Spatially Clustered Join on Heterogeneous Scientific Data Sets

Bin Dong, Surendra Byna, and Kesheng Wu
Lawrence Berkeley National Laboratory, 1 Cyclotron Rd, Berkeley, CA 94720
Email: DBin@lbl.gov, SByna@lbl.gov, KWu@lbl.gov

***Abstract*—In the era of data-intensive scientific discovery, data analysis is critical for scientists to identify essential information from the mountains of data generated by large-scale simulations or experiments. A generic operation in scientific data analysis is to combine information from multiple data sets, which are stored in heterogeneous file formats. This operation is typically known as a *Join* in database management field. Currently, a join operation involving multiple data sets in different file formats is time-consuming because of the need to prepare data (i.e., to convert data into a uniform format or to ingest into a database) and to run the join algorithms. Furthermore, data processing languages, such as SQL (Structured Query Language), can not easily express typical scientific analysis tasks such as interpolation. In this paper, we propose three techniques to address these challenges: a two-level data model to process data from different file formats without converting to a uniform format, a data organization structure known as Multi-Dimensional Binning (MDBin), and a join processing algorithm known as Spatially Clustered Join (SCJoin). Together, these techniques allow scientific data files to be used for query processing with less I/O cost and fast query response time without the extra cost to perform file format conversion and data ingestion. Evaluation of our proposed techniques in joining and interpolating data sets generated by a plasma physics simulation studying space weather phenomenon showed up to 8X improvement over FastQuery. Querying with our solution outperforms SciDB, a popular array data management system for scientific data, by 43X-143X. We also demonstrate that our methods scale to 64K CPU cores in analyzing 32TB data on a large-scale supercomputing system.**

I. Introduction

Scientific data analysis operations often need to combine multiple data sets to extract knowledge. For example, in climate science, the atmospheric carbon dioxide (CO_2) and the surface radiative forcing of the earth are used together to study the effects of greenhouse gases [10]; in biological sciences, images from different mass spectrometers are compared to develop new technologies for molecular imaging [20]. Moreover, many of the scientific data sets are massive. In this study, we aim to develop efficient techniques to support complex *join* operations that extract information from combining large heterogeneous scientific data sets.

Traditional relational database management systems, such as Oracle, PostgreSQL, and array-based data management systems such as SciDB can answer join queries [19], [3]. However, scientific data is often stored in files with formats such as HDF5, NetCDF, ROOT, etc. Using the database management systems need a time-consuming process of preparing and loading massive amounts of data into formats defined by these systems. Furthermore, the languages/interfaces used by these systems to perform data analysis are inefficient to support the operations such as interpolation that are frequently used in scientific data analysis. There are some efforts on answering queries directly using the scientific data files, such as FastQuery [5]. However, these systems either do not support joins or require all data files to be in the same format.

To address the above shortcomings, we present SDS-MD (a Scientific Data Service for Multiple Data sets) as set of new features of the Scientific Data Services (SDS) framework [26], [7]. In our previous work, we have demonstrated various data management services of SDS such as transparent data reorganization [6] and parallel range query processing [2], [8] on a single file. Here we develop new methods for efficient joins on multiple data sets in different formats. The key features of SDS-MD are the following:

- We design a lightweight two-level data model to enable querying directly on scientific data files without requiring format conversion or data ingestion. In this two-level data model, the top level abstraction captures a view of multi-dimensional array and the lower level abstraction contains the information of file format.
- We introduce a multi-dimensional binning structure named MDBin to organize the user data. The 1-D version of MDBin has inherited advantages of the clustered indexing technique we studied earlier [27], [1], but MDBin does not rely on any index and works with arbitrary number of variables. Our clustered data organization allows query processing to be performed on relatively large data blocks stored in large-scale parallel file systems.
- We develop a parallel query processing algorithm named spatially clustered join (SCJoin), to answer *join* queries efficiently. SCJoin uses the MDBin data structure to reduce I/O costs on parallel file systems. At the same time, SCJoin can use the abundant parallelism available on high performance computing (HPC) systems. Furthermore, SCJoin supports a convenient way to express common operations such as the multi-dimensional interpolation, avoiding the need for complex SQL statements.

We evaluate SDS-MD using the queries derived from analysis tasks from a plasma physics application that studies the behavior of plasma in space [4]. The queries involving linear, bilinear, and trilinear interpolations that are precursors to more advanced analysis operations. We compare the performance of SDS-MD with FastQuery that is currently used for analyzing the plasma data [4], and with SciDB, an array-based data management system developed for large-scale scientific data analysis. From the experimental results, we observed that SDS-MD significantly outperforms both FastQuery and SciDB.

The rest of the paper is organized as follows. In Section II, we provide various research efforts and the background for scientific data analysis using SciDB, FastQuery, and SDS. In Section III, we present the technical details of our two-level data model, MDBin, and SC-Join. Section IV describes our experimental setup. In Section V, we evaluate the performance of SDS-MD. We conclude the paper with a discussion of future work in Section VI.

II. Background and Related Work

A large class of methods to support scientific data analysis are based on relational database systems [19]. Realizing that most scientific data are stored in multi-dimensional arrays, array-based database systems have been recently developed [3]. The best known example of such a system is SciDB [3], which is designed for shared-nothing architectures. In addition, there are also tools and libraries that directly work on the data files without a database system [25], [5], [14], [15]. The example we plan to use from this group is FastQuery [25], [5], which is known to be efficient for processing range conditions on a single data set, but does not support joins on multiple data sets.

Scientific data is typically stored in files using one of the handful of common file formats such as HDF5 [11], NetCDF/PNetCDF [13], FITS[24], etc. Many scientific applications also use custom binary formats defined by application developers. Therefore, it is common for the data sets needed for a join operation are in different formats. For example, in the plasma physics application mentioned earlier [4], the particle data is in HDF5 files, while the magnetic field data is in a custom binary format. To establish a baseline for our work, we developed a program to convert the magnetic data into HDF5 so that queries could be processed with FastQuery [5]. For a second comparison with SciDB, we need to go through an extensive data preparation or loading processes, which are known to be time-consuming [2]. The metadata catalog systems like XMC cat and AMGA [12], provide efficient way to manage metadata but lack specific support for data querying.

