In-memory Query System for Scientific Datasets

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

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

***Abstract*—The growing gap between compute performance and I/O bandwidth coupled with the increasing data volumes has resulted in a bottleneck to the traditional post-simulation data processing method. Hence in-situ computing and query-driven data analysis are important techniques to minimize data movement. By taking advantage of the growing memory capacity on supercomputers, we developed an in-memory query system for scientific data analysis. Our approach is a combination of bitmap indexing, spatial data layout re-organization, distributed shared memory, and location-aware parallel execution. Our evaluations using real scientific datasets showed that we can aggregate the memory capacity from thousands of computes nodes to analyze a 750GB simulation dataset without transferring data to remote nodes or storage systems. Comparing to traditional solutions based on out-of-core parallel file systems, we achieve significant higher query performance.**

Keywords*—In-situ computing, query-driven analysis, indexing, scientific data, distributed shared memory

I. INTRODUCTION

Massive amount of scientific data is generated from scientific experiments, and large-scale simulations in many domains, such as astronomy, environment, and physics. The size of these datasets typically ranges from hundreds of gigabytes to tens of petabytes [2]. With the advancement of computing capacity, and the demand for higher resolution of scientific data, the data volume is expected to grow even further in the near future. On the other hand, I/O performance still relies on the convention methods of adding more faster disks. The inefficiency of I/O performance further exacerbated problem due to the complexity of the existing I/O methods and application I/O behaviors. The growing gap between compute performance and I/O bandwidth coupled with the increase in data volumes has resulted in a new bottleneck and obstacle to this traditional method of post-simulation data processing and analysis.

To bridge the gap, many new data analytic and visualization frameworks [25], [30] are based on the concepts of in-situ processing and query-driven analysis. The idea of in-situ processing [16] is to perform data processing and analysis at simulation time and in application memory. This approach not only can minimize data movement, but it also provides an opportunity for scientists to exploit all relevant data from the simulations that would be prohibitively expensive to collect again and compute during postprocessing. To analyze massive datasets more efficiently at simulation time, many in-situ processing solutions rely on query-driven analysis to extract valuable scientific information based on the priori knowledge from scientists about the region of interests to look at. Unfortunately, traditional databases suffer from a significant data-to-query delay due to the requirement to load data into the system before querying. Hence, several in-situ query solutions [15], [18], [19], [3] have been developed to provide SQL-like query capability directly on scientific data files based on various indexing and data compression techniques [20], [27], [14], and they have all shown significant performance improvement over the database solutions, such as SciDB [6] and MonetDB [4]. However, all these in-situ query systems [15], [18], [19], [3] were designed to interact with files, and depending on an underline parallel file system to store and load both data and indexes. Thus one of their main objectives is to achieve faster indexing and querying time by minimizing the size of index files, and maximizing the I/O efficiency between the query systems and file systems. However, as scientists demand for more sophisticated and interactive data analysis, any disk I/O could become a rate limiting factor as well.

In this paper, we propose an in-memory query system for interactive spatial data analytic directly on the compute nodes without writing data to a shared storage location. Our approach is to extend our previously work of FastQuery [9], [10] by adding the features of memory caching, spatial indexing and UDF data processing. Our primary goal is to prevent data movement between compute nodes and storage nodes. Through this tightly integration between data processing and storage management, we aim to support the following three tasks more efficiently.

Interactive query: User interaction is important part of the query driven analysis because queries are generated based on the knowledge and understanding from the users. Without sufficient prior knowledge about the data, and feedback from users at simulation time, it becomes a static data analysis process which is difficult to fully explore the dynamic results and information generated simulations. Hence, having a tight feedback loop of computation, query-driven analysis and user interaction can provide much more insight into the data and serve as a vehicle for steering the simulations. However, response time is critical to the usability and success of interactive data analysis. The current approaches that rely on parallel file systems to store and manage data and indexes will certainly not deliver sufficient I/O performance due to limited disk speed. Even with the help of asynchronous staging systems [26], it could only overlap the I/O time and computation time, but cannot reduce the data movement required between compute nodes and storage servers. Therefore, our system builds a distributed shared memory storage to achieve in-memory computing for interactive query analysis.

Data processing: Data refinery is a commonly required to transform raw data from the simulations into relevant and actionable information before being processed by the query

system. For example, in the VPIC plasma physics simulation dataset [5], the coordinates of particles generated from the simulator are local coordinates within each partition region in the simulated space. Thus the coordinates must be transformed into the global coordinates before querying. Embedded these transformation functions into a query string, such as "$sqrt(x^2 + y^2 + z^2) > 3$" can result in tedious and messy query syntax that is difficulty to be written by users and parsed by query systems if more sophisticated refinery process is required. More importantly, without the support of basic data manipulation, users have to write their own ad-hoc code for processing data then loading the results into a query system for analysis. The data movement between user program and query system could also cause unnecessary data copy and I/O overhead. Therefore, our query system includes a set of user defined function (UDF) to achieve more efficient data transformations with minimum data movement.

Spatial query: Finally, most scientific data has spatio-temporal characteristics. Not only the data is often generated and organized according to these characteristics, it is also queried upon them. For instance, the climate data is generated in regular grid, and in the study of atmospheric river, scientists only interest in the data features in the mid-latitude region of the globe that can cause disastrous floods. Flash simulation is another example that uses a block-structured adaptive grid to deliver sufficient resolution on each region where the astrophysical scientists are interested in features and observations with regarding to a particular spatial region block. However, all previous proposed in-situ query systems [15], [18], [19], [3] do not exploit spatial data property, and focus only on the query optimization of a linearized dataset. Therefore, in this work, we also aim to accelerate spatial query by exploiting the data structure and spatial information in our in-memory data management layer.

We evaluated our system on a NERSC supercomputer using a real scientific dataset. By aggregating the memory capacity from thousands of computes nodes, we demonstrate our system can analyze a 750GB dataset without introducing data transfer overhead to remote nodes or storage systems. Comparing to the traditional solutions based on out-of-core parallel file systems, we achieved over 10x speedup on the response time of query evaluation, indexing building and data processing. Therefore, our system is a promising approach to support interactive query and serve as a vehicle for steering simulations.

The rest of paper is structured as follows. Section II summarizes the related work. Section III gives a overview of the system architecture and API of our query system. Section IV describes our distributed shared memory storage layer. Then Section V $\sim$ Section VII details the operations for managing and query the datasets in our system. Finally, Section VIII shows our experimental results, and Section IX concludes the paper.

