

# SAGA: Array Storage as a DB with Support for Structural Aggregations

Yi Wang Arnab Nandi Gagan Agrawal  
Department of Computer Science and Engineering  
The Ohio State University Columbus OH 43210  
{wayi,arnab,agrawal}@cse.ohio-state.edu

## ABSTRACT

In recent years, many Array DBMSs, including SciDB and RasDaMan have emerged to meet the needs of data management applications where the natural structures are the arrays. These systems, like their relational counterparts, involve an expensive *data ingestion* phase. The paradigm of using native storage as a DB and providing database-like support (e.g., the NoDB approach) has recently been shown to be an effective approach for dealing with infrequently queried data, where data ingestion costs cannot be justified, though only in context of relational data.

Applications that generate massive arrays, such as the scientific simulations, often store the data in one of a small number of array storage formats, like NetCDF or HDF5. Thus, a natural question is, “*can database-like functionality be supported over native array storage?*”. In this paper, we present algorithms, different partitioning strategies, and an analytical model for supporting structural (*grid, sliding, hierarchical, and circular*) aggregations over native array storage, and describe implementation of this approach in a system we refer to as Structural AGgregations over Array storage (SAGA). We show how the relative performance of different partitioning strategies changes with varying amount of computation in the aggregation function and different levels of data skew, and our model is effective in choosing the best partitioning strategy. Performance comparison with SciDB shows that despite working on native array storage, the aggregation costs with our system are lower. Finally, we also show that our structural aggregation implementations achieve high parallel efficiency.

## Keywords

Scientific Databases, Array Databases, Structural Aggregations

## 1. INTRODUCTION

Large-scale arrays of different sizes and dimensions are used for storing images, sensor data, simulation outputs, and statistical data in a variety of scientific domains, including earth sciences, space sciences, life sciences, and social sciences [20]. With rapid growth in dataset sizes in each of these areas, effectively managing, querying, and processing such array data is one of the key ‘big-data’

challenges today. Many studies have observed the intrinsic mismatch between the array model and relational table view [14, 7, 13, 17, 11, 39, 8, 16, 51], and thus, it is well understood that the relational model (and the systems implementing this model, i.e., the relational databases) cannot be used (efficiently) for addressing this challenge.

In recent years, many Array DBMSs, including SciDB [8] and RasDaMan [7] have been designed to address this mismatch. As their names suggest, the key feature of these systems is that arrays, and not the relational tables, are the first-class citizens. Various complex operations desired in scientific applications, including correlations, curve fitting, and clustering, are naturally defined in terms of arrays (and array elements). Overall, the array-view, instead of the relational table view, often leads to better expressibility as well as performance.

A key functionality associated with array databases is the set of operations classified as *structural aggregations*. These operations collect and aggregate elements within each *group*, where these groups are created based on *positional relationships*. This is in contrast to groupings based on the same value that is common with the relational table view. As scientific applications where data needs to be stored and/or viewed as array typically need support for such operations, Array DBMSs like SciDB [8], RasDaMan [7], and MonetDB (which provides a prototype implementation of the language SciQL [51]) include the support for structural aggregations.

Array DBMSs, like their relational counterparts, involve an expensive *data ingestion* phase, where arrays are restructured so as to speedup query processing later. These systems are clearly well suited for situations where data needs to be loaded once and then queried frequently, and thus, the cost of data ingestion is well justified. On the other hand, there are applications, such as analysis of simulation data [40, 46, 45, 41], where massive amounts of data is analyzed only infrequently, and hence the cost of data ingestion cannot be justified. In fact, with current Array DBMSs, because of the use of specialized data formats in various scientific domains, ingesting data into a database system can require a series of transformations. For example, the current recommended way to import high-volume satellite imagery data, which is captured in the HDF-EOS format, into SciDB involves converting the HDF-EOS files into CSV text files, transforming the CSV files into the external files in the SciDB specialized format, loading the data into SciDB in the form of temporary 1-dimensional arrays, and finally, casting the loaded 1-dimensional arrays into n-dimensional ones within SciDB [32]. Clearly, the process is extremely time-consuming and requires availability of additional memory and disk space for transformations and storing data in intermediate formats. In practice, such data ingestion is extremely expensive, taking easily 100x the time for executing a simple query over the same amount of data.

As an alternate solution to databases requiring data ingestion costs, a new paradigm of using native storage as a DB and pro-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [Permissions@acm.org](mailto:Permissions@acm.org).  
SSDBM '14, June 30 - July 02 2014, Aalborg, Denmark  
Copyright 2014 ACM 978-1-4503-2722-0/14/06 ...\$15.00.  
<http://dx.doi.org/10.1145/2618243.2618270>

viding database-like support has recently been shown to be an effective approach for dealing with infrequently queried data. This paradigm aims to develop a database engine on top of the native storage. Examples include the NoDB approach [3] and automatic data virtualization [47]. Although this paradigm can be a promising approach for handling massive arrays that are not queried often, to our best knowledge, so far it has only been applied to relational tables stored in flat files [3] or for simple selection queries over arrays [47].

Applications that generate massive arrays, such as the scientific applications, often store the data in one of a small number of popular array storage formats, like NetCDF and HDF5. Thus, it is important to examine if database-like querying support can be provided on top of such storage of data. In this paper, we describe an approach of using array storage as a DB with support for structural aggregations. This approach has been implemented in a system we refer to as **Structural AGgregations over Array storage (SAGA)**. We focus on efficiently implementing and effectively parallelizing key *structural aggregation* operations, including *grid*, *sliding*, *hierarchical*, and *circular* aggregations. We propose different partitioning strategies, with the goal of handling both computationally expensive and inexpensive aggregations, as well as dealing with possibly skewed data. Moreover, we design an analytical model for choosing the best scheme for a given query and dataset. We have also developed aggregation methods for improving the performance on chunked array storage. Our system can process data from any array storage formats (including formats like HDF5 and NetCDF that are frequently used for scientific data) and even chunked or compressed array storage, as long as an interface for loading an array slab is provided [40, 45, 41].

We have extensively evaluated our structural aggregation approaches by using multiple real-world and a set of synthetic datasets of varying skew. We show how the relative performance of different partitioning strategies changes with varying amount of computation in the aggregation function and different levels of data skew. Moreover, we demonstrate the effectiveness of our cost models for choosing the best partitioning strategy. By comparing performance with SciDB, we show that despite working on native array storage, the aggregation costs with our system are significantly lower. Finally, we also show that our structural aggregation implementations achieve high parallel efficiency.

## 2. STRUCTURAL AGGREGATIONS

This section lists different types of structural aggregations that arise in scientific applications and are supported in our system.

### 2.1 Background: Structural Groupings and Structural Aggregations

Grouping has been one of the most important operations provided by databases, most often specified by a GROUP BY clause in standard SQL. Additionally, *aggregation functions* and corresponding *aggregation filters* can be added to the grouping. Most common grouping used in a database management system is the *value-based grouping*, where elements of the same *value* comprise one group. Over years, SQL extensions, particularly SQL:2003, introduced *structural grouping*, where elements are mapped to a group based on *positional* relationships. It turns out that such groupings are very common when arrays are the main structure. For example, calculating a simple *moving average* over array elements involves a positional relationship (i.e., a set of adjacent values).

In this paper, we refer to aggregations that are based on any structural or positional grouping as *structural aggregations* - they include not only traditional aggregates such as COUNT, SUM, AVG, as well as MIN and MAX, but also user-defined aggregation functions for domain-specific analysis. For our presentation,

we consider structural aggregations of three types, which are the *grid*, *sliding*, and *hierarchical* (or *circular*) aggregations, described in the following paragraphs.

The structural aggregations we consider process a rectilinear section of any size and dimensionality from an array (or an *array slab*). Unlike the aggregations based on array access patterns, which can load multiple array subareas into one group and then return a single aggregation result, structural aggregations divide an array slab into multiple and possibly overlapping groups and then return multiple corresponding aggregation results at one time. Structural aggregations can also be combined with value-based filtering conditions, as we will show through an example later.

Figure 1 shows the structural aggregation examples graphically, while the APIs and mathematical definitions are summarized in Table 1. This description assumes that we have a 2D array slab  $S$ , where the upper left element, the centroid, and the number of elements are denoted as  $(x_0, y_0)$ ,  $(x_c, y_c)$ , and  $|S|$ , respectively. For simplicity, we assume that the aggregation operator is SUM.

### 2.2 Grid Aggregation

Grid aggregation is widely used in many scientific applications. It involves dividing an array slab into multiple *disjoint* smaller blocks and then aggregating values within each smaller block. For example, in astrophysics, vast astronomy events are stored in massive multi-dimensional arrays. The data is often broken into smaller disjoint spatial grids of equal *grid size*, and a corresponding multi-dimensional histogram can be produced by binning the events over those spatial grids [20]. In Figure 1(a), a  $4 \times 4$  array is split into 4 non-overlapping  $2 \times 2$  grids, and then values within each smaller grid are aggregated.

### 2.3 Sliding Aggregation

Sliding aggregation calculates a sequence of aggregates within an array slab through a sliding grid of a fixed *grid size*. The grid moves from a starting element to an ending element with a *stride*, where the stride value is 1 by default. Sliding aggregation also arises in a number of scientific domains. For example, in earth sciences, it is common to apply certain image denoising algorithms, such as non-local means (NL-means), to preprocess satellite imagery data [9, 49]. More specifically, a Gaussian kernel function is applied on a *sliding window* to smooth out the outliers. As Figure 1(b) illustrates, a  $3 \times 3$  grid slides within a  $4 \times 4$  array in row-major fashion.

