Foundations and Trends® in Databases

Multidimensional Array Data Management

Suggested Citation: Florin Rusu (2023), "Multidimensional Array Data Management", Foundations and Trends® in Databases: Vol. 12, No. 2-3, pp 69–220. DOI: 10.1561/1900000069.

Florin Rusu
University of California Merced
frusu@ucmerced.edu

now
the essence of knowledge
Boston — Delft

Contents

1 Introduction **70**

2 Multidimensional Arrays **74**
2.1 Arrays 74
2.2 Types of Arrays 75
2.3 Array Dimensions 77
2.4 Arrays and Relations 78
2.5 Arrays and Tensors 80
2.6 Arrays and Data Cubes 81
2.7 Summary 82

3 Multidimensional Array Operations **84**
3.1 Array Operations in Scientific Applications 84
3.2 Relational Operations 88
3.3 Tensor Operations 89
3.4 Data Cube Operations 90
3.5 Summary 91

4 Algebras and Query Languages for Multidimensional Arrays **92**
4.1 Array Query Language (AQL) 93
4.2 Array Manipulation Language (AML) 95
4.3 Relational Array Mapping (RAM) 98

4.4 RasDaMan Query Language (RasQL) . . . 100
4.5 Science Query Language (SciQL) . . . 103
4.6 SciDB Query Languages . . . 104
4.7 Algebras for Domain Specific Data . . . 108
4.8 Relational Algebra . . . 109
4.9 Tensor Algebras . . . 110
4.10 Data Cube Algebras . . . 114
4.11 Summary . . . 115

5 Multidimensional Array Storage 116
5.1 Optimal Chunk Size . . . 117
5.2 Chunking Strategies . . . 118
5.3 Mapping Cells to Chunks . . . 130
5.4 Chunk Organization . . . 134
5.5 Mapping Chunks to Storage . . . 137
5.6 Relational Chunking . . . 142
5.7 Tensor Chunking . . . 143
5.8 Data Cube Chunking . . . 147
5.9 Summary . . . 147

6 Multidimensional Array Processing 148
6.1 Array Processing Paradigms . . . 150
6.2 Array Operators . . . 155
6.3 Advanced Array Processing Techniques . . . 162
6.4 In-situ Array Processing . . . 168
6.5 Relational Array Processing . . . 170
6.6 Tensor Processing . . . 171
6.7 Data Cube Processing . . . 177
6.8 Summary . . . 179

7 Multidimensional Array Systems 180
7.1 Array Databases . . . 180
7.2 Relational Array Systems . . . 186
7.3 Tensor Systems . . . 188
7.4 Data Cube Systems . . . 191
7.5 Summary . . . 194

8 Future Directions 196

Acknowledgments 200

References 201

Multidimensional Array Data Management

Florin Rusu

University of California Merced, USA; frusu@ucmerced.edu

ABSTRACT

Multidimensional arrays are a fundamental abstraction to represent data across scientific domains ranging from astronomy to genetics, medicine, business intelligence, and engineering. Arrays come under multiple shapes — from dense rasters to sparse data cubes and tensors — and have been studied extensively across many computing domains. In this survey, we provide a comprehensive guide for past, present, and future research in array data management from a database perspective. Unlike previous surveys that are limited to raster processing in the context of scientific data, we consider all types of arrays — rasters, data cubes, and tensors. We identify and analyze the most important research ideas on arrays proposed over time. We cover all data management aspects, from array algebras and query languages to storage strategies, execution techniques, and operator implementations. Moreover, we discuss which research ideas are adopted in real systems and how are they integrated in complete data processing pipelines. Finally, we compare arrays with the relational data model. The result is a thorough survey on array data management that should be consulted by anyone interested in this research topic — independent of experience level.

Florin Rusu (2023), "Multidimensional Array Data Management", Foundations and Trends® in Databases: Vol. 12, No. 2-3, pp 69–220. DOI: 10.1561/1900000069.