Data management systems typically use auxiliary data structures to accelerate query processing. For example, to accelerate range queries of the form *"$0<T<2$"* and *"$10<P<20$"*, database systems utilize data structures known as *indexes*, the most popular example is the B+tree. Besides, space filling curves and EDO [22] are also explored. It is much more important to use an index to process joins [23] because processing a join without any auxiliary data structure could be extremely time-consuming. Often it is also beneficial to reorder the data records following the index structure using a strategy known as clustered indexes [27]. Our work on MDBin takes inspiration from the clustered indexes. Given *"$0<T<2$"* is a bin for T and *"$10<P<20$"* is a bin for P, then the MDBin for T and P would include the bin *"$0<T<2$ and $10<P<20$"*. The values in the same bin are physically laid out together in the data file. This organization allows data records with similar values to be together, which captures the essence of Multi-Dimensional Clustering (MDC) [18], Order-preserving Bin-based Clustering (OrBiC) [27], and Bin-Hash Indexing [1]. We note that MDC uses the original values, while MDBin uses bins; and OrBiC and Bin-Hash Indexing are for one variable at a time and it also relies on the bitmap index for the same variable, while our proposed MDBin works for any number of variables and does not require a bitmap index.

A. SciDB

SciDB is an array database designed for *shared-nothing* architectures [3]. When the original file format and data organization are not recognized by SciDB, users need to prepare their data sets by converting them into one of the recognized formats. During the initialization, SciDB creates its system files and allocates disk space. Once all SciDB instances start to run, users can use the tools of SciDB to load data into the space of SciDB. The data preparation, system initialization, and data loading may happen only once in data analysis. Since large-scale HPC systems at centers such as NERSC[1] operate using batch submission systems, SciDB needs to run as a batch job. This requires starting the SciDB job before each data analysis run. To ensure the reuse of SciDB controlled data, it is also necessary to shutdown the SciDB batch job properly after each run. Inside SciDB, multiple data sets are often organized according to different attributes. When answering a query involving multiple attributes, SciDB reads each attribute separately. SciDB support SQL and AFL for users to perform their data analyses [3]

B. FastQuery

FastQuery [5], based on FastBit[25], is an open-source data processing library to build bitmap indexes and to query data with the bitmap indexes. Unlike FastBit, FastQuery works in parallel using multiple nodes and multiple cores within a processor. FastQuery also provides I/O drivers to work with different file formats, such as HDF5, PNetCDF, and ADIOS [17]. However, to work with queries on data in different file formats, FastQuery needs to convert data into a single file format and to merge the content of these files

[1]The National Energy Research Scientific Computing Center http://www.nersc.gov/

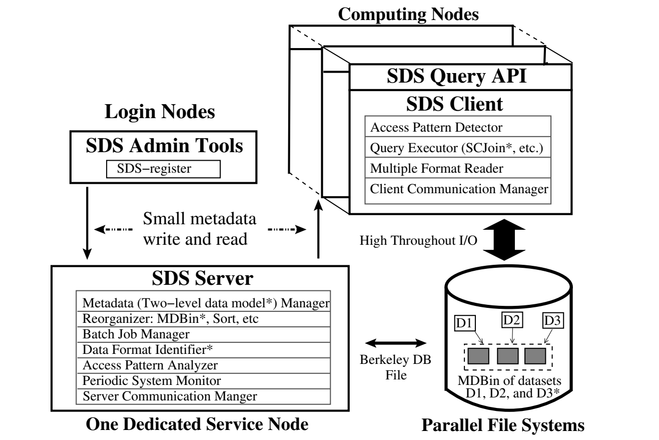


Figure 1. An overview of the SDS framework. Items marked with "*" are the new features developed for SDS-MD.

into one file. Data preparation step for FastQuery includes file format conversion, file content merging, and bitmap index building. To evaluate a query, FastQuery first uses bitmap indexes to find the locations of the interesting data and then reads these data from the original file. Thus, there are no *load data* and *initialize/start/stop* steps for using FastQuery in data analysis. In the file formats supported by FastQuery, multiple data sets are organized as different multiple-dimensional arrays and are stored separately. Hence, populating the selected data might results in a significant number of non-contiguous disk reads.

C. SDS

The Scientific Data Service (SDS) framework [7] provides data management optimizations as services. We present its overall architecture in Fig. 1. SDS has two main components, server and client. SDS server runs on a dedicated computer as a multi-threaded daemon, supporting metadata management, data reorganization, batch job management, and access pattern analysis services. SDS Client is a lightweight library linked with the user code to be run on compute nodes. SDS Query API is provided for applications to express SQL-like queries. When SDS Client receives a query, it sends a request to the Server to identify relevant metadata on reorganized data or indexes. If any reorganized data or indexes could benefit data access performance over reading the full data and scanning through it, the Server sends that information to the Client. The administrative tool also have the functions to provide hints of read patterns, to reorganize data and to build indexes. Since SDS uses a persistent server, there is no overhead to start and stop the system. As SDS directly works with the files in their original location, there is no data loading step.

The metadata from SDS Server to SDS Client is small in size and workload of SDS metadata server is lightweight. There is no metadata write/update requests from SDS Clients to SDS Server. The metadata is generated by services such as index building. SDS Server currently stores the file name of an reorganized file or bitmap index in Berkeley DB. To avoid having too many clients reading the metadata at the same time, the SDS Client library designates a single process for communication with the server, and that process broadcasts any information received from the server to all other MPI processes. As shown in our previous study [7], a single SDS server is sufficient to support the workload of Edison[2], a Cray XC30 peta-scale supercomputer at NERSC.

SDS Server runs a periodic service to identify potential (most recently used) files for reorganization or building indexes. Users can also start their reorganization after going through the performance evaluation component of SDS Server [7], [6], [16]. Based on data access patterns, SDS Server can also choose to read the subset of a big file and build index for the subset.

As the existing join methods in SDS has poor I/O performance in reading data from disk non-contiguously [2], we have developed a reorganization capability to organize multiple data sets into multi-dimensional bins (MDBin) and an algorithm named SCJoin to merge multiple data sets. Since current SDS only permits join on a single file format [2], a two-level data model and a data format identifier are designed to support data analysis directly on different formats.

III. SDS-MD: SDS for Multiple Data sets

The SDS-MD extension of the SDS framework enables efficient analyses on multiple data sets in different file formats. It employs three novel components: a two-level data model, a data structure named MDBin, and a spatially clustered join algorithm (SCJoin).

A. Two-Level Data Model

In supporting data analysis directly on different file formats without format conversion, the main challenge is to design a uniform data model to describe heterogeneous scientific file formats. This uniform data model should be easily constructed and should have a low overhead to serve a query. To this end, we propose a *two-level data abstraction* as shown in Figure 2. The top level of the abstraction is a multi-dimensional array, which is commonly used by many scientific applications. For example, climate data maps the globe as a 2D array corresponding to a mesh of latitude and longitude and stores different properties, such as temperature, pressure, at the nodes of each cell. The lower level of the abstraction refers to file formats, such as HDF5, user-defined binary format, NetCDF, PNetCDF, FITS, ADIOS, and so on.

