Noname manuscript No.
(will be inserted by the editor)

Formal Representation of the SS-DB Benchmark and Experimental Evaluation in EXTASCID

Yu Cheng · Florin Rusu

Received: date / Accepted: date

Abstract Evaluating the performance of scientific data processing systems is a difficult task considering the plethora of application-specific solutions available in this landscape and the lack of a generally-accepted benchmark. The dual structure of scientific data coupled with the complex nature of processing complicate the evaluation procedure further. SS-DB is the first attempt to define a general benchmark for complex scientific processing over raw and derived data. It fails to draw sufficient attention though because of the ambiguous plain language specification and the extraordinary SciDB results. In this paper, we remedy the shortcomings of the original SS-DB specification by providing a formal representation in terms of ArrayQL algebra operators and ArrayQL/SciQL constructs. These are the first formal representations of the SS-DB benchmark. Starting from the formal representation, we give a reference implementation and present benchmark results in EXTASCID, a novel system for scientific data processing. EXTASCID is complete in providing native support both for array and relational data and extensible in executing any user code inside the system by the means of a configurable metaoperator. These features result in significant improvement over SciDB at data loading, extracting derived data, and operations over derived data.

Yu Cheng
University of California, Merced
5200 N Lake Road
Merced, CA 95343
E-mail: ycheng4@ucmerced.edu

Florin Rusu
University of California, Merced
5200 N Lake Road
Merced, CA 95343
E-mail: frusu@ucmerced.edu

1 Introduction

Scientific investigation represents an important source of Big Data. Science generate massive amounts of data through high-rate measurements of physical conditions, environmental and astronomical observations, and high-precision simulations of physical phenomena. While effectively storing the data is a challenge in itself, the main problem scientists face is how to efficiently process data in order to obtain novel insights and gain knowledge. Considering the plethora of application-specific solutions available in the scientific data processing landscape, selecting the optimal solution for a given problem is a challenging task. The lack of standardized benchmarks that allow for a principled evaluation of the available alternatives makes the selection process even more difficult.

The *Standard Science DBMS Benchmark (SS-DB)* [17] is a recent attempt to create a general benchmark for the evaluation of scientific data processing systems. Similar to other popular benchmarks, e.g., the TPC benchmark suite [4], SS-DB is modeled after a real application workload based on a complete workflow for processing astronomical images. Nonetheless, the benchmark operations are representative for a large class of scientific data manipulations. Unlike the Sloan Digital Sky Survey [35] which is targeted at a specific aspect in the scientific processing pipeline, the SS-DB benchmark encompasses a full spectrum of operations over raw and derived data. While this is an important step toward generality, it also introduces some serious problems. The uttermost limitation which hinders a broader benchmark implementation is that operations are expressed in plain language. There is no formal representation for the benchmark operations. The lack of a generally accepted formalism to represent multi-dimensional array operations – the main component of the benchmark – is a valid argument in this respect. The immediate effect is nevertheless negative – we are aware of only two implementations of the benchmark, both presented in [17] – since lack of formalization makes impossible the definition of reference implementations. Other factors that drive the community away are the original results published in [17] and the evolving state of SciDB [2]—the reference system for the benchmark. The original benchmark results compare SciDB to a relational-based implementation on top of MySQL database. The difference between the two systems is enormous – 1 to 3 orders of magnitude – in favor of SciDB, mostly due to architectural differences and the inefficient mapping of arrays on top of relations in MySQL. The extraordinary SciDB performance discourages others from implementing the benchmark. Moreover, SciDB is only a prototype suffering considerable modifications from one version to another. These propagate to frequent modifications to the SciDB implementation of the benchmark resulting in frequent updates to the reference benchmark results.

As illustrated by the SS-DB benchmark, scientific data have dual structure. Raw data are ordered multi-dimensional arrays while derived data are best represented as unordered relations. At the same time, scientific data processing requires complex operations over arrays and relations. These operations cannot be expressed using only standard linear and relational algebra operators, respectively. Existing scientific data processing systems address only a subset of these requirements. They are typically

designed for a single data model, e.g., multi-dimensional arrays in SciDB, or they can handle complex processing only at the application level.

EXTASCID (EXTensible system for Analyzing SCIentific Data) [13, 15, 14] on the other hand is a complete and extensible system for scientific data processing. It supports natively both arrays as well as relational data. Complex processing is handled by a metaoperator that can execute any user code. EXTASCID provides unlimited extensibility by making the execution of arbitrary user code a central part of its design through the well-established User-Defined Aggregate (UDA) mechanism. As a result, EXTASCID supports in-database processing of full scientific workflows over both raw and derived data. Given all these desirable features provided in EXTASCID and the generality of the SS-DB benchmark, it is natural to ask what is the performance of EXTASCID on the SS-DB benchmark? Can all the complex benchmark operations be executed inside EXTASCID without moving data in the application layer? And how does the performance compare to SciDB?

In this paper, we address the aforementioned shortcomings of the SS-DB benchmark and answer the questions on the generality and performance of EXTASCID. Our end goal is to propose a sound formal representation for the benchmark operations together with a reference EXTASCID implementation and reference results. On one hand, this strengthens dramatically the relevance of the benchmark and enforces its position as the reference benchmark for scientific data processing. Given that no alternatives exist, a broad acceptance of the SS-DB benchmark fills an important void in the evaluation of a large class of Big Data applications. On the other hand, the EXTASCID implementation of the benchmark provides another reference point in the evaluation of scientific data processing systems. The results prove that the integrated EXTASCID architecture supporting natively both arrays and relations is more suited for complex scientific processing over raw and derived data requiring a high degree of extensibility. To this end, our contributions can be summarized as follows:

- We give a formal representation of the SS-DB benchmark in terms of ArrayQL algebra operators [27]. We provide statements for the benchmark queries in the ArrayQL [26] and SciQL [38] query languages. These are the first formal representations of the SS-DB benchmark. They can be used as reference for implementation in other systems.
- We present the design and implementation of EXTASCID—a novel system for scientific data processing. EXTASCID is complete in providing native support both for array and relational data and extensible in executing any user code inside the system by the means of a configurable metaoperator.
- We provide a reference SS-DB implementation in EXTASCID starting from the formal representation of the benchmark in ArrayQL algebra.
- We present results obtained by executing the SS-DB benchmark in EXTASCID. These are only the third reported results in the short history of the benchmark. When compared to the SciDB reference results, EXTASCID provides considerable improvement at data loading, extracting derived data, and operations over derived data. This is significant considering the initial performance gap between SciDB and other scientific data processing systems.

The rest of the paper is organized as follows. Section 2 presents array algebra formalisms and array query languages used to represent the SS-DB benchmark op-

erations which are described in detail in Section 3. The EXTASCID design and implementation are introduced in Section 4. The SS-DB implementation in EXTASCID starting from the formal array algebra representation is given in Section 5 while the benchmark results and the comparison with SciDB are presented in Section 6. Related work is presented in Section 7. We conclude in Section 8.

2 Array Query Languages

In order to analyze the SS-DB benchmark specification, we have to represent the benchmark operations in a formal query language. While relational algebra and SQL are standard formalisms for unordered relational data, there is no such algebra or query language commonly accepted for ordered array data. As a result, we settle for ArrayQL algebra [27] and ArrayQL [26] as our array algebra and query language, respectively. There are two reasons for our choice. First, these two formalisms are the most recent proposed in the literature. And second, they are part of the SciDB ecosystem [34], similar to SS-DB. We discuss alternative array algebra formulations and query languages in the related work.

2.1 ArrayQL Algebra

Arrays are formalized as 3-tuples of the form `(box, valid, content)`, where `box` represents the domain of the array with fixed bounds on all dimensions, `valid` is a boolean map indicating which cells have valid values, and `content` is a function providing the values for the array cells. This is the first algebra that represents cell validity explicitly. The benefit is that both dense and sparse arrays can be formalized within the same algebra constructs.

Given the representation of an array as a 3-tuple, a new array is created by each operator, with a corresponding new 3-tuple. Operators define mappings between the original 3-tuple components and the new components. Without going into details, we present the most important operators defined in ArrayQL algebra in the following:

- `SHIFT` array origin to a new position by changing the domain of the array components accordingly. It is useful when moving between coordinate systems.
- `REBOX` changes the dimension sizes. It can either clip or extend the array domain. `REBOX` implements subsampling or range queries over dimensions, one of the most important array operations.
- `FILTER` invalidates some array cells based on a content-only predicate. It is the direct equivalent of selection from relational algebra.
- `FILL` transforms all the invalid cells to valid and assigns them a default value. Essentially, `FILL` transforms a sparse array into a dense one.
- `APPLY` applies a function to each valid cell of an array.
- `COMBINE` combines the content of two arrays having the same shape, but not necessarily the same validity. The content of the new array is computed by a function over the content of the argument arrays.

- INNERDJOIN and INNEREJOIN are join operators over dimensions, and dimensions and attributes, respectively. Their semantics is equivalent to the natural join operator in relational algebra.
- REDUCE generates a reduced version of an array by aggregating over one or more dimensions. Supported aggregate functions include the standard SQL aggregates.

While ArrayQL algebra allows for a large variety of array operations to be expressed, there is an important feature that is completely missing from the algebra. This is the notion of adjacency or cell neighborhood. There is no operator that allows for aggregate functions to be applied to multiple adjacent cells centered on all the valid cells in the original array. While this operation can be expressed as a series of REBOX and REDUCE operators applied at all the cells in the original array, we argue that it is common enough to deserve an operator by itself.

An operator that handles adjacency is APPLY from AML [28]—this is a generalization of APPLY from ArrayQL algebra. We name this operator APPLY+ to avoid confusion. The argument function is defined over an array shape and generates as output another shape. It is applied to every cell in the input array—shape centered on each cell, to be precise. The most common case is when the output array has exactly the same shape as the input array. In this case, the output shape is a single array cell and there is direct correspondence between the origin cell in the input array and the output cell. The main difference from APPLY is that the value of the output cell is a function of multiple adjacent cells in the input array. Moreover, APPLY+ can specify which cells in the input array are considered as origin cells. In this situation, the output shapes are concatenated following the order in the input array.

2.2 ArrayQL

ArrayQL is an array creation and query language based on ArrayQL algebra. It is highly reminiscent of SQL and contains only two statements—CREATE ARRAY to create arrays at the schema level and SELECT FROM to query arrays. ArrayQL queries take as input arrays. The output can be either a new array – with dimensions specified explicitly in the query as brackets – or a relation—without any ordering constraint. Ranges on dimensions can be specified both for the input and the output arrays. In the case of input arrays, ranges correspond to sub-arrays, while in the case of the result array, ranges implement the SHIFT operator. If no ranges are provided, the complete dimension ranges of the input array(s) are automatically inherited. Structural joins between two arrays are specified by enumerating the arrays in the FROM clause and matching the dimension names. Overall, algebra operators are mostly implemented through index mappings. Not all ArrayQL algebra operators are specified in the language though. And not all the operations possible in the language by means of intricate index mappings are part of ArrayQL algebra.

SciQL [38] is a direct precursor of ArrayQL with almost identical syntax. It has a very important feature not present in ArrayQL though. It supports the APPLY+ operator as structural grouping. Thus, whenever the benchmark operations require it, we use SciQL queries instead of the less expressive ArrayQL.

3 SS-DB Benchmark

The SS-DB benchmark [17] is modeled based upon a real workflow for processing astronomical images. Although application-specific, SS-DB includes a full spectrum of operations over raw and derived data representative across various scientific domains. Queries in SS-DB are on 1-D arrays (e.g., polygon boundaries), dense and sparse 2-D arrays (e.g., images and astrophysical objects), and 3-D arrays (e.g., trajectories in space and time). Raw data ingestion and the computation of derived data are also part of the benchmark. In the following, we provide a detailed description of the SS-DB benchmark components and operations. Our contribution is to provide a formal representation for the benchmark operations based on the ArrayQL algebra and query language presented in Section 2—the original specification in [17] is in plain language. The abstract benchmark representation simplifies the understanding considerably and provides a clear specification for implementation in other systems.

3.1 Raw Data

The basic data element is represented by a 2-D grid, i.e., dense array, corresponding to a sky image. The default size of the grid is a configurable parameter. The origin of the grid lies on a large 2-D plane $\left(10^8 \text{ X } 10^8\right)$ corresponding to the entire sky. The origin has a higher chance to be placed towards the center of the domain than at other position in the space. This results in a dense region of grids lying in the central region of the domain and sparse everywhere else. Each grid cell contains 11 integer values corresponding to a set of measurements taken at that point. The distribution of the values is chosen to reflect as close as possible real scientific data. An instance of the benchmark consists of multiple grids spread across the domain. They are grouped into cycles according to the time when the image was taken, thus introducing a third dimension. In essence, the complexity of the benchmark is given by the size of the grid and the number of cycles, with more and larger grids corresponding to more difficult benchmarks.

