FDQ: Advance Analytics over Real Scientific Array Datasets

Roee Ebenstein
Department of Computer Science and Engineering
The Ohio State University
Columbus, Ohio
ebenstein.2@osu.edu

Gagan Agrawal
Department of Computer Science and Engineering
The Ohio State University
Columbus, Ohio
agrawal.28@osu.edu

Jiali Wang
Environmental Science Division
Argonne National Laboratory
Lemont, Illinois
jialiwang@anl.gov

Joshua Boley
Department of Computer Science
Illinois Institute of Technology
Chicago, Illinois
jboley1@hawk.iit.edu

Rajkumar Kettimuthu
Mathematics and Computer Science Division
Argonne National Laboratory
Lemont, Illinois
kettimut@anl.gov

Abstract—**Scientific data is not only rapidly increasing in size, but in complexity of operations performed upon as well. Compared to the prevalent use of ad-hoc approaches, structured operators provide many benefits. In this paper, we introduce FDQ - an Analytical Functions Distributed Querying Engine intended for Array Data. Motivated by needs of climate scientists in terms of both functionality and scalability, we make three major contributions: First, we introduce a new class of analytical querying – querying over windows where the planes that construct these windows are internally ordered. An example of this querying type is the introduced `MINUS` analytical function, a function that supports querying over accumulative measurements with data resets. Second, we describe in detail memory management optimizations for efficient processing of analytical (and other structured operators) querying over large datasets. Last, we provide efficient methods to execute these queries in parallel, using a sectioned (tiled) approach.**

We evaluate our methods using real multi-dimensional climate datasets, and show they outperform existing approaches. When running locally (not in a distributed manner), we observed an average performance improvement of 538% compared to other engines for analytical calculations. We also show our methods performance improve linearly with the provided computing resources (scale up and out).

I. Introduction

Scientific researchers generate data by using advanced (sensors or numerical computer simulations) tools and typically store it as multidimensional arrays. The analysis of such data is conducted by either manually written scripts or programs, or Array Database Management Systems (DBMS) such as SciDB [6] or ArrayDB [25]. Although DBMS that support scientific array data have been available for a while, scientists have reservations in using them, since they do not support all the required needs and/or have high initial data loading costs.

An example of a query that cannot be addressed by current systems is *"calculate the daily median of the differences between corresponding measurements (same time) of current and previous days"*. For example, if three samples are collected a day, at 10 am, 5 pm, and 10 pm, and we have 2 days of data – day 1 with [85, 75, 60] and day 2 with [80, 75, 50], the subtraction of day 2 from day 1 produces [-5, 0, -10]. The reported median in this case should be -5. This calculation cannot be phrased using one query (without the usage of "sub-queries" and joins) over array data, mainly since the notion of *matching samples* is lacking in current querying languages such as the Structured Querying Language (SQL) [26] (including SQL/MDA, an in development SQL extension for Multi-Dimensional Array data).

In this work we provide the ability to generically phrase and provide such functionality efficiently. This work builds on top of our earlier work, where we had developed structured query support on top of scientific data [13], [14]. We had focus on querying over native array storage formats (e.g. NetCDF), avoiding the costs other systems require (data duplication or reformatting). We use SQL as the interface to our querying systems – some semantical adjustments to the operators were made for fitting the array data context.

A. Motivation

This work builds on top of our earlier work and is driven by our collaboration with the climate research group at Argonne National Laboratory (ANL). Climate data generated by the climate research group at Argonne National Laboratory (ANL) [41], [48], [47], [40] has been analyzed by using manually written scripts and applications. As in other areas, the climate researcher analysis process often begins with a question, such as *"Why did X happen?"*, which leads to a group of variables that may be involved in X. These variables, together with the scientist intuition, produce hypotheses that lead to research questions. Once they are answered, the researcher is either satisfied, and can conclude and publish their results, or other questions are raised.

Currently, research is conducted in this community by programming applications for each question that arises. The development time for those programs had become a bottleneck – writing a separate (set of) program(s) for each hypothesis is expensive. Also, the programs inadvertently combine data querying, data analytics, and learning, which increases the application complexity. Last, software bugs can impact the result correctness, and with large datasets and complex applications, researchers are likely to not notice them. Using a declarative query engine can help address these concerns.

We introduce the FDQ, Analytical **F**unctions **D**istributed **Q**uerying, Engine. FDQ focus is on a class of operations called *Analytical Functions*, which had not been defined or supported before for array data. In analytical queries, the user provides dimensions that are used to partition the data (similar to the SQL `GROUP BY` partitions). Analytical functions process the data in each of these partitions (which are referred to as *windows*). However, unlike the partitions created by the `GROUP BY` clause, windows in our context are ordered, and therefore, the processing engine can access values of different windows.

B. Challenges and Contributions

Multiple challenges arise in efficiently calculating analytical functions over scientific array data. First is establishing the syntax and semantics of analytical querying over array data. Second, aggregations require heavy mathematical calculations, even more so for analytics that cross window boundaries. Minimizing the calculation overheads and processing queries efficiently is a challenge. Third, generating arrays requires pre-defining dimensions. Yet, the dimensions can be determined only after we know the results. Last, processing of such data requires a lot of memory – being memory efficient requires careful design of algorithms.

Overall, to summarize the contributions of this work:

- We introduce Analytical Functions (also known as "Window Querying") over scientific array data. We provide syntax and semantics, and extend Analytical Functions to not only allow windows ordering, but to also allow ordering each window internally.
- We optimize our querying engines by using "chunked" in-memory processing of array data. We refer to these memory chunks, or tiles, as *sections* in this work. Sectioning allows batching calculations and utilize memory caches to improve performance, decreasing disk accesses.
- We introduce methods to distribute windows calculations. We present approaches to distribute the query execution, based on both *sectioning* (or tiling) and dimensional distribution approaches, while discussing the benefits of each approach over the other.

We evaluate our system, showing our new functionality is performing well and that the new memory model improves query performance. Our new memory design is shown to not only improve performance of generic queries, but to also behave better with skew data. We also show that the calculations can be distributed efficiently; there is a nearly linear performance improvement when scaling (up and out) the system.

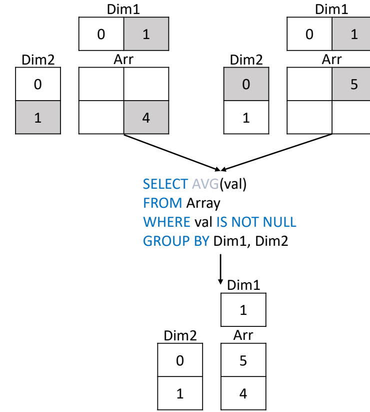

Fig. 1: Dimensions Reduction: An All `NULL` Dimensional Value (First Column) is Removed.