### 2.4 Hierarchical and Circular Aggregations

In space sciences, scientists may be interested in looking into the gradual influence of radiation from a source, which may be a pollution source or an explosion location, over its adjacent region. A set of regions of increasing radii (or sizes) are analyzed for this purpose. For hierarchical aggregation, the centroids of all the grids is the same as the centroid of the entire array slab, and each grid is entirely covered by a concentric outer grid. The radius of the innermost grid is specified by an *initial radius*. For the other grids, the radius increases by a fixed *step size*, until it reaches the array slab boundary. In Figure 1(c), three concentric grids are aggregated in a  $6 \times 6$  array: the innermost one is a  $2 \times 2$  grid, the middle one is a  $4 \times 4$  grid, and the outermost one is the entire array.

A variant of such hierarchical aggregation is *circular aggregation*, which calculates a sequence of aggregates of concentric but *disjoint* circles instead of regularly shaped grids. Figure 1(d) illustrates a circular aggregation that corresponds to the hierarchical aggregation shown in Figure 1(c).

There is a difference between grid aggregation and three other aggregations we have introduced so far. Particularly, we can also refer to grid aggregation as *non-overlapping* aggregation, since each input value is used towards aggregating a disjoint grid. In compari-

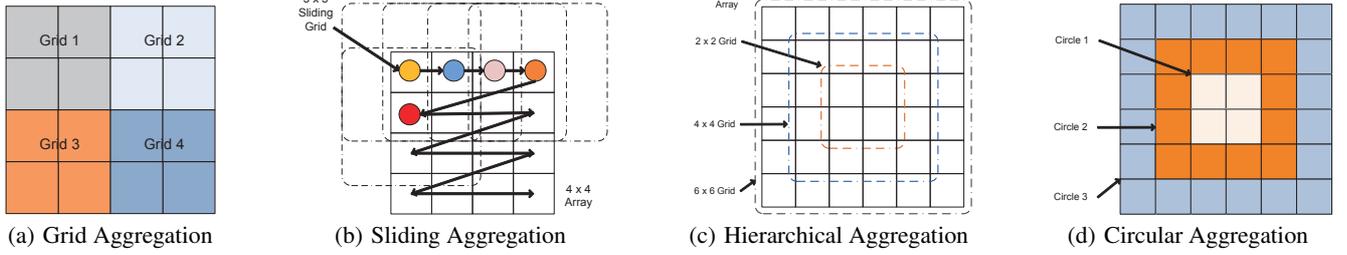


Figure 1: Structural Aggregation Examples

Table 1: Structural Aggregation APIs and Mathematical Definitions

API	Mathematic Definition (for the Aggregation Operator SUM in a 2-Dimensional Space)
$gridAgg(array\_slab S, grid\_size g)$	$\forall(x, y) \in S, (x - x_0) \bmod g_x = 0 \text{ and } (y - y_0) \bmod g_y = 0: \sum_{i=x}^{y_x} \sum_{j=y}^{y_y} S(i, j)$
$slidingAgg(array\_slab S, grid\_size g, stride s)$	$\forall(x, y) \in S, (x - x_0) \bmod s_x = 0 \text{ and } (y - y_0) \bmod s_y = 0: \sum_{i=x}^{y_x} \sum_{j=y}^{y_y} S(i, j)$
$hierarchicalAgg(array\_slab S, initial\_radius r, step\_size s)$	$\forall(x, y) \in S, \frac{x_0 - r_x - x}{s_x} = \frac{y_0 - r_y - y}{s_y} = n, n \geq 0: \sum_{i=x}^{2 \times (r_x + n \times s_x)} \sum_{j=y}^{2 \times (r_y + n \times s_y)} S(i, j)$
$circularAgg(array\_slab S, initial\_radius r, step\_size s)$	$\forall(x, y) \in S, \frac{x_0 - r_x - x}{s_x} = \frac{y_0 - r_y - y}{s_y} = n, n \geq 1: \sum_{i=x}^{2 \times (r_x + n \times s_x)} \sum_{j=y}^{2 \times (r_y + n \times s_y)} S(i, j) - \sum_{i=x}^{2 \times (r_x + (n-1) \times s_x)} \sum_{j=y}^{2 \times (r_y + (n-1) \times s_y)} S(i, j)$

son, sliding and hierarchical aggregations are all forms of *overlapping aggregation*. For all practical purposes, a circular aggregation is also an overlapping aggregation, since no library for scientific arrays provides an interface to exactly load a circle of array elements at a time.

### 3. GRID AGGREGATION ALGORITHMS

We now focus on parallel algorithms for performing grid aggregations. We assume that array is stored in a shared file system from which processes on different nodes and cores can access this data. This model matches the configuration of many high performance systems today, and also supports parallelization across cores within a node and nodes in a cluster/cloud in a uniform fashion. As stated previously, we also assume that array has been stored in a native form that is not controlled by our system. This leads to unique challenges in our work, since a system such as SciDB that has a data ingestion phase reorganizes the array during such a phase, and therefore, algorithms involved can be quite different.

Given an array slab, it is relatively easy to implement sequential grid aggregation, but efficient parallelization involves several challenges. To understand possible challenges, we first consider a simple approach for parallelization.

#### 3.1 Outline of a Simple Parallel Grid Aggregation Algorithm

The outline of the parallel grid aggregation method is shown as Algorithm 1. According to the grid size specified by the user, a set of grids are initialized and grids within this set are then partitioned. Note that a single grid may be within one partition or may be further divided, with sub-grids assigned to different partitions (or processors), since such partitioning is not performed specific to a particular query. There are two phases after the partitioning, i.e., a *local aggregation* phase and a *global aggregation* phase. In the local aggregation phase, each processor is responsible for aggregating each grid in its own partition. In the global aggregation phase, for those grids distributed among different partitions, one of the processors serves as a *master processor* and merges the local aggregates of all partial grids.

This simple algorithm has multiple advantages. First, by ensuring that each element is read by only one process, we avoid any redundant reads or computation. Second, workload distribution is static, and therefore, there is no runtime overhead of assigning work

#### Algorithm 1: GridAgg(slab $S$ , num\_procs $n$ )

- 1: Initialize  $m$  grids  $G$  and  $m$  corresponding aggregates  $A$
- 2: Divide  $S$  into  $n$  partitions (set  $P$ )
- 3: Each grid  $G_i (i = 1, \dots, m)$  is assigned to one or more partitions
- 4: Each processor loads its partition  $P_j (j = 1, \dots, n)$
- 5: **for** each (possibly partial) grid  $G'_i$  in  $P_j$  **do**
- 6:    $A_i \leftarrow aggregate(G'_i)$  *{\* local aggregation \**
- 7:   **if**  $G'_i \neq G_i$  **then**
- 8:     Merge  $A_i$  of all partial grids *{\* global aggregation \**
- 9:   **end if**
- 10: **end for**
- 11: **return**  $A$

to processors.

### 3.2 Challenges and Different Partitioning Strategies

The challenges in executing the algorithm effectively and achieving high parallel efficiency are: 1) how to ensure that the set of partitions to be processed by a particular process are physically contiguous on the disks, so as to minimize disk seek times, 2) how to balance the workload across processes when the datasets are skewed - particularly when a *value-based filtering condition* is added to a compute-intensive aggregation (as explained further below), and 3) how to minimize the communication cost when calculating global aggregates. The specific importance of each depends upon the nature of the queries.

Many aggregates of interest for scientific applications are domain-specific, user-defined, and compute-intensive (possibly even quadratic or cubic with respect to the grid size). Now, to further explain the second challenge we listed, consider the following query: suppose the aggregation involves computations over all the elements that are above a certain threshold (or not equal to a pre-defined missing value) within each grid. The use of this threshold (or excluding missing values) is what we also referred to as a *value-based filtering condition* above. Because of the distribution of values across different partitions, the number of arithmetic operations may vary significantly.

To address the above challenges, we have developed different partitioning strategies, each of which addresses one or more of the challenges.

P1 Grid 1	P1 Grid 2	P1 Grid 3 P2	P2 Grid 4	P2 Grid 5
P3 Grid 6	P3 Grid 7	P3 Grid 8 P4	P4 Grid 9	P4 Grid 10

(a) Coarse-Grained Partitioning Example

P1	P1	P1	P1	P1
P2 Grid 1	P2 Grid 2	P2 Grid 3	P2 Grid 4	P2 Grid 5
P3	P3	P3	P3	P3
P4	P4	P4	P4	P4
P1	P1	P1	P1	P1
P2 Grid 6	P2 Grid 7	P2 Grid 8	P2 Grid 9	P2 Grid 10
P3	P3	P3	P3	P3
P4	P4	P4	P4	P4

(b) Fine-Grained Partitioning Example

P1 Grid 1	P2 Grid 2	P3 Grid 3	P4 Grid 4	P1 Grid 5
P2 Grid 6	P3 Grid 7	P4 Grid 8	P1 Grid 9 P2	P3 Grid 10 P4

(c) Hybrid Partitioning Example

**Figure 2: Examples of Different Partitioning Strategies**

**Coarse-Grained Partitioning:** Coarse-grained partitioning divides the given array slab into multiple equal-sized partitions, each of which comprises contiguous grids. As an example, see Figure 2(a), where 10 grids in a 2D array slab are partitioned for 4 processors, with grid 3 evenly distributed to processors 1 and 2, and grid 8 to processors 3 and 4. Broadly, the array slab is evenly partitioned, using the *highest dimension* (e.g., rows when the layout is row-major), so that the grids are contiguous. If the number of grids is divisible by the number of processors, then all the aggregates can be calculated locally. Otherwise, any grid that contains the partition boundary will be covered by two adjacent partitions, resulting in an extra (parallel) merge operation over each two local partial aggregates.