1

Introduction

Multidimensional arrays are one of the fundamental computing abstractions to represent data across virtually all areas of science and engineering (Harris *et al.*, 2020) — and beyond. In science, spatio-temporal data acquired by sensors measuring environmental conditions or generated by simulations of physical phenomena are represented as 3- or 4-D dense arrays — also called *rasters* or *grids*. Concrete examples include spatial 3-D (x/y/z) arrays of Earth subsurface voxels, 3-D (x/y/t) time series of X-ray and fMRI medical images, and 4-D (x/y/z/t) optical or radio telescope signals in astronomy (Baumann *et al.*, 2021). In business analytics, *data cubes* aggregate statistical measures such as the mean, variance, and median across all the combinations of the values on a set of multiple — possibly hierarchical — dimensions (Gray *et al.*, 1996). For example, a retailer may want to compute the monthly average volume of sales for every city and every product category. Unlike the science rasters, the coordinates of a data cube do not necessarily have a strict ordering — they are categorical, not ordinal. Moreover, many entries in the data cube can be empty, resulting in a sparse array. In machine learning and artificial intelligence, highly-dimensional models are defined over features extracted from text and image data. For ex-

ample, the text synthesis models applied in natural language processing consist of embeddings with billions of features (Brown *et al.*, 2020). These models are represented as 1-D vectors, 2-D matrices, and their multidimensional *tensor* generalizations. Machine learning training and prediction consist of a sequence of linear algebra operations between the model and the training/testing data — also represented as tensors.

Due to their ubiquity, multidimensional arrays — rasters, data cubes, and tensors — have been studied extensively across many areas of computer science — including compilers, programming languages, scientific and high-performance computing, graphics, machine learning, and databases. With the exception of databases, the vast majority of these studies are focused on the computational aspects of array processing — not the data management issues. Within the database field, the first "call" to extend the unordered set-based relational model with ordered rasters dates back to 1993 (Maier and Vance, 1993). Although rasters had initially spurred research interest, they had been overshadowed by data cubes upon their introduction in 1996 (Gray *et al.*, 1996). However, the era of "Big Data" from the late 2000s and early 2010s has renewed the interest in raster and sparse ordered arrays. The main driver has been the large volume of spatio-temporal data generated by scientific applications. This has led to the creation of the SciDB array database (Cudre-Mauroux *et al.*, 2009) as a collaboration between data management researchers and astronomers fostered by the XLDB suite of conferences. The success of deep neural network models in classifying objects from images and text has brought the spotlight on tensor processing in the late 2010s — which continues by the time of this writing.

In this work, we survey the research on multidimensional array data management — including rasters, data cubes, and tensors — from a database perspective. Thus, our focus is on work published in database conferences and journals. Nonetheless, we also include references to relevant work from other computing domains whenever necessary. Our definitive goal is to identify and analyze the most important research ideas on arrays proposed over time. We cover the full spectrum of data management, from array algebras and query languages to storage strategies, execution techniques, and operator implementations. Moreover, we

discuss which research ideas are adopted in real systems and how are they integrated in complete data processing pipelines. Given that the unordered set-based relational data model is dominant across databases, we compare the differences to arrays at every step in the presentation.

The resulting survey aims to serve two main objectives. First, it summarizes concisely the most relevant work on multidimensional array data management by identifying the major research problems. Second, the survey organizes this material to provide an accurate perspective on the state-of-the-art and future directions in array processing. To the best of our knowledge, this is the first complete survey on array data management that includes rasters, data cubes, and tensors. Previous surveys are limited to raster processing in the context of scientific data. For example, the first survey on array storage and processing (Rusu and Cheng, 2013) does not consider data cubes and tensors. The VLDB 2021 tutorial on array DBMS (Zalipynis, 2021) discusses the design of the ChronosDB system for external raster processing.

