Parallel Index and Query for Large Scale Data Analysis

Jerry Chou
Mark Howison
Brian Austin

Kesheng Wu
Ji Qiang
E. Wes Bethel
Arie Shoshani

Oliver Rübel
Prabhat
Rob D. Ryne

ABSTRACT

Modern scientific datasets present numerous data management and analysis challenges. State-of-the-art index and query technologies are critical for facilitating interactive exploration of large datasets, but numerous challenges remain in terms of designing a system for processing general scientific datasets. The system needs to be able to run on distributed multi-core platforms, efficiently utilize underlying I/O infrastructure, and scale to massive datasets.

We present FastQuery, a novel software framework that address these challenges. FastQuery utilizes a state-of-the-art index and query technology (FastBit) and is designed to process massive datasets on modern supercomputing platforms. We apply FastQuery to processing of a massive 50TB dataset generated by a large scale accelerator modeling code. We demonstrate the scalability of the tool to 11,520 cores. Motivated by the scientific need to search for interesting particles in this dataset, we use our framework to reduce search time from hours to tens of seconds.

General Terms

Indexing Systems, Large Scale Data Analysis, Performance

1. INTRODUCTION

Modern scientific instruments and simulations produce vast amounts of data [12, 21]. The immense size of these data collections makes the process of scientific exploration unwieldy, thereby having a negative impact on the process of scientific discovery. We observe, however, that gaining insights from data typically involves interactive exploration of smaller regions or subsets of data. Can we somehow accelerate the search for smaller regions of interest? This problem has gained much attention in the database research community. A variety of indexing methods are used in commercial database systems to accelerate the search process [15].

However, most scientific datasets are not stored in these database systems; hence, they are unable to take advantage of these indexing techniques. It might be too expensive to make copies of datasets for indexing purposes, and a conversion of file formats might break downstream visualization and analysis tools. We take the approach of applying state-of-the-art indexing techniques *within* modern scientific data formats. This paper presents FastQuery, a massively parallel indexing and querying system that works with a variety of popular scientific data formats, such as HDF5 [25] and NetCDF [26]. We present a detailed discussion of FastQuery's design, implementation issues and performance studies to highlight the application to modern scientific datasets.

We need to overcome two significant design challenges in order to build a general index and query system for scientific data. The first challenge is to design a generic system that can accommodate the majority of scientific data models, and the second one is to enable high levels of concurrency in building and using indexes. Among the known scientific data storage formats, we see arrays as the most common data model used. Our system, therefore, supports arbitrary multi-dimensional arrays. In this regard, our design choice is similar to the one made by the SciDB effort [24]. In contrast to SciDB, however, we do not require the users to input their data into our system. This removes the need to copy the data into a separate data management system and avoids the need to rewrite data analysis functions for interacting with the new data management system.

To effectively utilize modern computing platforms, it is necessary for the index and query system to expose high levels of concurrency in all aspect of its operations. This imposes a number of restrictions on the type of applicable indexing methods. Our review of existing indexing methods shows that a compressed bitmap index is a good choice overall [28, 29], but such methods have not been applied in a parallel setting. FastQuery utilizes a parallel version of compressed bitmap indexes by breaking the user input data into non-overlapping subarrays.

We also need to tackle the challenge of sheer size (TBs-PBs) of modern datasets. I/O devices such as disks and SSDs, provide performance in the order of 100-500 MB/s, which makes it infeasible to read the entire dataset from a single device in any reasonable amount of time. It is, therefore, imperative that we aggregate I/O resources to both read and write data. FastQuery utilizes parallel I/O capabilities through the HDF5 layer to overcome this challenge.

Our key contributions are as follows:

- We design an index and query system that can work with arbitrary array-based data (Section 3). In contrast, existing database management systems require

SC11, November 12-18, Seattle, WA, USA

the data to be under their control and earlier attempts at building similar indexing systems required strict organizations of the data hierarchy [9].

- We implement a flexible strategy to index arbitrary subsets of the data at a time (Section 3.2). This feature reduces the memory requirement for creating the indexes and allows a higher number of concurrent tasks to build and use indexes.
- We parallelize the index building and query processing procedures to make efficient use of large numbers of distributed many-core processors (Section 4), and carefully study the trade offs among the various parameters.
- We demonstrate the effectiveness of our approach using a massive, 50 TB simulation data set, and conduct a performance study with 11,520 cores (Section 5 and 6).
- We conduct a careful study of the interactions between the index building and query software, and the I/O systems. From this, we derive a set of practical recommendations for index building and query processing on parallel platforms (Section 6).
- We demonstrate the practical applicability of FastQuery to real-world scientific problems; we accelerate query-based analysis of a large-scale high-resolution simulation dataset of beam dynamics in a electron linear particle accelerator (Section 7).

2. RELATED WORK

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

2.1 Scientific Data Formats

Scientific applications generally store their data in application-specific data formats. Over the years, a consensus has gradually emerged that arrays can be used to capture the main data structures required for scientific data. Thus, the commonly used scientific data formats are designed to store arrays efficiently [25, 26]. For this reason, we designed FastQuery to work with arbitrary array-type data.

A number of research database systems, such as, SciDB [24] and MonetDB [3], are based on similar array data models. In contrast to SciDB and MonetDB, however, our approach does not require the user to load their data into the database system, avoiding the need for additional data copies. This is a significant benefit, in particular considering the massive volume of many scientific data collections. In addition, our approach makes it possible to integrate the indexing capability directly with the scientific data formats themselves.

2.2 Indexing Techniques

A variety of indexing techniques are available in popular database systems [17], many of which are variations of the B-Tree [4]. These types of indexing methods are designed for transaction-type applications, exemplified by interactions between a bank and its customers. Typical interactions with scientific data, however, are significantly different from operations on transaction-type data.

First, a typical search operation in transactional data retrieves very few data records, such as a look-up of a single customer's banking account information. In contrast, search operations on scientific data commonly retrieve many more data records. For example, a scientist might be interested in studying how the ignition progresses in a combustion simulation from a spark into a flame engulfing the whole combustion chamber. In this case, resolving the query of interest might result in a few records in the beginning of the simulation, but might expand to include the majority of records towards the end.

Second, transactional data is frequently modified, one record at a time, whereas scientific data typically stays as is after it has been generated. The B-Tree data structure is designed to update quickly as the underlying data records are modified. This feature is unnecessary in indexes for the majority of scientific data sets. For such scientific data sets, the bitmap index is a more appropriate indexing structure [16][21, Ch. 6].