As a concrete example, consider the specifics of the normal SS-DB instance. The size of each grid is $7,500 \text{ X } 7,500$. There are a total of 400 grids in the dataset, grouped into cycles of 20, for a total of 20 cycles. The overall size of the dataset is approximately 1 TB—each grid is 2.48 GB.

The ArrayQL definition for the raw dataset is:

```
CREATE ARRAY images (
   img_id INTEGER DIMENSION [0:399],
   x INTEGER DIMENSION [0:7499],
   y INTEGER DIMENSION [0:7499],
   v1 INTEGER, ..., v11 INTEGER
)
```
(1)

It is important to notice that this representation is in the local coordinate system corresponding to each image. The origin of the images does not need to coincide.

The origin of the grids in the global coordinate system is stored in the 1-D array `image_origin`:

```
CREATE ARRAY image_origin (
  img_id INTEGER DIMENSION [0:399],
  x INTEGER, y INTEGER
)
```
(2)

`images` is a sparse array over the 3-D space $\left(400 \text{ X } 10^8 \text{ X } 10^8\right)$ with a single dense sub-array $(7,500 \text{ X } 7,500)$ at each `img_id` index in global coordinate system. The distinction between these two representations is significant for query execution. Depending on which coordinate system is used, the `SHIFT` operator in ArrayQL algebra has to be applied to change the origin of the array prior to query processing.

3.2 Derived Data

Raw data consist of dense grids with values associated to each cell in the domain. If we consider the entire 3-D space (`img_id-x-y`) though, raw data are very sparse, i.e., only a fraction of 0.5625×10^{-8} cells have values. Derived data are generated from the raw data through clustering. There are two types of clustering specified in the benchmark. Cooking, or observation creation, is local clustering inside each grid based on cell values and cell neighborhood relationships. Grouping is distance-based clustering applied to the previously obtained observations across the grids in the same cycle. It is important to notice that derived data represent sparse arrays both in the 2-D domain (`x-y`) as well as in the 3-D space (`img_id-x-y`).

3.2.1 Observations

Observations are extracted, i.e., ”cooked”, from raw data based on a user-defined function (UDF) over cell values. Intuitively, all the cells that are part of an observation satisfy two conditions: they are neighbors and the UDF holds for each individual cell. They form a cluster with a common property. As an example, consider adjacent pixels in an image with the R component in RGB having values greater than 100. The actual number of observations in a grid is strictly determined by the parameters of the UDF. It is likely though that only a small number of cells are part of observations. To enforce this explicitly, the benchmark imposes two conditions. It limits the size of the bounding box and the number of edges in the boundary polygon corresponding to each observation. In addition to the data corresponding to each individual cell, a series of aggregated attributes are defined for an observation: the center, the bounding box and the boundary polygon, and some additional domain-specific properties.

To understand the semantics of the cooking operation, we examine how it can be expressed as an array algebra formula—a sequence of array algebra operators, to be precise. Abstractly, cooking corresponds to the labeling operation from image processing, i.e., identify all the groups of adjacent cells, i.e., observations, satisfying some common property. In this case, the property is that the value of one attribute,

e.g., v1, is greater than a given threshold. The adjacent cells are defined as a rectangle of a configurable size and centered on each cell in the input array—the rectangle is statically specified at query time and is fixed for all the cells in the array. The result of cooking is an array with exactly the same shape and size. Cell values correspond to the unique identifier assigned to each observation. Other properties corresponding to the whole observation, e.g., center, bounding box, and boundary polygon, can also be computed once the observation is determined.

The sequence of array algebra operators – ArrayQL algebra enhanced with the generalized APPLY+ from AML – that implement cooking is the following:

1. FILTER the array with the cell predicate. valid is set to true only for the cells satisfying the predicate.
2. Assign a unique id to each cell that is still valid. This can be done using a function of the array indexes. The id is an additional attribute to the original array.
3. APPLY+ a function that sets each cell to the minimum id of all the neighbor cells. This is done for each cell in the input array. The result is a new array with the same id in the cells corresponding to an observation. To generate the entire observation, APPLY+ has to be invoked iteratively until the source and result arrays are identical—steady state.

Computing observation properties is not a straightforward algebra operation either because observations have arbitrary shapes. The following steps are executed for each observation, extracted iteratively by FILTER on the observation id:

1. INNERDJOIN is used to get the raw data corresponding to the cells in the observation.
2. REDUCE is applied to generate aggregate properties. These can be stored at all the cells that are part of the observation or only at a designated cell, e.g., the center.

Instead of providing the complete statement, we give only the SciQL query corresponding to APPLY+ for the first image in the array images since this is the defining cooking characteristic:

```
SELECT [x], [y], MIN(id)
FROM images[0]                                    (3)
GROUP BY images[0][x-1:x+1][y-1:y+1]
```

where id is the additional images attribute corresponding to the observation id. For cooking to work correctly, it is required that both the array algebra operator APPLY+ and the structural grouping in SciQL can identify the valid array cells.

3.2.2 Groups

The second form of derived data consist of groups of observations. A group contains observations from different grids in the same cycle having the centers close to each other—the centers are not required to coincide. The actual definition of closeness is specified through a UDF. Intuitively, a group can be imagined as a cluster in the 3-D space of grid cycles. There is no requirement though that the observations need to be in adjacent grids since the distance function already takes into account the distance across the time dimension, i.e., img_id. The center and bounding box of a group

are defined as in the case of observations, from the centers and the bounding boxes corresponding to member observations.

In order to write an array algebra expression for grouping, we need to get a better understanding of the operation. The important detail to remark is that the neighbors are determined according to a distance function rather than using a fixed shape centered on the observation center. Nonetheless, the neighbors can be represented as an irregular 3-D array computed based on a discretized version of the distance function. Thus, we can view grouping as a 3-D version of cooking with an irregular neighborhood shape operating on observation centers. Given an origin or reference observation, the neighborhood hypercube expands with the distance along the time dimension. To compute the group corresponding to a reference observation, the following ArrayQL algebra operators have to be invoked:

1. `APPLY+` labeling – set the observation id – to all the observations in the reference observation neighborhood. In this case, labeling is done in a single shot, not iteratively.
2. `APPLY+` the same labeling as above for all the observations labeled before—they are part of the same group. This is done step-by-step along the time dimension.
3. Once the observations in a group are determined, `REDUCE` is called on the result of the `INNERDJOIN` with the observation data corresponding to all the observations in the group to compute aggregate properties for the group.

The main difference between cooking and grouping is the irregular shape passed as argument to `APPLY+`. This difference is very significant in the case of SciQL though. Based on the examples given in [38], we argue that it is not possible to write grouping as a SciQL query because SciQL can handle only regular hypercubes.

3.3 Queries

The benchmark defines a series of nine queries—three on raw data, three on observations, and three on groups. A general characteristic across all the queries is that instead of applying them to an entire grid, they typically operate on a slab of the space which is specified as part of the query. The size and position of the slab are important parameters that control the difficulty level of the benchmark. This operation is known as *subsampling* or *range* query and is highly dependent on the storage strategy. The ideal situation is to execute all the queries by reading only the required data and nothing extra. This is hard to enforce across all the possible ranges.

Before we proceed to provide the array algebra expression for each query in the benchmark, we discuss how to handle subsampling since it is embedded in almost all the benchmark queries. Subsampling can be expressed in ArrayQL algebra by the `REBOX` operator. `REBOX` clips the original array to the query slab by effectively reducing the size of its `box`. The position of the array origin changes accordingly. If the dimension sizes have to be preserved, a second call to `REBOX` can extend the array to its original size while invalidating the cells that are not part of the subsample. Intuitively, the same effect can be achieved by a single call to `FILTER` with the range conditions on dimensions. This is not supported in the current ArrayQL algebra [27] since `FILTER` accepts conditions exclusively on the cell content and not on

dimensions. We assume that in all the queries that require subsampling, REBOX is first applied to clip the array to the query range. Thus, our array algebra expressions do not represent REBOX explicitly unless necessary.

3.3.1 Raw Data

Q1 Aggregation. "For the 20 images in each cycle and for a slab of size $[\texttt{T1}, \texttt{U1}]$ in the local coordinate space, starting at $[\texttt{X1}, \texttt{Y1}]$, compute the average value of vi for a random value of i." [17]

The ArrayQL algebra operators for the first cycle are:

```
R1 = REBOX(images, [img_id=0:19,
      x=X1:X1+T1, y=Y1:Y1+U1])                    (4)
R2 = REDUCE(R1, {img_id, x, y}, avg(vi))
```

Similar expressions can be written for the other cycles by changing the range on img_id in the REBOX operator. While Q1 supports different aggregate operators, it can be generalized further by allowing induced functions [6] over attribute values inside the aggregate. This can be done by adding a call to the APPLY operator before REDUCE:

```
APPLY(R1, f(v1, ..., v11))                        (5)
```

The ArrayQL syntax for Q1 is a simple SQL aggregation with the ranges specified after the array rather than in the WHERE clause:

```
SELECT AVG(Vi)
FROM images[0:19, X1:X1+T1, Y1:Y1+U1]             (6)
```

Q2 Recooking. "For a slab of size $[\texttt{T1}, \texttt{U1}]$ starting at $[\texttt{X1}, \texttt{Y1}]$, recook the raw imagery for the first image in the cycle with a different clustering function." [17]

The same cooking process for computing observations is applied to a slab of an image and with a different filtering condition. These are marginal modifications to the sequence of array algebra operators presented in Section 3.2.1.

Q3 Regridding. "For a slab of size $[\texttt{T1}, \texttt{U1}]$ starting at $[\texttt{X1}, \texttt{Y1}]$, regrid the raw data for the images in the cycle, such that the cells collapse $10 : 3$. All the vi values in the raw data are regridded in this process by an interpolation function." [17]

This operation modifies the size of the grid dimensions. While Q3 corresponds to image shrinking, it is equally possible to imagine a version that enlarges the grid with a specified ratio. The important aspect is that the number of cells reduces and the values in each new cell have to be determined accordingly. The standard solution is to compute the new value based on the values in adjacent cells using an interpolation function. Think of a mapping from a set of cells in the original grid to a single cell in the new grid, each making its share of contribution to the new value. In terms of array algebra operators, regridding is very similar to cooking, i.e., a neighborhood shape is applied at cell positions in the input array to generate cells in the output array. The main difference is that not all the cells in the input array are considered as origin in the case of regridding. How many and which is determined by the regrid ratio.

Out of all the array algebra formulations in the literature, only AML [28] supports cell selection through bit patterns. Thus, the same generalized APPLY+ operator used for cooking can be also used for regridding. In addition to the neighborhood shape and the aggregate function, a bit pattern identifying the cells where APPLY+ is invoked has to be specified. The bit pattern takes the form of a regular expression of 0's and 1's that is applied repetitively along the dimension domain. There is one such bit pattern for each dimension. APPLY+ is invoked only at those cells where the bit pattern is 1 for all the dimensions. For Q3, the bit pattern on both x and y is (1000000000)*. The neighborhood shape is a 10 X 10 square. The aggregate function regrid maps this square into a 3 X 3 square and specifies an analytical formula for each of the 9 cells as a function of the 100 cells in the input square. There is one such interpolation function corresponding to each attribute. The resulting 3 X 3 squares are finally concatenated to form the result array. With these parameters, the AML expression for Q3 is the following:

```
APPLY+(images, [1,10,10],
       regrid: [10 X 10] ↦ [3 X 3],          (7)
       (1000000000)*, (1000000000)*)
```

Since none of the array query languages in the literature implements the generalized APPLY+ operator – with the bit pattern selection – we argue that Q3 cannot be expressed directly as a query—only as a UDF over the entire array.

3.3.2 Observations

Q4 Observation Aggregation. "For the observations in the cycle with centers in a slab of size [T2, U2] starting at [X2, Y2] in the world coordinate space, compute the average value of observation attribute oi, for a randomly chosen i." [17]

Q4 has exactly the same array algebra representation (Eq. (4)) and ArrayQL format (Eq. (6)) as Q1 if we consider observation centers to be the only valid cells of a sparse array obs_center. The evaluation of the two queries is completely different though because obs_center is a sparse array and the range condition is given in the global coordinate space. As a result, not every image has observations in the given range—there are images that do not even overlap the range.

Q5 Polygons. "For the observations in the cycle and for a slab of size [T2, U2] starting at [X2, Y2] in the world coordinate space, compute the observations whose polygons overlap the slab." [17]

This query is similar to Q4 with the difference that instead of requiring the observation center to be contained in the slab, the query considers the observations for which the boundary polygon intersects with the slab. An alternative is to consider the bounding box instead of the polygon.