II. Background

A. Array Data

Scientific array data in our context is defined to be a set of multidimensional arrays, each referred to as a *variable*. Each variable is constructed from sequential values of a specific data type, e.g. `float` or a user defined structure. The dimensional values are either their indices (by the order they appear), or a value stored in another array and mapped by its *index location*. The array that contains such *dimension data* can only contain unique values and referred to as *mapped dimension*.

In particular, geophysical data, such as weather/climate numerical simulations and satellite data are often stored in files using formats such as NetCDF and HDF5 [29], [15]. Scientific data storage is made in raw files because it is accumulating over time in different locations and other formats are inefficient for its analysis, e.g. text files require parsing, relational DBMS increase the physical storage size for array data, etc. Currently, there are a few DBMS that allow querying scientific data – widely known ones are SciDB [7], MonetDB [38], and variants of Map-Reduce [12]. Yet, none provide all requested features [7], [43], [46], [17] – advanced analytics through declarative querying over distributed arrays.

B. DSDQuery

Scientific array data are often distributed over multiple files and sites, and it is desirable to support structured queries while not requiring data to be loaded into a database (or even re-formatted). DSDQuery [13] allows a query over multiple data sources appear like a query over one relation, while enabling SQL syntax to be used over scientific array data. This is done by using what DSDQuery refers to as *virtual dimensions* [13]. A common example of a virtual dimension for distributed arrays is the sampling time, which is often stored in each raw file's metadata.

DSDQuery uses multiple steps to generate its results. First, it divides each query to multiple sub-queries, each runs on a different source (raw file). Each sub-query resultset is translated to an *intermediate relation* that is stored on persistent storage. The format chosen by DSDQuery for its intermediate relation storage is the dimensional form, e.g. *[dim1, dim2, value]*, due to expected queries with low selectivity values. After all intermediate results are accessible, DSDQuery scans the results twice – once for generating condensed dimensions (where all values are used, for minimizing the resultset size), and once for populating the output variable; this process is referred to as *dimension reduction*. Figure 1 demonstrates the dimensions reduction process. Last, DSDQuery converts the resultset to the user requested format (the default is a NetCDF file). Since frameworks used to store scientific arrays do not allow to re-shape arrays after creation, it is necessary to extract dimensional values first, before populating the resultset.

III. Domain

A. Climate Datasets and Variables

The datasets generated by ANL's Climate Research Group are saved in NetCDF [29] files, each represents a specific epoch the file was sampled at or calculated for. Each file contains about 100 variables, each of which is structured from up to 4 dimensions (longitude, latitude, vertical levels, and time). In the datasets we have used, the time dimension has only 1 value, since the simulations generating the datasets are configured to output one time step [44], [33], [35] for each file. Last, not all variables contain vertical levels.

Each variable used can be large – in our case, the spatial resolution was 12 kilometers, and the model covers most of North America. The data used contain in total 2TB per year with a retention of 10 years – a total of 20TB. Each raw file is self-describing and contains the used dimensions, including their values. Usually, all the variables in the same raw file use the same dimensions definition – e.g., the precipitation and temperature variables within each file will often describe the same area.

The two variables we use for most of the queries in this work are temperature and precipitation, simulated by a regional climate model [41], [48], [47], [40]. Temperature outputs are stored as instantaneous absolute values, as measured in observatory stations or reported by weather forecast. Precipitation amounts are output in an *accumulated mode* – data is accumulated from the first epoch until last one, and reported as the value accumulated up to the sampling point. However, to avoid reporting a large number as the precipitation amount, the program (or operators) *reset* the precipitation amount at some time point and let the model output accumulated precipitation again from that time point. In calculating the amount of precipitation during a certain time-interval, if there is no reset, we simply subtract the largest reading from the smaller one. On the other hand, for those data sequences where reset has been applied, special processing is required. For example, for 5 samples with the values: 3, 5, 2, 2, and 4, the amount of rain between the first and last samples is 6 – 2 from sample 1 to 2, 2 from sample 2 to 3, which included a reset of the "counter", none between samples 3 to 4, and 2 more between the last two samples. Usually the counter reset is not often, leading to the ability to detect when a counter reset occurred.

B. Querying Needs and Tools

Currently, the workflow at ANL and their research communities includes manually extracting the variables and storing them locally by using tools such as NCO [45] and CDO [32]. The extracted data is used for simple mathematical and algebraic manipulations or loaded to a programming environment. The most common local data calculated are daily mean, or seasonal/annual averages. More advanced analysis of the data is conducted by scripts written and executed using NCL [21], Matlab [37], [3], Python [19], [20], or R [10], [30].

When it comes to collaborations with, or assistance provided to, other groups and other institutions to address different problems, transporting the larger than 20TB of generated raw data is tedious and most importantly, unnecessary. ANL offers pre-processing analytics to decrease the provided dataset size and address the needs of collaborators and colleagues better. Lowering the overhead on ANL's personnel in addressing these needs is important, and FDQ, in part, allows ANL to address these generic inquiries over long periods of times efficiently, while providing results faster.

IV. Analytical/Window Querying

A. Definition

Analytical Queries are queries over data partitions, each of which is referred to as a *window*. While analytical queries have some similarities to queries involving aggregations over partitions (the `GROUP BY` clause), Analytical Queries are unique in allowing access to other window's raw data while processing a specific window. For example, subtracting all days average values from their previous day cannot be generated directly by using the `GROUP BY` clause (this would require an inefficient use of sub-queries and joins).

Analytical Functions are usually listed in the `SELECT` clause of the query. As a consequence, Analytical Querying enables a single query to use multiple Analytical Functions – providing the ability to use multiple aggregations over different partitioning scheme within one query. The syntax for Analytical Functions is:

```
FUNC = FUNCTION(param_list) OVER
       ([PARTITION BY part_dim_list]
        [INTERNAL ORDER BY part_int_ord[INCOMPLETE]]
        ORDER BY part_ord)
```

`FUNC` is the syntax clause for analytical functions over array data, and is intended to appear instead of a column (or dimension) where *column_list* are used in the SQL standard [26]. Brackets mean the clause within it is optional. `FUNCTION` is the analytical function used – in our setting the available functions are average `AVG`, minimum `MIN`, maximum `MAX`, lead `LEAD`, lag `LAG`, median `MEDIAN`, and minus `MINUS`.

```
SELECT AVG (TEMP.val)
        OVER (PARTITION BY DAY_OF_YEAR(TEMP.date)
              ORDER BY DAY_OF_YEAR(TEMP.date))
        AS day_avg,
        LAG (day_avg, 1)
        OVER (ORDER BY DAY_OF_YEAR(TEMP.date))
        AS day_before_avg,
        day_avg - day_before_avg AS average_difference
FROM TEMP
```

Fig. 2: Analytical Query Example

The function parameters change based on the function, for example, the `MINUS` function takes 2 parameters, the variable upon it should operate, and the distance to the anchor window (the window which its values are used for the subtraction). The `OVER` clause marks that we are using an analytical function. The `PARTITION BY...` clause accepts a list of functions or dimensions (parallel to columns in relational settings) that are used to build the partitions upon the calculations occur. By using a list of functions or dimensions, the `ORDER BY` clause determines the external windows order – only dimensions or functions that are listed in the `PARTITION BY` clause can be used. Up to here, although discussed in the context of array data, similar functionality exists for relational systems and standardized under the SQL and SQL/MDA standards.