2.3 Bitmap Indexing Technology

A bitmap index logically contains the same information as a B-Tree, which 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 bitmaps can be processed efficiently, this index can answer queries efficiently as demonstrated by Patrick O'Neil in Model 204 [16].

The basic bitmap index uses one bitmap for each distinct key value. In the context of scientific data, the number of distinct values can be as large as the number of rows (i.e., every value is distinct). In this case, the number of bits required to represent an index may scale quadratically with the number of rows, i.e., 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 in practice.

A number of different strategies have been proposed to reduce the sizes of bitmap indexes and improve their overall effectiveness. Common methods include compressing individual bitmaps, encoding the bitmaps in different ways, and binning the original data [21, Ch. 6]. We here choose FastBit [27] — an open-source software that implements many of these methods — as a representative of general indexing methods. FastBit has been shown to perform well in a number of different scientific applications [27] and a series of theoretic computational complexity studies further establish its effectiveness [28, 29].

2.4 Distributed Indexing

The benefits of performing data analyses in parallel have been recognized since the dawn of parallel computing [10, 18]. Unlike many other high-performance computing applications, database operations are dominated by data-accesses to disk 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. Netezza uses an active storage approach where the disk controllers are modified to carry out certain database operations [6]. Teradata [2, 8] employs a specialized interconnect, called BYNET, to increase the bandwidth among data access modules. Alternatively, a number of researchers proposed to store the data in main memory [7, 14]. Such hardware-based solutions are typically not available to scientific users, or are unable to handle the

large data volumes required. To maximize the impact of our index and query system, our approach is purely software-based, using the popular message passing library MPI [11] and requiring only 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 [23]. 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 filesystem 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 applications and have implemented our parallel indexing system based on the sequential bitmap index software FastBit [27].

3. FASTQUERY

The goal of FastQuery is to provide a query selection mechanism for arbitrary scientific data formats that are array oriented. To achieve this objective, FastQuery utilizes the FastBit indexing and query technology to allow data selection based on arbitrary range conditions defined on the available data values, e.g., "energy $> 10^5$ *and* temperature $> 10^6$". Furthermore, we support subarray data operations to allow data querying and indexing of arbitrary subarray coordinates within a dataset to find, for example, the data satisfying the condition "vapor $> 10^5$" in the 5^{th} row of the dataset. We now briefly introduce the overall design and features of FastQuery.

3.1 Architecture

Figure 1 illustrates the FastQuery system architecture. The main system components are as follows:

The Query Processor and Index Builder are initiated by the user to perform index building and querying. The index builder builds the indexes for a whole or a subset of a dataset and stores the indexes in a file. The query processor accepts text-string queries from the user, then performs data selection using the stored indexes and returns the selection results to the users.

The FastQuery Parser is an internal component of FastQuery responsible for parsing user information given by the query processor and index builder. It defines and implements a simple, yet flexible, naming convention and query syntax for users to describe their requested variables and query constraints through the FastQuery API.

The array I/O Interface defines an I/O API for FastQuery to access data and bitmap indexes stored in an array model file. The primary functions of the interface are to read data, and load and write indexes. We expect the user to implement file format (e.g. HDF5 or NetCDF) specific calls to implement the API. Once the interface for the data format has been implemented, FastQuery can provide the data query and indexing functionality for the data format. This extensible design choice greatly increases the applicability of FastQuery to a wide range of present (and future)

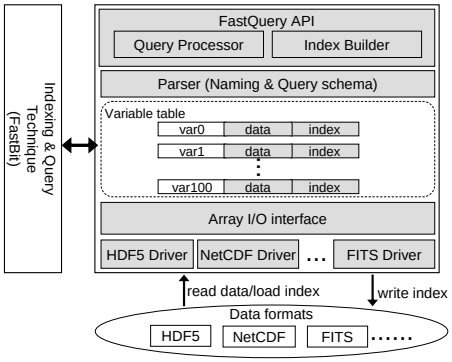

Figure 1: FastQuery architecture

file formats. All of the functionality and performance features presented in this paper use an HDF5 implementation of the I/O interface. We have, in addition, implemented a NetCDF interface, but we will not discuss that in this paper.

The Variable Table represents the relational data model used by FastQuery and other indexing and query technology [27, 4]. A variable table contains a list of columns, each of which maps to a variable (i.e. a dataset in HDF5 terminology), and each row maps to a record of a variable (i.e. data at a mesh point). FastQuery applies FastBit to build and query indexes by reading the data and bitmaps associated with each record in the table.

3.2 Subarray Indexing and Query

FastQuery deploys subarrays as means to partition data for indexing and querying in a parallel context. Since subarrays, by definition, are smaller than the complete dataset, the time for creating indexes and executing queries can be greatly reduced as compared to an approach which is constrained to processing the entire dataset as a whole. The usage of subarrays, furthermore, has the advantage that it allows for greater flexibility in the data analysis.

FastQuery adopts a subarray specification similar to the one used by Fortran and other programming languages. Here a subarray is specified by the general form "[*lower* : *upper* : *stride*]". The *lower* and *upper* indicate the first and last record position of the subarray in a dataset. The *stride* is the step size between *lower* and *upper* (default is *stride* = 1). For a n-dimensional dataset, the subarray range of each dimension must also be specified and separated by comma in the bracket.

4. APPROACH

In this section, we describe the parallel design and implementation of FastQuery.

4.1 Parallel Processing and Design

The fundamental goal of our parallel FastQuery design is to enable building and querying of indexes for subsets of data independently and concurrently. We achieve this goal by utilizing the subarray feature provided by the FastQuery design architecture. As described in Section 3.2, a subarray is a subset of data within a variable, and the size of a subarray can be specified with arbitrary length. Therefore, a single variable in a dataset can be divided into multiple subsets by specifying the subarray size. Then the bitmap

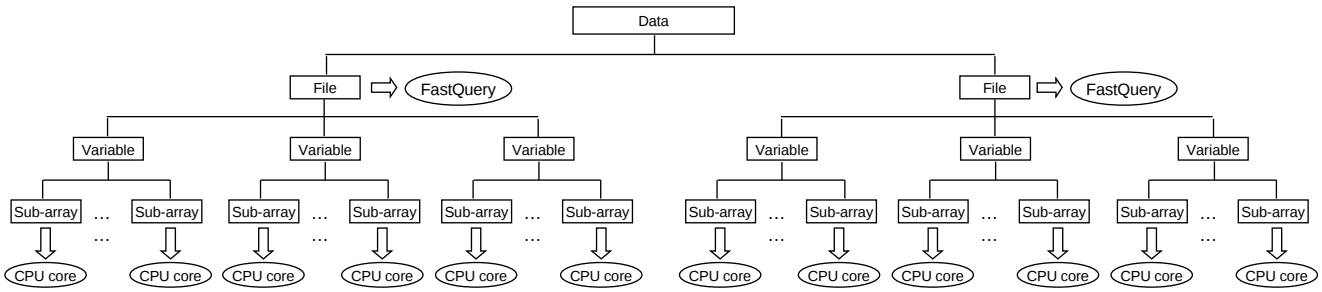

Figure 2: FastQuery exploits parallelism at three different levels: files, variables and subarrays. The degree of parallelism can be configured by specifying the number of cores to be used per file, per variable, and the size of subarrays.

indexes for each subarray can be built and queried independently for parallel processing. It is important to note that we can support query resolution at a different concurrency than what was used for creating the subarrays in the first place.