Consider observations to be represented as the only valid cells of a sparse array obs. The observation id is the single value stored in each cell that is part of an

observation. The ArrayQL algebra expression for Q5 can then be written as:

```
R1 = REBOX(obs, [img_id=0:19,
      x=X2:X2+T2, y=Y2:Y2+U2])
R2 = REDUCE(R1, {img_id, x, y},
      count distinct(obs_id))
```
(8)

Since the ArrayQL syntax is identical to Eq. (6) – the only difference is that `COUNT DISTINCT` is used instead of `AVG` – we do not include the query statement.

Q6 Density. "For the observations in the cycle and for a slab of size $[\texttt{T2}, \texttt{U2}]$ starting at $[\texttt{X2}, \texttt{Y2}]$ in the world coordinate space, group the observations spatially into `D4` by `D4` tiles, where each tile may be located at any integral coordinates within the slab. Find the tiles containing more than `D5` observations." [17]

It is not clear from the query definition when an observation is considered to be part of a tile—if the observation center is contained in the tile or if any observation cell is part of the tile. We consider the first version and use the array corresponding to observation centers in the array algebra expressions. As with all the other queries that require access to neighboring cells, Q6 cannot be expressed using only ArrayQL algebra operators. The generalized `APPLY+` operator has to be used for grouping at every cell with a neighborhood shape $\texttt{D4} \times \texttt{D4}$ having the upper left corner at the cell. With this extension, the ArrayQL algebra representation is:

```
R1 = REBOX(obs_center, [img_id=0:19,
      cx=X2:X2+T2, cy=Y2:Y2+U2])
R2 = APPLY+(R1[i], [1,D4,D4],
      count(center) AS density)
R3 = FILTER(R2, density ≥ D5)
```
(9)

Notice that `R2` and `R3` are computed separately for each image $i = [0 : 19]$ in the cycle. In SciQL, they correspond to structural grouping followed by `HAVING`:

```
SELECT cx, cy, COUNT(center) AS density
FROM obs_center[i][X2:X2+T2][Y2:Y2+U2]
GROUP BY obs_center[1][cx:cx+D4][cy:cy+D4]
HAVING density ≥ D5
```
(10)

3.3.3 Groups

Q7 Centroid. "Find each group whose center falls in the slab of size $[\texttt{T2}, \texttt{U2}]$ starting at $[\texttt{X2}, \texttt{Y2}]$ in the world coordinate space at any time `t`. The center is defined to be the average value of the centers recorded for all the observations in the group." [17]

This is a 3-D query over an array `group_center` containing valid cells only for the group centers. One such array corresponds to each cycle. This array can be

computed after the groups are determined. With this representation, the query for the first cycle can be expressed as a simple REBOX in ArrayQL algebra:

```
REBOX(group_center, [cycle=0,
      cx=X2:X2+T2, cy=Y2:Y2+U2])
```
(11)

and in ArrayQL as:

```
SELECT group_id
FROM group_center[0,X2:X2+T2,Y2:Y2+U2]
```
(12)

Q8 Center trajectory. "Define trajectory to be the sequence of centers of the observations in an observation group. For each trajectory that intersects a slab of size $[T3, U3]$ starting at $[X2, Y2]$ in the world coordinate space, produce the raw data for a D6 by D6 tile centered on each center for all images that intersect the tile." [17]

Q8 consists of two phases. First, the groups whose trajectory intersects a given slab are determined. Second, for each grid containing an observation in the group, the cells in a given tile centered on the group center in that grid are returned. It is important to notice that Q8 – as well as Q9 – requires access both to the raw images as well as to groups.

While in Q7 the center of a group is defined as a single 2-D point over all the observations in the group, in Q8 there is a center at each image that has observations in the group—there are at most 20 centers for a group. They define the trajectory of a group as a 3-D array in the global coordinate space. Then the trajectories that intersect the query slab, i.e., at least one center is contained in the slab, can be extracted with a simple REBOX operator applied to the corresponding group_center_img array:

```
R1 = REBOX(group_center_img,
     [cycle=0, img_id=0:19,
      cx=X2:X2+T3, cy=Y2:Y2+U3])
```
(13)

For each valid cell in R1, raw data have to be extracted from the corresponding image. This can be done with REBOX on the image representation in the global coordinate space. Assuming we operate on the first image in the cycle at observation center <ocx, ocy>, the ArrayQL algebra operator sequence is:

```
R2 = SHIFT(images[0], <orig_x, orig_y>)
R3 = REBOX(images[0],
     [ocx-D6:ocx+D6, ocy-D6:ocy+D6])
```
(14)

<orig_x, orig_y> is the image origin in the global coordinate system as stored in image_origin (Eq. (2)). Since the conversion of REBOX to ArrayQL is standard (see Eq. (11) and (12)), we do not include it here.

Q9 Polygon trajectory. "Define trajectory to be the sequence of polygons that correspond to the boundary of the observation group. For a slab $[T3, U3]$ starting at $[X2, Y2]$, find the groups whose trajectory overlaps the slab at some time t and produce the raw data for a D6 by D6 tile centered on each center for all images that intersect the slab." [17]

Q9 is similar to Q8. The only difference is the array used in REBOX over the derived data. The original observations array obs containing the groups the observation is part of and shifted to the origin in the global coordinate space has to be used instead of the group_center_img array. This array contains valid cells for all the points that are part of observations. It allows us to identify the groups overlapping the query slab and their centers at each position in the cycle.

3.4 Discussion

Given the algebra representation for all the operations in the benchmark, we analyze what array structures and operators are required for an actual implementation. The 3-D array corresponding to the raw images is given in Eq. (1). images is represented in the local coordinate system. This representation suffices to answer Q1–Q3 and to execute the cooking. For Q8 and Q9, the representation of images in the global coordinate system is required. This can be obtained by shifting the origin of the images based on image_origin. For grouping, only the observation centers are required. A sparse 3-D array obs_center over the global coordinates containing valid cells only where observation centers are located can do the job. The data inside the cell contain the observation id and a set of attributes (Q4, Q6). All the cells that are part of observations are required for Q5. They can be represented as a sparse 3-D array obs with the observation id as the single attribute. This array is also required to answer Q9. It is important to notice that this complete representation of observations in terms of the member cells eliminates the need to represent bounding polygons as arrays. With this representation, intersections between polygons can be expressed as a combination of the REBOX and REDUCE operators (Eq. (8)). Group centers are required to answer Q7 and Q8. In Q7, the single group centers can be represented as a sparse 3-D array group_center containing the group ids in the valid cells. In Q8, group centers are recorded for each image in the cycle—they can be different from one image to another. Thus, a sparse 4-D array group_center_img with the group ids stored in the valid cells is required. Notice that cycle represents the additional dimension in the arrays corresponding to groups. Table 1 summarizes the arrays required to answer the benchmark queries.

Name	Type	Coordinates
images	3-D grid	local
image_origin	1-D grid	global
obs	3-D sparse	local
obs_center	3-D sparse	global
group_center	3-D sparse	global
group_center_img	4-D sparse	global

Table 1: Arrays used in the SS-DB benchmark.

In terms of the ArrayQL algebra operators (Table 2), REBOX is used in all the range queries. The generalized form of APPLY+ that is present only in AML is used

for cooking, grouping, Q2, Q3, and Q6. The calls differ in terms of the cells where APPLY+ is invoked, the shape of the neighborhood, and the aggregator. REDUCE is applied whenever an aggregate has to be computed, e.g., Q1 and Q4. FILTER, SHIFT, and INNERDJOIN are the other operators used throughout the benchmark queries, with INNERDJOIN appearing only in cooking and grouping. From a critical point of view, the SS-DB benchmark emphasizes range queries and aggregation as the most significant array operations. This is in line with the previous work on array databases. In addition to these standard operators, APPLY+ variations and INNERDJOIN provide SS-DB generality beyond any other array benchmark.

Query	**Algebra operators**
Cooking, Q2	FILTER, APPLY+, INNERDJOIN, REDUCE
Grouping	APPLY+, INNERDJOIN, REDUCE
Q1, Q4, Q5	REBOX, REDUCE
Q3	APPLY+
Q6	REBOX, APPLY+, FILTER
Q7	REBOX
Q8, Q9	REBOX, SHIFT

Table 2: Algebra operators used in the SS-DB benchmark.

4 EXTASCID

GLADE. EXTASCID is built around the massively parallel GLADE [12] architecture for data aggregation. GLADE exposes the Generalized Linear Aggregate (GLA) [12] interface consisting of four user-defined functions – Init, Accumulate, Merge, and Finalize – and is optimized for GLA execution. A GLA is an associative-decomposable aggregate function. Essentially, what this means is that the order and the groups in which data items are aggregated does not affect the final result. This in turn allows for efficient asynchronous computation in a parallel environment along an aggregation tree. While it inherits the extensibility provided by the original GLA interface implemented in GLADE, EXTASCID enhances this interface considerably with functions specific to scientific processing. This requires significant extensions to the GLADE execution strategy in order to provide additional flexibility and to optimize array processing. The design of the EXTASCID parallel storage manager with native support for relations and arrays is entirely novel—GLADE works only for relational data.

Given its descent from GLADE, EXTASCID also satisfies the standard requirements for scientific data processing—support for massive datasets and parallel processing. Contrary to existent scientific data processing systems designed for a target architecture, typically shared-nothing, EXTASCID is *architecture-independent*. It runs optimally both on shared-memory, shared-disk servers as well as on shared-nothing clusters. The reason for this is the exclusive use of thread-level parallelism inside a processing node while process-level parallelism is used only across nodes.

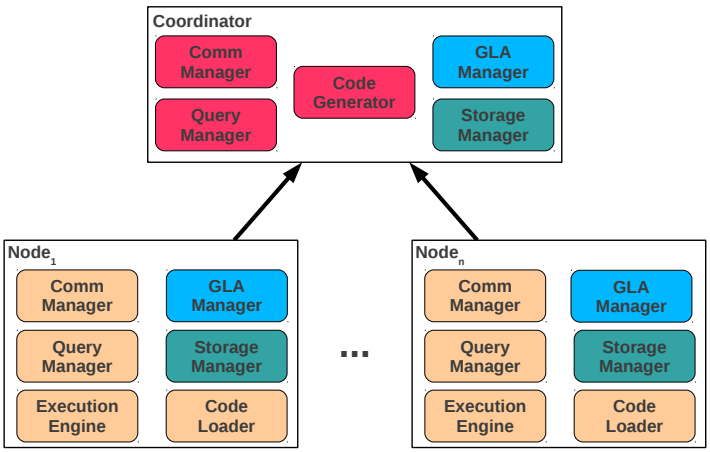

Fig. 1: EXTASCID system architecture.

EXTASCID. In a nutshell, EXTASCID is a parallel data processing system that executes any computation specified as a GLA using a merge-oriented execution strategy supported by a push-based storage manager. The storage manager is designed with special consideration for multi-dimensional range-based data partitioning in order to support efficient array processing. To allow for wide extensibility in terms of the supported user code and to extract maximum performance, GLAs are dynamically compiled inside EXTASCID at runtime following the optimized code generation mechanism proposed in the DataPath system [5].

Architecture. As shown in Figure 1, EXTASCID consists of two types of entities: a coordinator and one or more executor processes. The coordinator is the interface between the user and the system. Since it does not manage any data except the catalog metadata, the coordinator does not execute any data processing task. These are the responsibility of the executors, typically one for each physical processing node. It is important to notice that the executors act as completely independent entities, in charge of their data and of the physical resources. The coordinator as well as the executors consist of multiple components, depicted in Figure 1. While the components are inherited from the GLADE [12] architecture, significant changes are required in order to support array storage and processing in addition to the native relational data model. We discuss the changes to the two most important components – Storage Manager and GLA Manager – in separate sections. In the following, we summarize the main functionality of the other components.

Query Manager. The query manager at the coordinator is responsible for setting-up and managing the processing requested by the user. In the case of multiple executor processes, the query manager builds the aggregation tree used for merging the GLAs.

Code Generator. The code generator, represented at the coordinator in Figure 1, fills pre-defined M4 [1] templates with macros particular to the actual processing requested by the user, generating highly-efficient C++ code similar to direct hard-

coding of the processing for the current data. The resulting C++ code is subsequently compiled together with the system code into a dynamic library. This mechanism allows for the execution of arbitrary user code inside the execution engine through direct invocation of the GLA interface methods.

Code Loader. The code loader links the dynamic library to the core of the system allowing the execution engine and the GLA manager to directly invoke user-defined methods. While having the code generator at the coordinator is suitable for homogeneous systems, in the case of heterogeneous systems both the code generator and the code loader can reside at the executors.