Next, we discuss the extension we suggest. The `INTERNAL ORDER BY` is provided a list of functions or dimensions as well, but here only dimensions that are not listed in the `PARTITION BY` section can be used. This clause is used to determine the order of planes internal to the partition, as if there was a secondary partitioning, internal to the generated window, based on the provided list of dimensions. The `INCOMPLETE` clause instructs the engine to provide (and not dismiss, as the default `COMPLETE` behavior is) window's values that are calculated based on missing values (providing an advanced and nuanced handling for the case `NULL` values are addressed).

In Figure 2 we show an example of an analytical query for finding the differences between subsequent day's average temperatures. The syntax shown was simplified compared to the ANSI standard for simplicity and brevity. In the query there are three different parts: the first shows a call to the analytical function `Average`: the average is calculated over a window that corresponds to each day's data (the window is built through a function that extracts the day of year); the `ORDER BY` clause does not impact the results and can be omitted here. Next, we use the function `LAG`, which allows us access to data that was produced for the *previous window*, as determined by the provided `ORDER BY` clause which is critical here. The function `LEAD`, demonstrated in Figure 5, gives access to the subsequent window values. Last, we calculate the difference between the currently calculated day and the day before, showing the difference in temperatures across subsequent days.

Analytical Queries challenge the query optimization process since they require repeating calculations, large memory caches, and in some cases disk usage. Efficient algorithms have been developed to address these issues

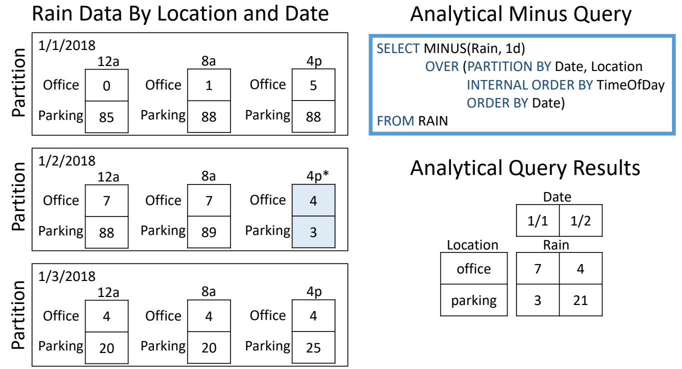

Fig. 3: Analytical Query Example: Demonstration of expected behavior of the `MINUS` analytical function with data reset (highlighted)

for relational databases [23], [4], [9], [18], [49], [42]. Yet, as to date, no engine for array data provides the full strength of analytical functions, therefore developing new methods, which utilize the sequential memory layout, for efficient processing of these calculations in that context is crucial.

B. Minus Function

We now introduce a new analytical function, `MINUS`, which is similar to what we described in the previous subsection. This function calculates the difference between the maximum value of each data point within each window, to the last value of that matching data point within the previous window. Mathematically, it can be described as

$$MAX2(val) - LAG(val) \tag{1}$$

where `MAX2` is calculating the maximum in a manner that compensates for data resets. This analytical function cannot be phrased in SQL without the `INTERNAL ORDER BY` clause, which was introduced here.

In Figure 3 we demonstrate the minus function over precipitation data. Data was collected from 2 locations during 3 days while the array contains 3 dimensions: day, time, and location. The query `PARTITION BY` clause contains only 2 out of the 3 dimensions – the 3rd dimension is aggregated upon. The `INTERNAL ORDER BY` clause lets the `MINUS` Analytical Function know the internal partitioning ordering (for determining the "matching value" to subtract). The traditional `ORDER BY` clause is used here as well, allowing the aggregation function access the previous (or next) partition values. We highlighted in the figure where data reset occurred (and marked the specific sample with an asterix).

In the figure, since the measuring begins at midnight, it should end at the next day midnight – although we have 3 days of data, we are missing a data point to calculate the full last day. For clarity, we demonstrate the calculation of the parking location for January 2nd:

$$(89 - 88) + 3 + (20 - 3) = 21 \tag{2}$$

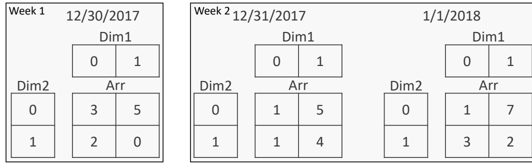

Fig. 4: A 3 dimensional variable

```
SELECT AVG(TEMP) – LEAD(AVG(TEMP))
        OVER (PARTITION BY extract_week(Time)
              ORDER BY extract_week(Time))
FROM    TEMP
```

Fig. 5: Query 1 - Analytical Function Query Example

V. Query Evaluation and Optimization Algorithms

A. Analytical Functions Implementations

Our discussion here considers three analytical functions that together cover all the different needs that arise in implementing any analytical function: `MIN` – Minimum, `AVG` – Average, and `MINUS` – Minus.

Before discussing the algorithms, we explain some of the issues that arise. Figure 4 shows one variable with three dimensions – Dim1, Dim2, and date (presented by its value above each partial array). Running the query in Figure 5 should produce results similar to those in Figure 6. The first question to address is should "Week 2" be produced at all? The `INCOMPLETE` clause allows fine grained control over this matter.

Had we decided not to issue the second window, the whole `Time` dimension should have been removed. Dimensional removals and additions introduce a challenge since, as mentioned before, current frameworks do not allow to modify dimensional sizes and variable definitions after the initial creation – enforcing two scans of the results in some cases.

Next, we describe the implementation of the three functions we are focusing on. Analytical functions require multiple data passes, while simultaneously performing additional calculations. Since many calculations are repeated it makes sense to pre-allocate memory in a way that allows caching the expected re-used values. For example, for *LAG* (a function to access previously produced values at a specific window offset), we can hold for every currently processed window the previous values in memory, allowing fast access to those data, without disk accesses or values recalculations.

However, memory is limited and this approach might require more memory than available. We limit the memory allocation to the available size, and work *section-by-section*, i.e., iteratively produce full output for *n* values at a time. Two options arise – to fill (calculate all values) each window before calculating the values of the next one, or to calculate a limited set of values for all the windows and iteratively do so, until all sections, or tiles, are calculated. The determination of which method should be used is based on the function used, for example `LAG` should use the latter approach.