II. Related Work

A. Array-based database systems

To mitigate the mismatching data model between the traditional row-major databases and array-based scientific data, several efforts have been made in two directions. One is to provide array-based SQL-like query language, so the data can be viewed and manipulated directly from the array data prospect [29]. The other one is to develop new scientific data management systems for more efficient data access and storage [4], [22]. Take one of the most well-known array-based databases SciDB [6] as an example. It is a shared-nothing parallel database system built on top of ArrayStore [24], which is a storage manager for storing array data using regular and arbitrary chunking strategies. SciDB supports both the Array Functional Language (AFL) and the Array Query Language (AQL) for analyzing array data. Although these systems provide convenient programming interface and efficient query performance, they suffer from the data-to-query delay for preparing and loading data into the databases. In contrast, our work aims to encapsulate the query logic on top of generic data access libraries, and provide instant access to data through a well-defined API. However, in the future, we also plan to develop a SQL-like interface on top of our query system, and to support more SQL-comparable functions, such as JOIN operation.

B. Query and indexing techniques

To prevent data-to-query delay, there are several attempts to provide query capability directly on scientific data format files, such as HDF5 and NetCDF, etc. For instance, [8] uses external tables from database to allow data to be queried in the original format using SQL. [1] enables PostgreSQL to run queries directly over comma-separated value files. Various indexing techniques have also been developed to accelerate the query process. FastBit [27] is a compressed bitmap indexing technique that has been successfully applied to data analysis in many science domains [7]. FastQuery [10], implemented on top of FastBit, parallelizes FastBit's index generation and query processing operations, and provides a programming interface for executing simple lookups. [15] integrates FastQuery with ADIOS [17] to achieve parallel in-situ indexing. [3] integrates FastQuery with the scientific data services SDS [12] and provides relational JOIN operation directly on HDF5 file format through the HDF5 Virtual Object Layer (VOL) abstraction. Recently, Sriram Lakshminarasimhan et al. proposed ISABELA-QA [19] and DIRAQ [18] to further eliminate the random data access pattern occurs in the query process through the combination of the indexing technique and data encoding and reorganization techniques. In this work, we choose FastQuery as the query engine to develop our in-memory query system. Different from the previous approaches based on files and focus only on the indexing techniques, we aim to explore the benefit of in-memory computing and in-memory data structure for spatial query in this work by integrating spatial tree structure with bitmaps indexing technique.

C. In-memory & parallel processing

Finally, in-memory computing and massive parallel processing(MPP) on share-nothing system architecture has proven to be a successful approach to exploit data parallelism and support data analysis in scale. MapReduce [11] is a pioneer in this approach, while systems like Dryad [13] generalized the data flows to DAG. However, in order to support interactive and iterative data analysis, in-memory computing technique must be explored, and Spark [28] is one such systems that develop its distributed execution engine on top of a resilient distributed

shared memory layer. Motivated by these approaches, our system adapts several similar design principals. For instance, it is also a share-nothing system architecture that exploit data parallelism. A self-managed shared memory layer is implemented for caching data and reducing I/O overhead. Finally, a set of user define functions is provided as a simplified and efficient programming interface for users to process data. But different from these data processing systems for byte stream data, our system can provide indexing and query capability on multi-dimensional scientific dataset with temporal-spatial property.

III. SYSTEM OVERVIEW

This paper describes our design and implementation of a in-memory query system for scientific data analysis. Our system consists of four layers. On the top is a lightweight, programming API for data query and processing. Then a data abstraction layer is defined to manage the data representation in our system, and store the metadata of user datasets for parallel execution and query. The third layer is a parallel execution engine that provides query, indexing and processing capability. Finally, a distributed shared memory(DSM) layer is at the bottom to store all the data managed by our system. In this section, we first state our system design principal. Then we describe the data abstraction layer and variable creation API. The parallel execution engine and distributed shared memory are introduced later in Section IV$\sim$Section VI.

A. Design Principal

Our goal is to develop a query system that allows scientists to perform efficient query and processing actions directly on the scientific datasets within their simulation program. To ensure the performance, scalability, and usability of our solution, we aim to achieve the following four design principals.

Lightweight: It means our system is easy to deploy, easy to integrate with user code, and easy to control resource usage. This is because our system is implemented as a MPI library that provides a set of API for programmers to load, query and process data. So no software installation is required, and it can be easily compiled with applications or simulation codes as a single MPI program, and then submitted to supercomputers for execution. The amount of resource usage can be easily controlled by the number of MPI tasks launched from the program, and all the allocated resources are freed after program terminates. Hence, our system is suitable for in-situ processing without requiring much changes to the existing user codes or computing environment.

Direct access to user data: As the data volume generated from simulators continues to grow, our system is designed to analyze these data directly on its original storage format without loading or converting the datasets before processing. In particularly, in this work, we not only consider the data that is stored in a scientific data format, such as HDF5, but also the one that still resides in the simulator memory not yet writing to files. To achieve this goal, our system implemented a unified data access interface to most of the scientific data formats, and defined a set of functions to load data directly from either files or memory buffer.

Scalable and efficient data processing capability: Our system achieves this requirement by taking three approaches. First is to exploit data parallelism, so that all the API provided from our system can be implemented in parallel and executed by individual MPI tasks. Second is to utilize indexing technique to accelerate query processing. As described in Section VII-B, our system combines a state-of-art bitmap indexing technique with spatial data structure information to reduce query response time. Finally, our system implemented its own memory storage layer to cache data or indexes. Hence our system can be fully operating in memory space without introducing any disk access to disks or parallel file systems.

Shared nothing architecture: To ensure the scalability of our system, and minimize the data movement and communication overhead between compute nodes, our system follows a shared nothing architecture. We achieve this goal by partitioning each dataset across all the MPI tasks, so that the expensive data processing operations can be performed independently on each MPI task. On the other hand, the metadata and aggregated statistics of datasets are simply duplicated on every MPI task, so that each MPI task has sufficient information to process its local data without communicating with other MPI tasks.

B. Data Abstraction

A data abstraction layer is defined to separate how the data is viewed by users, and how it is managed by our system internally. This abstraction allows us to define more convenient API for users, and implement more efficient data management techniques for system performance and scalability. In our system, data is represented as variables, and API is defined to perform operations on these variables. The variable creation API is in Section V, the query and indexing API is in Section VII-B, and the data processing API is in Section VI. Finally, these variables can be cached in our system to accelerate data analysis as described in Section IV. Here, we first briefly introduce what is a variable.

A variable is a multi-dimensional dataset with spatial locality. In other words, each data value not only has a corresponding coordinate in its storage dataset, but also has a corresponding spatial location in its space domain. By default, we assume variables have dense spatial locality, so that their spatial locations are the same as their dataset coordinates. For instance the climate simulation dataset is stored in a two-dimensional array according to the latitude and longitude of each measurement points. So the dataset coordinates can directly be used as the geographic location of these data elements. But we also consider datasets with sparse locality by allowing users to describe the spatial location by a separate set of variables with the same data dimensions. For instance, in the VPIC plasma physics simulation dataset, particles have a sparse spatial locality spreading over a 3 dimensional space domain. Hence, besides having a variable to store the energy measurements of particles, we also need to use other three variables to describe the (x, y, z) location of each particle. In this case, the dataset coordinates are used to represent the identifiers of data elements instead of the spatial location, and we said the variables x, y, z are the spatial variables of a "energy" data variable.