For example, a variable "x" with 1,000 records, could be treated as 10 subarrays, such as x[1:100], x[101:200], ..., x[901:1000]. The indexes of "x" could be built in parallel with 10 cores by using one core per subarray. Once the indexes are built, a query, such as "x>0", can be answered in parallel by dividing the query into 10 subarray queries (matching the indexed range). The subarray queries are then iteratively assigned to the cores. For instance, if we run the query on 5 cores, core 1 processes the subarray query "x[1:100]>0" and "x[501:600]", core 2 processed "x[101:200]" and x[601:700]", and so on. Needless to say, the 5 cores will process their subarray queries in parallel.

Moreover, as illustrated in Figure 2, we exploit the parallelism within FastQuery at three levels: files, variables and subarrays. At the topmost level, each FastQuery instance is associated with a file. Our parallel implementation allows a group of cores to collectively create a FastQuery instance and process the same file. Hence, we can easily create multiple FastQuery instances with different groups of cores to process files in parallel. Within a file, the group of cores for the file can be divided into sub-groups, and each sub-group can be responsible for the indexing and/or querying of one variable at a time. Finally, we divide the data of a variable into a set of subarrays, so that each core can index and/or query these subarrays in parallel.

At each level (i.e. files, variables, subarray), the degree of parallelism can be configured by specifying the group size per file, the sub-group size per variable, and the subarray size. Then during processing, the groups iterate through the files, the sub-groups iterate through the variables, and the cores iterate through the subarrays. Since the size of subarrays can be specified arbitrarily, the parallelism of FastQuery is only limited by the number records in the dataset. In other words, when the subarray size is 1, we could achieve the maximum parallelism by using the same amount of cores as the number of records in a dataset, and each core only processes one record. However, the advantage of using bitmap indexing for querying is to evaluate bitmaps instead of scanning through individual data values. As shown by our evaluation in Section 6, the subarray size should be chosen with a relatively large number, such as millions of records.

4.2 Parallel I/O and Implementation

While the computations of querying and indexing described in the previous subsection are independent across subarrays, the I/O operations of reading/writing indexes on a file could require coordinations that are imposed by the underlying I/O library, such as NetCDF or HDF5. For instance, a file library may require data to be read or write collectively by all the processors participating in the file. However, as shown in Figure 1, these parallel I/O issues are handled by the corresponding implementation of FastQuery array I/O interface rather than by the FastQuery itself. This is because the interface implementation is responsible for performing all the I/O operations. Therefore, in the following section, we use our current implementation on the HDF5 file library to describe our parallel I/O approaches in more details.

Our implementation is illustrated by the example in Figure 3. In the example, we attempt to use 4 cores to build the indexes of a variable divided into 10 subarrays. As mentioned in the previous subsection, these four cores would iteratively build the bitmap indexes of each subarray in 3 iterations. Once the indexes built, they would have to be stored in file as new variables (HDF5 dataset) for query processing in the future.

However, due to the parallel I/O requirement from HDF5, all the processors have to create new variables collectively. As a result, the indexes for each of the subarray cannot simply be stored in separated HDF5 dataset. Therefore, in our implementation, we compact all the built indexes of a variable into a single array and store to one bitmap variable. Thus, all processors could create and operate on the same HDF5 variable collectively.

Another challenge to our implementation is that although the data subarray size is fixed, their corresponding compressed bitmaps generated from FastBit could still have varied length. As shown in the figure, the bitmaps of the first subarray is 100, and the the bitmaps of the second subarray is 80, and so on so forth. Therefore, we use an additional variable "offset table" to record the position of the bitmaps for each subarray. When processing a query, each core can then find the corresponding bitmaps for its subarray based on the information available in the offset table.

Based on our implementation Figure 4 shows the steps of building and querying indexes in FastQuery. For building indexes, each processor could independently reading data and building indexes for its own subarray. But, processors would have to collectively create the variable to store bitmaps and the offset table. In particular, because bitmaps of subarrays

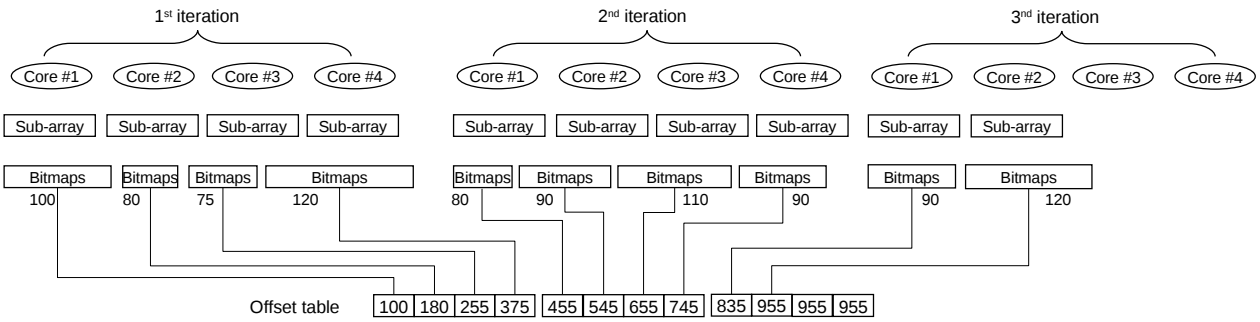

Figure 3: Bitmap indexes are built for each subarray and stored as a single compact array. An offset table is created to store the offset position of the bitmaps of each subarray, so that the cores can based on the information to load the bitmaps when processing queries.