Execution Engine. The EXTASCID execution engine is an enhanced instance of the GLADE-DataPath execution engine. It implements a series of relational operators – `SELECT`, `PROJECT`, `JOIN`, `AGGREGATE` – and a special operator for the execution of arbitrary user code specified using the GLA interface [12]. The extended version of the GLA interface we propose in this paper is depicted in Figure 3. The operators are configured at runtime with the actual code to execute based on the requested processing. The execution engine has two main tasks: manage the thread pool of available processing resources and route data chunks generated by the storage manager to the operators in the query execution plan. Parallelism is obtained by processing multiple data partitions simultaneously and by pipelining data from one operator to another.

GLA Manager. The GLA managers at the executors execute the `Merge` function – `LocalMerge` in Figure 3 – in the GLA interface, while the GLA manager at the coordinator executes `Terminate`. They are dynamically configured with the code to execute at runtime based on the actual processing requested by the user. Notice that the GLA manager merges only GLAs from different executors, with the local GLAs being merged inside the execution engine.

Communication Manager. The communication managers are in charge of transmitting data across process boundaries, between the coordinator and the executors, and between individual executors. Different inter-process communication strategies are used in a centralized environment with the coordinator and the executor residing on the same physical node and for a distributed shared-nothing system. The communication manager at the coordinator is also responsible for maintaining the list of all active executors. This is realized through a heartbeat mechanism in which the executors send *alive* messages at fixed time intervals.

4.1 Storage Manager

The storage manager is responsible for organizing data on disk, reading, and delivering the data to the execution engine for processing. There are multiple aspects that distinguish EXTASCID from traditional database storage managers. First, it supports natively relational data as well as multi-dimensional arrays. Second, and most important, the storage manager operates as an independent component that reads data asynchronously and pushes it for processing. It is the storage manager rather than the execution engine in control of the processing through the speed at which data are read from disk. And third, in order to support a highly-parallel execution engine

consisting of multiple execution threads, the storage manager itself uses parallelism for simultaneously reading multiple data partitions.

Data partitioning. Data partitioning [20] represents the main strategy for parallel data processing. In a relational setting, the tuples of a relation are split into multiple segments and assigned to different execution nodes for processing. Since each process works on a considerably smaller dataset, a speedup proportional to the number of processing nodes can be obtained in optimal conditions. In GLADE, data partitioning works as follows. The tuples of a relation are arbitrarily assigned to segments of fixed size—typically a few millions, to increase the size of sequential scans and reduce the number of seeks. This is done at loading by simply following the order in which tuples are ingested. The order of the segments on disk is again arbitrary, typically the order in which they are ingested. The assignment of segments to nodes is round-robin. The goal is to equally divide the data across nodes for load balancing. With this partitioning strategy, queries in GLADE have to always read all the data since there is no relationship between tuples and the segment they are part of. The only reduction in the amount of data read from disk is due to the vertical organization of the segments on disk which allows only for the attributes required by the query to be scanned. When the number of attributes is large, this reduction can be quite significant.

While the GLADE partitioning strategy works for relational data, it is suboptimal for array processing which often requires neighboring cells to be processed together. Specifically, the subsampling `REBOX` operator can be isolated to the segments overlapping the range selection if data are organized according to their position along the array dimensions. Moreover, the `APPLY+` operator requires adjacent data to be processed together. If data are not stored organized based on the dimensions, an expensive re-partitioning step is required as pre-processing. The EXTASCID storage manager addresses these problems and optimizes array organization while making minimal modifications to GLADE.

Chunking. Chunking [31] or tiling [22] is multi-dimensional range-based data partitioning for parallel array processing. What this means is that data having close values along the set of partitioning attributes are assigned to the same segment. For arrays, dimensions are used as partitioning attributes and the resulting data segments are called chunks or tiles. Possible chunking strategies are presented in [22,33]. Issues that need to be addressed include the shape of the chunk, the order in which to store the chunks on disk, and how to distribute chunks across processing nodes. Since GLADE already supports data partitioning for relational data, we examine how are these issues addressed for storing array data efficiently in EXTASCID.

Chunk shape. The shape of the chunk can be fixed across the entire array – regular chunking – or there can be multiple shapes, each of them containing the same number of array cells—irregular chunking [22,33]. Regular chunking is better suited for dense arrays, also known as grids, since each cell in the array contains data. The main issue with regular chunking is how to determine the optimal shape. The immediate alternative is to make the size of the chunk along each dimension proportional to the domain size of the corresponding dimension—aligned tiling [22] uses the same scaling factor across each dimension. Another alternative is to determine the shape based on the query workload as the solution to the optimization formulation that min-

imizes the overall number of chunks read from disk [31]. Irregular chunking is better suited to sparse arrays. The objective is to create chunks that contain the same number of data points rather than to have chunks with the same shape. This results in similar processing time across chunks and load balancing across processes—an important aspect for parallel processing. EXTASCID supports all these types of chunking as long as the chunks are computed by an external process and passed to EXTASCID during data loading. EXTASCID does not reorganize data once they are loaded inside the system. All the storage decisions are taken during loading. In order to change the chunking format, data have to be re-partitioned outside EXTASCID and reloaded.

Chunk ordering and distribution across disks. Once the chunk shape is determined, two additional problems require attention—how to order the chunks on disk and how to distribute the chunks across multiple processing nodes. It is important to notice that no matter what order is chosen, there will be tasks with suboptimal performance. Thus, the idea is to optimize the placement for a given workload or in the average case. Random placement of chunks on disk and across nodes is optimal in the average case. When workload information is available, the order of chunks on disk – the order in which dimensions are considered – can be chosen such that chunks that are accessed together are placed contiguously on disk. This results in larger sequential scans and fewer seeks, thus better I/O performance. Larger chunk sizes have a somehow similar effect. The assignment of chunks to nodes involves a more complicated tradeoff. On one side, i.e., subsampling, we aim for maximum parallelism. On the other, i.e., `APPLY+` operator, the amount of data transferred between nodes has to be minimized. Thus, it is not clear if chunks that are accessed together should be assigned to the same or different nodes. It depends on the actual task to be executed. The problem becomes even more complicated in EXTASCID due to the thread-level parallelism inside each processing node. In this situation, we opt for random chunk placement on disk and random chunk assignment to processing nodes as our default strategy. The user is given the possibility to change this and specify an arbitrary placement though.

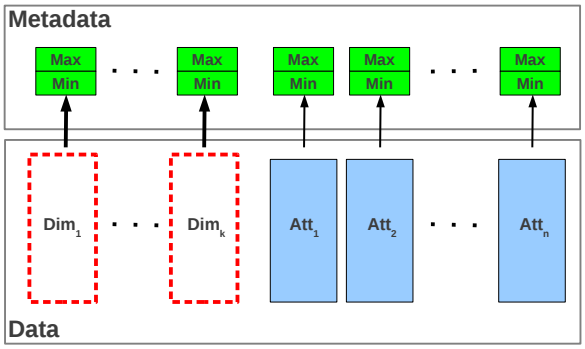

Fig. 2: Generic chunk structure for the storage of both relational data and arrays in EXTASCID.

Chunk structure. Figure 2 depicts the generic structure of an EXTASCID chunk containing metadata to support range-based data partitioning. It is important to point out that this structure is directly applicable both to relational data as well as arrays. For unordered relations, the dimensions do not exist—there are only attributes. For arrays, dimensions form a key. They have to be represented explicitly for sparse arrays, while in the case of dense arrays the dimensions can be inferred from the position in the chunk when data are stored in a pre-determined order. The metadata contain the minimum and maximum values for each dimension and attribute and are stored in the system catalog. They represent a primitive form of indexing. Different chunking strategies generate different ranges for the (Min, Max) metadata. For example, in the case of regular chunking, the (Min, Max) ranges are equal across all the chunks for all the dimensions and they represent the same fraction from the dimension size. The ranges allow for immediate detection of the chunks that need to be processed in subsampling queries—a large class in array processing. The actual data are vertically partitioned, with each column stored in a separate set of disk blocks. This allows only for the required columns to be read for each query, thus minimizing the I/O bandwidth required for processing. The impact of the (Min, Max) ranges on attributes is not that significant since there is no guarantee that attribute values are clustered. Nonetheless, if there is correlation between the cell position and its value, the (Min, Max) ranges can prune a significant number of chunks even for the value-based `FILTER` operator.

Given the generic chunk structure, it is important to determine what *optimizations* can be applied for different types of data. We are specifically interested in sparse and dense ordered arrays and unordered relations. While in the case of sparse arrays and relations there is not much beyond using the metadata to determine if a chunk is required for processing in a subsample or selection query, dense arrays provide further optimization opportunities. To be precise, the dimensions can be discarded altogether if the data inside the chunk are stored sorted along a known order of the dimensions. This optimization is known as *dimension* or *index suppression* and can reduce the amount of data read from disk even further. Notice that although index suppression reduces the amount of stored data, we do not consider it as a compression method. Compression is orthogonal to chunk organization. It can be applied at column level. Currently, EXTASCID does not support compression.

Storage manager operation. As already mentioned before, the EXTASCID storage manager supports a push-based execution model for a merge-oriented parallel execution strategy in which chunks are read from disk asynchronously and injected into the system. This is not a novel feature in EXTASCID but is rather inherited from DataPath [5] via GLADE [12]. The storage manger has to determine what chunks have to be generated for each user query. While in GLADE this decision is simple since all the chunks are read for every query, in EXTASCID the chunks required for a subsampling/selection query can be determined based on the (Min, Max) metadata, without actually reading the chunks from disk. This simple form of indexing can result in significant I/O reduction, especially for small range subsampling queries. Following the same strategy of runtime code generation, the storage manager is configured with code to select the chunks based on the query. This pre-processing step is executed during the storage manager setup phase, just before chunks are being generated. The actual process of reading and assembling chunks is highly-parallel,

with multiple requests being honored simultaneously. In essence, the storage manager is a complex module consisting of multiple components that operate in parallel and communicate asynchronously. This is completely different from ArrayStore [33] – the state-of-the-art storage manager for parallel array processing – which provides a standard pull-based interface centered around a `GetNext()` method.

4.2 GLA Execution

As identified in [32], there are only two strategies for parallel scientific processing—merge and overlap. In the merge strategy, each data partition is first processed independently by an executor process, followed by a merging phase in which the partial results are combined together. In overlapped execution, enough data are replicated across multiple partitions to isolate any computation to a single data partition, thus eliminating the subsequent merge phase. Merging is a more general strategy, applicable to any computation. Overlapping requires complicated data replication strategies and post-processing and is applicable only to bounded computations—otherwise the entire data have to be replicated at each data partition.

Merge-oriented execution. EXTASCID adopts a merge-oriented execution strategy, facilitated by the push-based storage manager and the GLA interface for complex task specification. Merging is supported by two components of the system—a GLA metaoperator that is part of the execution engine and the GLA manager. As all the other operators in the execution engine, the GLA metaoperator takes as input chunks. Unlike other operators though, its functionality is not restricted to a pre-determined template with a reduced number of configuration parameters. Instead, the GLA metaoperator can execute arbitrary user code as long as it is expressed using the GLA interface [12]. The role of the GLA manager is to merge together GLAs created at different nodes. Merging is executed on arbitrary tree structures, determined independently for each query.

The benefits of the EXTASCID execution strategy are twofold—completeness and extensibility. Since the GLA metaoperator processes chunks, it can handle both relational data and ordered arrays using the same framework. The only difference is the user code which can take advantage of the chunk structure. Extensibility is achieved by executing arbitrary user code expressed using the common GLA interface. In a typical workflow, an operation is first expressed as a GLA and executed by the GLA metaoperator. In time, the operation can be promoted to an independent operator and added to the execution engine.

