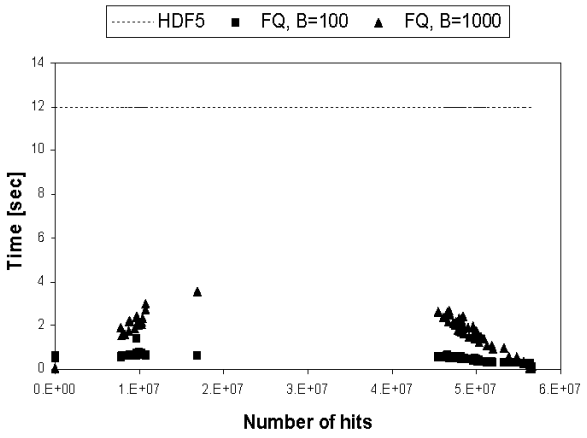


Figure 6. Response time of 1-dimensional queries using bitmap indices.

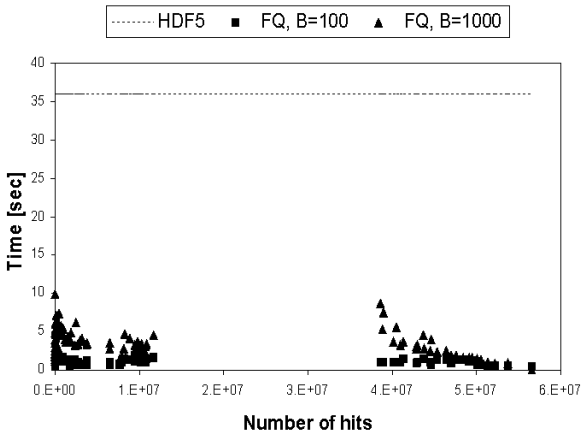


Figure 7. Response time of 3-dimensional queries using bitmap indices.

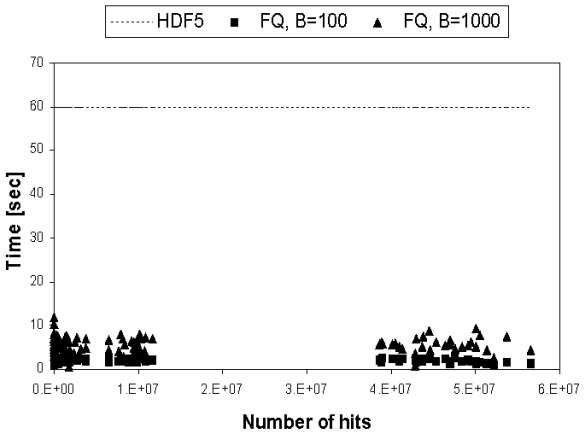


Figure 8. Response time of 5-dimensional queries using bitmap indices.

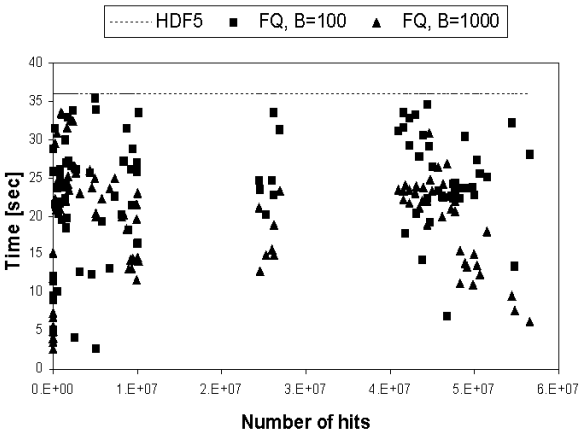


Figure 10. Response time of 3-dimensional queries using bitmap indices. Exact answers including candidate check.

evaluation costs increase as more bitmaps have to be scanned. On the other hand, however, as the number of bins increases, the cost for the candidate check decreases as the number of candidates per bin decreases [18]. Figures 9 to 11 show the query response times for 1, 3 and 5-dimensional queries. For all queries, HDF5-FastQuery with 1000 bins shows the best performances with an average response time of more than a factor of 2 faster than HDF5 for 5-dimensional queries. We can also see that as the number of query dimensions increases, the performance gain of HDF5-FastQuery over HDF5 increases as well.

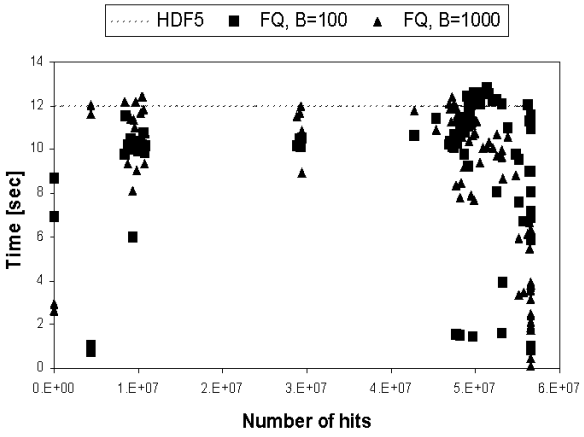


Figure 9. Response time of 1-dimensional queries using bitmap indices. Exact answers including candidate check.

