DSDQuery DSI - Querying Scientific Data Repositories with Structured Operators

Roee Ebenstein∗, Gagan Agrawal†
Department of Computer Science and Engineering
The Ohio State University
Email: ∗ebenstei@cse.osu.edu, †agrawal@cse.osu.edu

Abstract—Scientific data is often distributed through repositories that host a large number of files in formats such as NetCDF or HDF5. With recent and anticipated increases in the size of observational and simulation data, it is important to transport just the data that are of interest from a large distributed dataset. Unfortunately, existing portals provide limited querying interfaces – typically a set of predefined hard coded subsettings, limiting user’s querying flexibility.

This paper describes a system that addresses this gap. The relational algebra is adapted for scientific array querying allowing us to adapt a subset of SQL for this domain, which enables nuanced subsetting conditions to be applied on a set of dataset files within a repository. A query processing algorithm extracts and collects data from relevant datasets, based on metadata that was earlier extracted using an automatic metadata extraction engine. Finally, the system stitches a new structured, NetCDF, file to be returned as a resultset, allowing the returned data to be used and analyzed by existing tools. The system has been extensively evaluated to show its ability to handle increasing data and/or number of files.

I. INTRODUCTION

Data collected from instruments and simulations have been extremely valuable for a variety of scientific endeavors - it is reported that the climate change and atmospheric research communities published more than 500 papers between 2006 and 2010 using data distributed by the Earth System Grid (ESG)1. The key challenge is that while dataset sizes grow rapidly, disk speeds and wide-area transfer bandwidths have not increased. Thus, software tools for dealing with scientific data must incorporate new approaches, for data-driven scientific advances to be maintained in the future.

Scientific data is typically disseminated through portals such as ESG. These portals typically support processing a collection of files that are stored in formats such as NetCDF[1] or HDF5[2]. Though the data stored in these portals is of wide use, scientists are likely to be interested in a relatively small subset of data. These portals, however, provide only a limited querying functionality to obtain desired data subsets.

To motivate the need for more sophisticated querying abilities, we consider the following example. Climate simulations, like the Community Earth System Model (CESM), are producing massive datasets. The current output organization involves keeping all the variables for the entire globe for one time-slice in a single NetCDF file. However, most researchers focus on a small number of variables of a specific region within a specific time-range. Selecting the data of interest for the user involves spatial or spatio-temporal subsetting of data over a cartesian (non-rectilinear) grid. Unfortunately, it means that the user has to download orders of magnitude greater volume of data over the Wide Area Network (WAN) than they need, and then process the data locally.

Standard databases allow the querying flexibility required, but for various reasons scientific data are not stored in these mature popular systems. This gap has been addressed to an extent by new emerging array databases such as SciDB [3], which require loading the data while changing its storage format to a proprietary one, imposing overheads that cannot be justified for massive, read/append-only, infrequently accessed datasets.

In our recent work, we developed SDQuery DSI [4], which supports an efficient SQL-like selection operators over a single NetCDF or HDF5 file. Though useful, this work is limited to subset one file through one query. Our current system, DSDQuery DSI, addresses this issue in addition to the following: locating the data that is relevant for the query without requiring the user to intimately know the datasets, querying across multiple datasets of different formats and extracting relevant data, and, last, combining the output elements into one unified and combined output dataset (NetCDF, for example).

Our system allows the use of a subset of SQL to create a structured database, based on extraction of content from multiple datasets. Two components comprise our system: Bridger, which automatically detects new files and datasets, and extracts its metadata; and DSDQuery DSI, an extension on top of SDQuery that enables querying multiple datasets. Our system can be viewed as an application of the previously proposed No-DB approach [5], [6], [7]. Our work is unique in providing a database-like support over a set of files in scientific formats like NetCDF.

II. BACKGROUND AND DESIRED FUNCTIONALITY

A. SDQuery DSI

The work presented in this paper is built on top of SDQuery DSI [4], which was developed to minimize the amount of data needed to be transferred over a WAN, by subsetting data from a single file. SDQuery embeds data management functionality within Globus GridFTP [8] – an extendable File Transfer Protocol (FTP) server.