Based on this two-level format, we use a small amount of metadata (SDS-MD metadata) on the SDS server to map between these two levels. SDS-MD metadata contains the dimensions of a multi-dimensional array, basic data type,

[2]https://www.nersc.gov/users/computational-systems/edison/

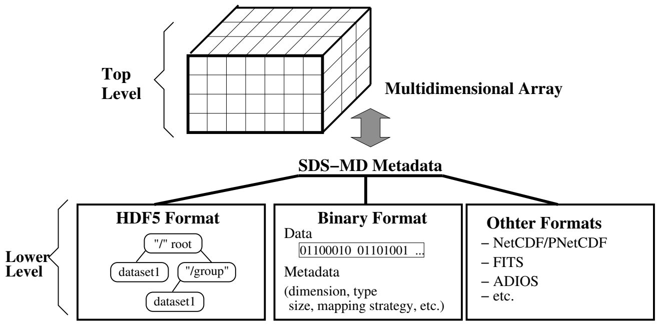


Figure 2. Two-level data model of SDS-MD

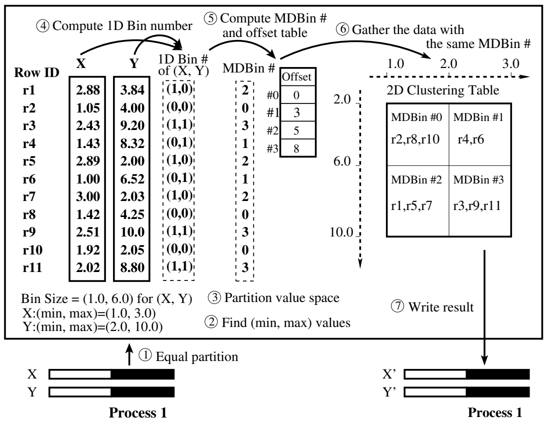
file format, and file layout strategy. Take a binary file as an example, where SDS-MD metadata would contain [*(file id, (3, (3,4,5)), (binary format id, float type id, row-major))*], where *(3, (3,4,5))* indicates that it is a *3*-D array with size $(3 \times 4 \times 5)$. The *row-major* and *float type id* are used to find the location of data on disk and then read with the driver specified by *binary format id*. For data sets stored is self-describing format like HDF5, we keep most SDS-MD metadata in the original file to reduce communication overhead to read it from SDS Server. SDS-MD metadata only records its *file format id* at server to choose I/O driver at runtime.

Consistency between the top and the lower levels in the *two-level data model* is supported by comparing the last modified time of the files. When a new file is added into SDS, its last modification time is recorded at the SDS Server. In serving a query, the recorded modification time and real modification time are compared. When a file modification is detected during a periodical scan, the SDS server updates its SDS-MD metadata accordingly. As most scientific data is *write-once-read-many*[21], updating metadata is infrequent. User can also update the top level metadata through SDS *Register-tool*.

Automatic and manual file format identification. When a new file is added, SDS-MD reads its format magic numbers to identify the type automatically. For example, HDF5 stores *(89 48 44 46 0d 0a 1a 0a)* at known offsets (e.g., 1024) as its magic number. SDS *Register-tool* also accepts the format information through its command-line parameters for users to specify by manually or when the file (e.g., binary file) does not contain format magic number.

B. Multi-dimensional Array Binning

To improve I/O performance of data analysis, multiple data sets required by a single query should be organized on disk correlatively. However, existing data layout strategies of file format such as HDF5 or array database such as SciDB do not provide a way to capture this correlation. Concurrently retrieving data from multiple data sets may translate into a large number of small sized read operations at scattered locations, which are typically much more time-consuming than a few number of sequential and large read operations. Many data organizations have been designed to turn these scattered read operations into sequential reads. For example, sorting a single data set reduces the time to access the values between a lower and an upper bound because all values in that range would be contiguous after sorting. To improve the accesses to multiple data sets, we introduce a parallel binning strategy, called *Multi-dimensional Array Binning* (MDBin).



Row ID	X	Y	1D Bin # of (X, Y)	MDBin #
r1	2.88	3.84	(1,0)	2
r2	1.05	4.00	(0,0)	0
r3	2.43	9.20	(1,1)	3
r4	1.43	8.32	(0,1)	1
r5	2.89	2.00	(1,0)	2
r6	1.00	6.52	(0,1)	1
r7	3.00	2.03	(1,0)	2
r8	1.42	4.25	(0,0)	0
r9	2.51	10.0	(1,1)	3
r10	1.92	2.05	(0,0)	0
r11	2.02	8.80	(1,1)	3

	Offset
#0	0
#1	3
#2	5
#3	8

Figure 3. An example of applying MDBin to data set X and data set Y. The number of processes is two and each builds the MDBin for its corresponding data locally without exchanging the data cross processes. Details on Process 1 are presented here.

The first step of MDBin is partitioning the data among available CPU cores evenly. Each core reads its corresponding data into memory, and then each core finds the minimum and the maximum values, denoted *(min, max)*. Then, global *(min, max)* values of all cores are obtained with a call to the *MPI_Reduce* function. In the example of Figure 3, the *(min, max)* values for *X* and *Y* on Process *1* are *(1.00, 3.00)* and *(2.00, 10.00)* respectively. Here, *X* and *Y* are two 1D arrays. We assume the *(min, max)* values on Process *1* are global for simplicity. Based on a *Bin Size*, the value space between *(min, max)* of each data set is equally partitioned among processors into 1D bins. We discuss the selection of *Bin size* in the later part of this subsection. In the example shown in Figure 3, the *Bin size* for *X* and *Y* are *1.0* and *6.0*, respectively. Starting from zero, *1D Bin Number* for each row is computed. For example, the *1D Bin Number* for the first row of *X* is 1 and for the second row of *Y* is 0. Then, the data is scanned for the second time to compute *MDBin Number*. We use the row-major mapping formula to compute the *MDBin Number*. In general, for an n-dimensional bins represented as $C_1 \times C_2 \times \ldots C_n$, a given *MDBin Number* is specified with *1D Bin Number* (i.e., $(c_1, c_2, \ldots, c_n)$) of each data set. The *MDBin Number* is computed with below equation.

$$MDBin\ Number = c_n + C_n(c_{n-1} + C_{n-1}(\ldots + C_2 c_1)).$$

In the example, the *1D Bin Number* for the first row is $(1, 0)$. The dimension size of the final results is 2×2. Hence, the *MDBin Number* for the first row is $0 + 2 \times 1 = 2$.