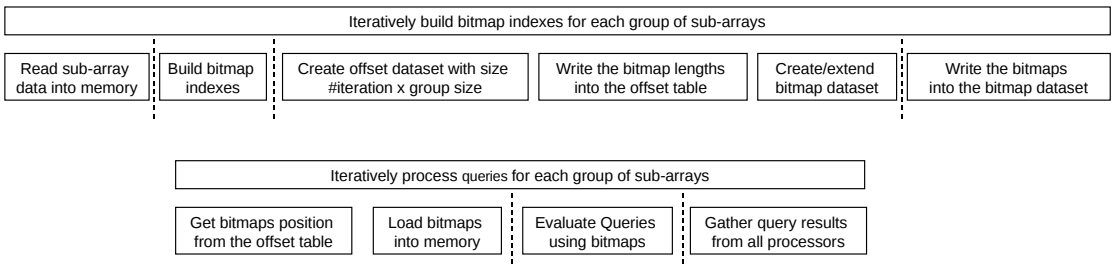

Figure 4: parallel indexes building and querying process.

are generated iteratively, the final length of all the bitmaps is not known. Therefore, the dataset for storing the bitmaps has to be extended iteratively after indexes are built for the iteration. On the other hand, the dataset for storing the offset is fixed and the same as the total number of subarrays. So, it could only be created as the fixed size and need not be changed in each iteration. Once the dataset for bitmap is created or extended collectively by all the processors, the bitmaps can then be written to file independently.

For the querying process, since processors only read bitmaps information from files, each processor could independently read its bitmap position from the offset table, then load the necessary bitmaps to memory to evaluate the query. Hence, query processing does not require any collectively file library call, but there still could be contention among processors when they attempt to access the file at the same time.

5. EXPERIMENTAL METHODOLOGY

5.1 Research Questions

Our FastQuery implementation has two major functions: creating indexes and resolving queries. We are interested in addressing the following questions for both of these tasks:

- Does the implementation scale as we increase the number of cores (strong scaling)?
- What is the optimal subarray size for this task?
- What is the optimal group size for this task?

In order to address these questions, we performed a number of detailed performance studies which are listed in Section 6. But first, we describe the hardware platform and the datasets used in the study.

5.2 Testbed

We conducted our experiments on the NERSC Cray XE6 supercomputing system Hopper [1]. The system has ≈6,500 compute nodes, with 24 cores and 32GB memory per node. Hopper uses Lustre as its filesystem, with a peak theoretical I/O bandwidth over 25GB/s. For each experiment, we launch FastQuery as a set of MPI tasks with one core per task. To prevent memory contention between MPI tasks on the same node, we limited the memory usage of each FastQuery process to 2GB, using 12 cores per node; the rest of memory space is reserved for system usage, such as initiating MPI tasks, etc. To achieve better I/O performance, we applied several I/O optimization strategies to tune HDF5 for Luster. As suggested in [13], we disable metadata cache evictions, use POSIX mode and increase Luster stripe count.

5.3 Dataset

For our evaluation we use a large-scale, high resolution physics simulation dataset generated by the IMPACT-T [19] code. The data set consists of 720 time steps with 1 billion particles per time step. Each time step is stored in a single HDF5 scientific data format file and contains measurements of each particle over 9 variables, such as the physical coordinates and particle momentum. The size of each file is around 68GB, and the total size of the whole dataset is around 50TB (=68*720). Section 7 discusses the simulation and application problem in more detail.

5.4 Performance Measurements

In our experiments, we measure the wall-clock time taken by FastQuery to perform two main functions: building and querying indexes. As shown in Figure 4, the process of building indexes of a file includes three steps: reading data, build-

[1] `http://www.nersc.gov/nusers/systems/hopper2/`

ing index and writing index. Although cores can read/write data and build indexes independently in each iteration, they do share a limited number of I/O servers and require collective I/O calls when modifying the metadata of a file. As a result, there is a certain amount of waiting time associated with each step from each core. To ease interpretation of the performance measurements we let all cores that participate in a single file, start each step simultaneously, and wait until all of them complete the step. For each step, we record the completion time and the average wait time of processors in each iteration. We report the aggregated numbers over several iterations for each step of the file. When indexes are built for multiple files and using different groups of processors, we report the average time among groups as well as the final completion time after all indexes are built and stored to files.

Similarly, the process of querying indexes consists of two iterative steps: reading index (or data if no index is available) and evaluating index/data. In contrast to the build index process, these two step do not require collective I/O calls from the HDF5 library. We, therefore, record the time of these two steps independently for each processor, and aggregate the numbers over several iterations. We report the average values among processors as well as the final completion time after the query is processed on all files.

6. PERFORMANCE EVALUATION

In this section, we present a variety of performance results for the steps of creating indexes and resolving queries. For both of these tasks, we vary three parameters (total number of cores, size of the subarray and the group size) and report on the observed performance characteristics.

While the FastQuery implementation allows us to specify arbitrary values for each of the three parameters, we chose a reasonable configuration space to simplify the discussion of results. For instance, we chose configurations wherein we have even load balancing across cores. In other words, each core processes the same or a similar number of subarrays. In almost all cases, we also constrain all the cores (12 in our case) on a single node to process a single file (at a time). Hence the group size (i.e. the number of cores assigned to a file) is a factor of 12. Again, the FastQuery implementation will entertain suboptimal configurations such as load imbalance across cores, and different cores on a single node processing different files, but we will not analyze those problem configurations for the ease of analysis.

6.1 Building Indexes

Here, we evaluate the performance of the task of building indexes. We specify a 3-digit precision binning option for the FastBit indexes [29]. The resulting indexes of each file is 45GB which is about 66% of the original data size of 68GB. Building indexes is an expensive operations, involving large amounts of time for doing file I/O, for reading the entire dataset and writing indexes. In a production setting, building indexes is a one-time preprocessing operation, but since we need to explore a multi-dimensional configuration space, we conducted our experiments using a smaller 4TB subset of the data (corresponding to 60). For the whole dataset, we present the scalability results under varied number of cores later on in Figure 15.

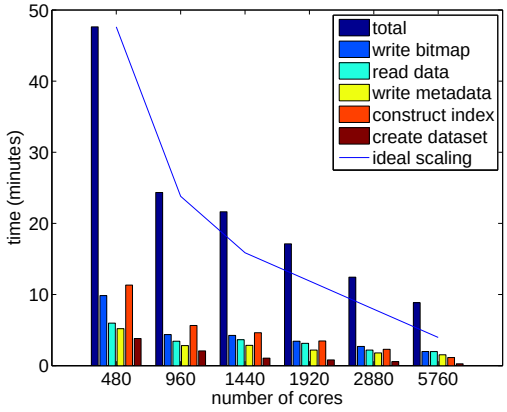

Figure 5: Building index time for 60 time steps (4TB) as the number of cores increases from 480 to 5760.