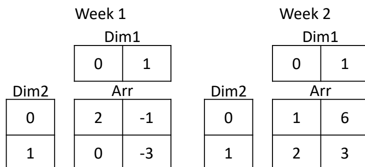

Fig. 6: Query 1 resultset

The Minimum function implementation is straightforward. Each value is initialized to *NULL*, the empty value, while a data scan updates values as necessary.

The average function implementation uses two arrays, one for a values counter and one for the data – both reset to 0. Every non-*NULL* value we process increase the corresponding counter by 1, while adding the value to the data array. After all the data for the currently processed section has been scanned, we divide the second array by the first one (a 'by place' division), and process the subsequent windows (after resetting the arrays values back to 0).

1) Implementation of Minus: The `MINUS` function was described in Section IV-B. A significant complexity of the implementation comes due to the *data counter resets*. This can be addressed by using three pre-allocated section caches – the previous values, the current maximums, and a careful maintenance of the current output (which is a temporary summation of all the reset values).

In Algorithm 1 we show the implementation of the `MINUS` analytical function in great detail, including memory optimization and proper handling of counter resets. First, we iterate over the sections of the relevant variable. We populate each coordinate with its appropriate value (based on the presented calculations). After each section is populated, the calculated values are cached for the next calculation to take place. For brevity, we assume that once a non-`NULL` value was assigned to a coordinate, at each section, at least one non-`NULL` value will appear for this coordinate again later.

As can be seen, there is an additional optimization – we maintain calculated sub-results in place. When a function requires access to the previous window, like the function `MINUS` does, we alternate pointers to where memory is allocated, switching the reading and writing address, instead of moving values to different memory areas. This results in a more complicated memory addressing mechanism, but it minimizes disk accesses while decreasing in-memory data movement as well.

We use multiple functions for handling the calculated results: `CopyToOutput`, `SwapPointers`, and `SendOutputOfSection`. Although for simplicity these are shown as independent function calls, in the implementation we delicately calculate where we should write to, and modify pointer addresses. These allow us to avoid unnecessary data access and movement.

B. Optimization: Dimensional Restructuring

Dimensional restructuring is an expensive phase in the scientific query execution process. This process entails removing dimensional values that all its matching variable values are empty (`NULL`), decreasing the resultset size.

Algorithm 1 Minus Analytical Function

```
1: procedure MINUS()
2:     curr ← ALLOCATESECTIONMEMORY
3:     out ← ALLOCATESECTIONMEMORY
4:     lastMax ← ALLOCATESECTIONMEMORY
5:     previousWindow ← ALLOCATESECTIONMEMORY
6:     RESETTONULL(lastMax)
7:     RESETTONULL(previousWindow)
8:     for each section s do
9:         for each ordered window w in s do
10:            RESETTONULL(curr)
11:            RESETTONULL(lastMax)
12:            for each raw data source r in w do
13:                for each coordinate c in r do
14:                    sc ← MAPTOSECTIONCOORDINATE(s, c)
15:                    DoAddition ← False
16:                    HadReset ← False
17:                    if r[c] == NULL then
18:                        continue
19:                    end if
20:                    if curr[sc] == NULL then
21:                        curr[sc] ← r[c]
22:                        lastMax[sc] ← r[c]
23:                        if previousWindow[sc] != NULL &&
24:                          curr[sc] < previousWindow[sc] then
25:                            HadReset ← True
26:                        end if
27:                    else
28:                        if lastMax[sc] == NULL then
29:                            curr[sc] ← r[c]
30:                        else
31:                            if lastMax[sc] ≤ r[c] then
32:                                if DoAddition then
33:                                    curr[sc] ← curr[sc] + r[c]
34:                                                − lastMax[sc]
35:                                else
36:                                    curr[sc] ← curr[sc]
37:                                end if
38:                            else
39:                                DoAddition ← True
40:                                curr[sc] ← curr[sc] + r[c]
41:                            end if
42:                            lastMax[sc] ← r[c]
43:                        end if
44:                    end if
45:                    if !HadReset &&
46:                      previousWindow[sc] != NULL then
47:                        out[sc] ← curr[sc] -
48:                                  previousWindow[sc]
49:                    else
50:                        out[sc] ← curr[sc]
51:                    end if
52:                end for
53:            end for
54:            COPYTOOUTPUT(curr, s, out)
55:            SWAPPOINTERS(curr, lastMax)
56:        end for
57:        SENDOUTPUTOFSECTION(out, s)
58:    end for
59: end procedure
```

Since this process requires not only a scan of the data, but also copying and remapping all existing values to new locations, it is an extremely expensive one.

The three approaches to restructure dimensions are:

- **A la carte –** Dimensions are built on the go. Once a new dimensional value is detected, it is added.
- **Post-building –** Execute sub-queries, accumulate results, and after all results are retrieved build the dimensions and the array.
- **Pre-building –** Build the dimensional values, and afterwards fill the data; requires executing the queries twice, or caching some queries results.

For the first approach listed above, multiple implementations are feasible. Since this process requires to completely rebuild the variable every time a dimensional value is added, we consider this option as too expensive. In addition, since the array changes its size while it is formed, memory pre-allocation cannot be easily used. The usage of current array data interfaces (NetCDF, for example) require the dimensional setting to be known before an array can be addressed, nullifying this approach as a viable option.

Post-building is the approach used in DSDQuery [13]. In this approach, each sub-query runs once, while caching (to disk, due to the potential size) the results in an intermediate format. Then, the cached results are scanned twice – the first scan is used for building and configuring the dimensions. The second data scan is used to populate the created variable. This approach does not fit well to queries with large selectivities – for example, DSDQuery failed in executing a simple aggregation query due to the need to store more than 200GB of intermediate results to a disk that is smaller than that, for generating a 2MB output.

The last approach, pre-building, includes a two step process as well. We first run queries to extract and build the dimensions (in a streamed manner). Then, we run data filling queries, over the source data, to assign the results to their place. This method enjoys the benefits of the previous approach, yet does not store intermediate results. We experienced a performance boost of up to 4,000% using this approach. This method works well mainly for condensed data, where executing the query twice is more beneficial than caching results to disk – the trade-off between the two last presented methods.

Working with climate data resulted in another optimization that is often valid for scientific data. If the data contains thousands of files, but all source from a specific simulation method or set of sensors, only one file of each source is needed to be used for building the dimensions (since simulations/data sources produce the same dimensional output).

C. Distributed Calculations

The calculations of the resultset can be distributed over two orthogonal options: sections and dimensions. In the first, section-based distribution, each section can be calculated by a different process (either locally or remotely). This results in each process accessing all input data sources, yet reading only smaller portions of it.

The performance with this approach depends on the properties of the distributed storage system.