To summarize, this approach gives good I/O performance and low communication cost. On the negative side, in presence of data skew, this approach does not provide load balance.

**Fine-Grained Partitioning:** Fine-grained partitioning evenly distributes elements within each grid from the given array slab across all processors. An example can be seen from Figure 2(b), where a 2D array slab that consists of 10 grids is partitioned across 4 processors. Each processor calculates a partial aggregate for each grid during the local aggregation, and then the global aggregation phase involves an all-to-one or all-to-all reduction over all the local aggregates.

As the example shows, the partitioned data, although contiguous within the same row of grids, is scattered among different rows. Thus, the I/O performance will not be as good as with the coarse-grained partitioning, and the communication cost can also be higher. However, a fine-grained distribution is likely to handle skew better than a coarse-grained distribution.

**Hybrid Partitioning:** Hybrid partitioning combines elements of coarse-grained and fine-grained partitioning. First, each partition covers an equal number of grids in the given array slab, with assignment in a round-robin fashion. If the number of grids is not divisible by the number of processors, the extra grids can be partitioned in a coarse-grained or fine-grained manner. As Figure 2(c) shows, 10 grids in a 2D array slab are partitioned into 4 processors. Starting from the grid 1, every 4 consecutive grids are mapped to 4 processors in order, and then grids 9 and 10 are partitioned in a coarse-grained manner.

Overall, hybrid partitioning has an advantage with respect to

communication cost and dealing with skew. This is because most aggregates can be calculated locally, and round-robin distribution ensures that one is likely to be more skew-tolerant. However, grids within a single partition are not contiguous.

**Auto-Grained Partitioning:** Among the partitioning strategies we have presented so far, both fine-grained and hybrid schemes can deal with data skew. However, they do so in an ad-hoc fashion, by round-robin distribution of grids or grid elements across the partitions, which is likely to balance the data amount satisfying value-based predicates across processes. In comparison, the strategy we introduce now is designed specifically for handling data skew and workload imbalance. Rather than evenly partitioning the input data (and hence balance the data loading costs), auto-grained partitioning aims to equally divide the total processing cost. Thus, the size of each partition is not necessarily equal.

Auto-grained partitioning assumes we have a certain estimate of data skews for different grids, and thus, the different underlying processing costs. Overall, it works as follows. First, given a value-based filtering condition, a *lightweight* uniform sampling is applied at runtime, to estimate the density of all the grids after filtering. In practice, such sampling is performed in parallel over different parts of the array slab. Second, based on the sampled density, for each grid we have an estimate of the total processing cost, which is sum of the loading cost (fixed) and the computation cost (which depends upon the density and the computation complexity of aggregation). Thus, the input of the partitioning algorithm can be reconsidered as a one-dimensional *cost array*, where the value of each element is the estimated cost of a corresponding grid. This problem can be modeled as a *balanced contiguous multi-way partitioning* problem, where the objective is to find a sequence of separator indices to divide this cost array into contiguous partitions such that the maximum cost of all the partitions is minimized.

We have designed two different algorithms to solve this problem. The first approach is a dynamic programming algorithm with a  $\Theta(NM^2)$ -time complexity, where  $N$  is the number of processors, i.e., the number of partitions, and  $M$  is the number of grids. Other than  $M$  and  $N$ , another input is the cost array of  $M$  elements, denoted as  $A$ . The decomposed sub-problem  $P(n, m)$  indicates the maximum partition cost after dividing the first  $m$  elements of  $A$  into  $n$  contiguous subarrays in a balanced way, so that the output is reformulated as  $P(N, M)$ . The problem can be decomposed as follows:

$$P(n, m) = \begin{cases} A_1, & \text{if } m = 1 \\ \sum_{i=1}^m A_i, & \text{if } n = 1 \\ \min(\max_{1 < k < m} (P(n-1, k), \sum_{i=k+1}^m A_i)) & \end{cases}$$

The second approach is a greedy  $O(M)$ -time algorithm to find a suboptimal partitioning. It can be completed with two passes over the cost array. First, we calculate the average partition cost with the given cost array and the number of partitions. Second, by linearly scanning the cost array, we remove a partition from the cost array when the accumulated cost is close to the average partitioning cost.

When the number of grids is relatively small, the system uses the dynamic programming algorithm to gain an optimal workload balance. When the number of grids is large, the system chooses the greedy algorithm. In this case, the greedy algorithm runs much faster, and it can produce a suboptimal solution that is almost as good as the optimal solution provided by the dynamic programming algorithm.

Overall, apart from the good workload balance, the auto-grained partitioning has a very low communication cost, since all the aggregates are calculated locally. Moreover, its I/O performance is also as good as the coarse-grained partitioning, as all the grids in the same partition are contiguous. The disadvantage is the overhead of parallel sampling and runtime partitioning.

**Table 2: Summary of Model Parameters**

Parameter	Type	Definition
$N_{glb}$	App	Number of elements in total
$N_{loc}$	App	Number of elements in a partition
$N_{glbGrids}$	App	Number of grids in total
$N_{locGrids}$	App	Number of grids in a partition
$N_{grid}$	App	Number of elements in a grid
$N_{procs}$	App	Number of processors (also the number of partitions)
$P_{elem}$	App	Processing cost per element
$N_{ftdGrid}$	Dataset	Number of elements after filtering in a grid
$F_{loc}$	Dataset	Local filtering factor for a partition
$F_{glb}$	Dataset	Global filtering factor for the entire dataset
$R$	Env	Ratio between the computation cost and the I/O cost for a single element
$f_{coarse}$	Env	Loading cost factor for the coarse-grained partitioning
$f_{fine}$	Env	Loading cost factor for the fine-grained partitioning
$f_{hybrid}$	Env	Loading cost factor for the hybrid-grained partitioning
$P_{hybrid}$	Env	Penalty factor for the hybrid-grained partitioning
$P_{auto}$	Env	Penalty factor for the auto-grained partitioning
$C_{samp}$	Env	Sampling cost for the auto-grained partitioning

### 3.3 A Cost Model for Choosing Partitioning Strategy

As we discussed above, the partitioning strategies we have introduced involve trade-offs with respect to communication and data loading costs, as well as their ability to handle skew. To automatically choose the partitioning strategy that results in the lowest total processing cost, we have developed a cost-based partitioning strategy decider.

Broadly, we can divide the total processing cost of parallel grid aggregation into five parts: the *initialization* cost for result values, the *loading* cost for partitions to be read, the *filtering* cost, i.e., applying a value-based predicate before aggregation, the *computation* cost of calculating aggregates, and the *communication* cost of merging local aggregates. For simplicity, we can ignore both the initialization cost and the filtering cost, since the former is trivial and the filtering phase costs can be merged with those of the data loading phase. Finally, it turns out that with an exception for fine-grained partitioning with very small grid sizes, communication costs can also be ignored. To keep our model simple, we assume that fine-grained partitioning is not an option when the grid sizes are small. Thus, we can simplify the total processing cost as:

$$C_{total} = C_{load} + C_{comp} \quad (1)$$

Here,  $C_{load}$  represents the loading cost, and  $C_{comp}$  is the computation cost.

The parameters used in the cost model are defined in Table 2. Those parameters can be categorized into three types: 1) *Application Parameters*, which correspond to the query parameters, including the given array slab, grid size, computation involved in calculating aggregates and other similar parameters, 2) *Dataset Parameters*, particularly, those reflecting data skew, which we assume we have broad knowledge of in advance, or acquire through a preprocessing round, and 3) *Environment Parameters*, which correspond to the architecture and disk system, which are learnt through sample runs.

Now, since all grids are dense, we have  $N_{grid} = N_{glb}/N_{glbGrids}$ . Second, with the exception of when auto-grained partitioning is used, the sizes of all the partitions are equal, i.e.,  $N_{loc} = N_{glb}/N_{procs}$ , and  $N_{locGrids} = N_{glbGrids}/N_{procs}$ .

**Loading Cost Analysis:** The loading cost is clearly independent of the data skew. In addition, through extensive experiments, we also observed that this cost is almost independent of grid size ( $N_{grid}$ ), unless the grid size is very small. At the same time, the loading cost does depend upon the partitioning strategy and the partitioned

data size ( $N_{loc}$ ), since they impact the disk seek times. Rather than modeling seek times analytically, we take an experimental approach, where the impact of disk seek for a particular partitioning strategy is measured for a particular environment. This cost is then represented as a weighing function, called the *loading cost factor*, and we estimate the loading cost by multiplying the loading cost factor with the size of partitioned data. We denote these loading cost factors by  $f_{coarse}$ ,  $f_{fine}$ , and  $f_{hybrid}$ , for coarse-grained, fine-grained, and hybrid partitioning, respectively. Then, the expression for the loading cost of a single partition, shown for coarse-grained partitioning as an example, will be

$$C_{load}^{coarse} = f_{coarse} \times N_{loc} = f_{coarse} \times \frac{N_{glb}}{N_{procs}} \quad (2)$$

In the environment we used, these three weighing functions were learnt using a set of representative queries. These queries did not involve the same array slab size or aggregation functions as the ones used in our experiments, and thus, our experiments validated the cost model. The specific values we obtained were:  $f_{coarse} - 12.2$ ,  $f_{fine} - 19.5$ , and  $f_{hybrid} - 16$ .