While scanning the data to compute the *MDBin Number*, we compute an *offset table* for all bins as well. This *offset table* is used to organize the resulting files and to serve query as index. Then, the rows with the same *MDBin Number* are gathered through the third scan to create an abstract multiple dimensional clustering table. In Figure 3, we use two 1D data sets and the values of those two data sets forms a two-dimensional clustering table. In the clustering table, the rows *r2*, *r8*, and *r10* of *X* and *Y* have same *MDBin Number*, and are written into bin zero of corresponding result data set. All rows within the MDBin bin zero fall into the range query that can be expressed with $1.0 < X < 2.0 \ and \ 2.0 < Y < 6.0$ and two contiguous disk reads (one for *X* and one for *Y*) are needed to access data satisfying the query.

Finally, based on the clustering table, the gathered data from a data set are contiguously written to another result data set by each core. In Figure 3, the value from data set X is written to X' and the value from data set Y is written to Y'. This design ensures the query performance on a single data set because *MDBin* on a single data set could be regarded as sorting. As discussed in Section II-C, SDS manages the size of the replicated data through identifying most frequently accessed files and indexes. For the MDBin replication in this example, the original data sets X and Y could be replaced with X' and Y' as they contain the same values. Hence, no extra storage space are needed. MDBin performs binning independently on each core and stores the resulting bin based on process rank. This avoids expensive communication among the cores to exchange partially binned data to create a global set of bins. When data is written to the file system, the *offset table* is also stored as a separate file. The (*min*, *max*) values and the MDBin size are stored on the SDS Server as metadata, which is used to answer queries.

Bin Size is determined based on the *(min, max)* values and the number of cores. In general, *Bin Size* $= (max - min)/number\ of\ cores$. This method maximizes the parallel processing on a single data set when the value distribution follows uniform distribution. On the other hand, when the value distribution between *(min, max)* is skewed, some bins could be assigned more values than others. To deal with this imbalance efficiently, we use the *Balanced Reader* component of SDS [6] to re-balance the load of reading the bins. In other words, SDS is optimized to use parallelism embedded in parallel file systems. Another method to determine the *Bin Size* is to find it from the access pattern history. As we described in Section II-C, SDS runs a service to monitor the access patterns of all files. Based on the observed access patterns, SDS could choose more suitable *Bin Size* for each data set. We assume that the data and queries come into our system the first time and we compute the *Bin Size* from *(min, max)* values in this study.

Frequency to rebuild MDBin and its transparent accesses. Since scientific data typically follows “write once and read many” [21], MDBin is built once and used many times. Hence, (*min*, *max*) values do not vary in the bins of MDBin indexes. Even though, SDS tracks the modification time of the files which it has built MDBin indexes for, when the file data changes, SDS Server merges new data or build new indexes, and then updates metadata. Transparent accesses to MDBin data is supported by the batch job management and read redirection capability of SDS framework. SDS Server is able to start batch jobs for creating MDBin for a proper file when the system load is low. Once the MDBin job finishes, the file name of offset table and global *(min, max)* values is stored in SDS Server as metadata. After receiving a query from the application, SDS Client read the metadata from SDS server and use it to find the bins to read.

```
function SCJoin($B$, $P_B$, $I$, $P_I$, $P_{B,I}$, $r$, $s$)
        $B$ : a set of $n$ data sets // $n < m$ for general case
        $I$ : a set of $m$ data sets //$m = 2^n + 1$ for interpolation
        $P_B$ : conditional restriction on $B$
        $P_I$ : conditional restriction on $I$
        $P_{B,I}$: join predicate on $B$ and $I$
        $r$ : process rank in MPI group
        $s$ : MPI process group size
 1. VAR $d_I$, $d_B$, $d_r$=NULL //Buffer for $I$, $B$ and result
 2. VAR $i$=0, $j$=0//Pointer to next block of $I$ and $B$
 3. VAR $b_B$, $b_I$ // MDBin number
 4. VAR $A$ // Total number of Bin(s)
 5. VAR $T_o$ //Pointer to offset table
 6. Read MDBin Metadata for $B$ and $I$
 7. Extract (max, min)s from MDBin Metadata
 8. Compute $b_B$ using $P_B$, (max, min)s, and Eq. III-B
 9. Compute $b_I$ using $P_I$, (max, min)s, and Eq. III-B
10.Compute $A$ using $s$ and (max, min)s
11.$T_o$ = Read offset table specified by MDBin Metadata
12.$d_I$=MDBin_next($I$, $b_I$, $r$, $s$, $i$, $T_o$, $A$)
13.$d_B$=MDBin_next($B$, $b_B$, $r$, $s$, $j$, $T_o$, $A$)
14.WHILE $d_I$ is not empty DO
15.  WHILE $d_B$ is not empty DO
16.  $d_r$ = $d_r \cup$ Join (e.g, interpolation) of $d_B$ and $d_I$ with $P_{B,I}$
17.  $j$ = $j$ + 1
18.  $d_B$=MDBin_next($B$, $b_B$, $r$, $s$, $j$, $T_o$, $A$)
19.  END WHILE
20.  $i$ = $i$ + 1 // $d_I$ is empty for interpolation when $i = 1$
21.  $d_I$=MDBin_next($I$, $b_I$, $r$, $s$, $i$++, $T_o$, $A$)
22.END WHILE
23.RETURN $d_r$

function MDBin-next($F$, $b$, $r$, $s$, $i$, $T_o$, $A$)
        $F$ : a set of $n$ data sets
        $b$ : MDBin number
        $r$ : process rank
        $s$ : process group size
        $i$ : next data block as bins of MDBin
        $T_o$ : pointer to MDBin offset table
        $A$: total number of Bin(s)
 1. VAR $start = 0$, $end = 0$ //file offset
 2. VAR $d$ //result buffer
 3. $start$ = $T_o[(r + s \times i) * A + b]$
 4. $end$ = $T_o[(r + s \times i) * A + b + 1]$
 5. $d$ = Read MDBin data of $F$ between ($start$, $end$)
 6. RETURN $d$
```

Figure 4. Algorithm of SCJoin.

C. Spatially Clustered Join Algorithm

A *Join* algorithm is a typical method used for combining multiple data sets (tables) into a single one based on the matched value of a certain key [2]. In a HPC environment, data exchange (also named as data shuffles) across network are usually required by join algorithms to match the keys located on different processes. The data exchange might hinder query processing as it usually involves expensive all-to-all communication. One method to avoid the data exchange is to let each process to retrieve its wanted data from parallel file system rather than from remote memory. To this end, the value distribution inside all data sets must be known by the join algorithm and at the same time, the join algorithm must have comparative I/O performance.