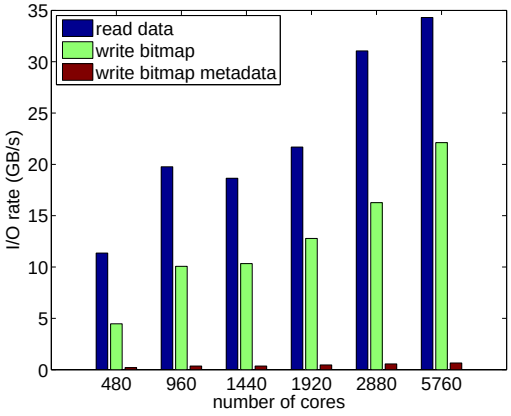

Figure 6: Building index I/O rates for 60 time steps (4TB) as the number of cores increases from 480 to 5760.

6.1.1 *Does the implementation scale with number of cores?*

For this set of experiments, we fixed the size of subarrays to 10 millions records, because it produced the best performance (as we will show in the next subsection). Since each variable in our dataset has 1 billion records, it results in 100 subarrays per variable under the given subarray size. Accordingly, we fixed the number of cores assigned to a file to be 96, so that the indexes of each variable can be built in one iteration (i.e. only the first 4 cores have to build indexes in the second iteration).

In Figure 5, we plot the results when the number of cores increases from 480 to 5760. The maximum number of cores used in this experiments is 5760 (=60*96) because each file is assigned to 96 cores and we only have 60 files. The other numbers are chosen to ensure all cores participate with the same number of files and are roughly load balanced. For instance, with 2880 cores, 30 (=2880/96) files could be built simultaneously, and the whole dataset (i.e. 60 files) would be built in 2 iterations. Accordingly, the indexes can be built in 3 iterations with 1920 cores, 4 iterations with 1440 cores, and so on.

As expected, from Figure 5 we observe that the total time to create indexes decreases as we add more concurrency.

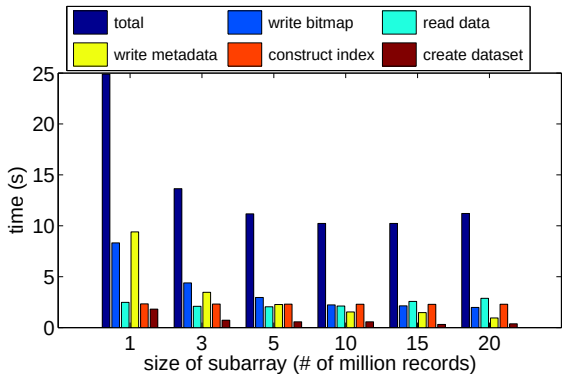

Figure 7: Building index time for 60 time steps (4TB) as the subarray size increases from 1 million records to 20 million records.

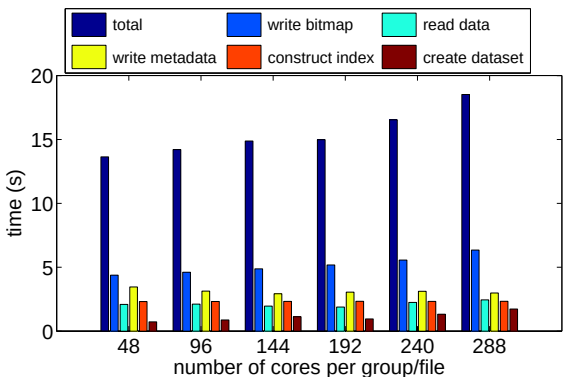

Figure 8: Building index time for 60 time steps (4TB) as the number of cores per file increases from 48 to 288.

More specifically, we find that the time for constructing index perfectly scales with the number of cores, as the time is reduced from 48min at 480 cores to 24min at 960 cores to 12min at 1920 cores. On the other hand, the I/O time, including read/write data, bitmap and bitmap metadata, reduces at at a sub-linear rate as the number of cores increases. Therefore, the total time also decreases at a sub-linear rate as compared to the ideal linear scaling results indicated in the figure by a solid line.

In Figure 6, we plot the corresponding I/O rate from the above results. The read rate is computed from dividing the total size of data files by the data read time observed, while the write rate is computed from dividing the total size of generated indexing files by the bitmap and bitmap metadata write time observed. We found reads (10-35GB/s) clearly have much better performance than writes (5-20 GB/s). We postulate a couple of possible explanations: Reads require less synchronization and coordinations among cores, and I/O is more efficient when data is accessed in big chunks. The raw data is read in subarrays of 10 million entries, while the generated bitmaps and bitmap metadata are much smaller in size. As observed for each file, the overall size of data, bitmap and bitmap metadata is 68GB, 44GB and 1GB, respectively. We also observe fairly poor performance for writing bitmap metadata.

6.1.2 *What is the optimal subarray size?*

Next, we vary the size of the subarray used in creating indexes. Figure 7 plots the results of subarray size across 1M, 3M, 5M, 10M, 15M and 20M entries. In these sets of experiments, the total number of cores was fixed to 2,880 (=60*48), and the number of cores assigned to each file was fixed at 48. Hence, all 60 files were built at the same time in one iteration.

From Figure 7, we observe that the performance improves as we move towards 10M entries, and then becomes worse as we further increase the subarray size. The detailed time breakdown indicates that the time for writing bitmap and bitmap metadata is greatly reduced as the subarray size increases from 1M to 20M. This is because a larger subarray size implies that a fewer number (but larger in size) bitmaps are constructed and written to file. But the difference does become negligible after the subarray size is larger than 5M.

On the other hand, the time for reading data reaches its minimum value, 2 minutes, at a subarray size of 5M, then the time actually becomes larger as the subarray size increases. In fact, we also observed the same minimum read time in Figure 5 at different parameter setting. We hypothesize that the reads have reached the maximum rate of 35GB/s (=60*68/120) that the system can sustain. Thus, the increasing subarray size would not improve read times after saturation.

Finally, the time for constructing indexes remains the same because similar amounts of indexes were built for the dataset regardless of the choice of subarray size.