For the auto-grained partitioning strategy, we estimate the total loading cost rather than the loading cost of a single partition. Because the partitioned grids are contiguous, the loading cost factor for coarse-grained partitioning can be used, and thus we have:

$$\sum^{N_{procs}} C_{load}^{auto} = f_{coarse} \times N_{glb} \quad (3)$$

**Computation Cost Analysis:** The computation cost for a partition is the sum of the computation cost for all the local grids. The computation cost for a grid is dependent upon the density of the grid as well as the nature of the aggregation computed (i.e., computation complexity). To capture the latter, the parameter  $P_{elem}$  denotes the per element processing cost for one grid aggregation. This parameter needs to be obtained for each aggregation function using sample datasets.

For the first three partitioning strategies, which evenly partition the input data, the overall computation cost should be the maximum cost among all the partitions, which can be formally stated as:

$$C_{comp} = \max_{N_{procs}} \left( \sum^{N_{locGrids}} R \times P_{elem} \times N_{ftdGrid} \right) \quad (4)$$

Here,  $R$  is the adjustable ratio between the computation cost and the I/O cost for a single element. The value of the parameter  $R$  is dependent upon the environment and can be determined during a learning phase, similar to how the loading factor parameters are obtained. We now elaborate on how other parameters are estimated for each of the strategies:

**Coarse-Grained Computation Cost:** Since the number of elements to be processed (those left after filtering) differs across partitions in the case of coarse-grained partitioning strategy, the computation cost is dependent on the partition where the least amount of data is filtered out. Particularly, we use a parameter  $F_{loc}$  to indicate the proportion of data after filtering in a given partition. We also have:

$$\sum^{N_{locGrids}} N_{ftdGrid} = F_{loc} \times N_{loc} \quad (5)$$

By using Equation 4 and 5, the coarse-grained computation cost can be estimated as follows:

$$C_{comp}^{coarse} = \max_{N_{procs}} \left( R \times P_{elem} \times F_{loc} \times \frac{N_{glb}}{N_{procs}} \right) \quad (6)$$

**Fine-Grained Computation Cost:** Because the fine-grained partitioning strategy is likely to evenly distribute the data after filtering,

the local filtering factor of any partition is equal to the global filtering factor. Thus, we have  $F_{loc} = F_{glb}$ . Because of such even distribution of filtered data among all the partitions, fine-grained partitioning strategy evenly distributes the computation workload. Therefore, to estimate the fine-grained computation cost, we can specialize Equation 4 and have:

$$\begin{aligned} C_{comp}^{fine} &= \sum^{N_{locGrids}} (R \times P_{elem} \times N_{ftdGrid}) \\ &= R \times P_{elem} \times F_{loc} \times \frac{N_{glb}}{N_{procs}} \quad (7) \\ &= R \times P_{elem} \times F_{glb} \times \frac{N_{glb}}{N_{procs}} \end{aligned}$$

**Hybrid Computation Cost:** Similar to the fine-grained partitioning strategy, the hybrid partitioning strategy can also lead to a reasonably well-balanced computation workload. Therefore, we can infer the hybrid computation cost from Equation 7, but to capture possible small workload imbalance, we empirically add a penalty factor  $p_{hybrid} = 0.06$ :

$$C_{comp}^{hybrid} = (R \times P_{elem} \times F_{glb} \times \frac{N_{glb}}{N_{procs}}) \times (1 + p_{hybrid}) \quad (8)$$

**Auto-Grained Computation Cost:** Calculation of costs with the auto-grained partitioning strategy involves additional considerations. Particularly, an extra *sampling cost* should be added, and because the auto-grained partitioning strategy does not evenly partition the input data, we have to estimate both the loading cost and the computation cost altogether. Through extensive experiments, we observed a slight workload imbalance for the auto-grained partitioning. To formally capture this workload imbalance, we empirically add a penalty factor  $p_{auto}$  to the total processing cost, where  $p_{auto} = 0.06$ :

$$C_{total}^{auto} = \frac{\sum^{N_{procs}} C_{load}^{auto} + \sum^{N_{procs}} C_{comp}^{auto}}{N_{procs}} \times (1 + p_{auto}) + C_{samp} \quad (9)$$

where  $C_{samp}$  refers to the sampling cost. Although the sampling cost can also be determined by the user with a given sampling probability, for simplicity, we consider it as a small constant, which is denoted as  $c_{samp} = 2$ .

Finally, the total computation cost in Equation 9 can be estimated as follows:

$$\sum^{N_{procs}} C_{comp}^{auto} = R \times P_{elem} \times F_{glb} \times N_{glb} \quad (10)$$

## 4. OVERLAPPING AGGREGATION ALGORITHMS

In this section, we present algorithms for overlapping aggregations, i.e., sliding, hierarchical, and circular aggregations.

Unlike non-overlapping aggregations, an overlapping aggregation can incur repeated loads and computations on the same data element. This leads to two important considerations in the design of algorithms: 1) *I/O Cost*: we will like to reuse the data loaded into memory to reduce repeated disk I/O operations, and 2) *Memory Accesses*: we will like to avoid repeated accesses to the same element even in memory, since they can cause unnecessary cache misses.

Based on these two considerations, we have designed three approaches: a *naive approach*, which, as the name suggests, is a simple scheme that ignores both I/O cost and memory access considerations, a *data-reuse approach* that addresses I/O costs but does not reduce memory accesses, and an *all-reuse approach*, which is conscious of both.

### 4.1 Naive Approach

To calculate each aggregate in an overlapping aggregation, one possibility is to simply load a grid of elements and then performs an aggregation over them, repeating the process for each aggregate to be computed. Thus, if there are  $N$  aggregates to calculate in total,  $N$  separate grids are loaded. To parallelize this operation, aggregations to be performed can be evenly distributed among each processor. Any of the partitioning strategies discussed in Section 3 can be used for this purpose. Obviously, the main drawback of this approach is that data resident in memory and/or cache is not reused, leading to redundant I/O and repeated memory accesses. Besides, significant workload imbalance may arise in hierarchical and circular aggregations due to different grid sizes.

### 4.2 Data-Reuse Approach

Unlike the naive approach, the data-reuse approach can reuse the data loaded into memory. We load a large array portion in each step, which can be either the entire array slab involved in the query, or its subset (e.g., the outermost grid in the hierarchical or circular aggregations). Aggregations that need this data can be processed without any further disk I/O operations. Now, to parallelize the aggregation, the loaded array slab can be evenly distributed to all the processors. Certain aggregations will require data from multiple partitions (or even different array portions, which is the case even for the sequential version). These cases can be handled by performing a local aggregation followed by a global aggregation, as is needed in the case of non-overlapping aggregations.

Clearly, this approach can massively reduce disk I/O costs. However, other limitations of the naive approach still remain, i.e., it involves repeated memory accesses to the same element.

### 4.3 All-Reuse Approach

Similar to the data-reuse approach, the all-reuse approach also loads a large array slab at a time and reuses the loaded data to reduce disk I/O. However, the distinguishing characteristic of this method is that each element is accessed only once - even for multiple aggregations - leading to much fewer memory accesses. The main underlying idea is that it is more computationally efficient to iterate over elements and update the associated aggregates in the process, rather than iterating over aggregation results that need to be calculated and then accessing the required elements for producing these results.

Algorithm 2 shows sliding aggregation by using the all-reuse approach. This method can only be applied if the aggregation operator is algebraic (also referred to as associative and commutative). The array slab is evenly partitioned in the highest dimension to ensure data contiguity. After a partition is loaded, local aggregates corresponding to all sliding grids that overlap with this partition are initialized. Once an element is read, it is used to update all the aggregates it contributes to. In this way, when each element is loaded into cache, it will be fully reused before being flushed out, leading to fewer cache misses. Because some of sliding grids overlap with two neighboring partitions, corresponding partial aggregation results need to be merged in a global aggregation phase.

The similar method can be applied for hierarchical aggregation. The main differences are, local aggregates are initialized for all concentric grids, rather than being initialized for all sliding grids that a partition overlaps with, and the global aggregation involves merging all local aggregates instead of the only a few ones that overlap with two neighboring partitions. Further, circular aggregation is a special case, since each element will contribute to exactly one aggregate that corresponds to a disjoint circle. Therefore, there is no need for an inner loop in Algorithm 2 to iterate over different aggregates.

---

**Algorithm 2:** SlidingAgg(*slab S, num\_procs n*)

---

```
1: Evenly divide  $S$  into  $n$  partitions (set  $P$ )
2: Each processor loads its partition  $P_i (i = 1, \dots, n)$ 
3: Initialize  $m$  local aggregates  $A$  for the  $m$  sliding grids that  $P_i$ 
   overlaps with
4: for each element  $e$  in  $P_i$  do
5:   for each aggregate  $A_j (j = 1, \dots, m)$  associated with  $e$  do
6:      $A_j \leftarrow \text{aggregate}(e, A_j)$ 
7:   end for
8: end for
9: Perform global aggregation
10: return  $A$ 
```

---

## 5. OPTIMIZATION FOR CHUNKED ARRAY STORAGE

Our goal in this paper has been to support an approach of directly using array storage as a DB, where no data ingestion costs are involved, and instead, operations are supported over native storage. The default array storage layout on disks is the same as the in-memory layout, which is a contiguous block stored in the *row-major* or the *column-major* fashion. It turns out that many libraries (e.g., NetCDF and HDF5) that manage array storage on disks can also support a more optimized layout, which is called the *chunked storage* [35]. The chunked storage involves splitting an array into chunks based on a multi-dimensional partitioning. As a result, array elements are not stored contiguously along any of the dimensions, and hence dimension dependency can be alleviated. Libraries that support such data storage also have the feature that when a single element is loaded to the memory, all the other elements in the same chunk are also cached.