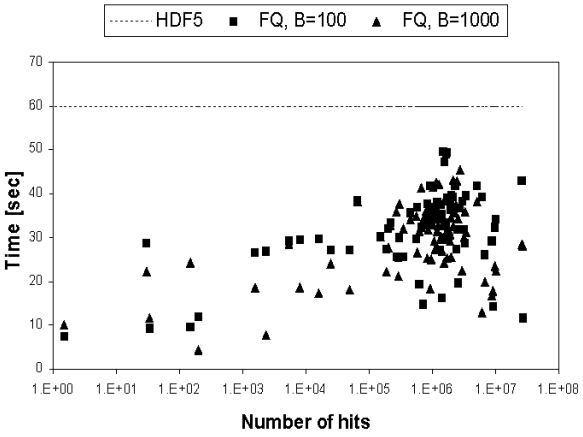
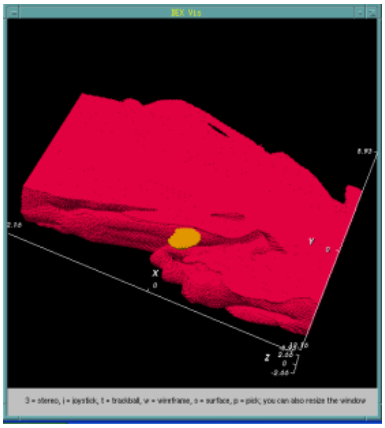


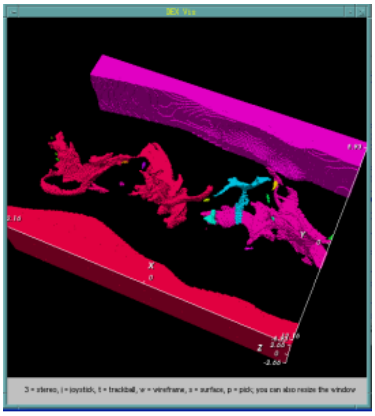
Figure 11. Response time of 5-dimensional queries using bitmap indices. Exact answers including candidate check.

5 Applications

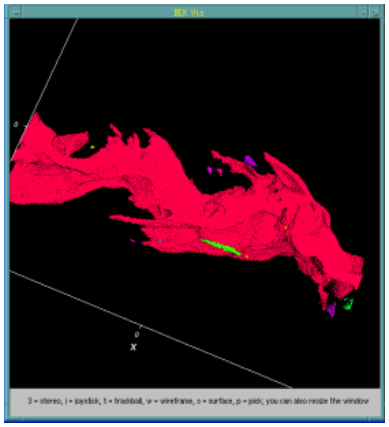
In addition to data analysis applications, we have applied bitmap indices for efficient query-based visualization within the DEX framework [15, 16]. DEX uses the bitmap indices for isosurface extraction as well as retrieval of high-dimensional volumes of data that can be used for post-processing in typical combustion simulation experiments. Figure 12 shows an interactive visualization process based



(a) $CH_4 > 0.3$



(b) $temp < 3$



(c) $CH_4 > 0.3$ AND $temp < 4$

Figure 12. A visualization of flames in a high-fidelity simulation of methane-air jet. The images show the cells in a 3D block-structured dataset that were returned by three different queries.

on a combustion simulation dataset similar to the one of the previous section. The example demonstrates how data is progressively interrogated to focus on cells that contain properties of interest. Figure 12 (a) shows the resulting isosurface based on the query $CH_4 > 0.3$. This query is refined in several steps Figure 12 (b), (c). and the isosurface is displayed in real-time.

6 Conclusions

In this paper we introduced HDF-FastQuery, a novel approach to accelerate data access of large HDF5-files by introducing semantic indexing. Our indexing technology is based on bitmap indices that are optimized for multi-dimensional queries on read-only data. We introduced the architectural layout of HDF-FastQuery that has the following advantages over traditional HDF5: (1) Simplified storage and retrieval of data in HDF5. (2) Ability to construct and store indices in HDF5. (3) Ability to query the data efficiently. We performed a detailed performance evaluation of our proposed solution on both synthetic and real data sets from a large combustion simulation. The performance results show that HDF5-FastQuery is up to a factor of 2 faster than HDF5 for multi-dimensional queries.

As our experiments have shown, there is quite some overhead for the candidate check which is due to the sequential skip scan of the HDF5-file. Currently, random access candidate checks are supported through the use of the HDF5 $H5Sselectelements()$ function call which allows a list of defined elements to be included in the selection of a given dataspace. While this approach, referred to as *scatter-gather*, is certainly faster than defining and accessing hyperslabs for each random access, we are currently investigating optimization techniques to improve the performance of scatter-gather and the random access to HDF5.