6.1.3 *What is the optimal group size?*

Finally, we varied the group size (i.e. number of cores assigned to a file) from 48 to 288. This corresponds to a fixed total number of cores of 2,880 and a subarray size of 3 million records. The minimum group size is 48 because we would like to utilize all cores for processing 60 files. For instance, using 24 cores per file will only utilize 1440 (=60*24) cores. The results plotted in Figure 8 show that increasing the group size is detrimental to overall performance. In particular, we find the time for creating dataset and writing bitmap both increase as more cores participate in a file, because more cores have to be coordinated and synchronized to perform these operations. However, we did find that the time for writing bitmap metadata decreases. It is likely because more metadata can be aggregated from different cores and written into file at the same time.

6.1.4 *Summary of Results*

In summary, for the task of building indexes, we observe that our implementation scales well to 5670 cores, we find that the optimal subarray size is 10M, and that 48 is a reasonable choice for group size.

6.2 Querying Indexes

We now consider the important task of resolving queries in parallel. We evaluate our system performance by resolving a query "$r > 4\sigma(r)$" on each of the 720 files from our 50TB dataset. As we will describe in more details in Section 7, the query is formulated by our particle physics collaborators to find the halo particles from the dataset. We resolve this query using the FastQuery implementation and present our performance results.

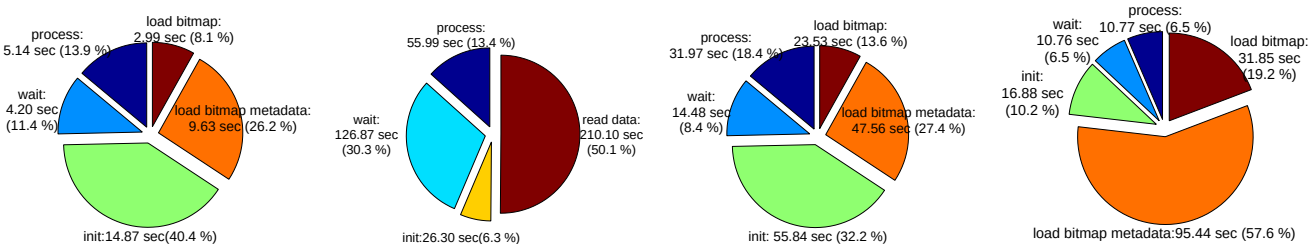

Figure 9: Query time percentage breakdown: 9(a) using indexing on 2880 cores and subarray size 10M, 9(b) without indexing on 2880 cores, 9(d) using indexing on 240 cores and subarray size 10M, and 9(d) using indexing on 2880 cores and subarray size 1M.

6.2.1 *Does the implementation scale with number of cores?*

Figure 10 shows performance results for the aforementioned query as we increase the number of cores from 240 to 2880. We also plot the results for a baseline technique which does not have access to index information, hence, is constrained to load and scan through the entire dataset. The results show that we achieve almost an *order of magnitude* faster performance across the board. On 2880 cores, the baseline techniques takes nearly 7 minutes, while FastBit indexing (with 10M subarray size) takes only 35 seconds. This corresponds to a performance improvement of over 93% and demonstrates the power of FastQuery accelerated analysis.

We now examine detailed timing information pertaining to these two cases for 2880 cores. Figure 9(a) and Figure 9(b) show the times taken by different stages of the query resolution task. We clearly see the majority of time is spent on reading the data from file in the non-indexed case (see Figure 9(b)). In contrast, FastQuery takes less than 10 seconds to load the indexes for evaluating the query. The "init" step shown in these figures corresponds to a one-time overhead cost of opening files and loading file metadata information. As a result, the high percentage of init time observed in Figure 9(b) is due to the fact that we only evaluate a single query per file. But, this overhead would become negligible as more queries are evaluated per file.

From Figure 10, we also see FastQuery scales well with increasing numbers of cores. As we go from 240 to 2880 cores, the absolute time (with 10M subarray size) reduces from 173 to 37s and from 816s to 166s (with 1M subarray size). If we compare the time percentage breakdown at 2880 cores and 240 cores in Figure 9(a) and Figure 9(d), we see the percentages are almost identical; this implies that all components in the FastQuery query resolution step are scaling at the same rate.

6.2.2 *What is the optimal subarray size?*

We build indexes for the 50TB dataset with subarray sizes of 1M, 10M and 20M and explore the performance implications for parallel query resolution. We fix the group size at 48 for all cases. The results are presented in Figure 10.

As indicated earlier, the strength of bitmap index based techniques lies in taking large datasets, and compressing them into smaller bitmaps. Hence using smaller "subarrays" is suboptimal from the point of view of query resolution. We can see this in Figure 10, where the 10M subarray consistently outperforms the 1M subarray. Examining detailed time breakdown in Figure 9(a) and Figure 9(d) indicates

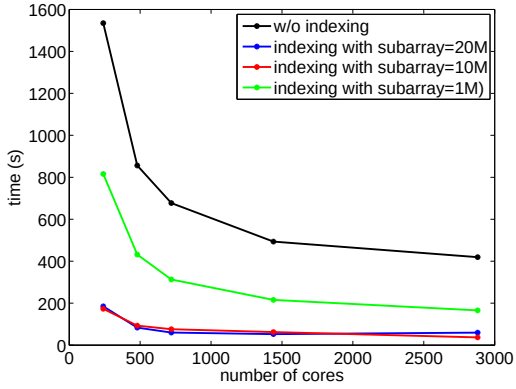

Figure 10: Query performance comparison for different subarray sizes.

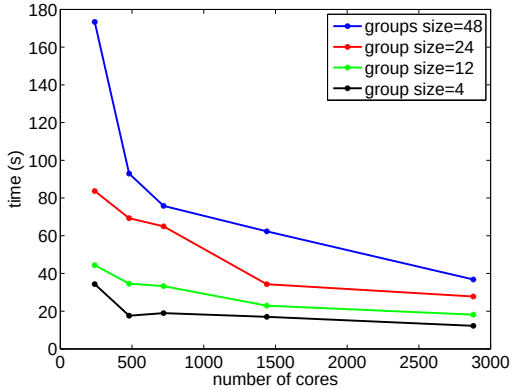

Figure 11: Query performance for different group sizes.

that FastBit appears to be taking the same amount of time to process a subarray. The 1M case (compared to the 10M case) results in 10 times more subarrays, hence FastQuery takes a proportionally longer time in loading bitmaps and bitmap metadata.

The open question at this stage is: will increasing subarray sizes further help improve performance? We tried using subarrays sized 20M, and observe comparable performance results. We are still conducting tests to better understand this behavior. It is important to note that we need to decide on a subarray size during the index creation process. It appears that in this case, both the index creation task, and query resolution task seem to hint at 10M entries as being a reasonable choice for subarray size.