Because the grid aggregation methods discussed in Section 3 do not consider the chunked array storage, the algorithms can be quite inefficient when chunked storage is used. For example, if a chunk is shared by multiple logical grids, this chunk may be loaded repeatedly to perform aggregations on different grids. Similarly, if a chunk is shared by different partitions, this chunk is loaded redundantly by multiple processors. Avoiding such overheads is the motivation for our method (Algorithm 3).

---

**Algorithm 3:** ChunkBasedGridAgg(*slab S, num\_procs n*)

---

```
1: Identify chunks comprising the queried array slab  $S$ 
2: Evenly divide chunks into  $n$  contiguous chunk sets (set  $C$ )
3: Set a load buffer  $B$  {* buffer size is a multiple of chunk size and less
   than cache size *}
4: Each processor divides  $C_i (i = 1, \dots, n)$  into subsets of chunks of
   size at most  $B$ 
5: for each subset of size at most  $B$  do
6:   Load chunk(s) into  $B$ 
7:   for each chunk  $c$  in  $B$  do
8:     for each element  $e$  in  $c$  do
9:       for each aggregate  $A_j (j = 1, \dots, m)$  associated with  $e$  do
10:         $A_j \leftarrow \text{aggregate}(e, A_j)$ 
11:      end for
12:     end for
13:   end for
14: end for
15: return  $A$ 
```

---

First, to minimize the number of chunks that have to be loaded by more than one processors, instead of mapping logical grids to processors, we map physical chunks to processors. Second, to favor I/O performance, the partitioning is performed in such a way that the chunks within one partition are contiguous to the extent possible. To fully reuse the cached data in the same chunk, once a chunk of data is read, all elements in it are used to update the aggregations they contribute to.

Note that the size of a load buffer needs to be carefully tuned, i.e., it should be a multiple of the chunk size and less than the cache size. If both the chunk size and the buffer size are very small, extra overheads can be caused by frequent small I/O requests. If the buffer size is too large compared with the cache size, the first a few cached chunks will be flushed out before they are actually used in aggregation. Since the actual number of chunks in the load buffer is usually less than the number of processors used, the fine-grained partitioning strategy is not supported. Otherwise, one chunk will be redundantly loaded by multiple processors.

The similar chunk-based optimization can also be applied for overlapping aggregations, since the all-reuse approach does not require data to be loaded in any specific manner.

## 6. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of our approach and implementations for various structural aggregations, using two real datasets, as well as synthetic datasets created to vary the amount of skew. Our experiments were conducted on a cluster of nodes with a shared file system (for evaluating parallel performance) and on the Amazon EC2 platform (for comparing with SciDB). We designed the experiments with the following goals: 1) to demonstrate the performance of our proposed partitioning strategies for grid aggregation (especially their ability to handle skewed data) and the effectiveness of the cost models for choosing the best scheme, 2) to compare performance of our system with SciDB, a popular Array DBMS, and 3) to evaluate the performance of parallel algorithms for different structural aggregations we have introduced.

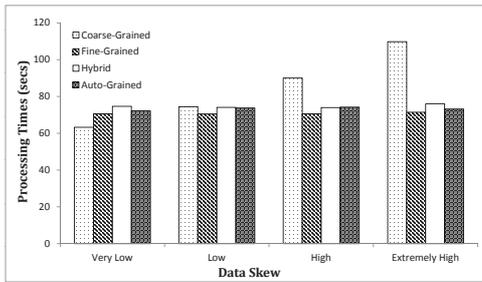
### 6.1 Experimental Setup

Our experiments were conducted using two real large-scale array-based datasets. These are both two-dimensional satellite imagery datasets, which use the HDF5 format that is popular in many scientific areas. These two datasets are referred to as *cloud pressure* and *terrain pressure*, since they record effective cloud pressure and terrain pressure, respectively. Each one comprises  $1,000K \times 1K$  double-precision elements, and has a size of 8 GB. They are downloaded from the Land Parameter Retrieval Model (LPRM) Level 2 (swath) collection [1]. Additionally, in Section 6.2 we also used 4-GB synthetic datasets of varying skew, each of which comprises  $512K \times 1K$  double-precision elements. Since we focused on evaluating the performance of our system on native array storage, the arrays are stored in the default row-major order on disks, i.e., the array storage was not preprocessed or reorganized. An exception is the set of experiments with the chunk-based grid aggregation. We used traditional aggregates including COUNT, SUM, AVG, as well as MIN and MAX in our experiments. Specifically, a compute-intensive user-defined aggregation function was used in Section 6.2 and 6.4.

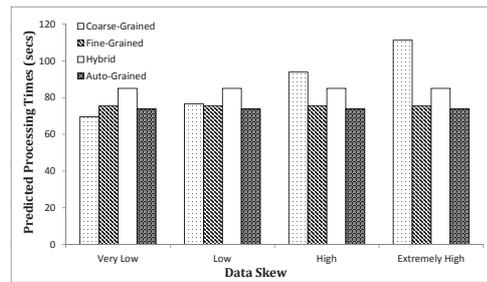
The performance comparison between our system and SciDB was conducted on an Amazon EC2 instance with 32 GB of main memory and 8-core Intel(R) Xeon(R) E5-2670 CPU, and the clock frequency of each core was 2.6 GHz. The version of SciDB was 13.12. All the other experiments were conducted on a cluster of machines, where each node had an AMD Opteron(TM) Processor 8218 with 4 dual-core CPUs (8 cores in all). The clock frequency of each core was 2.6 GHz, and the system had a 16 GB main memory. We used up to 16 compute nodes for our study. HDF5 version 1.8.10 was used for all our experiments.

### 6.2 Handling Skewed Data and Prediction Accuracy of Partitioning Strategy Decider

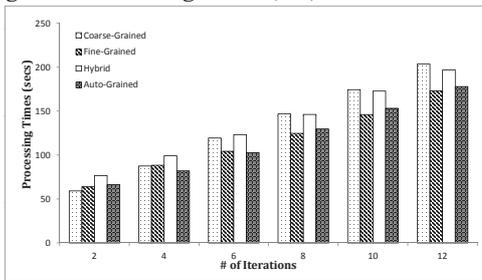
Our first experiment evaluates the performance of grid aggregation methods over skewed data and the prediction accuracy of the partitioning strategy decider. Recall that the relative performance



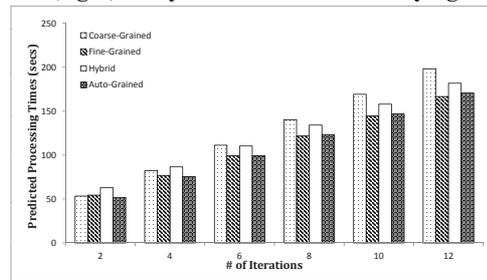
(a) Processing Times



(b) Predicted Processing Times

**Figure 3: Processing Times (left) and Predicted Processing Times (right) on Synthetic Datasets of Varying Skew**

(a) Processing Times



(b) Predicted Processing Times

**Figure 4: Processing Times (left) and Predicted Processing Times (right) on Cloud Pressure Dataset**

of different partitioning strategies and our cost models are primarily impacted by two factors: the amount of computation involved in the aggregation and the level of data skew. Thus, we designed experiments to vary both, and compared the performance of different partitioning strategies and the prediction accuracy of our models. Note that all the processors we used in this set of experiments were on the different machines.

We varied skew as follows. We designed synthetic datasets focusing on execution on 2 processors, where we applied a value-based predicate to filter out 50% of the data elements in each dataset. We created four synthetic datasets with different distributions of the data after filtering over the two partitions: 1) *very low skew*: 24% in the first half, and 26% in the second half; 2) *low skew*: 20% in the first half, and 30% in the second half; 3) *high skew*: 10% in the first half, and 40% in the second half; and 4) *extremely high skew*: 0% in the first half (empty), and 50% in the second half (full).

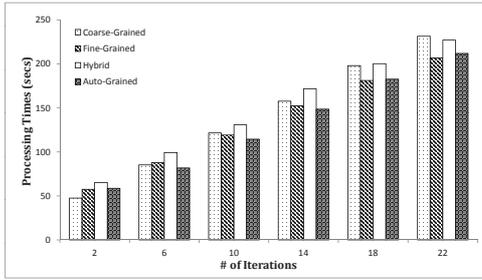
Also, the amount of computation in aggregation was varied in the following fashion. First, note that while calculating traditional aggregates like COUNT and AVG, the dominant cost is I/O, as the amount of computation is very small. Because I/O costs can be trivially balanced across processors with no impact from skew, we have used a user-defined and compute-intensive aggregation function - where data within each grid was clustered using the popular k-means algorithm. The amount of computation per element was further varied by using different number of iterations (before convergence) in the k-means algorithm.

The first set of experiments were with synthetic datasets of varying skew. The grid size was varied from 2,000K to 6,000K elements, and we set 30 centroids and 4 iterations for each grid. Figures 3(a) and 3(b) show the actual processing times and the predicted processing times from our models, respectively. With very low skew, the best results were obtained from the coarse-grained scheme. As the data skew increased, the performance of the coarse-grained partitioning began to get worse, and the best results were obtained from fine-grained and auto-grained schemes. As also predicted by our cost model, the processing costs with fine-grained, hybrid, and auto-grained schemes do not vary with skew.