As discussed in previous sections, the value space clustering of MDBin can describe the value distribution not only inside a single data set but also across different data sets. In MDBin, each process locally builds the MDBin bins for its assigned data and writes the final bins to result files based on the process rank. In that sense, we design SCJoin algorithm (shown in Figure 4) to support efficient combination of data sets. SDS Server builds MDBin for its input data sets without SCJoin involvement. SCJoin looks up the MDBin metadata to read only the necessary MDBin bins in large and contiguous disk blocks for joining and therefore avoids the expensive data shuffle. When MDBin file does not exist, hash or nested loop join could be used. Also, to improve the usability for scientists, the scientific operation, i.e., interpolation, is supported by the SCJoin.

As the inputs of SCJoin, the data set groups *B* and *I* contain the name of data sets to be joined. Without losing generality, *B* has smaller number of data sets than *I* (*n¡m*). In an interpolation, m is equal to 2^n+1. SCJoin also accepts the conditional restrictions P_F and P_I. Usually, users can use the conditional restriction to filter the data set in advance and therefore reduce the data size for join. For example, in two data sets *A* and *B*, the conditional restriction could be *"19<A<83 and 2<B<17"*. It also accepts join predicate $P_{B,I}$ such as *B.X=I.Y*, where *X* and *Y* are data sets of *B* and *I* respectively.

From line *6* to line *11*, SCJoin obtains the MDBin metadata, including *(min, max)* values and *offset table* file name for *B* and *I*, respectively. Then, SCJoin reads them into corresponding variable memory spaces. By employing the same method used to build MDBin file in Figure 3, SCJoin (line *12* and line *13*) computes the *MDBin Number* (b_B and b_I) and the total number of bins (A) for the F and I separately. In the two *WHILE* loops (line *16* to line *22*), SCJoin combines the data block d_B from *B* and the data block d_I from *I* iteratively. Function ***MDBin_next*** is used to read the next data block (specified by *i* and *j*) from the corresponding MDBin files. Inside ***MDBin_next***, the *start* and *end* address of the MDBin file are obtained though looking up the *offset table*. The next data block is related to the process rank *r* and the size of MPI process group *s* because MDBin is built and stored locally by each process. When the number of processes for analysis is smaller than the number of the processes used for building MDBin, a process might need to read multiple MDBin blocks.

Additionally, in the two *WHILE* loops, we choose data set group with smaller size (not the group with small number of data sets) as outer loop to keep most of its data in memory. For example, the file for interpolation usually has small size and we can set the $i = 0$. In the actually SCJoin operation (line *16*), the join operation (e.g., interpolation) on two data blocks is evaluated. Other scientific operation like linear regression on two files can also be supported through the same method. The merged results are return at the end. Assume the number of binning block for *B* and *I* are P_B and P_I separately. In the worst case, SCJoin will run using $O(P_b + P_I)$ I/O operations.

IV. Experimental Setup

We ran all our experiments reported in this paper on Edison, a Cray XC30 system located at NERSC. Edison has 133,824 compute cores and is able to deliver a peak performance of 2.57 petaflops/sec. We used a Lustre file system that has 144 object storage targets (OSTs) and a 72GB/s peak bandwidth. We stored the data for all experiments on Lustre with the default 1 MB stripe size [9] using all the OSTs. We used a monitoring node, known as MOM node[3] to run the SDS Server. For running SciDB on Edison, we have created a directory on the Lustre file system to store its system files and the loaded data. To use SciDB, users need to submit jobs to initialize, start, and stop the SciDB system, to load data, and to evaluate queries.

V. Results

In this section, we first compare the performance of a typical scientific data analysis using SDS-MD, FastQuery and SciDB. We then evaluate the scalability and the configuration of SDS-MD.

Representative 1D and 3D arrays from real scientific applications are used in our tests. Specifically, the data sets are generated by a plasma physics simulation [4]. This simulation produces particle data (named *P*) of a trillion electrons stored in the HDF5 format and their corresponding magnetic field data (named *FX*,*FY*, and *FZ*) stored in three user-defined binary files. The HDF5 file consists of seven 1D data sets, named $Energy$, X, Y, Z, U_x, U_y, and U_z, where X, Y, and Z are 3D spatial locations of particles, and U_x, U_y, and U_z are the three components of the particle velocities. A *2*TB subset of *P* is used to evaluate analysis performance and a full size of *P* file (*32*TB) is used for scalability tests. The magnetic field value is sampled on a regular 3D mesh grid of size 1000^3. At each mesh point, magnetic field value is recorded as three components in three separate files named *FX*, *FY*, and *FZ*. Each of them actually contain a 3D data array with 12GB size.

We have used *conditional selection* and *join* queries that represent two types of popular analysis operations used by

[3]http://www.nersc.gov/systems/edison-cray-xc30/

Table I
THE QUERIES TO EVALUATE SDS-MD. P.X MEANS DATA SET *X* IN HDF5 FILE *P*. OTHER NOTIONS HAVE THE SAME NAMING PATTERN.

ID	Query Description	SQL Semantic
Q1	Filtering particles on (*P.Energy*, *P.X*, *P.Y*, *P.Z*)	Conditional Select
Q2	Linear interpolation on *(P.X, $P.U_x$)* and *FX*	*3*-way Join
Q3	Bilinear interpolation on *(P.X, P.Y, $P.U_x$, $P.U_y$)* and *(FX, FY)*	*5*-way Join
Q4	Trilinear interpolation on *(P.X, P.Y, P.Z, $P.U_x$, $P.U_y$, $P.U_Z$)* and *(FX, FY, FZ)*	*9*-way Join

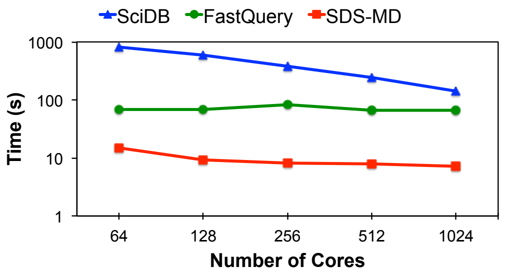


Figure 5. Time to execute query Q1 with SciDB, FastQuery, and SDS-MD. (The y-axis is in log-scale).