Extended GLA interface. Figure 3 depicts the stages of the merging strategy expressed in terms of the extended GLA interface specific to array processing. The semantic of the standard GLA methods is presented in [12]. In the following, we focus on the array-specific GLA methods we propose. `BeginChunk` is invoked before the data inside the chunk are processed, once for every chunk. `EndChunk` is similar to `BeginChunk`, invoked after processing the chunk instead. These two methods operate at chunk granularity. They are the places where side-effect operations are executed. For example, data can be sorted according to a dimension that makes the processing more efficient in `BeginChunk`. In `EndChunk`, data that

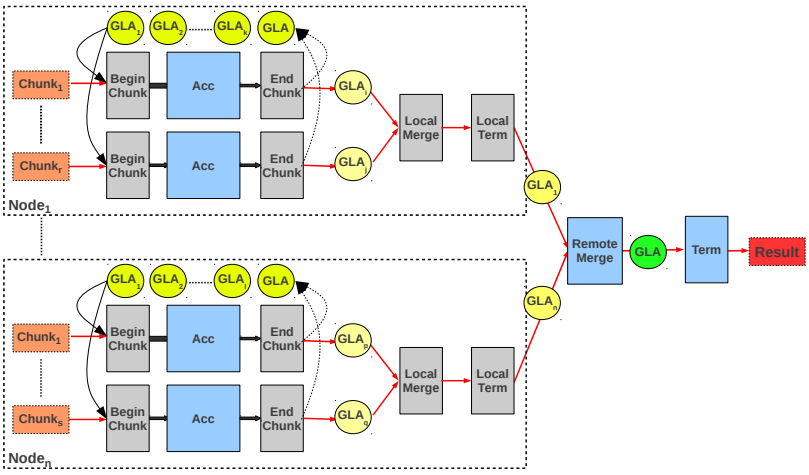

Fig. 3: EXTASCID merge-oriented execution strategy. The gray rectangles correspond to methods specific to array processing in the extended GLA interface.

are part of the GLA state and do not require further merging can be materialized to disk resulting in significant reduction in memory usage. The difference between `Init` and `BeginChunk`, and `Terminate` and `EndChunk`, respectively, is that `BeginChunk` and `EndChunk` can be invoked multiple times for the same GLA, once for every chunk. This is because GLAs are used across chunks. Merging is invoked in two places. In the GLA metaoperator, `LocalMerge` puts together local GLAs created on the same processing node, while `RemoteMerge` is invoked in the GLA manager for GLAs computed at different nodes. This distinction provides optimization opportunities depending on the chunking strategy—when chunks corresponding to the same array are stored on the same node, only `LocalMerge` is required. `Terminate` is called after all the GLAs are merged together in order to finalize the computation, while `LocalTerminate` is invoked after the GLAs at a processing node are merged. `LocalTerminate` allows for optimizations when the processing is confined to each node and no data transfer is required. It is important to notice that not all the interface methods have to be implemented for every type of processing.

4.3 Implementation

As mentioned earlier in the paper, EXTASCID is implemented on top of the GLADE parallel processing system. GLADE [12] in turn uses an extended version of the centralized DataPath [5] relational execution engine for local processing at each node. As a result, EXTASCID inherits the relational algebra operators implemented in DataPath. It also inherits the communication mechanism for parallel aggregation available in GLADE. In this section, we provide more details on how the array-specific

ArrayQL algebra operators are implemented in the DataPath execution engine. We consider parallel versions of the operators that process chunked arrays. Chunks are processed independently and in parallel, with minimal data sharing and transfer, in order to maximize the parallelism. Before discussing the array operators, we first look into the parallel implementation of the original relational algebra operators and of the GLA metaoperator.

4.3.1 Relational Algebra Operators

DataPath [5] implements four operators—`SELECTION`, `PROJECTION`, `JOIN`, and `AGGREGATE`. As with any relational database, operators can be combined into execution trees that support the execution of complex queries. The operators process chunks – a simplified version of the generic chunk structure depicted in Figure 2 – and generate chunks with a different structure or with different tuples. Inside every operator, the loop iterating over the tuples in the chunk is heavily optimized through compiler specific optimizations such as loop unrolling. This is done for every query in part by generating a hard-coded version of the operator specific to the query, compiling it, and linking it in the execution engine at runtime. The code generation process is driven by a parametrized template specific to each operator that is instantiated with query-specific arguments. This results in code that executes only operator-specific tasks. There is no tuple/attribute packing/unpacking or type conversion and value interpretation.

Specifically, `SELECT` plugs-in the selection condition in the loop and invalidates the tuples that do not satisfy the condition by resetting a corresponding bit. `PROJECT` simply drops all the columns in the chunk that are not required further up in the query tree. It is applied for any other operator, including the file-level access methods. `JOIN` is more complex since there are two stages in the processing—DataPath implements hash join under the assumption that one of the relations fits entirely in memory. In the build phase, the columns corresponding to the small relation are linearized into a hash table based on the join attributes. The hash function call is hard-coded with the specific attributes. In the probing phase, tuples from the large relation are iterated over and matched with corresponding tuples in the hash table to generate result tuples that are vertically partitioned in the chunk structure. All these operations are specific to the query at hand. In `AGGREGATE`, the function and the arguments – computed from the chunk attributes – are the parameters to the code generation template.

4.3.2 GLA Metaoperator

The GLA metaoperator takes as input chunks. It produces GLAs though. As a result, the GLA metaoperator can be placed only at the root of the query tree in the current implementation. Nonetheless, we can imagine a conversion function that transforms GLAs into chunks, thus allowing the GLA metaoperator to appear anywhere inside a query execution tree. The code generation template for the GLA metaoperator invokes the methods in the extended GLA interface. It is parametrized with the type of the GLA and the expressions the GLA is computed over. It is this additional level of indirection that allows for any user code to be injected into the system, not only

valid SQL expressions. To optimize the function call mechanism, the GLA methods are defined inline. While this increases the performance, it also represents a possible source of errors that can bring the entire system down. At the end of the day, the user is allowed to inject any code right in the heart of the system. We aim for maximum performance at the expense of safety and put the responsibility on the user.

4.3.3 Array Algebra Operators

In order to add new operators to GLADE, we first implement them as special cases of the GLA metaoperator, thus inheriting the parallel merge execution strategy. If the GLA metaoperator is used frequently enough, it can be promoted to become an independent operator with a dedicated keyword and syntactic rules in the query language. We apply this process when we implement the ArrayQL algebra operators in EXTASCID. We present the details for the operators used in the SS-DB benchmark in the following. Before we start though, we emphasize that in the case of sparse arrays – represented as relations – no modifications to the relational operators are required.

`SHIFT` only requires modifications to the chunk metadata since, when stored, the dimension indexes are relative to the chunk origin stored in the metadata. The chunk origin is stored in the global coordinate system.

`FILTER` is no different from the `SELECT` operator since the selection is on the values. Unless indexes are defined on the values, the only available solution is to scan all the tuples and check the condition for each of them—this is the DataPath solution. The primitive indexing solution provided in EXTASCID by the (Min, Max) ranges can improve dramatically upon the linear scan when the ranges are tight since a considerably smaller number of chunks have to be read from disk. If there is correlation between the position in the grid and the value, this is the case. `FILTER` is implemented at two granularity levels. The coarse grained part operates on chunks using the (Min, Max) ranges and is implemented as an access method at the file level. Only the chunks that contain at least a cell satisfying the selection condition are read from disk. The fine grained part is the standard `SELECT` operator in DataPath. It can be executed independently in parallel for each chunk read from disk.

`REBOX` is `FILTER` on dimensions. It follows the `FILTER` implementation directly, including the parallelism. Given that chunks are clustered on dimensions, it is guaranteed that the minimum number of chunks to be read from disk is always detected in the access method since the (Min, Max) ranges are compact. Moreover, the `REBOX` operator can take advantage of the chunk organization, i.e., dimension order, and identify the cells in the result box without iterating over all the cells. This represents an optimized version of `SELECT` for grid data.

`INNERDJOIN` is a structural join that ”glues” together cells at the same index in two different arrays having the same size. If the two arrays are chunked similarly, `INNERDJOIN` only requires that chunks at the same position are read into memory at the same time. This can easily be enforced in EXTASCID by specifying the same scan order for the two arrays and limiting the number of chunks that are processed inside the execution engine at any time. Chunks at the same index can be processed in parallel independently from all the other chunks. When chunks are non-aligned and have different sizes, `INNERDJOIN` becomes more complicated since a chunk from

one relation can join with many chunks from the other. In the current EXTASCID implementation, `INNERDJOIN` works only for aligned chunks. Moreover, if one of the arrays fits entirely in memory, the relational `JOIN` operator can be applied directly.

`REDUCE` is the exact equivalent of the GLA metaoperator. Thus, it is implemented as specific GLA instances parametrized with the aggregate function. A reduction across a subset of dimensions corresponds to group-by aggregation. It is implemented as a GLA that includes the grouping in all stages of the computation. Parallelism is automatically inherited from the GLA metaoperator parallel merge strategy.

4.3.4 APPLY+ Operator

At a high level, `APPLY+` requires grouping based on a user-defined neighborhood function followed by applying a user-defined aggregate function for all (or a subset of) cells in the array and their corresponding neighbors. The relational representation of `APPLY+` consists in a structural self-join based on the neighborhood function followed by a group-by aggregation using the user-defined aggregate function. The problem with this two-operator representation is caused by the standard relational `JOIN` operator which does not consider the ordered array structure. The neighborhood function can be implemented either as a nested-loop join due to the complex join condition or as a series of self-joins–one for each neighbor. Both solutions are inefficient. Consequently, `APPLY+` is implemented in EXTASCID as a single specialized instance of the GLA metaoperator—the `APPLY+` GLA.

`APPLY+` GLA works as follows. In `BeginChunk`, array cells are sorted such that they are accessed optimally in `Accumulate`. Any other pre-processing required by the neighborhood or aggregate functions, e.g., reinitialize the GLA state, is invoked in `BeginChunk`. Array cells are grouped according to the neighborhood function in `Accumulate`, while the aggregate is computed in `EndChunk`. The simplest implementation of neighborhood grouping is to assign each cell to all the groups it is part of. For SS-DB cooking, this can be further optimized based on the order in which array cells are processed in `Accumulate`. The completed aggregates can be materialized in `EndChunk` in order to reduce memory consumption.

Aggregate computation for cells that have neighbors outside of the chunk requires careful consideration. Essentially, the aggregate cannot be computed until all the cells become available in the same GLA. In EXTASCID, this is realized through the parallel merging mechanism – `LocalMerge` and `RemoteMerge` – provided by the GLA metaoperator. It is important to notice though that the amount of data transferred between GLAs is limited to what is required for the aggregate computation. This is represented exclusively by the GLA state. It is never the case that an entire chunk is passed from one worker node to another if only the border cells are used.

An alternative strategy that avoids merging altogether when the neighborhood function is bounded – at the expense of increased storage and more complicated chunk management – is to enforce that aggregate computation is always confined to a chunk, thus allowing for full parallel execution across chunks. This is known as overlapping [32] and requires cell replication across multiple chunks. Multiple overlapping strategies are discussed in [33]. They differ in the number of cell layers replicated across chunks. In single-layer overlap, a fixed number of border cells from

the neighboring chunks are stored together with the chunk—they can also be stored separately and accessed only when required. In multi-layer overlap through materialized overlap views, multiple such layers with increasing thickness are generated and stored. `APPLY+` GLA can take advantage of overlapping with minimal modifications. The `Merge` and `Terminate` methods are not required anymore since the entire computation is finalized in `EndChunk`. `BeginChunk` combines the overlapped data with the actual chunk to make them available as a whole in `Accumulate`.

4.4 Query Language

EXTASCID queries are specified at the execution plan level as a query tree that links together the different algebra operators. The plan is written in a query scripting language that requires the specification of the query tree structure – the operators and the links between them – and of the functionality of each operator, e.g., the selection ranges for `REBOX`. While the language might seem not declarative enough, it nonetheless provides a level of abstraction on top of the internal query representation that allows for easy specification of arbitrary user queries—the SS-DB queries have straightforward representations in the language. Essentially, the language is a direct representation for the physical query execution plan. In a full-fledged system with a declarative query language, this form is obtained at the end of a series of transformations and optimizations that take the high-level query to execution. They are all executed transparently to the user. In the current EXTASCID implementation, we bypass this entire process and, since the execution plan is followed exactly, the user is in charge of the query transformation and optimization. In a future implementation, we plan to integrate the upper part of the process starting from ArrayQL [26] or the more powerful SciQL [38] language.

4.5 Query Execution

In EXTASCID, query execution starts from the query plan written in the native scripting language. This is passed by the client to the coordinator. The coordinator takes this textual representation and transforms it into an internal format that drives the entire execution. The transformation consists in the instantiation of the generic algebra operators for the given query. The resulting instances behave as hard-coded operators specifically written for the query at hand. There is nothing generic anymore, nothing that requires interpretation. This results in maximum performance since everything is specific to the given query. The exact details of how the entire process works are inherited from the DataPath system and presented elsewhere [5]. The internal query representation is then distributed to all the processing nodes for execution. While this strategy might not be optimal since there are nodes that do not participate in the execution of a given query, e.g., a `REBOX` operator that accesses chunks from a single processing node, the EXTASCID coordinator cannot decide which nodes to send the query to since it has no global knowledge on data location. The nodes execute the query independently and the partial results are merged on the aggregation tree structure created by the coordinator at query initialization—details on the merging process

are presented in [30]. If the computation is confined to a chunk or to a processing node – the case for a large class of array operations – no merging is required. The extended interface of the GLA metaoperator provides the tools for this type of optimization. The last step of the query execution process is to pass the result to the user. This is done by the coordinator once merging on the aggregation tree has finalized.