SDQuery DSI reacts to an FTP PUT by receiving the sent file, and generating an index over it, and to an

1http://www-pcmdi.llnl.gov/ipcc/subproject_publications.php
FTP GET by examining the given command, and if it contains a query, processing it. If, for example, the user issues the following FTP command:

```
GET /home/user/POP.nc (SQL: SELECT TEMP FROM POP WHERE TEMP __GE 8.1 AND TEMP __LT 10.2)
```

SDQuery DSI would access the file POP.nc, and based on the inputted query, will extract the values of the variable TEMP that are in the range [8.1, 10.2].

B. Desired New Functionality and Challenges

The work presented here provides the following new functionality:

- DSDQuery DSI allows querying multiple files.
- DSDQuery DSI can locate the relevant files and datasets for a given query using its metadata repository, which is populated automatically.
- In contrast to SDQuery DSI, DSDQuery DSI generates a full structured database file in a format chosen by the user (currently only native coordinate, \((x, y, val)\), and NetCDF formats are supported).

To further illustrate the points above, we use an example. In SDQuery, a climate researcher query, rephrased in plain English, would look like “Get the evaporation data of a specific area from the file /server/2015/01/21/POP.evap.nc”. Evidently, the researcher has to know the file system structure of the source repository. In DSDQuery, the query would be phrased as “I want to see the evaporation data of a specific area from the 01/21/2015 simulation”. We believe this declarative statement forms a better communication with the data source provider for multiple reasons: The user does not need to know the repository file system structure, which now can be changed independently from its users, and more importantly, multiple data sources can be used.

However, multiple challenges are raised from this declarative query:

- **Query Meaning**: Many existing engines provide querying ability using declarative languages, but nuances and implications of such queries are not clear in the context of array data. In our work, we use relational algebra to assign meaning to operators.
- **File Detection and Metadata Extraction**: We show how using the dataset and additional metadata allows us to further assign semantics and context to queries. Specifically, non-unique array and dimension names can form unique meaning by using additional metadata.
- **Unionizing (and Aggregating) Data**: We show how our target operators should be run over multiple files.
- **Storage Location and Query Planning**: Locating data sources for queries to execute on is critical for efficient execution of queries using declarative language over large repositories. We developed algorithms for creating master queries and data sources adjusted queries, each of the latter being a subset of the master query optimized for a specific data source.

III. FORMAL DEFINITION OF QUERY OPERATORS

Our system supports a subset of SQL for querying array data. We have chosen SQL because of its popularity, and particularly, because none of the array query languages [3], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18] have reached level of popularity or maturity comparable to SQL. The subset of SQL we are using is similar to the SQL used by SciDB.

To make SQL operations applicable to array data, we first formalize a mapping between an array dataset and the relational algebra[19]. Without such a mapping, interpretation of query operators can be ambiguous in many cases. The relational algebra is described upon one dataset. When querying multiple datasets the data from these datasets needs to be unionized (and sometimes aggregated) to form the full and final dataset. In this work, we do not consider operations that involve processing across multiple relations/variables (for example, Joins).

A. Formalization - Relational Algebra

Array databases are comprised of dimensions and variables. Each variable is constructed of a data type and dimensions – the same dimension(s) can be shared among variables. Each dimension can have a mapping variable, with the same name, that is used for mapping different values by its coordinate. For example, a dimension longitude might have a mapped variable that maps the coordinate 0 to a value such as 39.997, etc.

In Fig.1 we demonstrate the dissonance between an array database and a relational one. In an array database, each dimensional coordinate in a variable, which can be referred to as a cell, has exactly one value. All of the cells together form a multi-dimensional array. Mapping the array model to relational one, we can define a relation to comprise the values of the variables that have the same set of dimensional coordinates. In the model we describe here, each array database source is a set of multiple such relations, each having different dimensional coordinates.