The other option is distribution by dimensions. In this option, each process accesses a set of dimensions from each relevant file. The advantage of using dimensions over sections comes due to virtual dimensions. Since virtual dimensions are stored in separate files, if the distribution is based on a virtual dimension, two processes rarely access the same file. Thus, this method allows geographical files distribution, and optimizes file access. However, certain analytical functions need access to previous windows, and it is possible that two processes will need to access the same file.

Another aspect is the output file generation, which can be done using two different options: writing the output file in a distributed manner, or adding a process to accumulate results and generate the output file locally. We experimented over both options, and concluded that the differences in performance are marginal. In settings where the storage system is well connected, like in ours, both approaches produce similar performance. In settings where the network is fast but storage is slow, adding a process near the end user that generates the final resultsets can be preferable. We chose to use the second approach, since it works well in both settings. The added process for writing the results will be referred to as the *management process* and is discussed next.

D. Results Accumulation

We designate a process to accumulate the resultsets. In Algorithm 1 we had shown how we transport results from the calculation nodes to the management process in units of sections (line 57). Next, for completeness, we describe the management process.

Algorithm 2 Accumulate Results

```
 1: procedure AccumulateResults()
 2:     curr ← allocateSectionMemory
 3:     output ← InitializeOutput
 4:     for each node n do
 5:         if WindowCanBeCopied then
 6:             address ← CalculateOffset(n, 0, 0)
 7:             address += output
 8:             GetResultsFromNode(n, address)
 9:         else
10:             GetResultsFromNode(n, input)
11:             for each section s do
12:                 for each window w do
13:                     address ← CalculateOffset(n, s, w)
14:                     address ← output + address
15:                     CopyWindow(address, input, w)
16:                 end for
17:             end for
18:         end if
19:     end for
20: end procedure
```

In Algorithm 2 we show how results are accumulated, taking sectioning into account. While when building the results we can remain agnostic to the final output layout, when accumulating results we may not. In the algorithm, we differ between two scenarios. The first is where each window is sequential and the memory contains the fully calculated window. In this scenario, we may just copy the input to the output. The more complicated scenario is when the output sent from the calculating node is not sequential (sections are used). In this case, we need to copy section-by-section and window-by-window, reducing the efficiency. As mentioned before, this use case is rare due to how scientific data is usually held, yet, for completeness, it must be addressed.

VI. System Implementation

A. FDQ Engine

We have developed two different versions of FDQ - a non-distributed version and a distributed one. The non-distributed version of FDQ is based on DSDQuery DSI [13] and is currently in use by scientists at ANL. The code base is implemented as a DSI plugin to Globus GridFTP [1], [11].

The distributed version uses MPI [16] for communication. For improving performances, we decrease the number of transported messages by calculating multiple sections and sending all of these to the management process together, within larger, but fewer, messages.

The implementation of both engines is in C++. The current system database is implemented using SQLITE [27]. Although it is not the most efficient relational system, it was chosen because it is the easiest to maintain and access, especially when user permissions are limited. This database is used for internal engine metadata queries, such as which variables are stored and where, as explained in DSDQuery [13].

B. ANL Web Portal

A web portal has been implemented at ANL to facilitate scientists' interaction with their data visually. In Figure 7, we show a screen-shot showing some of the provided functionality. This web portal is hosted on a Laboratory Computing Resource Center (LCRC)[1] web application server. The FDQ engine is run from a location accessible to this portal.

The portal page presents scientists with the relevant search parameter values and models that can be queried, and then maps user input back to the application server into an FDQ query. An extended SQL syntax that includes support for Analytical Window Querying is used. The web application uses `globus-url-copy`[2] (an open source GridFTP client) to submit the query to the FDQ-enabled server and retrieves the resulted NetCDF file. The NetCDF file is made available on an anonymous read-only GridFTP server. It can then be downloaded by the user using the Globus transfer service [2]. We chose this architecture since it is likely a user would like to download the results to their local environment or that different users would run the same query – this architecture eases the implementation for both.

[1] http://www.lcrc.anl.gov

[2] http://toolkit.globus.org/toolkit/docs/latest-stable/gridftp/user/

NRCM

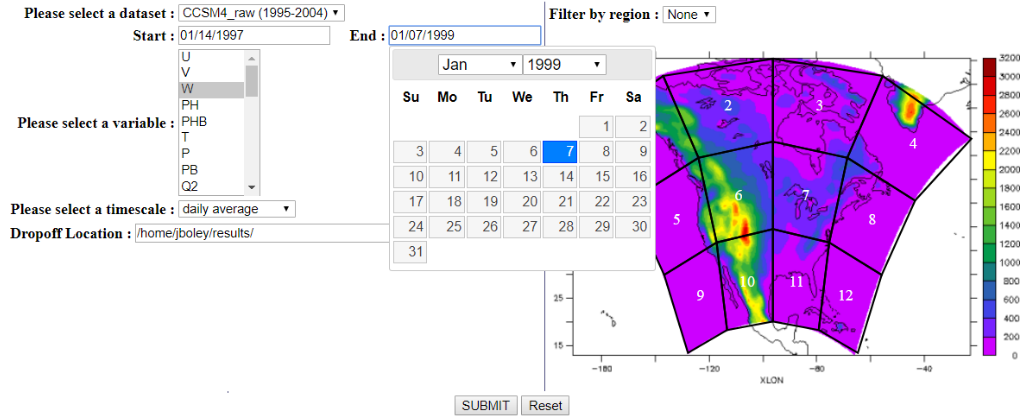

Fig. 7: ANL Querying Portal, backed by FDQ

VII. Evaluation

In this section, we evaluate our system. Our goal is to show the usage of FDQ is feasible with reasonable run times, is scalable, and that it outperforms earlier implementations (for supported queries). We consider both *group-by* queries and analytical queries, using both non-distributed and distributed versions of our engine.

With exception to experiments involving comparison with SciDB, experiments were executed on the Blues cluster at ANL. The cluster is constructed from about 350 nodes, each has 64GB memory, and uses Intel Xeon E5-2670 2.6GHz processors. All the data used for these experiments is real climate data, generated by ANL. The datasets comprise temperature and precipitation variables, Section III, have 2 common dimensions (latitude and longitude) and 1 virtual dimension (sampling time), and contain data for North America. Each day usually consists of 8 samples, distributed among different files. The daily precipitation data (not including any metadata) is 24MB, while the daily temperature data is 704MB (the latter has an additional dimension – vertical levels). In our experiments we accessed up to 20 TB of array data, and processed up to 1TB of it.

A. Average Function Performance

In this experiment we compared the execution of the Average analytical function performance to our predecessor engine and to the community leading Scientific Array DBMS, SciDB [34]. This function does not require cross-windows access, and therefore can be executed by using the `GROUP BY` clause for partitioning. We used an array that varied in size between 100MB and 90GB.