6.2.3 What is the optimal group size?

We varied the group size (i.e. number of cores processing a file) across 4, 12, 24 and 48. We hold the subarray size fixed at 10M. The results are shown in Figure 11. We observe that the query time improves as we reduce the group size. The best performance is obtained for a group size of 4. We postulate that as we decrease the number of cores processing a file, we are decreasing the synchronization requirements (as far as the HDF5 and filesystem is concerned). This raises the issue of whether reducing the group size to 1 (essentially file per proc) would give the best performance. We hypothesize that this might be the case, modulo I/O contention on a per-core basis. We were not able to test this particular configuration in our setup because we only had 700 files overall, and we were testing at higher concurrencies.

6.2.4 Summary of Results

To summarize our results from parallel query resolution, we observe that our implementation scales well to 2880 cores, we found that using a subarray size of 10M entries gives good performance, and that the group size should be as small as possible (at a reasonable concurrency) for best performance.

7. APPLICATIONS

Query-based data analysis as a general analysis tool has a wide range of applications. The index/query system FastBit has been used, e.g., to analyze extremely large computer network traffic data [22] and data from laser wakefield particle accelerator simulations [20]. Here we will focus on another example involving electron linear particle accelerators.

Particle accelerators are among the most versatile and important tools of scientific discovery. They are essential to a wealth of advances in material science, chemistry, bioscience, particle physics, and nuclear physics. They also have important applications to the environment, energy, and national security. Given the importance of particle accelerators, it is essential that the most advanced high performance computing tools be brought to bear on accelerator R&D, and on the design, commissioning, and operation of future accelerator facilities.

Here we focus on the analysis of data from large-scale, high resolution simulations of beam dynamics in electron linacs for a proposed next-generation x-ray free electron laser (FEL) at Lawrence Berkeley National Laboratory (LBNL) [5]. Particle-in-cell-based simulations of this type of accelerator require large numbers of macroparticles ($> 10^8$) to control the numerical macroparticle shot noise and avoid overestimation of the microbunching instability, resulting in massive particle datasets [19]. In the following we will demonstrate how our parallel index/query system can be used to enable fast, query-driven analysis of this type of data.

7.1 Experiment

For the purpose of this case study we focus on the analysis of characteristic subparts of a particle beam, in particular the transverse halo and core of the beam.

Transverse Halo: The transverse halo of a particle beam is a low density portion of the beam usually defined as those particles beyond some specified radius or transverse amplitude in physical space (Figure 12). Particles in the halo have the potential to reach very large transverse amplitude, eventually striking the beam pipe and causing radioactivation of accelerator components and possibly causing component

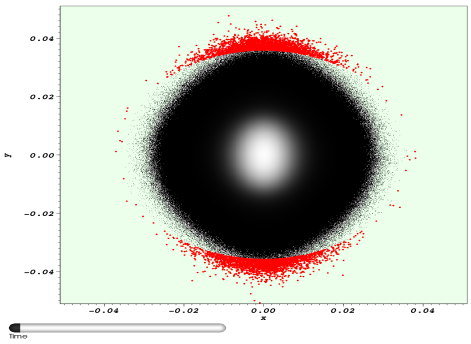

Figure 12: Particle density plot (gray) and particles selected (red) by the halo query for timestep 20 of the simulation.

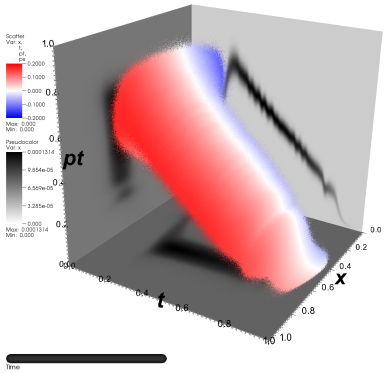

Figure 13: Scatter plot of all particles in x, t, pt space with particle color indicating the momentum in x direction px (red=positive, blue=negative). The axes of the plot have been normalized for display purposes. The gray plots on the side and bottom show the two-dimensional particle density in x/t, x/pt and t/pt, respectively.

damage. Controlling the beam halo is critical to many accelerator facilities and is often the limiting factor to increasing the beam intensity to support scientific experiments. In order to better understand this phenomenon, physicists need to be able to efficiently identify and extract the halo particles for subsequent analysis, e.g., to determine the origin of halo particles, to provide insight to help prevent the halo from forming, and to provide information to help develop halo mitigation strategies. To identify halo particles we use the query $r > 4\sigma_r$, with $r = \sqrt[2]{x^2 + y^2}$ and $\sigma_r = \sqrt[2]{\sigma_x^2 + \sigma_y^2}$, and where σ_x and σ_y denote the rms beam sizes in x and y, respectively. We here create an additional index for the derived quantity r in order to accelerate the query. The query as defined above is based on the assumption of an idealized circular beam cross section. To ensure that the query adapts more closely to the transverse shape of the beam one may consider scaling the x and y coordinates, using, e.g., a query of the form $r_s^2(x, y) > 16$ with $r_s^2(x, y) = (\frac{x}{\sigma_x})^2 + (\frac{y}{\sigma_y})^2$.

Core: Creating beams with good longitudinal beam quality (defined in terms of a quantity called the longitudinal rms emittance), and maintaining that beam quality during the acceleration process, is critical in certain types of accelerators involving intense electron beams. Frequently the "head" and "tail" of an electron bunch will be very different from

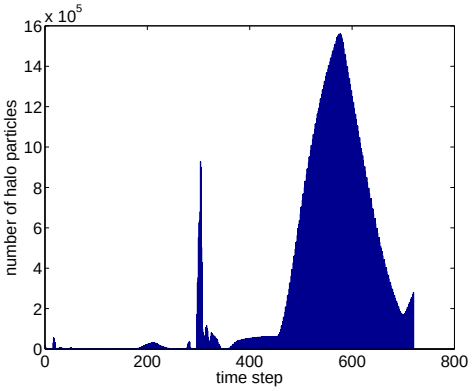

Figure 14: Plot showing the number of halo particles versus timestep.