**Projection**: In relational algebra, a projection is defined as a restriction of properties, or atoms, of a relation. This definition cannot be directly applied to array data because the data cannot be brought out of its context, i.e., dimensions, without using aggregation first. Therefore, we modify the definition of projection to apply only on non-dimensional atoms. For example, \(\Pi_{var2}(R)\) is a valid request for the data of the variable var2 from the repository. We allow querying multiple variables in one query – for example, \(\Pi_{var1, var2}(R)\) is a valid query.

**Selection**: A selection operation is of the form \(\sigma_\varphi(DS)\), where \(\varphi\) is a list of conditioned atoms that are connected by and operators, \((\wedge)\), or or operators \((\vee)\), and DS is the dataset at hand. The atoms in \(\varphi\) may be dimensions...
or variables names as in traditional relational algebra; each variable name and dimension name can be looked upon as a column name. For example, the query:

\[
\sigma_{\text{longitude} \leq 39.885 \land \text{depth} > 5} (\Pi_{\text{Temp}} (\text{DS}))
\]

implies: extract all the temperature values from the dataset DS that its longitude is smaller or equal to 39.885 and its depth is bigger than 5. In applying selection operation to array datasets remapping of the dimensions values might occur, and each data type has to have a NULL value – each value that is not selected, but for which a corresponding dimensional coordinate exists, is replaced with NULL. Dimensional coordinates that do not hold any values are discarded. If a dimension does not have a mapped variable we have two options: to copy the dimension as is, and replace all the values of the unselected coordinates with NULL, or to create a mapped dimension that maps the coordinates.

For example, if we have a variable with 2 dimensions, as DS (shown in Table I (a)), and we execute a query such as \(\sigma_{\text{val} = 1} (\text{DS})\), since the second column presented have no value equal to 1 this dimensional value can be dismissed. Assuming a mapped variable for the first dimension does not exist, the output would be either as shown in Table I (b), for the case we do not create a mapped variable. Alternatively, the result would be as shown in Table I (c) and (d), where the former holds the data for the dimension of the mapped dimension values and the latter holds the resulted variable. We implemented the second option since, although it requires a small additional processing time, it produces significantly smaller resultset size, which leads to shorter file transfer time. If the dimensional coordinates have not been modified, as is the case with the vertical dimension in the example, no additional processing is required.

**Set Operators:** We describe, using examples, the way one should think of set operations over array data.

**Union:** A union combines two different variables to appear as one in the output. In relational algebra, two relations need to be union compatible for the union to have meaning. For our case, a union compatible variables, are two variables that share the same dimensions. In the case the set of dimensions values are not identical, dimensional merge is performed and new cells are allocated with a value of NULL. Moreover, unlike the relational case, a union of two non-union-compatible variables is possible and allows constructing a dataset with multiple variables. For example, \(A \cup B\) for variables A and B in Table I (e) and (f) respectively is presented in Table I (g) where the first row contains the dimension values.

If the dimensional coordinates do not collide, as in the case where an array is spread over multiple nodes, we can avoid creating this additional dimension since no two different values will be assigned to the same dimensional coordinate. In our system, if there is no dimensional coordinate collision, we avoid creating the additional dimension, unless the unionization was user mandated explicitly.

**Intersection:** Intersection reports the values that are the same among the given variables, ignoring dimensions that do not match. The output has the same dimensions as of its sources. NULL is reported for dimensions that are created but when no actual coordinate match between the variables. In Table I (h) we demonstrate the result of intersection between variables A, presented in (e), and B, presented in (f).

**Aggregations:** The operator \(G_{\text{AGGR}(\text{dimension})} (\text{Var}, DS)\) executes the aggregation function \(\text{AGGR}\) using the given dimension on the variable Var that is stored in the dataset DS. For example \(G_{\text{sum}} (\text{TableI} (\text{VariableA}(e), DS))\), when ran on the data of variable A in Table I (e), would return a variable that has the value 4.0, since no dimensions were given for the action. \(G_{\text{avg}(\text{dim})} (\text{TableI} (\text{VariableA}(e), DS))\) will return the input variable, Variable A, since there is only one dimension, which the operation is on.