In Figure 8 we show the queries execution time and compare it to the precursor engine, DSDQuery (DSQL). Each queried day involves about 700MB of data, and thus 128 days query processes 90GB of data. We consider two queries, one which averages 16% of all data, and another that averages 100% of the data. The baseline system crossed the timeout threshold, 15 minutes, when querying 90GB of data, which is the reason we report

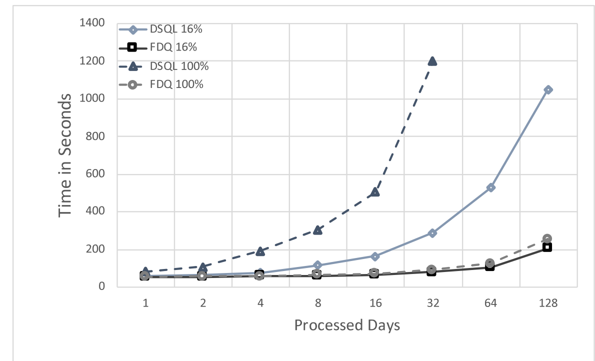

Fig. 8: Execution Time of The `AVG` Function Over Partitions With Different Selectivities (16% and 100%) [128 days - 90GB]

results for only up to 128 days (and 32 days for 100% selectivity). While FDQ is comparable for smaller datasets, when larger datasets are used FDQ performs several times faster. For example, for a query over 32 days (22GB), FDQ performs 3.5 times faster for queries with low selectivities and 13.3 times faster when querying all data. FDQ improvement increases with dataset sizes – for 128 days (90GB) the improvement of FDQ over DSDQuery increases to 5.05 times. This is sourced in the low impact different selectivities have on FDQ query execution – querying 6 times more data slowed DSQL by about 600% while FDQ was slowed by at most 22%. This is because of the more nuanced memory management that is increasing efficiency and decreasing the amount of data movement.

Next, we compare our engine to SciDB. We used a different environment for this experiment, where we could install SciDB. We used linux kernel (Debian) 4.4 on a machine with i5-4590 and 16GB of RAM. Both engines were set to use only 1 process. We used 3 3-dimensional arrays (100MB, 1GB, 10GB). It should be noted that there is a significant cost of loading data to SciDB, which we

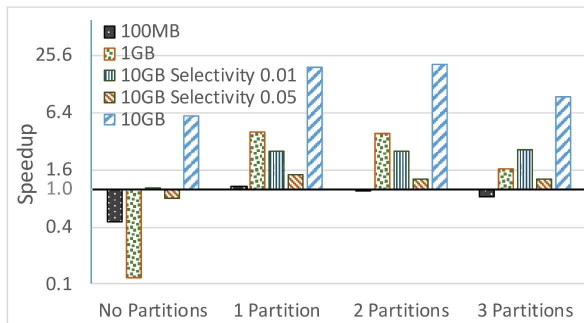

Fig. 9: Query Runtime Speedups: FDQ Over SciDB

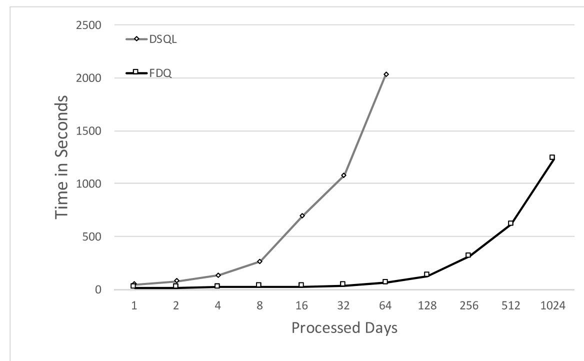

Fig. 10: Runtime of Minus Function: Benefits of Memory Optimizations in FDQ [1024 days - 24GB]

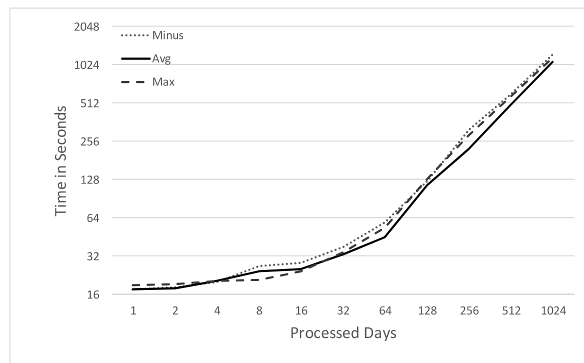

Fig. 11: Comparison of Analytical Functions Runtimes using FDQ [1024 days - 24GB]

do not account for here. Our queries calculated averages over different partitioning configurations. Specifically, the queries involve 1-partition (longitude, generate 5KB of data), 2-partitions (longitude and latitude, generate 1.8MB of data), or 3-partitions (which returns the original array). In some of the queries we used value based subsettings (`WHERE val < x`) to control selectivity – in FDQ, subsetting queries initiate the use of bitmap indexes to locate the queried data, and we wanted to measure its cost. For SciDB, we report the fastest execution out of 4 consecutive runs – this allows SciDB caches to fill correctly and to be utilized. FDQ does not use cross-query caches, therefore warm-up runs are not necessary.

In Figure 9 we present the results of the SciDB comparison. Each bar in the figure shows the speedup of FDQ compared to SciDB, while the bars are grouped based on the number of partitions used in the queries. The vertical axis, speedup, is logarithmic. The baseline of the figure is at the value 1; bars that are headed downwards signal that SciDB performed better. The larger the dataset, the better FDQ is – for array size of 100MB the execution time of both engines are similar, yet for larger arrays our speedup is between 2 to 21. When we use value based subsettings, the results are more moderate – an improvement of between 1.2 to 2.586. This is due to the dimensional restructure, the array rebuild in a more condense structure, and the generation of a NetCDF file as the query result. In comparison, SciDB emits coordinatal results that allow it to skip the restructure process and handle sparsity better, but prevents users from using their existing tools to visualize and analyze query resultsets. The last presented setting, 3 partitions, has a speedup of 9.4x for the 10GB dataset, lower than the speedup measured for the two other cases, 19.23x and 20.53x. This is because of the usage of virtual memory due to the resultset size.

The results for the "No Partition" group shows that for most datasets, when no partitioning is used SciDB, performs better, as expected. FDQ is not optimized for this case. Yet, when data does not fit in SciDB cache, it performs quite badly – a factor of 6 slowdown was observed for the 10GB array.

Overall, we can see the following: FDQ memory optimizations are desirable for array data querying, and outperform previous designs. In addition, FDQ behaves well for queries with changing selectivities and different columns order in the `PARTITION BY` clause; SciDB slows down between 2 to 4 times if the partitions used for the window generation are not in the optimized order.

B. Analytical Function Performance and Scalability

We focus on the execution of several analytical functions, including the most expensive analytical function, Minus. While the support for Minus queries was only introduced here, for comparison, we upgraded DSDQuery to support this function while using the same methods for memory access and management. No other engine provides the ability to run these queries.