But notice that our system does not distinguish between spatial variables and data variables. The spatial relationship

between variables only exists when the spatial query and indexing API is called as described in Section VII-B and Section VII-A. In these function calls, users is allowed to specify a data variable along with a set of spatial variables. Then our system will process the request involves all these variables together. Take the VPIC dataset as an example, we expect the scientists to first create four variables of equal size one-dimensional arrays to store the energy and (x, y, z) location of each particles. Then they can submit a nearest neighbor query by giving the input arguments of a range constraint "$energy > 1.2$", a list of spatial variable names "x, y, z", and a vector of origin location "(1,1,1)". As described in Section VII-A, a spatial data layout index tree according to the values in the spatial variables can be built to reduce the query time.

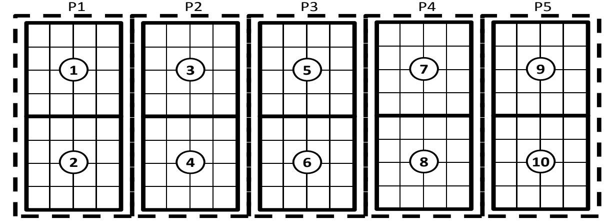
Variables are identified by a unique identifier in our system, and they are $immutable$ and $ephemeral$. Immutable means variables cannot be modified after creation. It allows us to simplify the management efforts on variables, and prevent synchronization and data consistency problem. But as described in Section VI, a set of data transformation API is provided to generate new variables by applying a user defined function on the values a set input variables. Ephemeral means the data values of a variable are not computed and stored inside the DSM storage layer at creation time by default. Instead, our system records the arguments and data flow for generating the variables. Only when a "cached" or "query" action API is called on a variable, then we materialize the variable by re-computing its values and caching the results in our DSM storage layer. As discussed in Section IV, caching a variable can eliminate disk access, exploit data locality, and prevent recomputing values. Due to the limited cache space, it is critical to ensure the cache efficiency and utilization. But in this work, we don't discuss when to cache a variable, and leave it for users to decide based on our data analysis logics and behaviors. But it is a critical and interesting question that we would like to explore further in the future.

C. API & Use Case Example

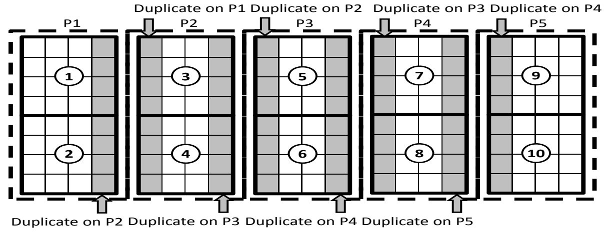
We briefly introduce the four types of function calls in our system API in below. All these functions are implemented as collective and synchronous MPI calls.

Create: There are four ways to create a new variable in our system. (a) `LoadFile()`: Load a dataset from a file. (b) `LoadMem()`: Loading an array buffer from user program memory space. (c) `Select()`: Selecting a subarray region from a existing variable. (d) `UDF()`: Use a user defined function to generate data from other existing variables. As mentioned, all the variables created by these API are not cached or materialize. But their values can be retrieved by re-computing at runtime. These functions are detailed in Section V and Section VI.

Index: There are two indexing API calls supported in our system. (a) `BuildBitmapIndex()`: Build bitmap indexes on a single variable individually for answering range value queries. (b) `BuildSpatialIndex()`: Build a spatial index tree structure on a set of variables for answering spatial queries. The index process is detailed in Section VII-A.



(a) Partitioned by 4x4 data blocks, and distributed to 4 processes.



(b) Data partition with boundary width=1.

Fig. 1. Data partition in DSM storage layer.

Query: There are a set of query API provided by our system. We do not list each of them here. But they are divided into two types. One type is for range value query, such as "$energy > 1.2$". The other type is for spatial query, such as a nearest neighbor query with a origin location and a value constraint. Both types of queries have corresponding API for retrieving the hit count and the coordinates or values of the selected data. The query process is detailed in Section VII-B.

Checkpoint and cache: Finally, we have a checkpoint API to write variables into files for fault tolerance and data output. More importantly, we have a cache API to explicitly force a variable to be materialized and cached in a DSM. The details of our DSM storage layer is discussed in Section IV.

IV. DSM Storage Layer

A distributed shared memory(DSM) storage layer is implemented in our query system to cache data and support in-memory computing. The cached data is partitioned and stored across the memory space of all the MPI tasks participated in the program. When a MPI task needs to read a given part of the dataset, it will base on the partition information to determine the data location, and retrieve the data either locally or remotely. There are three key challenges for designing this DSM storage layer: (1) How to partition the data. (2) How to minimize the remote data access. (3) How to overcome limited memory space. This section discusses each of these issues as follows.

A. Data Partition

To utilize the memory space located at MPI tasks, the data of a cached variable is partitioned and distributed across all the processes. Since each variable is a multi-dimensional dataset, our system let user to specify a fixed block size for partitioning. For example, given a 8x20 two-dimensional variable, if the block size is 4x4, the dataset will be partitioned into 10 data blocks. After data is partitioned, our system first linearizes these data block along a given dimension. Then evenly place continuous data blocks at the same storage location. Therefore, as shown by the example in Figure 1 (a), if 5 MPI tasks are given, the first two blocks will be placed on the first MPI

task, then the next two blocks will be placed on the second MPI task, and so on so forth. By doing so, we can reduce data redundancy as explained in Section IV-B, and we can aggregate the data process or query results by a single MPI_Allgather call to append the returned data.

B. Data Replication

We do not have data replication because we do not consider fault recovery at individual process level in this work. For most MPI-based scientific applications, fault tolerance is handled by writing checkpoint, and fault recovery is performed at global-level by re-running the whole program from previous check-point. Thus, data replication doesn't provide any benefit. In addition, if only the memory content is lost due to unexpected reason, the cached variable in our system can always be re-generated from the origin data source of files or user memory buffers.

However, we do allow redundant data to be copied at the boundary between data blocks like the technique used in SciDB. The data redundancy can reduce the amount of remote data access when processing a user defined function that needs to read the neighbors of each data point, such as a smooth function. In our system, user can use a configuration parameter to control the width of the boundary. Furthermore, because continues data blocks are placed at the same location in our DSM, duplicated data only occurs at the boundary between processes instead of individual data blocks. Therefore, if the boundary width is 1, the size of each data block is expanded by 1 along each dimension as shown in Figure 1(b), and the duplicated data is indicated in gray color.