**Mathematical Operators:** Mathematical operators can be given over variables or dimensions. In either case, the operator may be applied on a singular value, or on an array of values. In the case of a singular value, for both, dimensions and variables, the mathematical operation is performed for each value of the relevant dimension or variable. If for example the user specifies \(\text{var1} \times 2\), or \(\text{dim1} \times 2\), each of the atom value’s is multiplied by 2. In a case where a variable or a dimension is multiplied by another variable, dimension, or a given array, one can either perform a matrix operation (which is the default in our system), or alternatively, perform a by place operation by using the “\(\cdot\text{op}\)” suffix.

### B. Example Queries

As concrete examples, we use two queries – they involve filtering results by value and dimensions. We consider a climate simulation output stored in a repository across a number of files. Some of the files contain the queried variable, TEMP. The queried variable dimensions are: longitude, latitude, and depth, though in certain datasets, addition dimensions also appear.

\[
\begin{align*}
&\text{SELECT AVG(TEMP)} \\
&\text{FROM TEMP} \\
&\text{WHERE TEMP. TEMP >= 5} \\
&\text{AND TEMP. depth >= 15}
\end{align*}
\]

\[
\begin{align*}
&\text{SELECT AVG(TEMP)} \\
&\text{FROM TEMP} \\
&\text{WHERE TEMP. TEMP < 15} \\
&\text{AND TEMP. latitude < 0}
\end{align*}
\]
The relational algebra for the query in Fig 2 (a) is
\[ \bigcup_{d \in DS} G_{AVG}(\text{longitude}, \text{latitude}, \text{depth}) \sigma_{\text{\text{TEMP}} \geq 5 \land \text{\text{TEMP}} < 15} (\Pi_{\text{TEMP}}(d)) \]

Here, we unionize the results of a query from multiple datasets. The query extracts the variable \text{TEMP} from each queried dataset \( d \), afterwards it subsets the variable by value, and then aggregates the results by three of its dimensions. Because certain files contain additional dimensions beyond these three, an aggregation needs to be executed first to conform all datasets to the three common dimensions. To recap, first, the system finds the datasets that are relevant for the query as described in the next section, then, for each dataset, a selection by variable values is performed. Afterwards, if needed, an aggregation is performed. Last, the data is collected from each dataset and unionized (and possibly aggregated) to form the final result.

In the second query, Fig 2 (b), the condition is not on the value of the variable, but on its dimensions. The relational algebra that represents this query is:
\[ \bigcup_{d \in DS} G_{AVG}(\text{longitude}, \text{latitude}, \text{depth}) \sigma_{\text{\text{depth}} \geq 15 \land \text{\text{latitude}} < 0} (\Pi_{\text{TEMP}}(d)) \]

IV. SYSTEM DESIGN AND IMPLEMENTATION

A. Metadata Extraction

The metadata extraction is divided into five parts: \textit{new file detection}, \textit{atoms extraction}, \textit{statistical analysis}, \textit{atoms unification}, and \textit{DB enrichment}. The first part, new file detection, is done using two methods. First, the system registers a set of \textit{inotify}[20] file system listeners on the relevant directories. When a file is added or modified, a metadata scan is initiated and the new file is registered. Afterwards, on initialization, the system verifies its repository contains up to date data by scanning the file system. Differences between the repository and the file system are processed.

The next part, atoms detection, finds the relevant \textit{metadata atoms} within the detected dataset. The metadata atoms are mostly the variables and their dimensions. As will be described in Subsection IV-C, additional metadata can be added to each dataset and/or atom. Assuming we detected a new file, with the variable presented in Subsection III-B, the output would be variable: \text{TEMP}, dimensions: \text{depth}, \text{longitude}, \text{latitude}. Additional atoms can appear in the dataset’s metadata or in the additional metadata file.