The most recent survey on array databases (Baumann *et al.*, 2021) gives a thorough analytical and experimental comparison among several raster systems — nineteen systems are compared and four of them are experimentally benchmarked. While there is some unavoidable overlap between our work and this survey, their approach and take-away message are quite different. The focus of Baumann *et al.* (2021) is RasDaMan — the array database developed by the same authors. The presentation of the main concepts — including algebra and query language, storage, and processing — is centered on the solutions implemented in RasDaMan. Alternative solutions are only briefly referenced. The end message is that RasDaMan is the most feature-complete array database system available — which is true given its 30+ years of development. This work delves considerably deeper in each of the topics and covers more breadth. Our perspective on multidimensional array data management considers all types of arrays — rasters, data cubes, and tensors. The focus is first on methods and then on their realization in a particular system. Our end goal is to summarize all the methods in a systematic presentation and provide a thorough analysis. In summary, we view Baumann *et al.* (2021) as system-centric while this work is technique-centered.

This work surveys a large body of work on multidimensional arrays published over three decades and is organized as follows. We start with a theoretical formalization of arrays and their categorization in Section 2. The defining operations for every type of array are presented in Section 3. The next three sections follow the architecture of a data processing system — going top-down from the user interface to the execution internals. Array algebras and query languages are introduced in Section 4. Array storage techniques are presented in detail in Section 5. Execution strategies and array operators are discussed in Section 6. The implementation of these ideas and their integration in real systems are analyzed in Section 7. We conclude with a summary of the most relevant ideas and an outlook to future directions on array data management in Section 8.

2

Multidimensional Arrays

In this section, we provide a formalization of multidimensional arrays from a relational database perspective. At a high level, arrays can be viewed as an instance of a particular type of relation — or table — in which the set of attributes that form the unique key have domains that are finite integer intervals. This set of attributes are called dimensions and they functionally determine the values of the other attributes in the array. This abstraction considers arrays as functions — which is first introduced by Maier and Vance (1993) — and is different from the representation of arrays as collection types (Libkin *et al.*, 1996). Based on this functional abstraction, we give a classification of array types and a conceptual comparison between arrays — on one side — and tensors, relations, and data cubes — on the other.

2.1 Arrays

A multidimensional array with N dimensions and M attributes — or N-dimensional array — is defined by a set of *dimensions* $\mathcal{D} = \{D_1, \ldots, D_N\}$ and a set of *attributes* $\mathcal{A} = \{A_1, \ldots, A_M\}$. Every dimension D_i, $i \in \{1, \ldots, N\}$, is a finite ordered set over a discrete domain $[l_i, u_i]$ that contains the integers between l_i and u_i. For simplicity, we

assume that l_i is always 1. Every combination of dimension values, or indices, $[i_1, i_2, \ldots, i_N]$, defines a *cell*. Cells have the same tuple — or `struct` — type, given by the set of attributes $\mathcal{A}$. Dimensions and attributes define the schema of the array. Based on these concepts, an array can be thought of as a function from dimensions to attribute tuples:

$$Array : [D_1, D_2, \ldots, D_N] \longmapsto \langle A_1, A_2, \ldots, A_M \rangle \tag{2.1}$$

This formalization corresponds to the definition of N-dimensional tensors (Kolda and Bader, 2009). It is different from multidimensional vectors, which do not differentiate between dimensions and attributes. In the vector model (Salton *et al.*, 1975), the array with N dimensions and M attributes is represented as a vector with $(N + M)$ dimensions/attributes. The functional formalization of arrays also makes explicit the functional dependence between dimensions and attributes specific to the relational data model (Codd, 1970), in which the N dimensions form a key of the corresponding relation.