V. Variable Creation

There are four ways to create a new variable in our system, we details them individually as follows.

From file: It creates a variable by loading a multi-dimensional dataset from a file that is stored on a shared file system. An unified file interface is implemented in our system to support various file formats for user to choose from, including HDF5 and NetCDF, etc. But this API call only loads the necessary information to retrieve data from files, such as the filename and the dataset metadata. Therefore, the file must remain opened, and the dataset must remain unchanged until the variable is deleted or cached.

From memory: It creates a variable by loading a linearized multi-dimensional dataset from a memory buffer directly out-put from simulators or applications. The input argument from users simply contains a pointer to the memory buffer, a vector of dimension lengths and the data type. By storing these input arguments, the system can retrieve data values directly from the user memory buffer without making another duplicate copy in the DSM storage layer. But the memory content must not be freed or modified until the variable is deleted or cached. To detect unexpected changes, a checksum mechanism is implemented inside our system to verify the integrity of data content.

From subarray: It creates a variable by loading a subset of data from another existing variable. The subset region is a subarray specified by three arguments including the starting position of the selected region of each data dimension, the stripe distance of each data dimension, and the number of selected elements of each data dimension. Again, by storing these input arguments, our system can retrieve the correct data values by re-computing the data coordinates in its origin variable at runtime.

From transform: It generates a new variable by using a user defined function(UDF) to transform and aggregate data values from existing variables. A new variable will have the same dimensions as its input variables, but only the values are re-computed by a UDF. Hence the data values can be retrieved at runtime by calling the UDF function on the data from its origin variables. Three types of UDFs are defined by our system, and they are detailed in Section VI.

VI. Variable Transformation

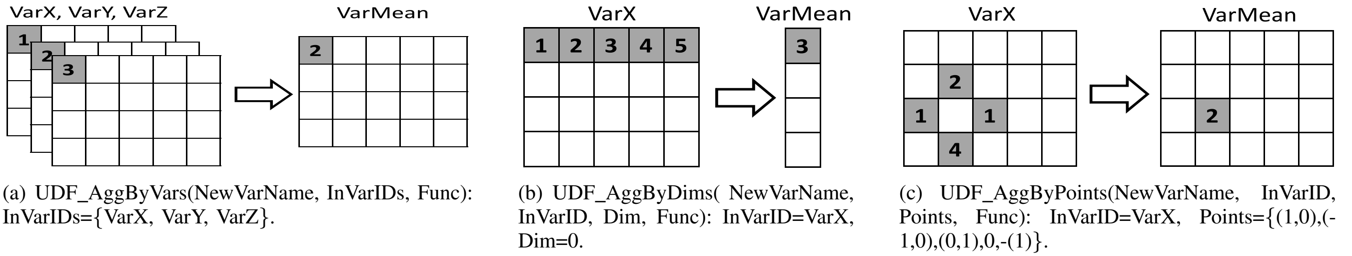
During data analysis, scientists often need to refine data into values that can be understood and meaningful before the interesting data points can be queries and retrieved. However, writing parallel code to process dataset on large scale com-puting cluster are proven to be complicated. Especially for interactive or iterative data analysis, it is often driven by a feedback loop between data refinery and query exploration. Therefore, we add analytical capability to our query system by supporting a set of user defined functions (UDF) API for users to transform their variables by aggregating and mapping the data values.

There have been several similar attempts on database systems by using the extensibility features of SQL such as User-Defined Functions (UDF) and User-Defined Aggregates (UDA). A generate distributed framework was also developed by GLADE [23] to compute user defined aggregates in a paradigm and runtime environment similar to MapReduce. But different from those approaches, our UDF is designed for multi-dimensional datasets instead of key-value pair tuples or table columns. Therefore, depending on the mapping pattern, there are three types of UDF defined in our system. We describe each of them as follows.

Aggregate by variables(AggByVars): It applies a user define function to aggregate the data values across multiple datasets at the same coordinates. It is commonly used to compute values that are derived from other variables. For instance in Figure 2 (a), it is used to compute the Euclidean distance of a given (x, y, z) location stored in three separate datasets. If only one variable is specified, it becomes a one-to-one mapping data transformation.

Aggregate by dimension(AggByDims): It applies a user define function to aggregate the data values across a given dimension of a dataset. It is commonly used to compute the aggregates of a dataset. For instance in Figure 2 (b), it is used to compute the sum of each column in a two-dimensional dataset. The sum of all elements can be computed by applying the UDF again on the new resulting dataset along the row.

Aggregate by coordinates(AggByPoints): It applies a user defined function to aggregate the data values across a given set of coordinates in relative location. It is commonly used to compute values that are derived from the neighbors in a dataset. For instance, Figure 2 (c) shows that a smooth function



(a) UDF_AggByVars(NewVarName, InVarIDs, Func): InVarIDs={VarX, VarY, VarZ}.

(b) UDF_AggByDims(NewVarName, InVarID, Dim, Func): InVarID=VarX, Dim=0.

(c) UDF_AggByPoints(NewVarName, InVarID, Points, Func): InVarID=VarX, Points={(1,0),(-1,0),(0,1),0,-(1)}.

Fig. 2. Examples of using the three types of UDF to compute the average values. NewVarName="VarMean" in all cases. Func is a function pointer to the UDF implemented by users.

can be implemented by this type of UDF to re-generate the values of a dataset. A set of relative coordinates "{(-1,0), (1,0), (0,-1), (0,1)}" is given by users to specify the four neighbors locating at the top, bottom, left and right positions when computing the value of every data point.

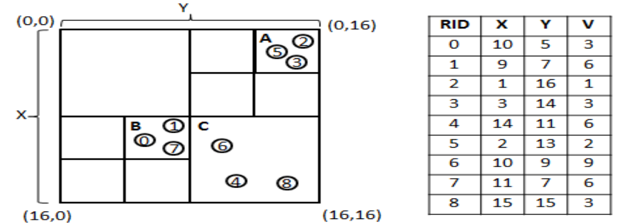
Similar to the programming paradigm of MapReduce, a generalized UDF interface is defined by our system as shown in Figure 3. The function is called to compute the value at each coordinate in the new variable dataset. According to the type of UDF and the input argument specified by the programmer, our system is responsible for collecting the input values that need to be aggregated at each coordinate, and preparing them as an input argument array when executing the UDF. Hence, programmers simply implement the aggregation function with respect to each coordinate in a sequential code manner, and our system will parallelize the computations across all the data points, and store the results dataset distributively in the DSM storage layer. Thus, the UDF feature not only add the analytical capability to our system, but also simplify the programming efforts from users.

```
void eculid::UDF(vector<double> vals, void* ret) {
    double x = vals[0];
    double y = vals[1];
    double z = vals[2];
    *ret =\sqrt{(x^2 + y^2 + z^2)};
}
```

Fig. 3. An implementation of the UDF interface in our system. Input is a list of values collected according to the type of UDF. Output is the aggregated result value. This function is called to determine the value of each element for the new variable from transformation.