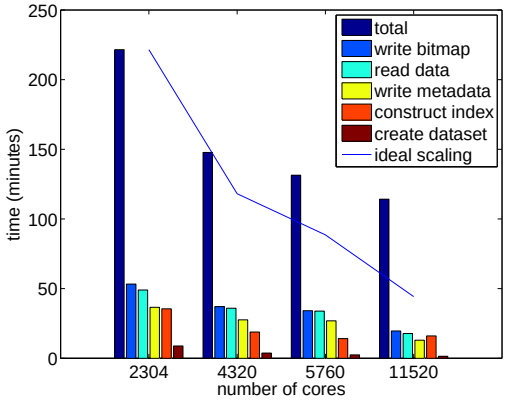

Figure 15: Build index time for 720 time steps (50TB) as the number of cores increases from 2304 to 11,520.

its "center" region, the longitudinal "core." As can be seen in Figure 13, focusing on the longitudinal core of a bunch eliminates strong variations at the head and tail which would otherwise distort analysis results. The longitudinal particle motion can be described by a longitudinal coordinate, z, or equivalently, by a particle's arrival time, t, at the location z. We here define the core of the beam using the query $\bar{t} - 200\delta \leq t \leq \bar{t} + 200\delta$, with $\bar{t}$ being the longitudinal bunch centroid and $\delta = \frac{10^{-9}}{0.036728}$. The parameter δ has been provided by domain scientists and corresponds to $\approx 1nm$ (i.e. the time it takes an electron to travel $1nm$). For analysis purposes, physicists typically further subdivide the core into slices to identify within-core differences in particle density, uncorrelated energy spread and transverse emittance. In order to divide the core into, e.g., 400 $1nm$ wide slices, we could evaluate at each timestep multiple queries of the form $t_{min}(i) \leq t \leq t_{max}(i)$ with $t_{min} = \bar{t} - (i - 201)\delta$, $t_{max}(i) = \bar{t} - (i - 200)\delta$ and $i \in [1, 400]$. Such an analysis of all 720 timesteps of our example dataset, hence, would require $288,000$ queries, further highlighting the need for efficient query methods.

7.2 Results

Figure 14 shows for each timestep the number of particles selected by the halo query. For the halo query we observe large variations in the number of particles that satisfy the query. The larger numbers of halo particles at late timesteps provide impetus to explore the source of the halo. In this case the halo particles and observation of an increase in the maximum particle amplitude were found to be due to a mismatch in the beam as it travels from one section of the accelerator to the next. These diagnostics provide accelerator designers with evidence that further improvement of the design may be possible, and they provide quantitative information useful for optimizing the design to reduce halo formation and beam interception with the beam pipe, and ultimately to improve accelerator performance.

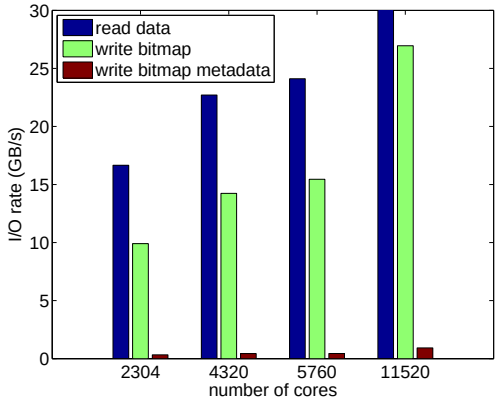

Figure 16: I/O rate for building indexes for 720 time steps (50TB) as the number of cores increases from 2304 to 11,520 (also see Figure 15).

As shown in Figure 15, FastQuery requires approximately 2 hours of preprocessing time to build all indexes for all timesteps of the 50TB datasets. Based on the bitmap indexes, FastQuery is then able to evaluate data queries very efficiently. FastQuery was able to process both the halo and core query in 12 seconds. For the halo query we observe speed-ups of one order of magnitude compared to the sequential scan (see Figure 10). Notably, FastQuery requires only $\approx$ 500 compute cores to process all queries in times that would be reasonable in practice. FastQuery enables in this way repeated, complex, large-scale query-based analysis of massive datasets, which would otherwise not be practical with respect to both time as well as computational cost.

8. CONCLUSIONS

We have presented FastQuery, a system for applying state-of-the-art index and query technology to massive datasets. The FastQuery framework allows us to utilize parallel I/O resources and distributed multi-core resources available on modern supercomputing platforms. We presented in-depth study of various design decisions and parameters choices that went into implementing the system. We have demonstrated the strong scalability of FastQuery upto 11,520 cores. We have successfully applied FastQuery to a massive 50TB dataset from a particle physics simulation, providing new insights and results for our science collaborators.

9. ACKNOWLEDGMENTS

This work is supported by the Director, Office of Laboratory Policy and Infrastructure Management of the U.S. Department of Energy under Contract No. AC02-05CH11231.

10. REFERENCES

[1] M. K. Aguilera, W. Golab, and M. A. Shah. A practical scalable distributed b-tree. *Proc. VLDB Endow.*, 1:598–609, August 2008. http://dx.doi.org/10.1145/1453856.1453922.

[2] C. Ballinger and R. Fryer. Born to be parallel: Why parallel origins give teradata an enduring performance edge. *IEEE Data Eng. Bull.*, 20(2):3–12, 1997.

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

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

[5] J. Corlett, K. Baptiste, J. Byrd, P. Denes, R. Donahue, L. Doolittle, R. Falcone, D. Filippetto, J. Kirz, D. Li, H. Padmore, C. F. Papadopoulos, G. Pappas, G. Penn, M. Placidi, S. Prestemon, J. Qiang, A. Ratti, M. Reinsch, F. Sannibale, D. Schlueter, R. Schoenlein, J. Staples, T. Vecchione, M. Venturini, R. Wells, R. Wilcox, J. Wurtele, A. Charman, E. Kur, and A. Zholents. A next generation light source facility at lbnl. In *Proceedings of PAC 2011*, New York, USA, April 2011.

[6] 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.

[7] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *SIGMOD Conference*, pages 1–8, 1984. http://doi.acm.org/10.1145/602259.602261.

[8] D. J. DeWitt, M. Smith, and H. 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.

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

[10] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25:73–169, June 1993.

[11] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI: The Complete Reference. Volume 2, The MPI-2 Extensions*. Scientific and Engineering Computation. MIT Press, Cambridge, MA, USA, second edition, 1998.

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

[13] M. Howison, Q. Koziol, D. Knaak, J. Mainzer, and J. Shalf. Tuning hdf5 for lustre file systems. In *Workshop on Interfaces and Abstractions for Scientific Data Storage*, 2010.

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

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

[18] M. T. Ozsu. *Principles of Distributed Database Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[19] 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.

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

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

[22] K. Stockinger, E. W. Bethel, S. Campbell, E. Dart, , and K. Wu. Detecting Distributed Scans Using High-Performance Query-Driven Visualization. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, November 2006.

[23] M. Stonebraker. The case for shared nothing. *IEEE Database Engineering Bulletin*, 9(1):4–9, 1986. http://sites.computer.org/debull/86MAR-CD.pdf.

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

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

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

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