Example 2.1 (Arrays). Two arrays A and B with dimensions `i` and `j`, and attributes `r` and `s` of integer type are depicted in Figure 2.1. Their notation in SciDB's AQL language (Maier, 2012; Lim *et al.*, 2012) is `A,B<r:int,s:int>[i=1,3;j=1,4]`. The range of `i` is `[1,3]`, while the range of `j` is `[1,4]`. The numbers in every non-empty cell are the values of attributes `r` and `s`, e.g., `A[i=1,j=2]` $\mapsto$ ⟨`r=2,s=5`⟩. It is important to notice that ⟨`2,5`⟩ represents a tuple — not a nested array — and can be decomposed into two primitive values ⟨`2`⟩ and ⟨`5`⟩ — both identified by the index combination `[i=1,j=2]`. From the relational perspective, arrays A and B are relations with schema `A,B(i:[1,3],j:[1,4],r:int,s:int)` having the pair of attributes `(i,j)` as primary key.

2.2 Types of Arrays

There are two kinds of array data — dense and sparse — classified according to the number of entries defined for the *Array* function.

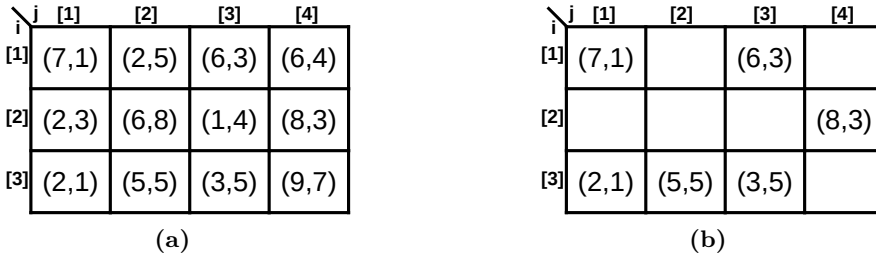

Figure 2.1: (a) Dense array `A`. (b) Sparse array `B`.

An example of each type of array is depicted in Figure 2.1. If *Array* is defined for every entry in the input domain, i.e., for each of the $|D_1| * |D_2| * \cdots * |D_N|$ entries, then the array is considered *dense*, also known as *grid* or *multidimensional discrete data (MDD)* (Furtado and Baumann, 1999). Grids contain values in every cell.

Sparse arrays can be thought of as incomplete grids with missing cells. Intuitively, sparse arrays are obtained by making the size of the domain for every dimension extremely large, while providing values only for a limited number of cells. For example, consider the case of dimensions defined over real-valued domains. Notice that it is also possible to go the other direction — transform sparse arrays into grids. The idea is to condense multiple index values across every dimension into a single scalar, such that all the cells contain at least one value. However, this may result in cells that contain more than a single tuple, case in which there are two alternatives — store the tuples with all — or a part — of their attributes independently or create a single tuple that aggregates the multiple values across the merged cell. This process is similar to histogram binning for data compression or approximation.

An intuitive way to understand the difference between dense and sparse arrays is to look at the expression $Array\,[d_1, d_2, \ldots, d_N]$. In the case of a grid, this expression always returns a valid cell containing data. That is not the case for sparse arrays, in which it is possible that a cell is empty and does not contain valid data, e.g., $B[i = 1, j = 2] \mapsto \langle\rangle$ in Figure 2.1 (b). This can be dealt with by returning a special value —

such as zero in the case of numerical matrices. Thus, from a high-level logical perspective, all arrays can be treated as dense. However, always storing arrays in dense format cannot be practically implemented at the physical layer due to the extensive space requirements. As a result, the strategies to manage dense and sparse arrays are quite different.