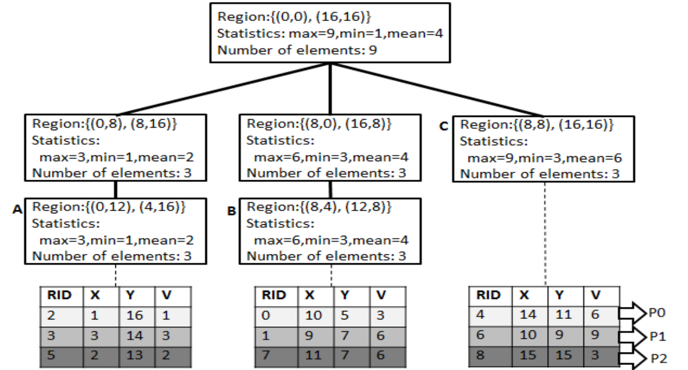
VII. Query & Indexing

We use our previous work, FastQuery, as the query engine in this work. FastQuery provides the API to build and query bitmaps indexes for a multi-dimensional dataset in parallel by partitioning data into chunks (i.e. subarray), and then build or query those chunks independently across MPI tasks. Details can be referred to our FastQuery paper [10]. In this work, we extend our query capability to spatial data or multi-datasets. In the current implementation, indexes are only built upon users' requests. But in the future, we plan to record the query history and build indexes automatically based on the query pattern observed from the history. Indexes are annotated metadata, and they can be stored separately from the input data with their own data structure and storage organization. A variable must be materialized in memory or loaded from file before its indexes are built. The built indexes can be either cached in memory or written to files for repeatedly query processing.



RID	X	Y	V
0	10	5	3
1	9	7	6
2	1	16	1
3	3	14	3
4	14	11	6
5	2	13	2
6	10	9	9
7	11	7	6
8	15	15	3

(a) A spatial data using 3 variables to record the value and (X, Y) location of 9 measurements identified by their RID(record ID).



RID	X	Y	V
2	1	16	1
3	3	14	3
5	2	13	2

RID	X	Y	V
0	10	5	3
1	9	7	6
7	11	7	6

RID	X	Y	V
4	14	11	6
6	10	9	9
8	15	15	3

(b) Spatial index tree build from our system.

Fig. 4. An example of a spatial indexing tree. The tree structure and metadata are duplicated on all MPI tasks, but the datasets at leaf nodes are distributed across processes.

A. Spatial Indexing

When the `BuildSpatialIndex()` API is called, we build a spatial index tree as shown in Figure 4. Each intermediate node is a bounded box of a space region represented by its upper left coordinate and lower right coordinate. The space region of a parent node is partitioned by its children into a set of non-overlapping sub-space regions. The space region is iteratively partitioned until reaching a partitioning threshold, such as the minimum number of elements in a box, or the minimum region size of a box. At each leaf node, only the dataset coordinates of all the elements within the box are stored, and the corresponding data values can be retrieved from the variable dataset. As described more detailed in Section VII-B, a query with spatial constraints can be handled more efficiently by traversing through the tree structure to narrow down the search space. Any space partition tree can be considered to be implemented in our system as an index technique. In the current implementation, we build quad tree for two-dimensional data, and octree for three dimensional data. We plan to implement k-d tree for data with even higher dimensions in the future.

To further accelerate query process, we also annotate more indexing information on the tree structure. At the intermediate nodes, we record the aggregated statistic metadata from all its children nodes, such as the max/min values or the total number of elements, etc. These metadata can be used as a branch and bound condition when searching the tree. For instance, let us consider a query for finding the data with value between $7 < V < 9$, and located within the space region between $8 < X < 16$ and $0 < Y < 16$ in Figure 4. Without statistic metadata at intermediate nodes, we have to examine the values in region B and C. But with the annotated metadata, only region C needs to be searched. Therefore, if the selected data only falls in a few number of leaf nodes, the search time can be greatly reduced. In addition, at leaf nodes, we use can build bitmap indexes to accelerate the query within its box region. Notice that bitmap indexes can also be built for the whole dataset without creating the spatial tree by simply calling the `BuildBitmapIndex()` API. It is used when the query only involves one variable without any spatial constraint.

We utilize the FastQuery API to build an index tree in the following steps. (1) The spatial index tree is built iteratively from the root by querying the data falls inside the box of each intermediate nodes. The query can be accelerated if bitmaps indexes are built for individual spatial variables in advanced. (2) When reaching a leaf node, we create a temporary variable to include all the data within it. Then we build bitmaps indexes for the temporary variable. (3) The aggregated statistic metadata is collected iteratively from the leaf nodes to the root, and store the results along the intermediate nodes.

The data size of a tree structure and the statistics metadata at intermediate nodes is insignificant comparing to the size of data or bitmap indexes. Therefore, we duplicate these information on all the MPI tasks to provide a global view in query process. On the other hand, the temporary variable created at leaf nodes are partitioned into data chunks and stored distributed across MPI tasks as described in Section IV. So the data coordinates and bitmaps indexes of leaf nodes are also generated with respect to each data chunk individually, and stored across processes.

B. Spatial Query

As mentioned, FastQuery is designed to provide query API on individual dataset. So if a query involves constraints on multiple variables, the indexes are built and queried for each variable separately. Then the results are filtered by finding the intersection among all variables. In this work, with the spatial index tree structure, we not only can process multi-dataset query more efficiently, but also can support spatial query, such as nearest neighbor search.

Since a spatial index tree is duplicated on all MPI tasks, given a query, each MPI task can independently traverse the tree in the same order to find the leaf nodes that satisfy the given spatial constraints one by one. Then the temporary variable at each leaf node can be queried in parallel through the FastQuery API using bitmap indexes. Finally, the query results, such as the hit counts, or the values and coordinates of selected data, are gathered and returned to users. To support nearest neighbor query, we first locate the leaf node that covers the origin location given from users. Then backtrack the tree structure to locate the next nearest leaf nodes until the requested hit counts or maximum distance constraint given from users is reached.