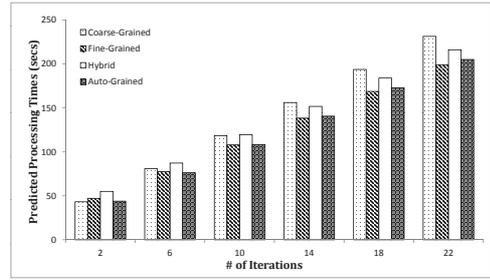
Second, we experimented on both the cloud pressure and the ter-

rain pressure datasets with 4 processors. The number of iterations for clustering was varied from 2 to 12 for the cloud pressure, and from 2 to 22 for the terrain pressure, with increasing the processing time per element. The grid size was varied from 2,000K to 6,000K elements, and we set 20 centroids for each grid. For the cloud pressure dataset, we filtered out all the missing values. This introduced significant skew - specifically, among the 4 partitions obtained using the coarse-grained partitioning, the proportions of data after filtering were 55%, 100%, 100%, and 56%, respectively. For the terrain pressure dataset, we filtered out all the values less than a threshold, and the fraction of the data in all the 4 partitions were 65%, 43%, 40%, and 62%, respectively. Figures 4(a) and 4(b) show the results on the cloud pressure dataset, and Figures 5(a) and 5(b) show the results on the terrain pressure dataset. The observations from these charts are as follows. When there are only 2 iterations on both datasets, the I/O cost is the dominating cost. As a result, the coarse-grained partitioning gives the best performance, because it minimizes disk seek times and hence has the best I/O performance. When more iterations are involved, the computation cost begins to dominate the total processing cost, and skew potentially impacts workload balance. Therefore, both the fine-grained and auto-grained schemes result in better performance, as they are able to load balance even in the presence of skew. Additionally, we can also see that when the number of iterations is 10 in Figure 4(a) and 22 in Figure 5(b), the hybrid partitioning can outperform the coarse-grained partitioning, because the former can handle skew better. Lastly, when neither I/O cost nor computation cost overwhelmingly outweighs the other, the auto-grained partitioning will have the best performance. This is because it has a load balancing advantage over the coarse-grained partitioning strategy, and it has better I/O performance than the fine-grained and hybrid partitioning strategies.

We can also see that the predicted cost follows actual processing times, and in almost all cases, it can predict the relative performance of different partitioning strategies accurately. The only case where the model is not correct in predicting performance is when the performances of two strategies are very close. This is because it is a simple model, which has not considered factors like



(a) Processing Times



(b) Predicted Processing Times

**Figure 5: Processing Times (left) and Predicted Processing Times (right) on Terrain Pressure Dataset**

communication cost (this cost is trivial only with an exception for fine-grained partitioning with a very small grid size). Overall, we have shown that the model can help choose the appropriate partitioning strategy for a given query, in view of the dataset skew and amount of computation involved per element in the aggregation.

Additionally, two issues from this experiment need to be clarified. First, to demonstrate the predication accuracy, we have focused on the “crossover” of the performance of different partitioning schemes. These results may give the appearance that the best scheme here might only marginally outperform the others. However, if we were to further increase the amount of computation, it turns out that the skew-tolerant partitioning schemes (fine-grained and auto-grained schemes) could outperform the coarse-grained scheme by more than a factor of 2. Second, for handling data skew, the fine-grained partitioning might appear to be sufficient, which may seem to imply that the other schemes are unnecessary. However, two disadvantages of the fine-grained scheme are not apparent from our experiments: 1) this scheme can incur significant communication costs when the queries involve a small grid size, and 2) as mentioned in Section 5, this scheme does not work for chunked array storage.

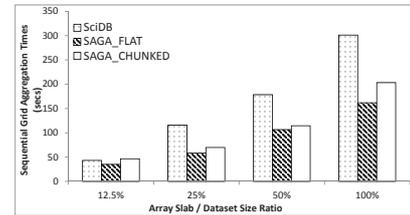
### 6.3 Performance Comparison with SciDB

Our next experiment compared the performance of our system with SciDB, a popular Array DBMS that could support many of the operations we support. We performed comparison on an EC2 instance with 8 cores. As we have discussed throughout, SciDB involves a considerable upfront cost of data transformation and reloading. Our figures only report SciDB’s query processing times after all the data was loaded, i.e., *excluding the data ingestion time*, since our goal is to show that our approach of using array storage as a DB can still allow efficient structural aggregations over array data. It should be noted that the data ingestion time for the dataset we have used could easily take 100x the time for executing a simple query over the same amount of data - this included the time for the transforming HDF5 datasets into CSV files and casting the temporary 1D arrays within SciDB into 2D arrays.

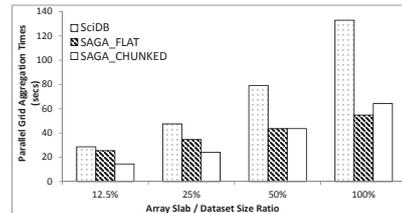
Because SciDB does not support hierarchical or circular aggregation, we only compared our performance of grid and sliding aggregations (referred to as window aggregations in SciDB). We used the 8-GB cloud pressure dataset, and we evaluated the performance of our system on both the default flat array storage and the chunked storage (with chunk size set as  $1K \times 512$ , also equal to the chunk size used when the dataset was loaded into SciDB). Note that this chunk size (4 MB) matches SciDB recommended chunk sizes (4 - 8 MB)<sup>1</sup>.

To thoroughly evaluate the performance differences between the two systems, we created queries using different aggregation operators and varied different parameters like the grid size and the size

of the array slab. Specifically, the grid size was varied from 32K to 256K elements for grid aggregation and from  $3 \times 3$  to  $7 \times 7$  for sliding aggregation. We also aggregated over array slabs of varying sizes, with ratio between the queried array slab size and the dataset size varied from 12.5% to 100%. With our system, the coarse-grained partitioning was used for grid aggregations, and the all-reuse approach was used for sliding aggregations, since they gave the best performance on this dataset and the queries we used.



(a) Sequential Performance Comparison



(b) Parallel Performance Comparison

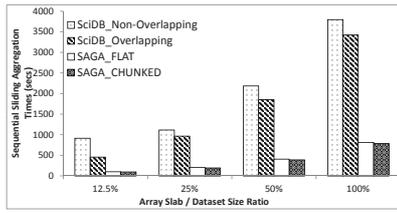
**Figure 6: Grid Aggregation Performance Comparison**

**Performance Comparison of Grid Aggregation:** The comparison results of grid aggregations are shown in Figure 6. Since we did not observe any measurable difference in relative performance across different aggregation operators or grid sizes, we report only a single (average) processing time with each approach, and each ratio between array slab size and dataset size. We can see that our system had better performance irrespective of whether the array storage is flat or chunked, or whether the queried array slab is a subarray or the entire array. As shown in Figure 6(a), the sequential performance of our system is better by an average of 39% and 25% on flat arrays and chunked arrays, respectively. To the best of our understanding, this difference arises because our system loads grids or chunks in order, leading to better I/O performance, whereas SciDB loads and processes chunks in random order and does not preserve sequential I/O.

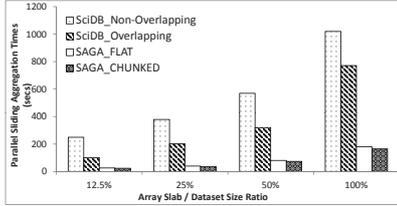
Moreover, as shown in Figure 6(b), the parallel performance of our system is also superior to SciDB. We believe SciDB scales less well mainly due to the nontrivial overhead of process scheduling. Note that although our sequential performance on chunked array is worse than the performance on flat array, due to the overhead of addressing the mismatch between the logical grid layout and the phys-

<sup>1</sup><http://www.scidb.org/forum/viewtopic.php?f=11&t=334>

ical chunk layout, processing queries on chunked array is likely to provide better scalability especially for a data subset.



(a) Sequential Performance Comparison



(b) Parallel Performance Comparison

**Figure 7: Sliding Aggregation Performance Comparison**

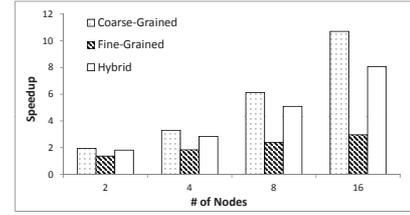
**Performance Comparison of Sliding Aggregation:** Some additional background is important before we elaborate on the performance difference for sliding aggregations. SciDB potentially simplifies sliding aggregations by replicating or overlapping chunk boundary elements while loading data [28]. The number of elements to be replicated across neighboring chunks should be set by the user at the time when the data is loaded into SciDB. Assuming sufficient number of boundary elements have been overlapped, sliding aggregation results can be obtained by processing data from each chunk independently, or what we will refer to as *aggregation over overlapping chunks*. If such chunk overlap has not been set while loading data, or if sliding aggregation with a grid size greater than the chunk overlap is performed, SciDB has to stitch neighboring chunks together when processing chunk boundary elements (*aggregation over non-overlapping chunks*), and this results in considerable performance loss. Because all possible queries and proper chunk overlap size may not be known before the data is loaded into databases, both aggregations over overlapping chunks and non-overlapping chunks are likely, and we have evaluated both of them. Note that overlapping chunked array storage is not supported by popular libraries for scientific datasets, like NetCDF or HDF5.