5 Benchmark Implementation

In this section, we provide the details of how we implement the SS-DB benchmark normal scale, i.e., 400 $(7,500 \times 7,500)$ grids, in EXTASCID. We present the structures used to represent the raw images and the derived data – observations and groups – the implementation of the array algebra operators used in queries, and the query implementation. We also discuss alternative implementations and their trade-offs.

5.1 Raw Data

The raw image array given in Eq. (1) is represented in the local coordinate system. Each image is stored separately and it is chunked regularly into 100 chunks of equal size (750×750). Each chunk is approximately 25 MB in size. Grid cells inside the chunk are organized in row-major order without storing the coordinates, i.e., dimension suppression. This results in a 15% reduction in chunk size. Inside the chunk, attributes are stored vertically partitioned and are read only when the query demands it. Each column is approximately 2.25 MB in size. The chunks of each image are distributed in round-robin fashion over the available processing nodes. For example, in our experimental setup consisting of 9 processing nodes with one coordinator and 8 workers, half of the nodes store 12 chunks while the other half store 13 chunks. Overall, each node stores $5,000$ chunks with a total size of 125 GB. Notice also that each node stores chunks from every image. While other strategies are available for chunk shape, the organization of the cells inside the chunk, and the distribution of chunks across nodes – see [29] for a comprehensive survey – we settled for this approach due to its simplicity and proven efficiency [33].

The generic structure shown in Figure 2 stores the chunk borders along each dimension in the chunk metadata. The borders are loaded initially from disk when the chunk is read into memory. During processing, as the chunk passes through operators, the borders are modified accordingly. For example, in order to change to the global coordinate system using the `SHIFT` operator, chunk borders have to be modified for every chunk in the image based on the origin of the image. This entails only the modification of the borders in the chunk metadata.

5.2 Derived Data

Due to its sparse structure in the global coordinate space, we store derived data in sparse array format. Essentially, dimensions are stored explicitly since their value

cannot be inferred anymore from the position in the chunk. In order to support an efficient SHIFT operator, the dimension value is stored relatively to the chunk origin—the minimum value along each dimension stored in the chunk metadata. In the following, we discuss the details specific to the observation and group implementations.

5.2.1 Observations

Observations are represented as two sparse arrays with dimensions explicitly materialized (Table 1). obs stores the observation id for every cell that is part of the observation. It has the same dimensionality as images. obs_center represents the observation center in the global coordinate system and it has as attributes the aggregated properties of the observation, e.g., the average pixel value.

obs is populated by the APPLY+ algebra operator during the cooking process. The implementation of the cooking APPLY+ operator is standard (Section 4.3.4). It has two stages. The first stage identifies the observations internal to an images chunk. It is a fully parallel process without any data transfer. The observations at chunk boundaries have to be merged together in the second stage. This requires transferring the boundary observation data between chunks and even across nodes. Since the number of boundary observations tends to be small, the amount of transferred data is relatively reduced. Moreover, observations are materialized as soon as adjacent chunks are merged in RemoteMerge. obs_center is populated at the end of the cooking process by combining data from images and obs.

There are multiple strategies to chunk obs and obs_center. For instance, an obs chunk can be created for every corresponding images chunk. The problem with this strategy is that the number of observations in a chunk is too small, e.g., since there are $20,000$ observations on average in an image, less than 200 observations end-up in the same chunk. Chunks with small size incur reduced I/O throughput due to the frequent disk seeks and short scans. The solution we adopt for increasing the chunk size is to merge observations from multiple chunks together. Merging can be done for the images chunks resident at the same node or, at the extreme, only a single obs chunk is created for every image. After experimenting with these alternatives, we found that having a single obs chunk per image provides optimal performance. This is the solution we implement in EXTASCID. We apply the same chunking strategy, i.e., single chunk per image, for obs_center. To guarantee uniform distribution of chunks across processing nodes, we build aggregation trees having the root at different nodes while cooking the images array.

5.2.2 Groups

Groups are represented as two high-dimensional arrays – group_center (3-D) and group_center_img (4-D) – having the dimensions materialized. group_center stores the single group center across all the observations in the group in the global coordinate system. The third dimension corresponds to the cycle the group is part of. group_center_img stores a center for every image that contains observations in the group. This center is computed from the observation centers in that particular image. img_id, taking values between 0 and 19, is the fourth dimension in this array.

The only solution to parallelize the grouping process is across different cycles. A merge-based parallel implementation inside the cycle is not possible since all the combinations have to be considered whenever merging observations from multiple images. Essentially, no work is saved through parallelization. As a result, the first step in the grouping process is to bring all the observations corresponding to a cycle, i.e., `obs_center`, on the same node. Notice that different cycles end-up on different nodes though. For example, at most 3 cycles from the normal SS-DB instance are processed on the same node in our experimental setup consisting of 8 workers. In terms of the extended GLA interface, all the action happens in `Terminate`. The global coordinate space (`img_id-x-y`) corresponding to a cycle is regularly chunked along the `x-y` dimensions such that nearby observations are grouped together. The resulting grid index is meant to reduce the number of observations that have to be compared for membership to the same group. Group creation proceeds iteratively from the first image in the cycle with a series of calls to the distance-based `APPLY+` operator. Once the member observations are determined, the two group arrays – `group_center` and `group_center_img` – can be filled.

In `group_center`, we create a single chunk for all the groups contained in a cycle since the number of groups is at most a constant factor larger than the number of observations in the first image of the cycle. The chunk is obtained directly as a result of grouping. `group_center_img` is sliced additionally along the `img_id` dimension, with a chunk generated for every image. Although this requires all the group centers to be inspected for every range query, we have not seen a significant performance degradation when compared to chunking along all the dimensions. The reason is the relatively small number of groups in a cycle.

5.3 Queries

Query implementation follows directly the implementation of the corresponding array algebra operators given in Section 3.3. The EXTASCID GLA-based implementation for each array operator used in SS-DB is presented in Section 4.3.3. Without going into the details specific to every query in the benchmark, we emphasize two important aspects of our implementation. First, `REBOX` is pushed for execution into the storage manager. This guarantees that only the chunks required by the query at hand are read from disk, thus minimizing the overall I/O. And second, the chunks that reach the memory are asynchronously processed in parallel both inside the same operator as well as across operators. When combined, these two execution strategies guarantee optimal performance both for range queries as well as for the `APPLY+` operator, as shown by our experimental results.

6 Benchmark Results

In this section, we present the results obtained by executing the SS-DB benchmark in EXTASCID and SciDB. We use SciDB 13.11 – the latest version at the time we perform the experiments (December 2013) – and the corresponding SS-DB benchmark

implementation made available with the source code [2]. We optimize the SciDB implementation based on detailed instructions from the SS-DB benchmark maintainers [21]. In order to obtain a complete picture on the EXTASCID performance, we run experiments on two system configurations – single-node and cluster – and over two SS-DB instances having different size. In addition to comparing the two systems in two entirely different environments, this also provides an insight on their scalability.

Methodology. Benchmark execution consists of three distinct stages. First, raw image data are loaded into the processing system. This is a translation step that maps images from their original representation into the internal system representation. In the second stage, derived data – observations and groups – are extracted from the raw data. The overall execution time is reported for each of the three operations—loading, cooking, and grouping. Queries are executed in the third stage as follows. A series of five different configuration parameters are randomly generated for every query. The query is executed ten times for every parameter configuration and the average execution time is reported for the (query, parameter configuration) pair. The same is repeated for every parameter configuration, for a total of five execution times per query. The sum of these execution times is reported as the query execution time. The reason we sum-up the execution time for different configurations is the high variance incurred by different selection ranges, especially in the global coordinate system. We present three results for every benchmark operation—the EXTASCID execution time, the SciDB execution time, and the ratio between the SciDB and the EXTASCID execution time. Notice that all the experiments are executed with cold caches.

6.1 Single-Node Configuration

System. We execute the single-node experiments on a standard server with 2 AMD Opteron 6128 series 8-core processors (64 bit) – 16 cores – 40 GB of memory, and four 2 TB 7200 RPM SAS hard-drives configured RAID-0 in software. Each processor has 12 MB L3 cache while each core has 128 KB L1 and 512 KB L2 local caches. The storage system supports 240, 436 and 1600 MB/second minimum, average, and maximum read rates, respectively—based on the Ubuntu disk utility. According to `hdparm`, the cached and buffered read rates are 3 GB/second and 565 MB/second, respectively. Ubuntu 12.04.3 SMP 64-bit with Linux kernel 3.2.0-56 is the operating system.

For the single-node configuration, the EXTASCID coordinator and worker execute as separate processes exchanging messages through TCP/IP sockets. All the parallelism is implemented at thread-level inside the worker. Configuring SciDB is trickier since there are two levels of parallelism—process and thread. The 16 physical cores can be divided in five different combinations between the two while maintaining full coverage. Not all the combinations are equal though since the larger the number of processes, the larger the memory consumption. After experimenting with all of the five combinations, we found that (16 processes, 1 thread/process) provides the optimal performance—especially for data loading. Thus, we present the results for this configuration which uses only process-level parallelism and is opposite to EXTASCID which uses only thread-level parallelism.

Data. The dataset used for this configuration is the SS-DB small instance consisting of 160 $(3,750 \times 3,750)$ 2-D grids. The grids are grouped into cycles of 20, for a total of 8 cycles. The overall size of the dataset is approximately 100 GB—each grid is 625 MB. We use the SS-DB configuration with the medium benchmark parameters as defined in [17].

6.1.1 Data Loading

The results for loading the 160 benchmark images in EXTASCID and SciDB are presented in Table 3. Although loading is parallel in both systems, there are significant implementation differences that account for the different results. Similar to the overall system, loading in EXTASCID is thread-level parallel. We split the 160 images into four groups of 40 and assign them to four threads that read and chunk them in parallel. The chunks are (375×375) in this case. Writing the converted chunks inside EXTASCID is done by a separate thread. We limit the number of reading threads to four in order to reduce disk interference. Disk I/O is the bottleneck when loading data in EXTASCID. Parallel loading of multi-dimensional arrays in SciDB is a two stage process. First, the images are loaded into a temporary 1-D array with a surrogate dimension. Image dimensions are attributes in the temporary array. This process is the equivalent of loading in EXTASCID. In the second stage, the temporary 1-D array is reorganized into the actual 3-D array used in processing. The reorganization consists in building the (375×375) chunks together with a 50 band overlap along each side of a dimension. The overlap is an artifact to fully parallelize cooking. Even when raw data are already grouped into chunks, reorganization is required because parallel loading in SciDB 13.11 can be executed only into 1-D arrays [2]. EXTASCID does not require reorganization because it supports direct parallel loading of multi-dimensional arrays.

As shown in Table 3, loading in EXTASCID takes $1,351$ seconds. Parallel loading in SciDB takes $3,475.15$ seconds – a factor of 2.57 larger – for the configuration with 16 processes or instances. With fewer SciDB instances running on the server, the loading time is considerably higher—$28,257$ for 1 instance, $7,715.32$ for 4, and $4,406.27$ for 8, respectively. This shows that parallel loading in SciDB is implemented only at process or instance-level. There is no thread-level parallelism inside an instance. As a result, $15/16^{\text{th}}$ of the data have to be moved across process boundaries. This takes time and affects the SciDB performance negatively when compared to EXTASCID. The results get only worse when the number of SciDB instances is smaller—a clear sign that converting data into the internal SciDB representation is a time-consuming process. Essentially, loading data in SciDB is CPU-bound. The SciDB reorganization process on the 16 instance configuration takes $36,006$ seconds (10 hours)—10 times more than the actual loading. Given that this single-time process is related to index building in relational databases, the high execution time is somehow expected. As long as the workload is query-dominated and the query performance is significantly improved, reorganization makes sense. When compared to EXTASCID – which achieves the same chunking output but without overlap – the overall SciDB performance is almost a factor of 30 worse—a significant difference.

System	Execution Time [seconds]		
	Load	Cook	Group
EXTASCID	1,351	79.71	2.82
SciDB	3,475.15 + 36,006 = 39,481.15	81.00	21.00
SciDB/EXTASCID	2.57 (29.22)	1.02	7.45

Table 3: Loading, cooking, and grouping execution time for single-node.

6.1.2 Derived Data

Cooking. The results for cooking (Table 3) provide a comparison between the merge and overlap parallel execution strategies [32]. In EXTASCID merge, chunks are processed concurrently and boundary observations are merged across threads using the GLA interface methods. The more boundary observations exist, the more merges are required. In SciDB overlap, merging is avoided altogether by replicating a bounded amount of data across chunks. The amount of replicated data guarantees that an observation is fully contained inside a chunk. Overlap drawbacks include an increase in chunk size – quite significant even when the overlap is relatively small to the size of the chunk – and an increase in the time to process a chunk. For this particular dataset, merge and overlap perform almost identically—a factor of 1.02 advantage for EXTASCID merge. In general, we argue that there are cases when each method is better than the other. It depends on how observations are distributed relative to the chunks. A detailed study on the performance of the two strategies is presented in [32].