In Figure 10 we show a performance comparison of DSDQuery [13] with FDQ. The X-axis scale is logarithmic, while the Y-axis is not. As can be seen, DSDQuery is slower, and its run times are not feasible for a scientific workflow – querying 16 days of data (about 384MB) takes more than 10 minutes using DSDQuery, while it takes 28 seconds using FDQ. Querying larger amounts of data, 64 days (1.5GB) takes more than 30 minutes using DSDQuery while taking only 59 seconds using FDQ.

In Figure 11 we show run times comparison amongst the most commonly used functions. The graph shows the most simple (maximum) and the most complicated (minus) analytical functions perform similarly with the same pattern, suggesting any needed analytical function will execute efficiently, within these two boundaries.

Next, we evaluated how the system scales on a machine with 8 cores. We ran a query using the Average

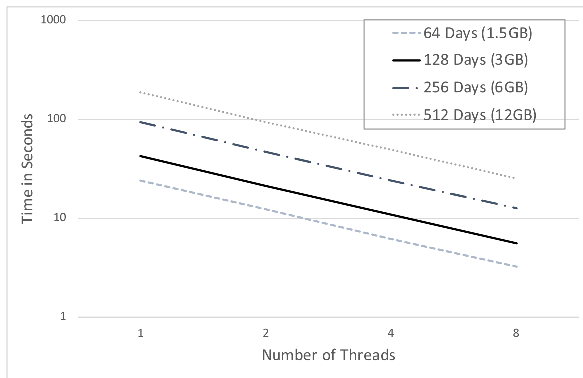

Fig. 12: Execution time of Average Function with Increasing Number of Threads

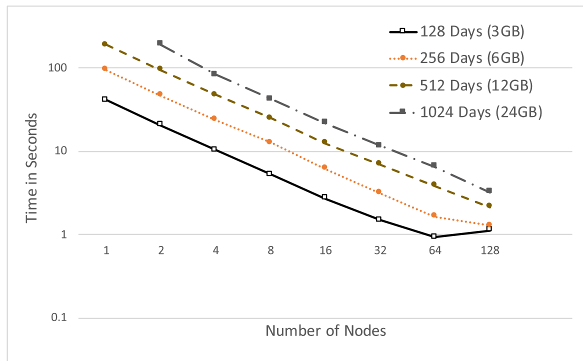

Fig. 13: Execution Time of the Average Function With Increasing Number of Nodes

analytical function over changing number of days – varying from 64 (1.5GB) to 512 (12GB). As can be seen in Figure 12, the system scales well, up to the number of cores available on the machine. The improvement is linear, as expected.

Next, we ran an average query in a fully distributed manner. We executed the same query for different number of days, calculating the daily average. All results were consistent – for example, processing of 128 days on one node took 40.99S, while processing of 256 days on one node took 81.71S; an additional example, processing of 1024 days took 6.57s using 64 nodes, while processing 256 days took 1.66s and 0.93s for 128 days on the same amount of nodes.

In Figure 13 we report the run times of a query using the Average analytical function for changing amount of days. It is noticeable that as the parallelism increases, the performance improve linearly. An interesting pattern that was uncovered is that when an extremely small number of days are processed on one node (1 day in the 128 days experiment over 128 nodes), the performance degrade. This is a result of the overhead in the communication and reduction algorithms.

In conclusion, FDQ performs and scales well. The new memory management architecture together with the optimizations described before allow scientists to use structured query operators efficiently. Calculations are distributed in a manner that allows linear improvement with the number of used nodes. A noticeable advantage is the ability to run an analytical function for large amount of data and windows in only a few seconds while processing many raw files. For example, we got to 1024 days over 64 nodes in 6.57 seconds, times that are unheard of within these communities.

VIII. Related Work

Scientific Data Management Systems: SciDB [6] is a centralized DBMS that uses database-like caches to process array queries; but it provides limited querying abilities and requires manual array storage configuration. Storage engines such as TileDB [28] and ArrayDB [25] provide different approaches for configuring array storage. MapReduce [12], [46], [17] systems use HDFS [5], or a variant, to store data and allow users to query data by either programing an application to do so or using a high-level querying framework like Hive [36]. MapReduce frameworks for processing array data, e.g. SciHadoop [8] and MERRA [31], provide native querying abilities over array data, yet, they do not provide generic declarative querying support, do not support geographically distributed data, nor provide the join operator or analytical functions support. Finally, Wanalytics [39], although addressing joins in distributed settings, does not handle window querying nor array data.

All these systems require pre-processing and data loading. Our system allows executing queries over native storage, which is desirable if data is generated at a high rate and queried infrequently.

A few querying languages have been suggested for querying array data. AQL [24] is the most referred and used one. In an attempt to standardize multi-dimensional array querying syntax, there is an effort to extend SQL [26] to address these data – referred to as SQL/MDA. Both do not consider advanced analtyical querying. In addition both implicitly assume array data is not distributed through virtual dimensions.

Analytical/window functions: Analytical functions have been available in relational system for a while [18], [23], [9]. The introduction of these functions to the scientific array DBMS is still limited. Specifically, SciDB [22] supports a limited syntax for analytical functions. It does not provide cross windows data access, but does allow a window to be defined. With the SciDB window, joining two different window queries is a possible mechanism to manually implement functionality such as `lead` and `lag`. However, because functionality is broken across different queries efficiency is limited – memory and execution optimizations cannot be used since both queries need to execute before the join can be executed.

IX. Conclusions

In this work we introduced analytical functions to the domain of scientific array data. We shown the current tools used by scientists are not sufficient and introduced new syntax, algorithms, and techniques that enable the usage of a structured querying engine to efficiently process complex analytical queries. We evaluated our work over real dataset, and shown that we allow near real-time analytical querying, outperforming other options.

Acknowledgement

This material is based upon work supported by the National Science Foundation, under grants ACI-1339757 and ACI-1339798, and the U.S. Department of Energy, Office of Science, under contract number DE- AC02-06CH11357.