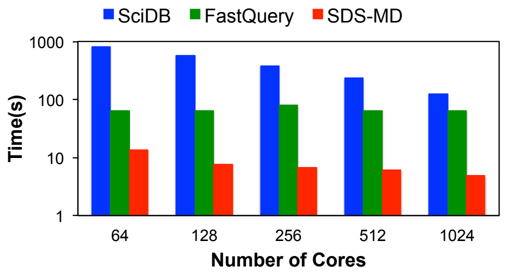


Figure 6. Time for reading the selected data for query Q1 with SciDB, FastQuery, and SDS-MD. (The y-axis is in log-scale).

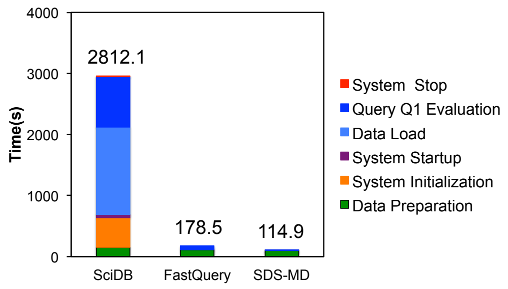


Figure 7. The overheads for preparing SciDB, FastQuery, and SDS-MD to run query Q1

scientists. We list the four queries used in this study in Table I. These queries come from real data analysis of plasma physics data [4]. Query Q1 filters the particle file *P* with value range conditions on its spatial and energy attributes. We use Q1 to compare the performance of querying data sets organized by the MDBin of SDS-MD, FastQuery, and, SciDB. We use Q2, Q3 and Q4, with multi-file SQL joins, to test the performance of SCJoin directly on multiple files in different formats. Domain scientists use these queries to explore the magnetic reconnection phenomenon, where particle data is combined with magnetic field values based on the location of the particles in X, Y, and Z directions. Since magnetic field value is sampled at regular mesh points, a trilinear interpolation is required to find the magnetic field value at the location of the particle.

A. *MDBin with Q1 on multiple data sets*

To filter particles with certain Energy and within a spatial range in X, Y, and Z directions, we evaluate Q1 with the following highly selective conditions: "*1.2<Energy<1.3 and 140<x<150 and 65<y<75 and 4<z<14*". This query identifies *148K* particles (hits) out of the 80 billion particles stored in the particle data.

In Figure 5, we show the overall performance of running Q1 with SciDB, FastQuery, and SDS-MD using different number of CPU cores ranging from 64 to 1024. SDS-MD outperforms SciDB by 20X and FastQuery by 9X when using 1024 processes. To evaluate where SDS-MD is gaining performance, we analyzed the time spent in reading the data. As shown in Figure 6, ≈95% of the overall query execution time for all the systems is spent in reading the data. Since SciDB does not use indexes to query the data, it performs a full scan of the entire data sets, resulting in the longest reading time. FastQuery first finds the coordinates of selected values from a bitmap in parallel. FastQuery provides two options to retrieve selected data, 1) sequential read (by gathering coordinates on a single thread) and 2) parallel read (by letting each thread broadcast its coordinates to all other threads and each thread reads all selected data). Parallel read has worse performance because of all-to-all coordinates information exchange and reading data non-contiguously, which is proved in our previous work [8]. In this experiment, we choose sequential read and therefore we observe poor scaling performance with FastQuery in Figure 6.

Both SciDB and FastQuery also spend a significant portion of time on read because the values select by the given query conditions are scatter randomly in the HDF5 files. With MDBin of SDS-MD, the four data sets used in setting the query are reorganized based on their clustering relationship that is embedded in their value space. In this test, the *Bin size* is determined with the number of cores and the (min, max) values of the four data sets. If we use the range granularity of the query string, MDBin could deliver even better performance. Reading the data satisfying the query only need a few contiguous disk read from MDBin, giving the observed advantage over FastQuery and SciDB.

We now analyze the overheads of using the three systems. Figure 7 shows the overheads involved in preparing the systems for running queries. The index generation time (110s), as part of data preparation, is the main overhead for FastQuery. To load the data into SciDB, we first converted the particle file (from the HDF5 format) into multiple binary

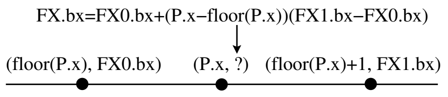


Figure 8. An example of linear interpolation.

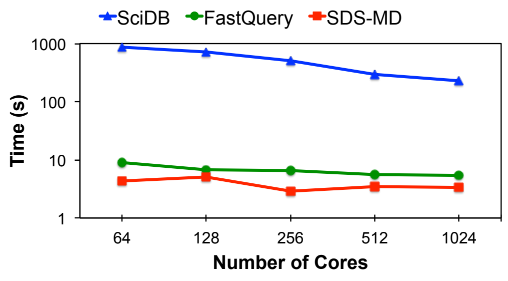


Figure 9. Time to run query Q2 with SciDB, FastQuery, and SDS-MD

files. We then loaded the data to SciDB with its *load ()* Array Functional Language (AFL) operator. SciDB also has substantial one-time initialization cost. The overheads of SciDB, including system initialization and data load times, are higher than the time to executing query Q1. With SDS-MD, the main overhead is with reorganizing the data into multi-dimensional bins using the MDBin approach. This cost is much less than the data preparation and loading costs of the other two systems.

B. SCJoin to query multiple data sets

In this section, we evaluate the performance of different systems running three *join queries* on multiple data sets. These queries are used by domain scientists to perform interpolations on the particle data and the magnetic field data. The three queries come from linear interpolation with 1D data (named Q2), bilinear interpolation with 2D data (Q3), and tri-linear interpolation with 3D data (Q4). Figure 8 gives an example of computing linear interpolation with 1D data, where a magnetic field value $FX.bx$ (i.e., value of variable *bx* from the file named *FX*) at point $P.x$. To compute $FX.bx$, two values at locations *floor($P.x$)* and *floor($P.x$)+1* are required to read from binary file *FX*. Using *FX0* and *FX1* as two aliases of *FX* and *FX0.bx* and *FX1.bx* to denote the magnetic field values at *floor($P.x$)* and *floor($P.x$)+1*, respectively, the $FX.bx$ is computed as: *FX0.bx + (P.x - floor(P.x)) (FX1.bx-FX0.bx)*. In all, to perform the linear interpolation for a particle, *X* needs to be read from file *P* in HDF5 format and two *bx* values need to read from file *FX* that is in binary format. This is actually a 3-way join in SQL semantics. Through the same analysis, a bilinear interpolation requires a *5*-way join and a tri-linear interpolation requires a *9*-way join, as shown in Table I. In queries Q2, Q3 and Q4, a conditional string on *Energy*, *X*, *Y*, and *Z* is also applied to filter the particles to a specific spatial region used in Q1. The query conditions are needed as the magnetic reconnection occurs in a specific region [4].