2.3 Array Dimensions

The dimensions of an array functionally determine the values of the attributes in every cell. Moreover, they define a candidate key on the array. Thus, dropping any dimension — dimensionality reduction — results in the loss of the functional property. The remaining dimensions define a domain having as many duplicates as the size of the range of the eliminated dimension. In order to preserve the functional property under dimensionality reduction, the original array has to be *split* or *sliced* into multiple arrays with lower dimensionality. The number of these arrays is given by the range of the eliminated dimension. For example, let us assume that we reduce the dimensionality of *Array* by eliminating the first dimension D_1. The result is $u_1 - l_1 + 1$ arrays of dimensionality $N - 1$ given by:

$$\begin{aligned} &Array_{l_1} : D_2 \times \cdots \times D_N \longmapsto (A_1, A_2, \ldots, A_M) \\ &Array_{l_1+1} : D_2 \times \cdots \times D_N \longmapsto (A_1, A_2, \ldots, A_M) \\ &\ldots \\ &Array_{u_1} : D_2 \times \cdots \times D_N \longmapsto (A_1, A_2, \ldots, A_M) \end{aligned} \tag{2.2}$$

A concrete example of slicing the array `A` from Figure 2.1 along its first dimension i is depicted in Figure 2.2.

Evaluating the expression $Array\,[d_1, d_2, \ldots, d_N]$ requires two steps in the sliced representation. First, the d_1 array has to be identified. Then, the expression $Array_{d_1}\,[d_2, \ldots, d_N]$ has to be evaluated instead. Dimensionality reduction can be generalized to any number of dimensions. The result is multiple lower dimensionality arrays. Although the benefits of slicing may not be clear immediately, there are classes of queries that benefit from this representation.

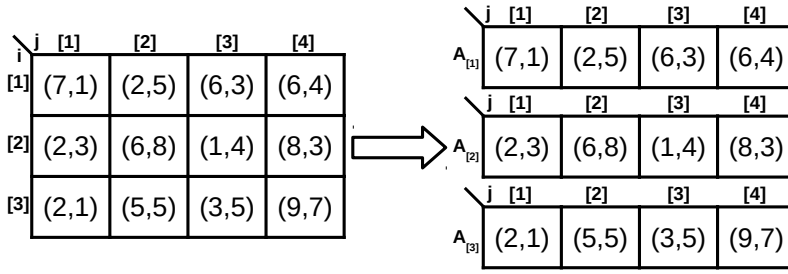

Figure 2.2: Array slicing through dimensionality reduction.

2.4 Arrays and Relations

An array has multiple physical representations as a relation. We present the most common representations for the two 2-D arrays `A` and `B` given in Example 2.1 following the nomenclature used previously in the literature (Luo *et al.*, 2017; Yuan *et al.*, 2021):

- Array as table `T(i,j,r,s)`: The array dimensions and attributes become primitive attributes of the table. There is a tuple for every valid cell in the array. In data warehousing, the array table `T` is called a fact table while the array attributes `r` and `s` are called measures. The table representation is optimal for the sparse array `B` since only the valid cells are included. In the case of the dense array `A`, the dimensions `i` and `j` are redundant across the tuples since a cell can be identified based on its position. Thus, the dimensions do not have to be physically stored.
- Array as tuple `ST(r[][],s[][])`: The entire array is represented as a single tuple, with every array attribute becoming an attribute of the relation. These relation attributes are composite multidimensional arrays that maintain the dimensions of the original array. However, the dimensions are included only in the composite attribute, not as relation attributes. Positional indexing in the array is delegated to the composite attribute. In order to reassemble the attributes inside an array cell, the indexes in `r[][]` and `s[][]` have to be matched. Similar to columnar storage in relational databases, this can be done either positionally or through an equi-join across all the dimensions.

- Array as set of tiles `TI(i_t,j_t,r_t[][],s_t[][])`: The array is decomposed into a set of identical sub-arrays — or tiles — that group contiguous blocks of cells. The tiles are identified by their indices on the contracted dimensions, which become primitive attributes of the table. The tiles are represented as a single tuple. Thus, the number of tuples is equal to the number of tiles. This representation is a generalization of *array as tuple*.
- Array as set of slices: This representation corresponds to array slicing and combines array as table and array as tuple. The dimensions are split into two groups. The dimensions in the first group are represented explicitly as primitive attributes — as is the case for array as table. The dimensions in the second group form a lower-dimensional array for every combination of the explicit dimensions. The array attributes are also represented as lower-dimensional arrays. While the arrays in array as tuple have the same dimensionality as the original array, the arrays in this representation have lower dimensionality. For 2-D arrays, there are two symmetric instances of array as set of slices — array as set of row vectors `RV(i,j[],r[],s[])` and array as set of column vectors `CV(i[],j,r[],s[])`. In the case of row vectors, dimension `i` becomes a primitive attribute while dimension `j` and the array attributes `r` and `s` become composite vector attributes of the table. There is a tuple for every row of the array. This eliminates the redundancy in the representation of the row index — which is stored at most once. For the dense array `A`, there is still redundancy on the column index `j`. The column vectors representation is identical, but applied to columns.