Utilizing a spatial tree structure to handle multi-datasets query is more efficient then only using bitmaps indexing for two main reasons. First, an index tree is built based on the values of all spatial variables together. So the query can be resolved in a single search, and the search results don't need to filtered again by a post-processing step. Secondly, as the number of spatial variables involved in a query is increased, it becomes more expensive to build bitmaps indexes for each of the variables. In general, the total size of bitmaps indexes will be proportional to the number of variables. In contrast, with spatial index tree structure, we don't build bitmaps indexes for each of the spatial variables, and the size of index tree can always be controlled by the minimum number of elements in a box, or the minimum region size of a box. Especially, if we implement our index tree using k-d tree, the combinatorial explosion problem can be prevented, and the tree size is only related to the number of elements instead of the number of variables.

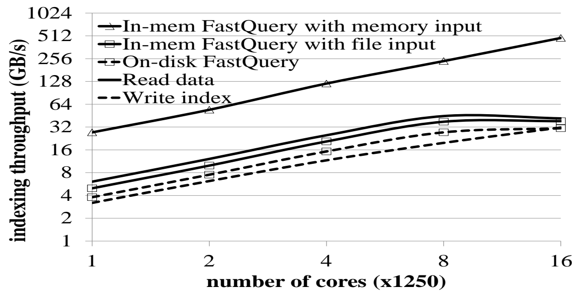
VIII. Experimental Evaluation

A. Experimental Setup

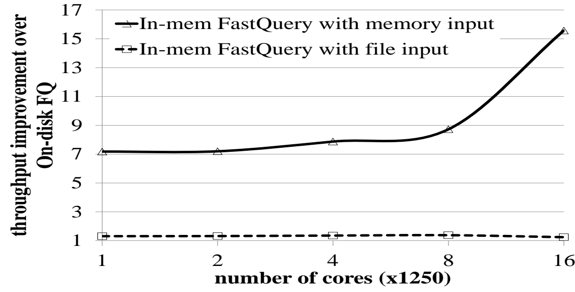
We have conducted all our experiments on the Cray XC30 system, called Edison, at the National Energy Research Scientific Computing Center (NERSC). Edison has 5,576 compute nodes, where each node has two 12-core Intel Ivy Bridge 2.4GHz CPU and 64GB of memory. To prevent memory contention between the MPI tasks on the same node, we have limited the memory usage of each FastQuery process to 2 GB, using 24 cores per node; the rest of memory space is reserved for system usage, such as initiating MPI tasks, etc. Edison has a Lustre parallel file system to provide storage spaces. Its peak performance at 72 GB/s with 144 OSTs.

The test datasets for our experiments are produced by a plasma physics simulation called VPIC. VPIC writes a significant amount of data at a user-prescribed interval. In this study, we use the data files from a simulation of 186 billion electrons. Each particle is associated with seven one-dimensional datasets. For evaluating spatial query, our experiments use the four datasets that record the energy and (x, y, z) locations, and these dataasets are sorted according to the energy values. For each dataset, its data size is around 750GB, and its index size is around 150GB. Therefore, even with the smallest scale in our experiments(1296 processes), both data and index can be cached in memory. But our experiments consider the data is loaded from a file initially unless we state otherwise.

Because the file system used in our testbed is a shared resource among hundreds of users, the file system experiences contention from other jobs running on the system. In addition, as observed in a previous study [21], many other issues, such as the complexity of the I/O software stack and the contention of non-IO MPI communication, could also contribute to the variability of performance results. Therefore, we repeat each experiment at least 5 times over the course of a week, and report the medium value over the experiment runs.

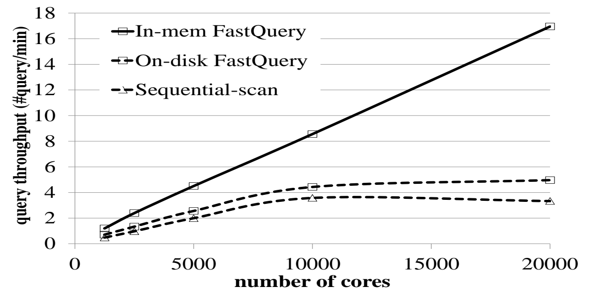


(a) Throughput comparison for indexing and file read/write performance.

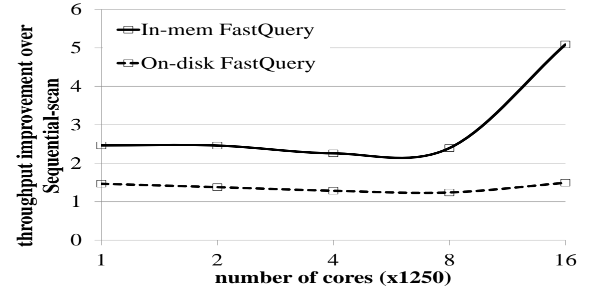


(b) Indexing throughput improvement over On-disk FastQuery.

Fig. 5. Scalability and performance comparison of bitmaps indexing.



(a) Throughput comparison for query.



(b) Query throughput improvement over Sequential-scan.

Fig. 6. Scalability and performance comparison of bitmaps query processing.

Our experiments compare the query performance of following three query methods:

Sequential-scan: It always reads out all the data values sequentially, and filter the data according to query constraints.

On-disk FastQuery: It is our previous FastQuery implementation, which doesn't have the capability to cache index or data in memory. For spatial query, programmer must implement their ad-hoc code to query each spatial variable individually, and then use a posting-processing to filter the results. However, it does use bitmaps indexing to accelerate range query.

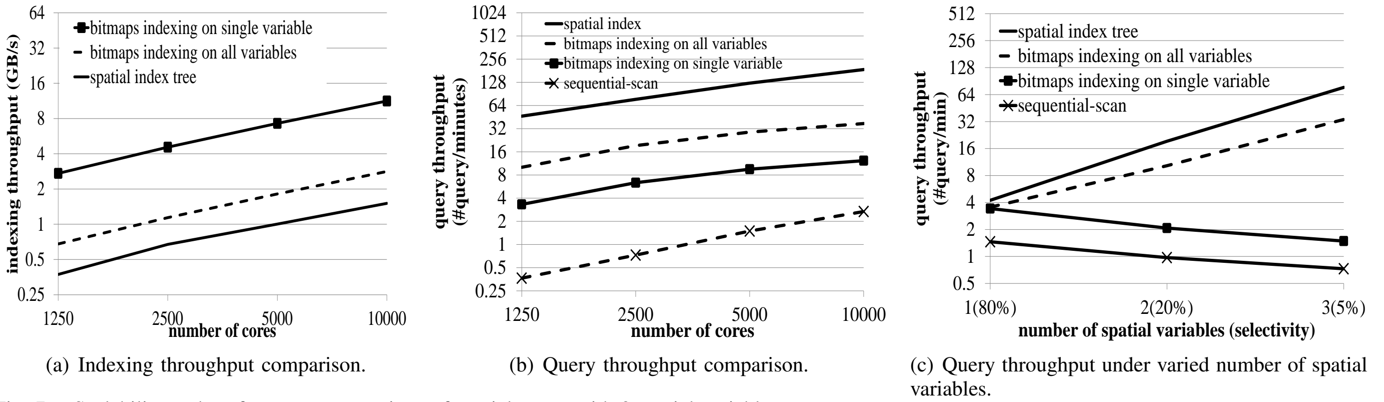
In-memory FastQuery: It is the implementation from this work, with the three additional features: caching, spatial indexing and UDF processing.