Next, for each atom, statistical information extraction is activated, which help execute and/or optimize the queries. A query can be optimized by, for example, removing datasets that are irrelevant, or allocating resources by the portion of results that are expected or the dataset size. This part finds the minimum and the maximum values for each atom we detect.

Atoms unification makes sure all the atoms, manually added metadata enrichments, and statistical information look the same in the unified repository. The unification of the metadata from all the different dataset formats (for example: NetCDF or HDF5) makes sure the execution engine is agnostic to the source dataset format. The final part, DB enrichment, sends the unified atoms to the shared system’s repository.

B. Query Analysis

Algorithm 1 Query Analysis

```
1: function QUERYANALYZE(query)
2:    Maps = \emptyset
3:    pt ← ParseTree(q)
4:    for each v ∈ variable in pt.FROM do
5:        allq = \emptyset
6:        s ← v.dimensions (in all query section)
7:        mdq ← build query for v with all s
8:        DS ← executeQuery(Repository, mdq)
9:        cd ← FindCommonDimensions(v, DS)
10:       for each ds ∈ DS do
11:          stat ← queryDataSourceStats(v,ds)
12:          q ← regenerateQuery(query, cd, ds, stat)
13:          allq = allq U q
14:       end for
15:    Maps = Maps U (v,allq)
16: end for
17: return Maps
18: end function
```

In Algorithm 1 we describe how we analyze a query. This algorithm generates a map between a variable to the multiple single-file queries that needs to be run on each dataset. The query performance is mainly dependent on the query execution engine, SDQuery [4]. The query shown in Fig. 2 (b) is used here for demonstration.

The input to the algorithm is an SQL query. The query, in addition to stating the user intended results, is a \textit{dataset definition} corresponding to the output generated at runtime. The output dataset definition is based on the given variables and dimension, and the current repository content – if there are three files that contain the queried variable with the dimensions mentioned in the query, the output variable will contain all the dimensions that intersect across these files even if some of the dimensions were not explicitly queried.

When a query is submitted, we parse it, build a parsed query tree (line 3), and determine which variables are required for the query execution. These are iterated on (line 4). In the example query the only variable is \text{TEMP}. Afterwards, in line 6, for each variable we extract all the dimensions that are mentioned in the original input query. In the case a user did not explicitly relate a dimension to a variable the system deducts what variable should the dimension be related to by: using the metadata in the repository, the number of variables within the query, and/or query analysis. In our running example, the created map would be \( \text{TEMP} \{ \text{depth}, \text{latitude} \} \).

In line 7, we build a repository query for getting the datasets that contain the atoms that are mentioned in the original query, which is executed in line 8. For the example query mentioned above, the repository query asks for the datasets that has a variable named \text{TEMP} with both of the dimensions \text{depth} and \text{latitude}. Next, in line 9, we find the variable dimensions that intersects across all the detected datasets. In our example, the output would be \( \{ \text{depth}, \text{longitude}, \text{latitude} \} \); although longitude is not mentioned in the query, it appears in all the datasets.
that have the variable \texttt{TEMP} and the dimensions \texttt{latitude} and \texttt{depth}.

For each dataset and variable combination, in line 12 we generate an SDQuery query that is based on a subset of the given original query. The new query selects only the portion of the original query that is relevant for the specific variable and dataset, based on the dimensions, aggregations, and statistics that are relevant. The common dimensions are used here for building SD-Query queries that aggregate the necessary data before the merge process begins, which improves performance significantly since it can be executed in parallel and operates on smaller datasets (pre-unionization).

In Fig. 3, there is an example repository that has three files, two of these have the variable \texttt{TEMP}, while the last one does not. For the example query, given the repository described above, the common dimensions are: \texttt{depth}, \texttt{longitude}, \texttt{latitude}. The dimension \texttt{longitude} was added since both the files in the repository contain it, while the dimension frequency was not because it is not contained in all the “\texttt{TEMP}” variables that are in the repository. Assuming the statistical analysis concluded all the depths in the first file are lower than 15, the optimized SDQuery request for that file would be:

```sql
GET 1.nc (SQL: SELECT TEMP FROM TEMP WHERE latitude < 0)
```

while the SDQuery request for the second file would be:

```sql
GET 2.nc (SELECT AVG(TEMP) FROM TEMP WHERE depth > 15 and latitude < 0 GROUP BY depth, latitude, longitude).
```

Notice an aggregation is required in the second query since the dimension \texttt{frequency} is not in the common dimensions, and therefore cannot be unionized with the variable that is in the previous file.