We will also extend HDF5-FastQuery to handle irregular data such as AMR [3] that are very typical for scientific applications. These data sets are stored in multiple resolution levels and need special indexing for efficient access. We will also work on more sophisticated visualization techniques to analyze complex scientific processes that are often difficult to understand with traditional analysis tools.

7 Acknowledgment

This work is supported by the Director, Office of Science, Office of Advanced Scientific Computing, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

References

[1] F. Alted, M. Fernandez-Alonso, PyTables: Processing and Analyzing Extremly Large Amounts of Data in Python In *PyCon 2003*, Washington D.C., USA March 2003.

[2] J. B. Bell, M. S. Day, I. G. Shepherd, M. Johnson, R. K. Cheng, J. F. Grcar, V. E. Beckner, M. J. Lijewski. Numerical Simulation of a Laboratory-Scale Turbulent V-flame. In *Proc. Natl. Acad. Sci. USA, 102(29), 10006-10011* 2005.

[3] M. Berger, P. Colella, Local Adaptive Mesh Refinement for Shock Hydrodynamics, In *Journal of Computational Physics*, May 1989

[4] C.-Y. Chan and Y. E. Ioannidis. Bitmap Index Design and Evaluation. In *SIGMOD*, Seattle, Washington, USA, June 1998. ACM Press.

[5] FITS - Flexible Image Transport System, http://heasarc.gsfc.nasa.gov/docs/heasarc/fits.html

[6] J. Gray, D. T. Liu, M. Nieto-Santisteban, A. Szalay, D. DeWitt and G. Heber, Scientific Data Management in the Coming Decade, In SIGMOD Record, 34(4), December 2005.

[7] J. M. Hellerstein, E. Koutsoupias, C. H. Papadimitriou, On the Analysis of Indexing Schemes. In *Symposium on Principles of Database Systems (PODS)*, Tucson, Arizona, USA, May, 1997.

[8] B. Nam, A. Sussman, Improving Access to Multi-dimensional Self-describing Scientific Dataset. In *International Symposium on Cluster Computing and the Grid (CCGrid)*, May 2003, Tokyo, Japan. IEEE Computer Society Press.

[9] NetCDF - Network Common Data Form, http://www.unidata.ucar.edu/software/netcdf/

[10] H5IN: Indexing Interface, http://hdf.ncsa.uiuc.edu/RFC/H5IN/RM_H5IN.html, November 2005.

[11] HDF - Hierarchical Data Format, http://hdf.ncsa.uiuc.edu/

[12] FastBit: An Efficient Compressed Bitmap Index Technology, http:://sdm.lbl.gov/fastbit

[13] T. Johnson. Performance Measurements of Compressed Bitmap Indices. In *International Conference on Very Large Data Bases (VLDB)*, Edinburgh, Scotland, September 1999. Morgan Kaufmann.

[14] P. O'Neil and D. Quass. Improved Query Performance with Variant Indexes. In *International Conference on Management of Data (SIGMOD)*, Tucson, Arizona, USA, May 1997. ACM Press.

[15] K. Stockinger, John Shalf, Wes Bethel, and K. Wu. DEX: Increasing the Capability of Scientific Data Analysis Pipelines by Using Efficient Bitmap Indices to Accelerate Scientific Visualization. In *International Conference on Scientific and Statistical Database Management (SSDBM)*, Santa Barbara, California, USA, June 2005. IEEE Computer Society Press.

[16] K. Stockinger, J. Shalf, W. Bethel, and K. Wu. Query-Driven Visualization of Large Data Sets. In *IEEE Visualization 2005*, Minneapolis, MN, October 23-25, 2005, IEEE Computer Society Press.

[17] K. Stockinger, K. Wu, S. Campbell, S. Lau, M. Fisk, E. Gavrilov, A. Kent, C.E. Davis, R. Olinger, R. Young, J.E. Prewett, P. Weber, T.P. Caudell, E.W. Bethel, S. Smith. "Network Traffic Analysis With Query Driven Visualization SC 2005 HPC Analytics Results." In *Supercomputing 2005, HPC Analytics Challenge*, November 2005.

[18] K. Stockinger, K. Wu, and A. Shoshani. Evaluation Strategies for Bitmap Indices with Binning. In *International Conference on Database and Expert Systems Applications (DEXA)*, Zaragoza, Spain, September 2004. Springer-Verlag.

[19] K. Wu, E. J. Otoo, and A. Shoshani. An Efficient Compression Scheme for Bitmap Indices. To appear in *ACM Transcations on Database Systems*, 2006. ACM Press.

[20] K. Wu, E. J. Otoo, and A. Shoshani. On the Performance of Bitmap Indices for High Cardinality Attributes. In *International Conference on Very Large Data Bases (VLDB)*, Toronto, Canada, August 31 - September 3, 2004, Morgan Kaufmann.

[21] K. Wu, W. Koegler, J. Chen, and A. Shoshani. Using Bitmap Index for Interactive Exploration of Large Datasets. In *International Conference on Scientific and Statistical Database Management (SSDBM)*, Cambridge, MA, 2003. IEEE Computer Society Press.