As we can see in Figure 7, our system outperforms SciDB with sliding aggregation over non-overlapping chunks (average of 83% for sequential performance and 88% for parallel performance), and even outperforms SciDB over overlapping chunks by an average of 79% for both sequential and parallel performance. To the best of our understanding, such performance difference is because of our all-reuse approach, which can provide completely sequential I/O and significantly reduce cache misses.

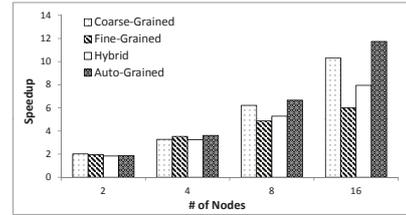
## 6.4 Performance of Parallel Structural Aggregations

**Parallel Performance of Grid Aggregations:** To evaluate our algorithm, we considered all four partitioning strategies and used the terrain pressure dataset, with both I/O bound and CPU-bound grid aggregations. The grid size was varied from 8K to 256K elements in the I/O bound aggregation that calculates traditional aggregates, and it was varied from 1,000K to 8,000K elements in the CPU-bound aggregation (k-means clustering, with 20 centroids and 20 iterations for each grid). The number of nodes we used varied from

1 to 16. Note that since there is no workload imbalance in the I/O bound aggregation, the auto-grained partitioning strategy becomes equivalent to coarse-grained one and is not shown here.



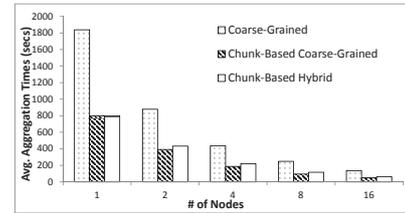
(a) I/O Bound Grid Aggregation



(b) CPU Bound Grid Aggregation

**Figure 8: Parallel Performance of Grid Aggregations Implemented by Different Partitioning Strategies**

Figures 8(a) and 8(b) show the results of I/O bound and CPU bound parallel grid aggregations, respectively. Not surprisingly, for I/O bound parallel grid aggregations, the coarse-grained partitioning results in the best performance, as it preserves better data contiguity. Both coarse-grained and hybrid schemes have better scalability than the fine-grained scheme, because of the lower communication costs for each. For CPU bound parallel grid aggregations, as the number of nodes grows, both coarse-grained and auto-grained schemes have better scalability because the grids are contiguous in all partitions. Again, not surprisingly, auto-grained scheme which explicitly accounts for skew has the best performance with 16 nodes. Overall, we can see that the best scheme has achieved at least 75% parallel efficiency.



**Figure 9: Parallel Performance of Grid Aggregations with and without Chunk-Based Optimization**

**Chunk-Based Optimization Evaluation:** To evaluate the optimization for chunked array storage, we compared the performance of (I/O bound) grid aggregations with chunk-based optimization against the ones without such optimization. As mentioned earlier in Section 5, to avoid redundant chunk read by multiple processors, the fine-grained partitioning is not supported over chunked arrays. Thus, here we only compared the chunk-based grid aggregations implemented by coarse-grained and hybrid partitioning, with the original grid aggregation implemented by the coarse-grained partitioning, which has proved to have the best I/O performance on flat arrays. We transformed both two real datasets to use chunked storage. We used the same I/O bound grid aggregation parameters on both two real datasets in Section 6.4 with up to 16 nodes. The chunk size was  $64K \times 1$ .

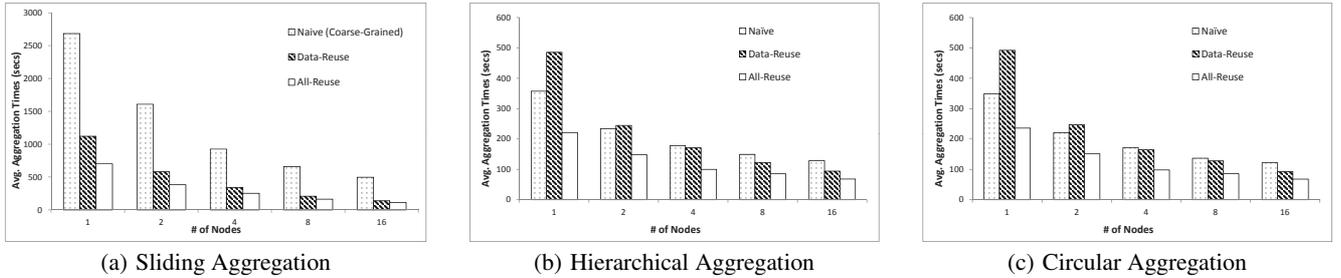


Figure 10: Parallel Performance of Overlapping Aggregations

As Figure 9 shows, the chunk-based grid aggregations implemented by both coarse-grained and hybrid partitioning can outperform the original implementation with the coarse-grained partitioning, by a factor of up to 2.75 and 2.30, respectively. This is because that, the original implementation allows different subsets of a data chunk to be read by different processors, and hence the same chunk is very likely to be loaded by multiple processors redundantly. In contrast, with our chunk-based optimization, each array chunk is loaded by exactly one processor, leading to better I/O performance. Note that the results here are not in contrast to the results in Figure 6. This is because that, although we can improve the grid aggregation performance over the chunked storage with our optimized algorithm, it is still worse than the performance over the flat storage, due to the extra overheads of handling the mismatch between the logical grid layout and the physical chunk layout.

**Parallel Performance of Sliding Aggregation:** We experimented on the two real datasets to evaluate performance of sliding aggregation algorithms, i.e., the naive approach, the data-reuse approach, and the all-reuse approach. The number of nodes we used was varied from 1 to 16. The sliding grid size was varied from  $3 \times 3$  to  $7 \times 7$ . We used the coarse-grained partitioning strategy for the naive approach.

Figure 10(a) shows the results. We can see that the aggregations implemented by the naive approach is dramatically slower than the other two approaches, as we will expect. The all-reuse approach can outperform the data-reuse approach by a factor of 37.5%, 33.9%, 26.1%, 21.5% and 17.7%, for 1, 2, 4, 8 and 16 nodes, respectively, because it is more cache-friendly.

**Parallel Performance of Hierarchical and Circular Aggregations:** The last experiment evaluates the parallel performance of hierarchical and circular aggregations, implemented by the naive approach, the data-reuse approach, and the all-reuse approach. We experimented on the two real datasets. Up to 16 nodes were used for these two aggregations. The size of innermost grid was  $512K \times 512$ , and the grid expanded by 32K in the first dimension and 32 in the second dimension in each step, with a total of 16 concentric grids in all.

Figures 10(b) and 10(c) show the parallel performance of hierarchical and circular aggregations, respectively. Again, we can see that the all-reuse approach leads to the best performance, for the same reasons we had discussed earlier. However, unlike the previous results, we can see that the naive approach can even outperform the data-reuse approach at the beginning. This is because, unlike sliding aggregation, with hierarchical and circular aggregations, there are not too many frequent and small I/O requests even in the naive approach. Further, for the naive approach, the data loaded for each grid is always contiguous in memory, and the data will be sequentially accessed by cache during aggregation.

## 7. RELATED WORK

We now compare our work with existing efforts on array database systems, array query languages, work on parallel aggregations in

databases, and closely related techniques for handling data skew.

SciDB [8] is a popular Array DBMS which has drawn considerable attention recently. We have conducted extensive performance comparison between our system and SciDB, and demonstrated that we can process structural aggregations faster, in addition to alleviating the need for expensive data ingestion steps. RasDaMan [7] is another robust system, which is based on an array algebraic framework [6]. ArrayDB [30] is a prototype array database system, which is mainly used for processing small 2D images. MonetDB [43] is a column-store DBMS for spatial applications. RasDaMan, MonetDB, and SciDB all support certain structural aggregations similar to our system.

Query languages or operators for array operations have been studied by the above efforts, as well as a number of other projects. RasDaMan uses RasQL [7], SciDB [8] supports both an SQL-like query language AQL (Array Query Language) and a functional language AFL (Array Functional Language), whereas MonetDB initially used RAM [42], and now uses SciQL [51]. In other efforts, the open source Postgres database [2] contains an array data type, with a fairly extensive language binding to SQL. AQL [27] involves a nested relational calculus to manipulate arrays. AML [29] is a generic array query language, which takes bit patterns as parameters for all the array operations. AQuery [26] makes use of ordered relational tables to process natural order-dependent queries, specifically over 1D arrays. Others projects have also made similar proposals [10, 11, 33, 31].

Parallelization of structural aggregations has already been carried out in context of RasDaMan [7] and SciDB [8]. Prior to that, Shatdal and Naughton developed parallel algorithms for value-based aggregations in relational databases [36]. Our all-reuse approach for overlapping aggregations has some similarities to the aggregations at multiple granularity on data cube [15], because such multi-level aggregation also involves overlaps among aggregated regions. However, the difference is that our method accumulates associated aggregation results concurrently, while OLAP multi-level aggregation executes in a bottom-up fashion.

Load imbalance caused by data skew has been recognized as an important issue in supporting array operations. Since prior systems involved a data ingestion phase, they addressed the problem by reorganizing data in a way that impact of skew was minimized [38, 37, 34]. Particularly, the techniques included storing the array in irregular chunks of unequal sizes but with the same amount of data, or shuffling the chunks to randomly distribute them across processors. Data skew problem has recently been addressed in the context of MapReduce also. As also summarized by Kwon *et al.* [25], static optimizers [23, 18] leverage cost models to balance the expected processing cost, and dynamic optimizers [24, 4, 44] detect and repartitions the bottleneck tasks at runtime. Other efforts [50, 5] optimize MapReduce speculative execution.