The third file is not requested at all since it does not contain the variable \texttt{TEMP}. A query such as \texttt{SELECT TEMP FROM TEMP WHERE FREQUENCY = 1} would query only the file 2.nc, while a query such as \texttt{SELECT TEMP FROM TEMP WHERE NO = 1} would produce an empty dataset. The content of the repository determines the content of the output, and therefore the same query on different repositories might produce not only different content, but also different data structures.

C. System Implementation

In Fig. 4 we show our system architecture. The two parts the system is constructed from are Bridger and DSDQuery. Bridger’s detects and extracts metadata. DSDQuery receives user queries, executes each, and sends the query resultset to the user using its embedded transportation service.

1) Bridger: Bridger maintains the system’s repository. Since array databases are stored in a file system, changes can occur only when a file is created, modified, or deleted. For detecting when a file is added we add inotify listeners[20] to each directory the system is configured on. The system can be configured to automatically, and recursively, add listeners to subdirectories of the configured directories on initialization or as these subdirectories are created. When a file system event occurs, the system determines if the detected file is relevant or not.

Once a relevant file is detected, the system uses the file-type plugin to extract the file metadata, and the file-type specific statistical engine to extract statistical information. All the extracted metadata is unified and normalized for storage in a relational sqlite[21] database.

2) Additional Meta Data: The in-file dataset’s metadata is often not sufficient for semantical querying using declarative query language, since the data is brought out of context when variable and dimension names are used. The ability to enhance the dataset’s metadata is necessary. Since array datasets are physically stored in file systems, the file system hierarchy often has a meaning and therefore there are multiple types of metadata enhancement that are required:

- Enhancing a directory, and all of its subdirectories.
- Enhancing a dataset, and all of its atoms.
- Enhancing an atom within a specific dataset.

Each file with the extension .metadata is adding metadata to the system’s repository. The metadata file name, without the extension, determines which directory or dataset the additional metadata should apply for. The metadata file is structured from text lines, each line provides a metadata atom to add to the repository or instructions for the scanning process (such as which subdirectories should be added and if it should be done recursively and online).
Since SQL supports hierarchical data structures, we added the capability to subselect based on the metadata by using the prefix `metadata. *` where * is the metadata atom name. For limiting the results of a specific array, the keyword metadata will be followed by the array name. For example - for limiting the variables to those provided by a specific sensor, while giving a meaningful semantical context to the query, one could use the where clause: `metadata.TEMP.sensor = 'sensor name}'.

3) DSI Implementation: DSDQuery DSI is implemented as a plugin to Globus GridFTP [8], in the same way the precursor system SDQuery [4] was. If the request is a DSDQuery request, which is detected by having the string DSQL: right in front of a valid SQL statement, the engine extracts the SQL statements and applies the algorithm described in Section IV-B for analyzing the query, locating the relevant array datasets, and for generating the queries that should run on each individual dataset. The output of Algorithm 1 is a map between the variables that should appear in the resulting array dataset to a set of SDQuery queries that a union of, and possibly an aggregation over, would fill.

For example, the following FTP GET call:

```
GET / (DSQL:SELECT TEMP FROM TEMP
WHERE TEMP __GE 8.1 AND TEMP __LT 10.2)
```

would obtain from the process above a set of SDQuery queries similar to:

```
GET /home/user/POP.nc (SQL:SELECT TEMP FROM POP
WHERE TEMP __GE 8.1 AND TEMP __LT 10.2)
```

The generated queries differ in the location and name of the dataset that should be used, and in some cases with its aggregations and subsetting as described in Section IV. Next, for each expected variable and dataset we call the SDQuery engine with the given query. We modified the SDQuery engine to return the variable metadata, and for each returned value its coordinates – data that was missing in the original SDQuery engine. The results of each SDQuery’s query are unionized and aggregated into an intermediate resultset file. This file holds for each output variable its metadata, its source files, and the result set in a coordinate format: \(\{(x, y, z, value)\}\).