We compare the time to run query Q2 on different systems in Figure 9. This plot does not include the time to prepare and load the data and to initialize the systems. These overheads are shown separately in Figure 10. SciDB takes the longest time to answer this query as it needs to scan the entire data to evaluate the conditional string, to exchange data for applying the join, and to read data from non-contiguous locations in the magnetic field file *FX*. Without considering the data preparation time and data load time shown in Figure 10, SCJoin of SDS-MD performs 69X faster than SciDB for executing Q2 using 1024 cores. The value space clustering from MDBin is used by SCJoin to read only the necessary bins from storage and also in a contiguous manner. As FastQuery can only work with a single file, *FX0.bx* and *FX1.bx* values for all *floor($P.x$)* and *floor($P.x$)+1* points had to be calculated as part of the data preparation step and stored in the same particle data file (P) in the HDF5 format. Hence, as shown in Figure 10, the data preparation time is significantly high. For SDS-MD, the data preparation involves building MDBin files for P and binary data *FX*. Excluding the enormous data preparation cost of FastQuery, SDS-MD on average performs 1.6X faster than FastQuery with the cores number from 64 to 1024. SDS-MD outperforms FastQuery by 45X when the data preparation costs are included.

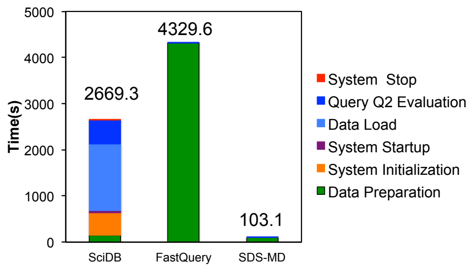


Figure 10. The overheads for preparing SciDB, FastQuery, and SDS-MD to run query Q2

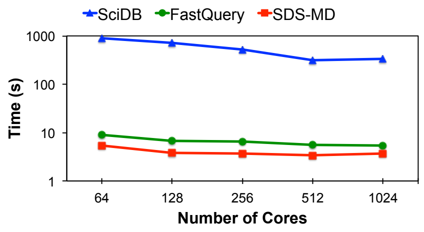


Figure 11. Time to run query Q3 with SciDB, FastQuery, and SDS-MD

We compare the time to run the queries Q3 (bilinear interpolation) and Q4 (tri-linear interpolation) on all three systems in Figures 11 and 12, respectively. Similar to the previous evaluation, these times do not include the data preparation and the data load times for all three systems. For Q3, SDS-MD outperforms SciDB by $135X$ on average and FastQuery by $1.7X$ to execute query Q3 with increasing the number of CPU cores from 64 to 1024. With the overhead costs included, speedups of SDS-MD are 42X and 26X over SciDB and FastQuery, respectively. For Q4, SDS-MD is

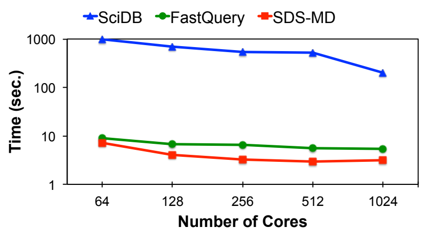


Figure 12. Time to run query Q4 with SciDB, FastQuery, and SciDB

Table II
SUMMARY OF THE SDS-MD SPEEDUP OVER SCIDB AND FASTQUERY

Query ID	Query Running Time		Total Time	
	SciDB	FastQuery	SciDB	FastQuery
Q1	43X	8X	16X	1.5X
Q2	135X	1.5X	26X	41X
Q3	135X	1.6X	26X	42X
Q4	143X	1.7X	26X	42X

143X faster than SciDB and 1.7X faster than FastQuery. With the overhead costs included, SDS-MD outperforms SciDB and FastQuery by 41.8X and 26.4X, respectively.

C. Evaluation of SDS-MD configurations

In this section, we report strong and weak scaling of building MDBin, and evaluate the impact of *Bin Size* on the performance of reorganizing data and on running queries with SDS-MD. We have executed query Q1 on different particle files, with their size varying between $2TB$ and 32TB.

In Figures 13 and 14, we show the performance of reorganizing the particle data using MDBin strategy as we increase the number of CPU cores. We conducted weak and strong scaling tests separately and report the data read time, binning time, bin write time, and offset table write time. In all, the read and the write times are dominant factors and the time for building MDBin. While the cost of building MDBin ($\approx 7\%$ of the total time) scales well from *2,000* to *32,000*, the overhead of reading and writing data dominate the overall cost.

In Figure 15, we show the variability in performance for running query Q1 with different *Bin Sizes*. The optimal performance for running query Q1 is obtained when *Bin size* for *(Energy, x, y, z)* is *(0.1, 10, 10, 10)* (denoted as *Bin Size** in the X-axis. With this *Bin size*, the boundary of query Q1 matches well with the bin boundaries. When the *Bin Size* is smaller than *Bin Size*, multiple smaller bins are read from storage and merged to answer query Q1. When the *Bin Size* is greater than *Bin Size**, SDS has to read the entire bin and then filter the data that satisfies the given condition, which is costly.

In summary, we compare the speedups of SDS-MD over SciDB and FastQuery in Table II to run various queries without the data preparation time (labeled as “Query Running Time”) and with the data preparation time included (labeled as “Total Time”). Once the data is prepared and loaded,

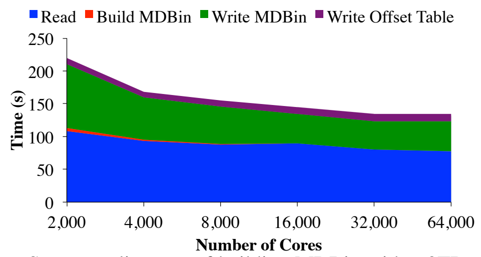


Figure 13. Strong scaling test of building MDBin with a 2TB particle file

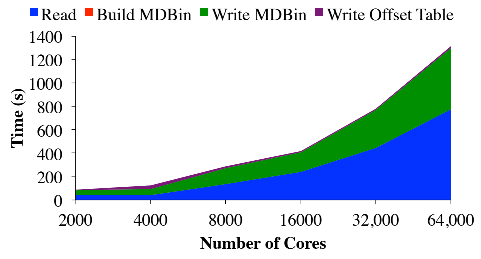


Figure 14. Weak scaling of building MDBin. Data size for each core is fixed *534*MB. In 64, 000 test case, the total particle file data size is *32*TB

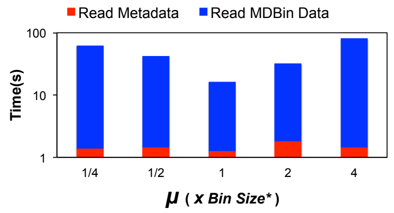


Figure 15. Time to serve query Q1 with different *Bin Sizes*. *Bin Size** is *(0.1, 10, 10, 10)* for *(Energy, x, y, z)*. The $\mu\times$(0.1, 10, 10, 10) is *(*$\mu\times$*0.1,* $\mu\times$*10,* $\mu\times$*10,* $\mu\times$*10)*.