References

[1] W. Allcock, J. Bresnahan, R. Kettimuthu, et al. The globus striped gridftp framework and server. In *SC*, page 54. IEEE, 2005.
[2] B. Allen, J. Bresnahan, L. Childers, et al. Software as a service for data scientists. *CACM*, 55(2):81–88, 2012.
[3] C. A. Andersson and R. Bro. The n-way toolbox for matlab. *Chemometrics and intelligent laboratory systems*, 52(1):1–4, 2000.
[4] I. Ben-Gan. *Microsoft SQL Server 2012 High-performance T-SQL Using Window Functions*. Pearson Education, 2012.
[5] D. Borthakur et al. Hdfs architecture guide. *Hadoop Apache Project*, 53, 2008.
[6] P. G. Brown. Overview of scidb: large scale array storage, processing and analysis. In *SIGMOD*, pages 963–968. ACM, 2010.
[7] P. G. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *SIGMOD*, pages 963–968, 2010.
[8] J. B. Buck, N. Watkins, et al. Scihadoop: array-based query processing in hadoop. In *SC*, pages 1–11. IEEE, 2011.
[9] Y. Cao, C.-Y. Chan, et al. Optimization of analytic window functions. *VLDB*, 5(11):1244–1255, 2012.
[10] J. Chambers. *Software for data analysis: programming with R*. Springer Science & Business Media, 2008.
[11] K. Chard, I. Foster, et al. Globus: Research data management as service and platform. In *PEARC*, pages 1–5. ACM, 2017.
[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[13] R. Ebenstein and G. Agrawal. Dsdquery dsi-querying scientific data repositories with structured operators. In *Big Data*, pages 485–492. IEEE, 2015.
[14] R. Ebenstein and G. Agrawal. Distriplan-an optimized join execution framework for geo-distributed scientific data. *SSDBMS*, 15(39):44–45, 2017.
[15] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson. An overview of the hdf5 technology suite and its applications. In *EDBT/ICDT*, AD, pages 36–47. ACM, 2011.
[16] W. Gropp, E. Lusk, et al. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.
[17] T. Gunarathne, T.-L. Wu, J. Qiu, and G. Fox. Mapreduce in the clouds for science. In *CloudCom*, pages 565–572. IEEE, 2010.
[18] A. Gupta. Method for minimizing the number of sorts required for a query block containing window functions, May 14 2002. US Patent 6,389,410.
[19] J. Helmus and S. Collis. The python arm radar toolkit (py-art), a library for working with weather radar data in the python programming language. *Journal of Open Research Software*, 4(1), 2016.
[20] S. Hoyer and J. Hamman. xarray: Nd labeled arrays and datasets in python. *Journal of Open Research Software*, 5(1), 2017.
[21] W. Huang. Using ncl to visualize and analyse of nasa/noaa satellite data in format of netcdf, hdf, hdf-eos. In *AGU Fall Meeting Abstracts*, 2014.
[22] L. Jiang, H. Kawashima, and O. Tatebe. Incremental window aggregates over array database. In *Big Data*, pages 183–188. IEEE, 2014.
[23] V. Leis, K. Kundhikanjana, A. Kemper, et al. Efficient processing of window functions in analytical sql queries. *VLDB*, 8(10):1058–1069, 2015.
[24] L. Libkin, R. Machlin, and L. Wong. A query language for multidimensional arrays: design, implementation, and optimization techniques. In *ACM SIGMOD Record*, volume 25, pages 228–239. ACM, 1996.
[25] A. P. Marathe and K. Salem. Query processing techniques for arrays. *VLDB*, 11(1):68–91, 2002.
[26] J. Melton. Iso/ansi: Database language sql. *ISO/IEC SQL Revision. New York: American National Standards Institute*, 1992.
[27] M. Owens. Embedding an sql database with sqlite. *Linux J.*, 2003(110):2, June 2003.
[28] S. Papadopoulos, K. Datta, et al. The tiledb array data storage manager. *VLDB*, 10(4):349–360, 2016.
[29] R. Rew and G. Davis. Netcdf: an interface for scientific data access. *Computer Graphics and Applications, IEEE*, 10(4):76–82, 1990.
[30] D. R. Roberts and A. Hamann. Predicting potential climate change impacts with bioclimate envelope models: a palaeoecological perspective. *Global Ecology and Biogeography*, 21(2):121–133, 2012.
[31] J. L. Schnase, D. Q. Duffy, et al. Merra analytic services: Meeting the big data challenges of climate science through cloud-enabled climate analytics-as-a-service. *Computers, Environment and Urban Systems*, 61:198–211, 2017.
[32] U. Schulzweida, L. Kornblueh, and R. Quast. Cdo user's guide. *Climate Data Operators, Version*, 1(6), 2006.
[33] M. Shaffer, B. Wylie, et al. Using climate/weather data with the nleap model to manage soil nitrogen. *Agricultural and Forest Meteorology*, 69(1-2):111–123, 1994.
[34] M. Stonebraker, P. Brown, et al. The architecture of scidb. In *SSDBM*, pages 1–16. Springer, 2011.
[35] R. Stratton. A high resolution amip integration using the hadley centre model hadam2b. *Climate Dynamics*, 15(1):9–28, 1999.
[36] A. Thusoo, J. S. Sarma, et al. Hive: a warehousing solution over a map-reduce framework. *VLDB*, 2(2):1626–1629, 2009.
[37] M. H. Trauth, R. Gebbers, N. Marwan, and E. Sillmann. *MATLAB recipes for earth sciences*. Springer, 2006.
[38] M. Vermeij, W. Quak, M. Kersten, and N. Nes. MonetDB, a novel spatial columnstore DBMS. In *FOSS4G*, pages 193–199, 2008.
[39] A. Vulimiri, C. Curino, et al. Wanalytics: Analytics for a geo-distributed data-intensive world. In *CIDR*, 2015.
[40] J. Wang and V. R. Kotamarthi. Downscaling with a nested regional climate model in near-surface fields over the contiguous united states. *JGR: Atmospheres*, 119(14):8778–8797, 2014.
[41] J. Wang and V. R. Kotamarthi. High-resolution dynamically downscaled projections of precipitation in the mid and late 21st century over north america. *Earth's Future*, 3(7):268–288, 2015.
[42] A. Witkowski, S. Bellamkonda, T. Bozkaya, N. Folkert, A. Gupta, J. Haydu, L. Sheng, and S. Subramanian. Advanced sql modeling in rdbms. *ACM Transactions on Database Systems (TODS)*, 30(1):83–121, 2005.
[43] H. Xing, S. Floratos, et al. Arraybridge: Interweaving declarative array processing in scidb with imperative hdf5-based programs. *ICDE*, 2018.
[44] C. Yavuzturk and J. D. Spitler. Comparative study of operating and control strategies for hybrid ground-source heat pump systems using a short time step simulation model. *Ashrae Transactions*, 106:192, 2000.
[45] C. Zender. Nco user's guide, 2004.
[46] H. Zhao, S. Ai, Z. Lv, and B. Li. Parallel Accessing Massive NetCDF Data Based on MapReduce. In *WISM*, pages 425–431. Springer-Verlag, 2010.
[47] Z. Zobel, J. Wang, et al. Evaluations of high-resolution dynamically downscaled ensembles over the contiguous united states. *Climate Dynamics*, pages 1–22, 2017.
[48] Z. Zobel, J. Wang, et al. High-resolution dynamical downscaling ensemble projections of future extreme temperature distributions for the united states. *Earth's Future*, 2017.
[49] C. Zuzarte, H. Pirahesh, et al. Winmagic: Subquery elimination using window aggregation. In *SIGMOD*, pages 652–656. ACM, 2003.