There is one file for each queried variable, this file includes the data from all the datasets and each file can be built in parallel. After the intermediate files are generated, we use the requested output format to decide if and how to aggregate the data.

The aggregations over dimensions are performed at the SDQuery level, which creates a unified format for each variable, yet additional aggregations might be needed in the case of joins or SET operators. The resulting dataset is sent to the user using the GridFTP standard parallel file sending mechanism.

V. EVALUATION

We evaluate DDSQuery DSI’s overall performance. Our emphasis is on testing how well the system “scales”.

We use two simple queries: The first query is `SELECT TEMP FROM TEMP WHERE TEMP ≤ X` – we change the value of \(X\) to vary the selectivity. The second query is `SELECT TEMP FROM TEMP WHERE depth ≤ 10.0` and longitude > 250.0. Here, the values of the depth and the longitude change based on the datasets, to maintain a small selectivity of 0.5%. A set of NetCDF files, each containing 500 MB of data from which the queried variable is 100 MB, was used for our experiments.

Although we only experiment on one variable spreaded over multiple files, the engine is implemented in a way that the performance of queries over multiple variables is the same. The number of files aggregated can alternatively represent the number of different variables.

All experiments were performed on 8 cores, 2.53 GHz Intel(R) Xeon(R) processors, with 12 GB of memory servers. We used The Ohio State University Computer Science and Engineering RI cluster for these experiments. The reported results are an average of three executions each performed after a system warmup. Because in none of the experiments did we witness high values of standard deviation, we do not report these.

A. Metadata Processing Overhead

In this experiment we increase the number of files in the system, the added files do not contain the queried variable. The result set size is 8.4 KB for both queries – a selectivity of 0.005%. The results are shown in Fig. 5, where only the X-axis is logarithmic. The increase in execution time is modest, and in fact, almost non-existent till the number of files becomes quite large. This shows that our system is able to process metadata efficiently.

B. Increasing Number of File and Distributed Variable

In this experiment we increase the number of files that contain the queried variable (which is equivalent to increasing the number of variables queried, as explained before). The resultset size increases linearly with the number of queried datasets. The selectivity of the first query, a “by value” query, is 20%. Although not all the content of the variable is returned, all the dimensions are created enforcing the output variable size to be the same as of the queried variable. For the second query, a “by dimensions” query, the selectivity is low, i.e., 0.1%. The output size of the first query is 100 MB per input dataset. The second query outputs 557 KB per input dataset.

In Fig. 6, we show the effect of increasing the number of files on the performance, while in Fig. 7 we present the matching resultset sizes. Both axes presented are logarithmic. The processing time is linear in the number of files touched, as expected. Surprisingly, the duration
of the query processing is unrelated to the query selectivity – both queries take nearly the same time, yet the dataset output size is different by scales: the duration of the NetCDF interface calls is nearly the same in both scenarios, DSDQuery engine processing duration is negligible compared to it.

C. Impact of Selectivity

Here, we experiment with varying selectivities for dimension based queries and for value based queries.

Using a series of queries where dimensional ranges are varied to control the selectivity, we measure the output file size and the query execution performance. We show in Fig. 8 the performance of “by dimension” queries in a logarithmic scaled graphs. The output sizes are linear in both selectivity and the number of files. The query duration increases linearly within each selectivity, as well as the difference between each selectivity is linear: The query duration for 0.1% is about 10 times faster than for 1% etc.