Grouping. EXTASCID is faster than SciDB by a factor of 7.45 on grouping (Table 3). While EXTASCID exploits parallelism only for extracting groups across cycles, the SciDB implementation is entirely sequential. The sequential SciDB implementation is argued to be simpler and not considerably slower than a parallel version in [17]. This has to do with the representation of observations as arrays in SciDB. Since observations have a relational representation in EXTASCID, implementing parallel grouping is considerably simpler. And the performance gains are as expected. The parallel EXTASCID implementation achieves almost a linear speedup, i.e., 7.45 instead of 8—the number of cycles.

6.1.3 Queries

Raw Data. The results for executing the SS-DB queries on raw data are presented in Table 4. We remark immediately more variability, with EXTASCID still outperforming SciDB at cooking (Q2) and SciDB being faster for Q1 and Q3. Since Q2 is executed over 8 images – the first in each cycle – the SciDB execution time of 4 seconds is exactly as expected. EXTASCID performs better due to a reduced number of merges. SciDB outperforms EXTASCID by a factor of 5.59 at aggregation over raw data (Q1). While both systems use columnar storage and read only the required data from disk, the difference is made by two features implemented only in SciDB—compression and caching. The amount of data stored on disk for every column is further reduced through lossless compression. This results in shorter I/O delays. In

addition, SciDB also implements a chunk-level buffer pool that caches the recently accessed chunks in memory, thus avoiding I/O operations entirely. We observed the effect of caching on Q1, i.e., the first run is four times slower than the subsequent ones. SciDB has a slight advantage – a factor of 1.72 – over EXTASCID on Q3 mostly due to compression. Beyond that, the two implementations are identical. It is important to notice that caching does not play a role in this case since Q3 is executed over a full image cycle.

System	**Execution Time [seconds]**		
	Q1	Q2	Q3
EXTASCID	39.16	2.56	181.41
SciDB	7.00	4.00	105.24
SciDB/EXTASCID	0.18	1.56	0.58

Table 4: Execution time for queries on raw data for single-node.

Observations. Table 5 contains the results for executing the SS-DB queries over the observation data. EXTASCID is consistently faster than SciDB in this situation, by as much as a factor of 120 for Q6. After careful inspection of the execution mechanisms in the two systems, we found the reasons to explain the difference. The main cause is the representation of sparse arrays in the two systems. While EXTASCID supports natively the relational representation of sparse arrays with dimensions stored explicitly, SciDB uses a unified representation for sparse and dense arrays alike. Although run-length encoding (RLE) and delta compression are used extensively to reduce the size of sparse arrays, SciDB still has to handle more data and to express queries over sparse arrays on the unified dense array representation. This has a negative impact on the execution time. Another reason that explains the gap in execution time is query invocation. In SciDB, each query is executed as 20 separate queries—one per cycle image. In EXTASCID, a single query is sufficient to get the desired result. The iterative invocation of each query and the overhead incurred by the parallel dissemination across the 16 instances when accumulated over that many queries become a noticeable fraction in the overall execution time, as the results in Table 5 confirm.

System	**Execution Time [seconds]**		
	Q4	Q5	Q6
EXTASCID	0.28	0.17	0.15
SciDB	5.00	20.00	18.00
SciDB/EXTASCID	17.86	117.65	120.00

Table 5: Execution time for queries on observation data for single-node.

Groups. The results for queries over groups of observations are included in Table 6. With the exception of query Q7, we observe a similar trend as for the queries

over observations. And the reason is the same—the representation of sparse arrays as relations in EXTASCID. SciDB outperforms EXTASCID on Q7 because group centers are not stored explicitly in EXTASCID but they are rather derived from the member observations every time.

System	Execution Time [seconds]		
	Q7	Q8	Q9
EXTASCID	0.62	17.62	17.66
SciDB	0.31	102.86	98.44
SciDB/EXTASCID	0.50	5.84	5.57

Table 6: Execution time for queries on groups of observations for single-node.

6.2 Cluster Configuration

System. For the cluster configuration, we report experimental results on a 9-node shared nothing system. The coordinator in both EXTASCID and SciDB runs on the server used for the single-node experiments. The other 8 nodes are identical and they have almost the same configuration as the server. The only differences are 20 GB of RAM instead of 40 GB and 1 TB hard-drives instead of 2 TB. The nodes are mounted inside the same rack and are inter-connected through a Gigabit Ethernet switch. In EXTASCID, the server node is configured as the coordinator while the other 8 identical machines are workers. In SciDB, we settled for a configuration with 8 instances 2 threads each for each worker node. The server has a single instance. Thus, there are 65 SciDB instances across the entire cluster. The reason we use 8 instances instead of 16 per node is the smaller memory footprint.

Data. The dataset used for the cluster configuration is the SS-DB normal instance consisting of four hundred $(7,500 \times 7,500)$ 2-D grids. The grids are grouped into cycles of 20, for a total of 20 cycles. The overall size of the dataset is 1 TB – each grid is 2.48 GB – which corresponds to approximately 125 GB allocated to each node. This is a $1/4^{\text{th}}$ increase compared to the server configuration. Similar to the single-node configuration, each image is split into 100 (750×750) chunks. These chunks are 4 times larger though. We use the SS-DB configuration with the medium benchmark parameters as defined in [17].

6.2.1 Data Loading

The results for loading the 400 benchmark images in EXTASCID and SciDB are presented in Table 7. The difference between the two systems is considerably larger than for the single-node configuration. The reason is the parallel loading mechanism in SciDB. The 1 TB raw file hosted at the server is read sequentially and chunked. The chunks are sent in round-robin order to the 65 SciDB instances for parallel loading. While mapping to the internal SciDB representation and loading are executed

concurrently, reading is sequential. In EXTASCID, data are initially split across the 8 workers, thus reading is also parallel. Moreover, no data transfer between workers is required since data are loaded locally. When we apply the same strategy in SciDB the results become even worse – $30,361$ seconds instead of $23,265.82$ – due to the all-to-all communication across nodes. Data reorganization – chunking and overlapping – increases the performance gap even further. Overall, EXTASCID is faster at loading data by a cumulative factor of 169.22—25 minutes compared to 3 days.

System	Execution Time [seconds]		
	Load	Cook	Group
EXTASCID	1,509	82.19	10.48
SciDB	23,265.82 + 232,080 = 255,345.82	144.42	69
SciDB/EXTASCID	15.42 (169.22)	1.76	6.58

Table 7: Data loading, cooking, and grouping execution time for cluster.

6.2.2 Derived Data

Cooking. EXTASCID performs cooking fully parallel at chunk level using the merge execution strategy. The amount of data transferred between nodes is reduced to the minimum and merging of adjacent chunks is executed as early as possible in order to optimize the performance. SciDB cooking is also parallel and confined to each chunk. While there is no merging, the time to process a chunk is larger since the size of the chunk is larger. This is reflected in the results in Table 7 which show a slightly better execution time for EXTASCID.

Grouping. The results for grouping follow the same trend as for the single-node configuration with EXTASCID outperforming SciDB by a factor of 6.58 (Table 7). This is due to the parallel grouping implementation in EXTASCID as opposed to sequential in SciDB.

6.2.3 Queries

Raw Data. The results for queries over raw data are presented in Table 8. The same trend from Table 4 maintains with SciDB faster for Q1 and Q3 and EXTASCID slightly faster for Q2. The explanations given in the single-node configuration propagate to the cluster configuration with the corresponding decrease in execution time. Q2 is the only exception – the execution time increases – due to processing a larger number of larger images—20 instead of 8.

Observations. Query execution time over observation data in the cluster configuration is shown in Table 9. As the results show, this is the type of queries for which EXTASCID outperforms SciDB significantly. The maximum difference is a factor of 40.20 for Q6 and is due to the optimal handling of sparse arrays in EXTASCID. The increase in query execution time when compared to the single-node results in Table 5 is due to the larger number of observations that are extracted from larger images.

System	Execution Time [seconds]		
	Q1	Q2	Q3
EXTASCID	22.11	8.85	129.46
SciDB	2.76	9.17	87.38
SciDB/EXTASCID	0.12	1.04	0.67

Table 8: Execution time for queries on raw data for cluster.

System	Execution Time [seconds]		
	Q4	Q5	Q6
EXTASCID	0.79	0.72	0.66
SciDB	8.62	24.96	26.53
SciDB/EXTASCID	10.91	34.67	40.20

Table 9: Execution time for queries on observation data for cluster.

Groups. The conclusions drawn for queries over observations can be immediately extended to queries over groups of observations (Table 10). The gap between EXTASCID and SciDB is not that significant though.

System	Execution Time [seconds]		
	Q7	Q8	Q9
EXTASCID	3.36	34.25	35.87
SciDB	2.18	157.70	168.51
SciDB/EXTASCID	0.65	4.60	4.70

Table 10: Execution time for queries on groups of observations for cluster.

6.3 Scalability

Table 11 and 12 present EXTASCID and SciDB scaleup for loading and querying, respectively. Since the amount of data increases 10 times while the number of workers increases only 8 times, we expect a scaleup value of 1.25 when processing the entire raw data. For a single image, the expected scaleup is 0.5 since the number of chunks is the same and the size of a chunk grows only 4 times. The scaleup for processing derived data – observations and groups – is more difficult to quantify theoretically since it depends on the actual number of observations and groups identified in an image or cycle, respectively.

The EXTASCID scaleup for loading and cooking – two operations executed over the entire raw data – is close to the expected value of 1.25. The same is true for cooking in SciDB. Loading data in the SciDB cluster configuration is significantly worse than expected due to the large amounts of data that have to be transferred between

System	Scaleup		
	Load	Cook	Group
EXTASCID	1.12	1.03	3.72
SciDB	6.69 (6.46)	1.78	3.29

Table 11: Scaleup from single-node to cluster for load, cook, group.

instances. The same trend manifests in the case of data reorganization. Grouping in SciDB is executed sequentially over all the cycles. Since there are 2.5 more cycles in the normal SS-DB instance, a scaleup of 2.5 is expected. The larger number of observations account for the difference to the 3.29 value. In EXTASCID, grouping is thread-level parallel even for the single-node configuration. In the cluster environment, merges across workers require data transfer such that all the groups in a cycle end-up at the same node for a final merge. This impacts negatively the scalability.

System	Scaleup								
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
EXTASCID	0.56	3.46	0.71	2.82	4.24	4.40	5.42	1.94	2.03
SciDB	0.39	2.29	0.83	1.72	1.25	1.47	7.03	1.53	1.71

Table 12: Scaleup from single-node to cluster for queries.

The scaleup for query processing is depicted in Table 12. The reason for the large variation is that some queries are executed over ranges, some over full images, and some other over an entire cycle. Moreover, the number of observations and groups varies across the two SS-DB instances, thus the larger scaleup values for queries over derived data. As a general trend, we observe that parallel processing on the cluster configuration improves execution time for queries over raw data even when the amount of data are larger. This is due entirely to chunking the raw data across multiple workers.

6.4 Discussion

There are multiple conclusions we can draw from the experimental results presented in this section. The most important point to remark is that although EXTASCID and SciDB share common design and implementation features, their performance is quite different for the SS-DB benchmark. EXTASCID outperforms SciDB at data loading and grouping while their performance is similar for cooking. The difference for loading is two orders of magnitude when the reorganization time is included. The relationship between the two systems is more nuanced when considering query performance. SciDB is slightly faster on queries over raw data whenever compression and chunk caching can be applied. For queries over derived data – amenable to a relational representation – EXTASCID provides the better execution time since it supports natively both dense as well as sparse arrays. Since SciDB is targeted at dense arrays,

mapping between the two representations is required. These trends manifest both for a single-node configuration as well as in a cluster environment. Whenever processing raw data the two systems exhibit linear scaleup, thus confirming their scalability over large datasets partitioned across many processing nodes.

7 Related Work

There are three lines of research on scientific and array data processing that we consider related to the topics addressed in this paper—benchmarking, array query algebras and languages, and scientific data processing systems. To the best of our knowledge, Sloan Digital Sky Survey (SDSS) [3,35] is the single other benchmark targeted specifically at scientific processing. Similar to SS-DB, SDSS is also based on processing astronomical images. Unlike SS-DB though, SDSS operates exclusively on relational data obtained as a bi-product of astronomical observations. SS-DB is more general and contains a full spectrum of operations ranging from raw data processing to the creation and querying of derived observation data. These operations manipulate array-oriented data through relatively sophisticated user-defined functions, not always expressible in SQL. While typical implementations of the SDSS benchmark are SQL-based, e.g., MS SQL Server and MonetDB [24], SciDB and MySQL [17] are the only two implementations of the SS-DB benchmark we are aware of. With our implementation in EXTASCID, we provide another reference point for a larger adoption of the SS-DB benchmark—the only general benchmark for scientific data processing.