B. Range Query Indexing & Query

In this set of experiments we compare the overall performance and scalability of bitmaps indexing and query. The binning indexing option is percision=2, and the query is "$energy > 1.2$". For indexing performance, we only compare between On-disk FastQuery and In-memory FastQuery. Both systems use the same way to build indexes. However, On-disk FastQuery can only read data and write index to files, while In-memory FastQuery can load data and cache index directly in memory without going through a out-of-core storage systems. Therefore, we also show the results of In-memory FastQuery with input from file and from memory, respectively. But for both cases, the indexes are cached in memory without writing to file.

Figure 5 (a) shows the overall throughput of indexing as the number cores is exponentially increased from 1,250 to 20,000. The In-memory FastQuery with input from memory always achieves the highest throughput because it doesn't read data from or write indexes to file. On the other hand, On-disk FastQuery always has the lowest throughput because both of its input and output involve disk I/O. To investigate the I/O issue, we added two lines ("read data" and "write index") in the plot to highlight the I/O throughput. As can be observed, the peak I/O throughput can be achieved from our experiments is only around $40 \sim 45$ GB/s. As a result, the throughput speedup factors starts to gradually decreases after 10,000 cores for both On-disk FastQuery and In-memory FastQuery with file input. In contrast, In-memory FastQuery with memory input can always achieves close to linear speedup because it only involves memory operations, and the memory bandwidth always increases proportionally to the number of computing cores or nodes. Therefore, the throughput improvement of our In-memory FastQuery over On-disk FastQuery could grow as the scale increases. As shown in Figure 5 (c), initially at the smallest scale with 1250 cores, the throughput of In-memory FastQuery is 1.3 times faster than the On-disk FastQuery when the input is from file, and it is 7.2 times faster when the input is from memory. But when the scale increases to 20,000 cores, In-memory FastQuery with memory input is 15.5 times faster than On-disk FastQuery, and we can expect the gap will continue to grow in larger scales. Therefore, In-memory FastQuery not only significantly reduces indexing time, but it also achieves better scalability and stability.

Next, we show the query performance comparison in Figure 6. Here we compare In-memory FastQuery, On-disk FastQuery and Sequential-scan. For In-memory FastQuery, the indexes are assumed to cached in memory. So bitmaps can be loaded by copying a memory pointer only. In contrast, On-disk FastQuery needs to load bitmaps from file for query evaluation. Finally, Sequential-scan always reads all the data from disk. As shown in Figure 6 (a), the throughput of On-disk FastQuery is faster than Sequential-scan because of the indexing can reduce the amount of data reading from file. On the other hand, the throughput of In-memory FastQuery is faster than On-disk FastQuery because the indexes can be reading from



(a) Indexing throughput comparison.

(b) Query throughput comparison.

(c) Query throughput under varied number of spatial variables.

Fig. 7. Scalability and performance comparison of spatial query with 3 spatial variables.

memory instead of file. Similar to indexing performance, the query performance of On-disk FastQuery and Sequential-scan can also suffer from the I/O bandwidth limitation of file system. Especially the query evaluation time is relatively fast, so the I/O overhead has even greater impact to the overall performance. As shown in Figure 6 (b), On-disk FastQuery is always around x1.5 times faster than Sequential-scan. But the improvement from In-memory FastQuery over Sequential-scan significantly increases from x2.5 to x5. Therefore, eliminating disk access is critical to query performance, and our In-memory FastQuery significantly improve both indexing and query performance by utilizing memory cache.

C. Spatial Indexing & Query

In this set of experiments, we show the performance improvement of spatial query from having a spatial index tree. All the results were measured at a fixed scale using 2,500 cores. Our query for evaluation is to find the particles whose location is within a space region and energy is greater than 1.2. We compare the following four query strategies: "Spatial index", "bitmaps indexing on single variable", "Bitmap index on all variables", and "Sequential-scan". "Spatial index tree" assumes an index tree has been built based on the "x, y, z" variables as described in Section VII-A. "Bitmaps indexing on single variable" assumes the bitmaps indexes are only built for the "energy". So after finding the particles with energy larger than 1.2, we still have to retrieve their x, y, z values, and filter the data outside the search space region. In contrast, "Bitmap indexing on all variables" assumes bitmaps indexes have been built for all four variables. So we can query these variables individually, and then scan through the search results to find the intersection. Finally, Sequential-scan simply reads all the variable values of each particles one-by-one, and keeps the ones that satisfy all the query constraints.

Figure 7 (a) shows the throughput of indexing as the number cores increases from 1,250 to 10,000. The indexing throughput of "single variable" is four times faster than the throughput of "all variables", because it only builds bitmap indexes on "Energy" instead of on all four variables. The indexing throughput of our spatial index is the slowest, because of the extra cost of building the index tree. However, as shown in Figure 7 (b), the query throughput of our spatial index is significantly faster than bitmaps indexing and Sequential-scan. Furthermore, as shown in Figure 7 (c), using spatial index can achieve faster query time as the query becomes more complex because the query selectivity is reduced from 80% to 5%, and fewer number of leaf nodes needs to be examined. On the other hand, the query time of Sequential-scan and single variable bitmaps indexing increases proportionally to the number of spatial variables, because it needs to scan the data of each spatial variable.

D. Data Caching & Processing

Finally, we demonstrate the scalability and performance improvement of processing data throughout our UDF API and data caching. The use case we consider here is to transforms a dataset using a smooth function that recomputes the value of each data point by taking the mean of its neighbors. The process includes three steps: read input dataset from file, compute smooth function, and write new dataset to file. As shown in Figure 8, all three steps can be performed in parallel, and our UDF operations can achieve close to linear speedup. But we also observed that the computation time is much smaller than the I/O time, and the I/O scalability can suffer after the bandwidth of file system is saturated. As a result, I/O is an expensive operation in data transformation as well.