When a query subsetting is by value, the output resultset sourced in different datasets can have different dimensions. As mentioned before, the variable size in each of our files is 100 MB. When selecting 1%, about 1 MB of data can be expected – however, it turns out to be more complicated, because of the coordinates that are added due to dimensions remap. For example, when selecting two values that are at the coordinates (0,0), (1,1), the coordinates (0,1), (1,0) have to be created as well (the value for these will be NULL).

<table>
<thead>
<tr>
<th>S</th>
<th>F</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1%</td>
<td>(0.02%)</td>
<td>20K</td>
<td>701K</td>
<td>12M</td>
<td>83M</td>
<td>600M</td>
</tr>
<tr>
<td>1%</td>
<td>(0.2%)</td>
<td>17M</td>
<td>123M</td>
<td>373M</td>
<td>797M</td>
<td>1.6G</td>
</tr>
<tr>
<td>10%</td>
<td>(2%)</td>
<td>100M</td>
<td>200M</td>
<td>400M</td>
<td>800M</td>
<td>1.6G</td>
</tr>
</tbody>
</table>

TABLE II: Size of file based on selectivity (S) and number of files (F) for value based query

In Table II we show the output dataset sizes based on the number of datasets being queried and the selectivity. We write the selectivity of the values from the variable while in parenthesis we write the selectivity from the whole dataset file. For selectivity of 10%, although most values are not selected, a full dimensional map is created, resulting in a file size of 100MB. The same happens for selectivity of 1% when 16 files are touched. The content of each dataset affects on how fast the resultset size increases: if the queried values are at different coordinates in each dataset, the resultset size will increase faster than if the values are condensed at the same coordinate area.

In Fig. 9 we show the growth in execution time of different selectivities is linear, and is nearly the same for all. Although the higher the selectivity the more values need to be written to the output variable, the time difference is negligible thanks to buffering of the writings which limits us to one NetCDF library call (which is very expensive) in both cases. Overall, this shows that the system can handle growing output sizes efficiently.
VI. RELATED WORK

The work presented here provides an alternative to the vast body of work on array databases [3], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18]. These systems require that the data be ingested by a central system before it could be queried. DSDQuery DSI is intended to provide basic querying support without having to load the data inside a traditional system. It is suitable for read-only or append-only data that is queried relatively infrequently.

Prior to our predecessor work on SDQuery, we had supported similar functionality on flat-files containing array data [5]. More recently, NoDB [6] systems allow querying over raw data files, though focus is on record-based data. Adaptive query engine [7] is an extension of NoDB. The key novel aspect of our work is the focus on a collection of files in the scientific datasets domain.

Data dissemination portals, such as ESGF [22], provide limited, either basic or pre-specified and hard-coded, subsetting query options. In comparison, our system allows a variety of queries through combination of structured operators, while transporting the data. OPenDAP [23] provides ability to share and access remote data. OPenDAP does not provide any querying capabilities directly – and in fact, requires data in different formats to be converted to its internal format. UFI [24] is a tool that allows viewing a semi-structured local file as if it was a database – including files in NetCDF and HDF5 formats. However, it does not directly support a high-level query language nor distributed data.

VII. CONCLUSIONS AND FUTURE WORK

We have presented a system that can process structured queries on scientific data repositories. In the process, we have addressed several challenges, including 1) extending the relational algebra to apply on array datasets, 2) locating files for processing a user query, 3) querying over multiple array datasets, and 4) producing an output dataset that contains an aggregated union of the user requested data. Our experimental evaluation demonstrates that our system can scale well while handling increasing number of datasets, amount of data queried, and amount of data output.

In the future, we will make DSDQuery a distributed system – multiple repositories will communicate with each other. In addition we intend to provide the ability to apply aggregations and joins over results from all distributed repositories. Some optimizations can be performed on top of our current work. In particular, we will research the ability to reduce memory and disk space while processing large aggregations and unions, and parallelization options (via distribution and SIMD).

Acknowledgements

This work was supported by DOE Office of Science, Advanced Scientific Computing Research, under award number DE-DC0012495, program manager Lucy Nowell, and by NSF under the award ACI-1339757.