Array algebras and query languages. ArrayQL algebra [27] and query language [26] are formalisms introduced in the SciDB context—similar to the SS-DB benchmark. This is the main reason we choose these formalisms to represent the SS-DB operations. They are not commonly accepted though as the representative array algebra and query language. In fact, there is no commonly accepted array algebra and query language. In the following, we discuss other such alternatives proposed in the literature. AQL [25] is a declarative query language for multi-dimensional arrays that treats arrays as functions from index sets to values rather than as collection types. AQL is based on the nested relational calculus with arrays which plays the same role relational calculus and algebra play for the relational data model. The RasDaMan [6] array algebra conceptualizes arrays as functions from rectangular domains to cell values. Three core constructs – MARRAY, COND, and SORT – that can express every array operation when composed together are introduced. In the corresponding RasQL query language [7], arrays are treated as a composite attribute type with a set of corresponding operators. AML [28] is an algebra consisting of three operators that manipulate dense arrays and take bit patterns as parameters. A significant AML limitation is that it contains only structural operators, i.e., operators that consider the indexes. At query language level, AML is more like an elevated execution plan description than a declarative array query language. RAM [36] and SRAM [16] are array algebras for dense and sparse arrays, respectively, developed in the context of the MonetDB [23] columnar database system. Since the execution happens inside a relational database engine, array queries follow a sequence of transformations that

take arrays represented in the comprehension syntax to relational operators through an intermediate array algebra stage. Although a series of rewriting rules and optimizations are applied at each of these two steps, relying on the relational algebra operators to map and process array operations introduces inefficiencies due to the impedance mismatch in representation. SciQL [38] is the most comprehensive extension to the SQL:2003 standard with support for arrays. It provides seamless integration of set, sequence, and array semantics. The goal is to make minimal modifications to the SQL syntax while allowing for maximum expressiveness in the array operations supported by the language. An interesting characteristic of all the array algebras discussed above is their equivalence. In [8], it is shown that all the array algebras can be reduced to RasQL—both in array representation as well as operations. This is primarily due to the equivalence between comprehensions and the MARRAY operator for creating arrays. We point the interested reader to [29] for a comprehensive discussion on these and other array algebras and query languages.

Array processing systems. EXTASCID is part of a long series of parallel systems for scientific data processing. Titan [11] and T2 [10] are the first systems designed with extensibility in mind. They adopt an execution strategy closely related to the Map-Reduce [19] paradigm. More recently, SciHadoop [9] implements array processing on top of the popular Hadoop Map-Reduce framework. The main differences between EXTASCID and these systems are the different execution strategy, i.e., UDA vs. Map-Reduce, and the native support for arrays and relations in EXTASCID. RasDaMan [7] is a general middleware for array processing with array chunks stored as BLOBs in a back-end database. The processing is specified through a limited number of second-order operators integrated into SQL and executed entirely inside the middleware. RasDaMan is targeted only at array data – relational data have to be processed either at the application level or inside the middleware – and it is not parallel. The RAM [36] and SRAM [16] systems provide support for array processing on top of the MonetDB [23] columnar database. They do not provide native support for arrays since arrays are represented as relations and array operations are mapped over relational algebra operators. Moreover, RAM and SRAM are not parallel. RIOT [37] is a prototype system for linear algebra operations over large vectors and matrices mapped into a standard relational database representation. Linear algebra operations are rewritten into SQL views and evaluated lazily. Since a relational database is used both for array storage and processing, RIOT falls in the category of middleware systems similar to RasDaMan. ArrayStore [33] and TrajStore [18] are storage managers optimized for multi-dimensional arrays and trajectories, respectively. They do not provide a query execution strategy and operators to implement it. We point the interested reader to [29] for a comprehensive discussion on array database systems.

SciDB. SciDB [34] is the system EXTASCID resembles the most. Both are parallel systems designed to be extensible. Parallelism is obtained through data partitioning into chunks further optimized by dimension suppression. While SciDB makes heavy use of array decomposition and compression, EXTASCID resorts to composite chunks organized vertically. SciDB supports natively only arrays. EXTASCID provides native support both for arrays and relations. While SciDB adopts a traditional pull-based execution model, EXTASCID is entirely push-driven. The execution strategy in EXTASCID is well-defined through the UDA interface which makes reasoning

about parallelism clear. The same is not true in SciDB where a series of UDFs are arbitrarily interconnected and the parallelization strategy is based on chunk overlapping rather than merging. It is important to emphasize that in SciDB UDAs are considered only as a mechanism to implement aggregate computations for User-Defined Types (UDT). They are never considered a general processing paradigm capable of expressing almost any type of computation, the case in EXTASCID. While there are discussions on the implementation of the SS-DB benchmark in multiple of these systems, the SciDB implementation is the only we are aware of at the time when the paper is written. Consequently, this is our reference for the EXTASCID evaluation on the SS-DB benchmark.

8 Conclusions

In this paper, we present a formal representation of the SS-DB benchmark in terms of array algebra operators and array query language constructs. These are meant to simplify the implementation in other systems and foster the acceptance of SS-DB as the standard benchmark to evaluate scientific data processing applications. Given that no alternatives exist, SS-DB fills an important void in the evaluation of a large class of Big Data applications. To verify the soundness of our formalization, we give a reference implementation and present benchmark results in EXTASCID, a novel system for scientific data processing we have developed. EXTASCID is complete in providing native support both for array and relational data and extensible in executing any user code inside the system by the means of a configurable metaoperator. These features result in considerable improvement over SciDB at data loading, extracting derived data, and operations over derived data. The results prove that the integrated EXTASCID architecture supporting natively both arrays and relations is more suited for complex scientific processing over raw and derived data requiring a high degree of extensibility.

References

1. M4. `http://www.gnu.org/software/m4/`. [Online; December 2013].
2. SciDB. `http://www.scidb.org/`. [Online; January 2014].
3. Sloan Digital Sky Survey. `www.sdss3.org`. [Online; April 2013].
4. TPC Benchmark Suite. `http://www.tpc.org/`. [Online; February 2013].
5. S. Arumugam, A. Dobra, C. Jermaine, N. Pansare, and L. Perez. The DataPath System: A Data-Centric Analytic Processing Engine for Large Data Warehouses. In *Proceedings of 2010 ACM SIGMOD International Conference on Management of Data*, pages 519–530, 2010.
6. P. Baumann. A Database Array Algebra for Spatio-Temporal Data and Beyond. In *Proceedings of 1999 NGITS International Workshop on 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 *Proceedings of 1998 ACM SIGMOD International Conference on Management of Data*, pages 575–577, 1998.
8. P. Baumann and S. Holsten. A Comparative Analysis of Array Models for Databases. In *Proceedings of 2011 FGIT-DTA/BSBT*, pages 80–89, 2011.
9. J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt. Sci-Hadoop: Array-based Query Processing in Hadoop. In *Proceedings of 2011 SC International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 66:1–66:11, 2011.

10. C. Chang, A. Acharya, A. Sussman, and J. H. Saltz. T2: A Customizable Parallel Database for Multi-Dimensional Data. *SIGMOD Rec.*, 27(1):58–66, 1998.
11. C. Chang, B. Moon, A. Acharya, C. Shock, A. Sussman, and J. H. Saltz. Titan: A High-Performance Remote Sensing Database. In *Proceedings of 1997 IEEE ICDE International Conference on Data Engineering*, pages 375–384, 1997.
12. Y. Cheng, C. Qin, and F. Rusu. GLADE: Big Data Analytics Made Easy. In *Proceedings of 2012 ACM SIGMOD International Conference on Management of Data*, pages 697–700, 2012.
13. Y. Cheng and F. Rusu. EXTASCID: An Extensible System for the Analysis of Scientific Data. `http://www-conf.slac.stanford.edu/xldb2012/demos.asp`. [XLDB 2012 abstract & demo].
14. Y. Cheng and F. Rusu. Astronomical Data Processing in EXTASCID. In *Proceedings of 2013 SSDBM International Conference on Scientific and Statistical Database Management*, pages 387–390, 2013.
15. Y. Cheng and F. Rusu. Formal Representation of the SSDB Benchmark and Experimental Evaluation in EXTASCID. *CoRR*, abs/1305.1609, 2013.
16. R. Cornacchia, S. Héman, M. Zukowski, A. P. de Vries, and P. Boncz. Flexible and Efficient IR using Array Databases. *VLDB Journal (VLDBJ)*, 17:151–168, 2008.
17. P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, S. Madden, M. Stonebraker, S. B. Zdonik, and P. G. Brown. SS-DB: A Standard Science DBMS Benchmark. `http://www.xldb.org/science-benchmark/`. [Online; August 2012].
18. P. Cudre-Mauroux, E. Wu, and S. Madden. TrajStore: An Adaptive Storage System for Very Large Trajectory Data Sets. In *Proceedings of 2010 IEEE ICDE International Conference on Data Engineering*, pages 109–120, 2010.
19. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, 2008.
20. D. J. DeWitt and J. Gray. Parallel Database Systems: The Future of Database Processing or a Passing Fad? *SIGMOD Rec.*, 19, 1991.
21. D. E. Difallah and P. Cudre-Mauroux. Private communication.
22. P. Furtado and P. Baumann. Storage of Multidimensional Arrays Based on Arbitrary Tiling. In *Proceedings of 1999 IEEE ICDE International Conference on Data Engineering*, pages 480–489, 1999.
23. S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two Decades of Research in Column-Oriented Database Architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.
24. M. Ivanova, N. Nes, R. Goncalves, and M. Kersten. MonetDB/SQL Meets SkyServer: the Challenges of a Scientific Database. In *Proceedings of 2007 SSDBM International Conference on Scientific and Statistical Database Management*, pages 38–46, 2007.
25. L. Libkin, R. Machlin, and L. Wong. A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques. In *Proceedings of 1996 ACM SIGMOD International Conference on Management of Data*, pages 228–239, 1996.
26. K.-T. Lim, D. Maier, J. Becla, M. Kersten, Y. Zhang, and M. Stonebraker. Array QL Syntax. `http://www.xldb.org/wp-content/uploads/2012/09/ArrayQL-Draft-4.pdf`. [Online; January 2013].
27. D. Maier. ArrayQL Algebra: version 3. `http://www.xldb.org/wp-content/uploads/2012/09/ArrayQL_Algebra_v3+.pdf`. [Online; January 2013].
28. A. P. Marathe and K. Salem. Query Processing Techniques for Arrays. *VLDB Journal (VLDBJ)*, 11(1):68–91, 2002.
29. F. Rusu and Y. Cheng. A Survey on Array Storage, Query Languages, and Systems. *CoRR*, abs/1302.0103, 2013.
30. F. Rusu and A. Dobra. GLADE: A Scalable Framework for Efficient Analytics. *OS Review*, 46(1), 2012.
31. S. Sarawagi and M. Stonebraker. Efficient Organization of Large Multidimensional Arrays. In *Proceedings of 1994 IEEE ICDE International Conference on Data Engineering*, pages 328–336, 1994.
32. E. Soroush and M. Balazinska. Hybrid Merge/Overlap Execution Technique for Parallel Array Processing. In *Proceedings of 2011 AD EDBT/ICDT Array Databases Workshop*, pages 20–30, 2011.
33. E. Soroush, M. Balazinska, and D. L. Wang. ArrayStore: A Storage Manager for Complex Parallel Array Processing. In *Proceedings of 2011 ACM SIGMOD International Conference on Management of Data*, pages 253–264, 2011.

34. M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. The Architecture of SciDB. In *Proceedings of 2011 SSDBM International Conference on Scientific and Statistical Database Management*, pages 1–16, 2011.
35. A. S. Szalay and al. Designing and Mining Multi-Terabyte Astronomy Archives: the Sloan Digital Sky Survey. *SIGMOD Rec.*, 29(2), 2000.
36. A. R. van Ballegooij. RAM: A Multidimensional Array DBMS. In *Proceedings of 2004 EDBT Extended Database Technology Workshops*, pages 154–165, 2004.
37. Y. Zhang, H. Herodotos, and J. Yang. RIOT: I/O-Efficient Numerical Computing without SQL. In *Proceedings of 2009 CIDR Conference on Innovative Database Research*, 2009.
38. Y. Zhang, M. Kersten, M. Ivanova, and N. Nes. SciQL: Bridging the Gap between Science and Relational DBMS. In *Proceedings of 2011 IDEAS Symposium on International Database Engineering and Applications*, pages 124–133, 2011.