Our work takes inspiration from the NoDB approach [3] and related work on automatic data virtualization [47]. Our contribution, however, is in building a database engine on top of the native array storage rather than the flat files in CSV format, which essentially

present a relational table view. We have implemented our system on top of earlier efforts that were also based on the native array storage, but supported only the selection queries [40, 45, 41]. Other projects with a similar focus include the Oracle Database Filesystem (DBFS) [22], which can extend certain database features to unstructured data storage. PostgresRaw [3] supports advanced query processing over raw files on the fly, assuming they can be mapped to relational tables. MonetDB data vault [19] also intends to support transparent access to the data from an external scientific file repository. FastBit [48] and FastQuery [12, 21] accelerate the query processing over array storage using bitmap indices. By contrast, our system can support additional types of aggregations as well as optimized aggregations over skewed or chunked data.

## 8. CONCLUSIONS

This paper has focused on providing the database-like support over native array storage, with a specific focus on implementing a number of structural aggregation operations. These operations arise in a number of scientific disciplines. We have demonstrated that by development of nuanced algorithms and cost models, efficient array operations can be supported over native array storage. Detailed performance comparisons with SciDB further confirm this - despite no data ingestion overhead, our aggregation processing costs are lower. We have also shown high parallel efficiency with our algorithms.

## 9. REFERENCES

- [1] Land Parameter Retrieval Model (LPRM) Level 2 (Swath) Collection. <http://disc.sci.gsfc.nasa.gov/Aura/data-holdings/OMI/index.shtml>.
- [2] PostgreSQL. <http://www.postgresql.org>.
- [3] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: efficient query execution on raw data files. In *SIGMOD*, pages 241–252, 2012.
- [4] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica. True elasticity in multi-tenant data-intensive compute clusters. In *SoCC*, page 24. ACM, 2012.
- [5] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *OSDI*, pages 1–16, 2010.
- [6] P. Baumann. A Database Array Algebra for Spatio-Temporal Data and Beyond. In *Next Generation Information Technologies and Systems*, pages 76–93, 1999.
- [7] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The Multidimensional Database System RasDaMan. In *SIGMOD*, pages 575–577, 1998.
- [8] P. G. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *SIGMOD*, pages 963–968, 2010.
- [9] A. Buades and B. Coll. A non-local algorithm for image denoising. In *CVPR*, pages 60–65, 2005.
- [10] J. a. P. Cerveira Cordeiro, G. Câmara, U. Moura De Freitas, and F. Almeida. Yet Another Map Algebra. *Geoinformatica*, 13(2):183–202, June 2009.
- [11] R. Cornacchia, S. Héman, M. Zukowski, A. P. Vries, and P. Boncz. Flexible and efficient IR using array databases. *VLDB J.*, 17(1):151–168, 2008.
- [12] L. Gosink, J. Shalf, K. Stockinger, K. Wu, and W. Bethel. HDF5-FastQuery: Accelerating complex queries on HDF datasets using fast bitmap indices. In *SSDBM*, pages 149–158. IEEE, 2006.
- [13] J. Gray, D. T. Liu, M. Nieto-Santesteban, A. Szalay, D. J. DeWitt, and G. Heber. Scientific data management in the coming decade. *SIGMOD Rec.*, 34:34–41, 2005.
- [14] M. Gyssens and L. V. S. Lakshmanan. A foundation for multi-dimensional databases. In *VLDB*, pages 106–115, 1997.
- [15] J. Han, M. Kamber, and J. Pei. *Data mining: concepts and techniques*. Morgan kaufmann, 2006.
- [16] A. J. Hey, S. Tansley, K. M. Tolle, et al., editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.
- [17] B. Howe and D. Maier. Algebraic manipulation of scientific datasets. *VLDB J.*, 14(4):397–416, 2005.
- [18] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi. LEEN: Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud. In *CLOUDCOM*, pages 17–24, Washington, DC, USA, 2010.
- [19] M. Ivanova, M. L. Kersten, and S. Manegold. Data Vaults: A Symbiosis Between Database Technology And Scientific File Repositories. In *SSDBM*, pages 485–494, June 2012.
- [20] M. Kersten, Y. Zhang, M. Ivanova, and N. Nes. SciQL, A Query Language for Science Applications. In *EDBT/ICDT Array Databases Workshop*, pages 1–12. ACM, 2011.
- [21] J. Kim, H. Abbasi, L. Chacon, C. Docan, S. Klasky, Q. Liu, N. Podhorszki, A. Shoshani, and K. Wu. Parallel in situ indexing for data-intensive computing. In *LDAV*, pages 65–72. IEEE, 2011.
- [22] K. Kunchithapadam, W. Zhang, A. Ganesh, and N. Mukherjee. Oracle Database Filesystem. In *SIGMOD*, pages 1149–1160, 2011.
- [23] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *SoCC*, pages 75–86. ACM, 2010.
- [24] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: mitigating skew in mapreduce applications. In *SIGMOD*, pages 25–36. ACM, 2012.
- [25] Y. Kwon, K. Ren, M. Balazinska, B. Howe, and J. Rolia. Managing skew in hadoop. *IEEE Data Eng. Bull.*, 36(1):24–33, 2013.
- [26] A. Lerner and D. Shasha. AQuery: query language for ordered data, optimization techniques, and experiments. In *VLDB*, pages 345–356, 2003.
- [27] L. Libkin, R. Machlin, and L. Wong. A query language for multidimensional arrays: design, implementation, and optimization techniques. In *SIGMOD Rec.*, pages 228–239, 1996.
- [28] Y. Liu and V. Vlassov. Replication in distributed storage systems: State of the art, possible directions, and open issues. In *CyberC*, pages 225–232. IEEE, 2013.
- [29] A. P. Marathe and K. Salem. A Language for Manipulating Arrays. In *VLDB*, pages 46–55, 1997.
- [30] A. P. Marathe and K. Salem. Query processing techniques for arrays. *VLDB J.*, 11(1):68–91, 2002.
- [31] J. Mennis and C. D. Tomlin. Cubic map algebra functions for spatio-temporal analysis. *CaGIS*, 32:17–32, 2005.
- [32] G. Planthaber, M. Stonebraker, and J. Frew. EarthDB: scalable analysis of MODIS data using SciDB. In *BigSpatial*, pages 11–19, 2012.
- [33] D. Pullar. MapScript: A Map Algebra Programming Language Incorporating Neighborhood Analysis. *Geoinformatica*, 5(2):145–163, June 2001.
- [34] B. Reiner, K. Hahn, G. Höfling, and P. Baumann. Hierarchical Storage Support and Management for LargeScale Multidimensional Array Database Management Systems. In *DEXA*, 2002.
- [35] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *ICDE*, pages 328–336, 1994.
- [36] A. Shatdal and J. F. Naughton. Adaptive parallel aggregation algorithms. In *SIGMOD*, pages 104–114. ACM, 1995.
- [37] T. Shimada, T. Tsuji, and K. Higuchi. A storage scheme for multidimensional data alleviating dimension dependency. In *ICDIM*, pages 662–668. IEEE, 2008.
- [38] E. Soroush, M. Balazinska, and D. Wang. ArrayStore: A Storage Manager for Complex Parallel Array Processing. In *SIGMOD*, pages 253–264, 2011.
- [39] M. Stonebraker, J. Becla, D. J. DeWitt, K.-T. Lim, D. Maier, O. Ratzesberger, and S. B. Zdonik. Requirements for Science Data Bases and SciDB. In *CIDR*, 2009.
- [40] Y. Su and G. Agrawal. Supporting User-Defined Subsetting and Aggregation over Parallel NetCDF Datasets. In *CCGRID*, pages 212–219, may 2012.
- [41] Y. Su, Y. Wang, G. Agrawal, and R. Kettimuthu. SDQuery DSI: integrating data management support with a wide area data transfer protocol. In *SC*, page 47. ACM, 2013.
- [42] A. R. van Ballegooij. RAM: a multidimensional array DBMS. In *EDBT 2004 Workshops*, pages 154–165, 2005.
- [43] M. Vermeij, W. Quak, M. Kersten, and N. Nes. MonetDB, a novel spatial columnstore DBMS. In *FOSS4G*, pages 193–199, 2008.
- [44] R. Vernica, A. Balmin, K. S. Beyer, and V. Ercegovac. Adaptive MapReduce using situation-aware mappers. In *EDBT*, pages 420–431. ACM, 2012.
- [45] Y. Wang, Y. Su, and G. Agrawal. Supporting a Light-Weight Data Management Layer Over HDF5. In *CCGRID*, pages 335–342, may 2013.
- [46] Y. Wang, J. Wei, and G. Agrawal. SciMATE: A Novel MapReduce-Like Framework for Multiple Scientific Data Formats. In *CCGRID*, pages 443–450, may 2012.
- [47] L. Weng, G. Agrawal, U. Catalyurek, T. Kur, S. Narayanan, and J. Saltz. An Approach for Automatic Data Virtualization. In *HPDC*, pages 24–33, 2004.
- [48] K. Wu, S. Ahern, et al. FastBit: interactively searching massive data. In *Journal of Physics: Conference Series*, volume 180, page 012053, 2009.
- [49] J. Yang and Z. Fei. Broadcasting with prediction and selective forwarding in vehicular networks. *International Journal of Distributed Sensor Networks*, 2013, 2013.
- [50] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*, volume 8, pages 29–42, 2008.
- [51] Y. Zhang, M. Kersten, M. Ivanova, and N. Nes. SciQL: Bridging the Gap Between Science and Relational DBMS. In *IDEAS*, pages 124–133, Sept. 2011.