SDS-MD performs 43X to 143X faster than SciDB and 1.5X to 8X faster than FastQuery. With the data preparation time included, SDS-MD performs 26X faster than SciDB for queries with multiple variables and with joins. SDS-MD outperforms FastQuery by ≈40X for the same queries with data preparation included.

VI. CONCLUSIONS AND FUTURE WORK

Diverse data formats, efficient data structures, and algorithm for join operations are nontrivial problems of data analysis in extreme-scale scientific data era. In this work, we introduce two-level data model, MDBin, and SCJoin to provide solutions for scientific data analysis. We demonstrate the advantages of our methods in answering the join queries from a real scientific application. We are currently working on providing a querying interface for making use of the SDS framework for different data and file formats.

ACKNOWLEDGMENT

This work was supported by the DOE Office of Science, Advanced Scientific Computing Research, under contract number DE-AC02-05CH11231 (Program manager: Lucy Nowell). This research used resources of the National Energy Research Scientific Computing Center (NERSC).

REFERENCES

[1] Bin-Hash Indexing: A Parallel Method For Fast Query Processing. Technical report, Laurence Berkeley National Laboratories, LBNL-729E, 2008.

[2] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani. Parallel data analysis directly on scientific file formats. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 385–396, New York, NY, USA, 2014. ACM.

[3] P. G. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 963–968. ACM, 2010.

[4] S. Byna, J. Chou, O. Rübel, Prabhat, H. Karimabadi, et al. Parallel I/O, analysis, and visualization of a trillion particle simulation. In *Proceedings of the Supercomputing Conference 2012*, SC '12, 2012.

[5] J. Chou, K. Wu, and Prabhat. FastQuery: A Parallel Indexing System for Scientific Data. In *CLUSTER*, pages 455–464. IEEE, 2011.

[6] B. Dong, S. Byna, and K. Wu. Expediting scientific data analysis with reorganization of data. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8, Sept. 2013.

[7] B. Dong, S. Byna, and K. Wu. SDS: A Framework for Scientific Data Services. In *Proceedings of the 8th Parallel Data Storage Workshop*, PDSW '13, pages 27–32, New York, NY, USA, 2013. ACM.

[8] B. Dong, S. Byna, and K. Wu. Parallel query evaluation as a Scientific Data Service. In *2014 IEEE International Conference on Cluster Computing, CLUSTER 2014, Madrid, Spain, September 22-26, 2014*, pages 194–202, 2014.

[9] B. Dong, X. Li, L. Xiao, and L. Ruan. A new file-specific stripe size selection method for highly concurrent data access. In *Proceedings of the 2012 ACM/IEEE 13th International Conference on Grid Computing*, GRID '12, pages 22–30, 2012.

[10] D. R. Feldman, W. D. Collins, P. J. Gero, M. S. Torn, E. J. Mlawer, and T. R. Shippert. Observational determination of surface radiative forcing by co2 from 2000 to 2010. *Nature*, 519(7543):339–343, 03 2015.

[11] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson. An overview of the HDF5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, AD '11, pages 36–47, 2011.

[12] B. Koblitz, N. Santos, and V. Pose. The amga metadata service. *Journal of Grid Computing*, 6(1):61–76, 2008.

[13] J. Li, W. keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A High-Performance Scientific I/O Interface. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 39–, New York, NY, USA, 2003. ACM.

[14] J. Liu, S. Byna, and Y. Chen. Segmented analysis for reducing data movement. In *Big Data, 2013 IEEE International Conference on*, pages 344–349, Oct 2013.

[15] J. Liu and Y. Chen. Fast data analysis with integrated statistical metadata in scientific datasets. In *CLUSTER*, pages 1–8. IEEE, 2013.

[16] J. Liu, B. Dong, S. Byna, and K. Wu. Model-driven data layout selection for improving read performance. In *Proceedings of High Performance Data Intensive Computing*, HPDIC'14, 2014.

[17] Q. Liu, J. Logan, Y. Tian, et al. Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks. *Concurrency and Computation: Practice and Experience*, 26(7):1453–1473, 2014.

[18] S. Padmanabhan, B. Bhattacharjee, T. Malkemus, L. Cranston, and M. Huras. Multi-dimensional clustering: A new data layout scheme in db2. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 637–641, New York, NY, USA, 2003. ACM.

[19] PostgreSQL Global Development Group. PostgreSQL. http://www.postgresql.org/.

[20] A. Römpp, T. Schramm, A. Hester, I. Klinkert, J.-P. P. Both, R. M. Heeren, M. Stöckli, and B. Spengler. imzML: Imaging mass spectrometry markup language: A common data format for mass spectrometry imaging. *Methods in molecular biology (Clifton, N.J.)*, 696:205–224, 2011.

[21] A. Shoshani and D. Rotem. *Scientific Data Management: Challenges, Technology, and Deployment*. Chapman & Hall/CRC, 1st edition, 2009.

[22] Y. Tian et al. EDO: Improving Read Performance for Scientific Applications through Elastic Data Organization. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, CLUSTER '11, pages 93–102, 2011.

[23] P. Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2):218–246, 1987.

[24] D. C. Wells, E. W. Greisen, and R. H. Harten. FITS: a flexible image transport system. *Astronomy and Astrophysics Supplement Series*, 44:363–370, 1981.

[25] K. Wu, S. Ahern, E. W. Bethel, et al. FastBit: Interactively Searching Massive Data. *Journal of Physics Conference Series, Proceedings of SciDAC 2009*, 180:012053, June 2009. LBNL-2164E.

[26] K. Wu, S. Byna, D. Rotem, and A. Shoshani. Scientific data services – A high-performance I/O system with array semantics. In *HPCDB*. IEEE, 2011.

[27] K. Wu, K. Stockinger, and A. Shoshani. Breaking the curse of cardinality on bitmap indexes. In *Proceedings of the 20th International Conference on Scientific and Statistical Database Management*, SSDBM '08, pages 348–365, Berlin, Heidelberg, 2008. Springer-Verlag.