To mitigate the I/O overhead, in our system, we can take advantage of our DSM storage layer to cache re-used data, or the data that needs to be repeatedly transformed. For example, lets considering a case that users would like to iteratively re-compute a dataset by a smooth function until the maximum value is lower than a threshold. Without UDF and caching from our system, users need to write their ad-hoc code to implement a parallel smooth function. The computing results will be written to file, and then be loaded into a query system for evaluating the termination condition in each iteration. Therefore, the total processing time increases proportionally to the number of iteration. But without In-memory FastQuery, after the data is read from file initially, it can be cached and queried in memory, and only written to file at the end of the iterations. Figure 9 shows the performance speedup factor from our system under varied number of iterations, and scales. The improvement increases over iterations because the data I/O data can be prevented in each iteration. The improvement also increase over the number of cores, because of the increasing I/O contention of file system at larger scale. Overall, the above results show we provide scalable and efficient data processing capability to our query system.

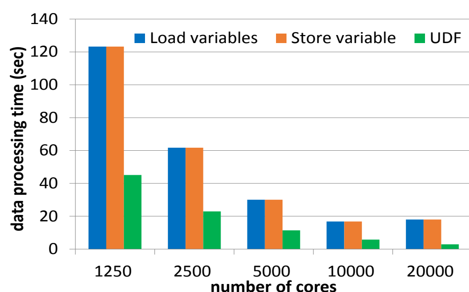


Fig. 8. Scalability of data transformation using user defined function.

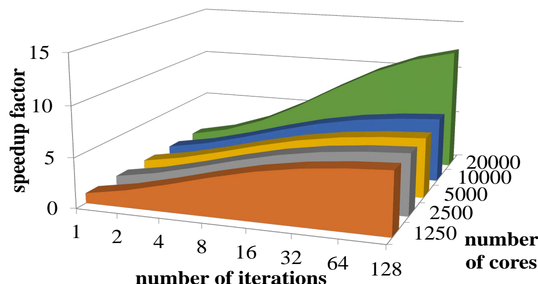


Fig. 9. Speedup over non-cached data processing under varied number of cores and number of processing iterations.

IX. Conclusion

In this paper, we present the design and implementation of an in-memory parallel query system. It utilizes a spatial index tree technique, and bitmaps index technique to accelerate query with range and spatial constraint across multiple datasets. The query system is also integrated with a distributed shared memory caching layer and a user defined function programming interface to provide the capability of caching and transform dataset for data analysis. Our evaluation based on real scientific datasets shows several important contributions and performance improvement from our system. (1) In-memory query system can achieve close to linear speedup at any scale, while the query system relied on file system cause suffer from limited I/O bandwidth when the scale grows over 10,000 cores. (2) Caching data and indexes in memory can significantly improve indexing and query performance by a factor of x10 to x100. (3) Our spatial index technique can achieve the best query performance, and efficient reduce the query time as the number of spatial variable increases. (4) Our UDF programming interface can simplify the programming effort in data analysis, and provide data processing capability to our query system.

References

[1] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. "nodb: Efficient query execution on raw data files". In *ACM SIGMOD*, pages 241–252, 2012.

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

[3] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani. Parallel data analysis directly on scientific file formats. In *ACM SIGMOD*, pages 385–396, 2014.

[4] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the Memory Wall in MonetDB. *Commun. ACM*, 51(12):77–85, Dec. 2008.

[5] K. J. Bowers, B. Albright, L. Yin, B. Bergen, and T. Kwan. Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulationa). *Physics of Plasmas*, 15(5):055703, 2008.

[6] P. G. Brown. Overview of SciDB: Large Scale Array Storage, Processing and Analysis. In *SIGMOD*, pages 963–968, 2010.

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

[8] Y. Cheng and F. Rusu. Parallel In-situ Data Processing with Speculative Loading. In *ACM SIGMOD*, pages 1287–1298, 2014.

[9] J. Chou, K. Wu, and Prabhat. FastQuery: A parallel indexing system for scientific data. In *IASDS*, 2011.

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

[11] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[12] B. Dong, S. Byna, and K. Wu. SDS: A Framework for Scientific Data Services. In *PDSW*, pages 27–32, 2013.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *ACM EuroSys*, EuroSys '07, pages 59–72, 2007.

[14] J. Jenkins, I. Arkatkar, et al. ALACRITY: Analytics-Driven Lossless Data Compression for Rapid In-Situ Indexing, Storing, and Querying. *Transactions on Large-Scale Data- and Knowledge-Centered Systems X*, 8220:95–114, 2013.

[15] J. Kim, H. Abbasi, L. Chacon, C. Docan, S. Klasky, Q. Liu, N. Podhorszki, A. Shoshani, and K. Wu. Parallel in situ indexing for data-intensive computing. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 65–72, Oct 2011.

[16] S. Klasky, H. Abbasi, et al. In Situ Data Processing for Extreme-Scale Computing. In *SciDAC*, July 2011.

[17] ADIOS. http://www.nccs.gov/user-support/center-projects/adios/.

[18] S. Lakshminarasimhan, D. A. Boyuka, S. V. Pendse, X. Zou, J. Jenkins, V. Vishwanath, M. E. Papka, and N. F. Samatova. Scalable in situ scientific data encoding for analytical query processing. In *HPDC*, pages 1–12, 2013.

[19] S. Lakshminarasimhan, J. Jenkins, et al. ISABELA-QA: Query-driven analytics with ISABELA-compressed extreme-scale scientific data. In *SC*, pages 1–11, Nov 2011.

[20] S. Lakshminarasimhan, N. Shah, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. F. Samatova. Compressing the Incompressible with ISABELA: In-situ Reduction of Spatio-temporal Data. In *Euro-Par*, pages 366–379, 2011.

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

[22] A. P. Marathe and K. Salem. Query processing techniques for arrays. *The VLDB Journal*, 11(1):68–91, Aug. 2002.

[23] F. Rusu and A. Dobra. Glade: A scalable framework for efficient analytics. *SIGOPS Oper. Syst. Rev.*, 46(1):12–18, Feb. 2012.

[24] E. Soroush, M. Balazinska, and D. Wang. ArrayStore: A Storage Manager for Complex Parallel Array Processing. In *SIGMOD*, pages 253–264, 2011.

[25] T. Tu, H. Yu, et al. Remote runtime steering of integrated terascale simulation and visualization. In *ACM/IEEE Supercomputing Conference HPC Analytics Challenge*, 2006.

[26] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka. Topology-aware Data Movement and Staging for I/O Acceleration on Blue Gene/P Supercomputing Systems. In *SC*, pages 19:1–19:11, 2011.

[27] K. Wu, S. Ahern, et al. FastBit: Interactively searching massive data. In *SciDAC*, 2009.

[28] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Usenix HotCloud*, pages 10–10, 2010.

[29] Y. Zhang, M. Kersten, and S. Manegold. SciQL: Array Data Processing Inside an RDBMS. In *SIGMOD*, pages 1049–1052, 2013.

[30] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. PreDatA: Preparatory Data Analytics on Peta-scale Machines. In *IPDPS*, pages 1–12, April 2010.