The main difference among these representations is the number of dimension indices that are stored explicitly. At one extreme, we have *array as table*, which represents all the indices, while at the other, *array as tuple*, which represents only the array attributes. The other representations implement various tradeoffs between the two. *Array as table* is the only representation that conforms with the original relational data model (Codd, 1970). All the other representations require support for attributes having a composite array data type and corresponding

functions, which are characteristics of object-relational databases such as PostgreSQL (The PostgreSQL Development Team, 2020). If composite data types are not supported, arrays can be mapped to binary large objects (BLOB), case in which the database system provides only storage while the array manipulations are delegated to the application.

To better understand the difference between arrays and relations, it is important to clarify the distinction between dimensions and attributes. A relation can be viewed as an array without dimensions, only with attributes. Thus, there is no ordering function that allows the identification of a tuple based on dimensional indices. Going from relations to arrays, it is required that dimension attributes form a key in the corresponding relation, i.e., there is a functional dependence from the dimension attributes to all the other attributes in the relation. Since a key is maximal, any attribute can be immediately transformed into a dimension. Converting dimensions into attributes results in breaking the functional dependence property and losing the ordering information. As such, any array can be viewed as a particular type of relation organized along dimensions.

The expression $Array\,[d_1, d_2, \ldots, d_N]$, where $d_i \in [l_i, u_i]$, makes sense for an array and is uniquely determined. The same is true for a relation in which $(d_1, d_2, \ldots, d_N)$ represents a key. However, what distinguishes an array from a relation is that the array is organized such that finding the entry $Array\,[d_1, d_2, \ldots, d_N]$ can be done directly from the value of the indices — the position — without looking at any other entries. This is not possible in a relation since there is no correspondence between the indices and the actual position in the physical representation. Consequently, multidimensional arrays are specialized relations stored physically sorted according to their dimensions.

2.5 Arrays and Tensors

From a mathematical perspective, a tensor can be defined implicitly through a vector space product over a scalar field or explicitly as a concrete multidimensional array (Guo, 2021). Given a basis for the abstract vector space, the tensor is represented by the coordinates in this basis — which are themselves a multidimensional array. For

example, the natural basis of an N-dimensional tensor is given by the set of vectors $\{e_1 = (1, 0, \ldots, 0), e_2 = (0, 1, \ldots, 0), \ldots, e_N = (0, 0, \ldots, 1)\}$. When the basis changes, the tensor coordinates also change according to the transformations in the vector space. However, in the case of explicitly defined tensors, a basis change operation is not common.

In this work, we adopt the explicit mathematical definition of tensors. From this perspective, tensors are data structures that generalize 1-D vectors and 2-D matrices to N-dimensional arrays with three or more dimensions. Moreover, the number of attributes in a tensor is restricted to one — which has to be of a primitive scalar data type (Kim, 2014). Consequently, the two arrays `A` and `B` from Example 2.1 generate four tensors: `A_r<int>[i=1,3;j=1,4]`, `A_s<int>[i=1,3;j=1,4]`, `B_r<int>[i=1,3;j=1,4]`, and `B_s<int>[i=1,3;j=1,4]`, respectively.

2.6 Arrays and Data Cubes

A data cube (Gray *et al.*, 1996; Harinarayan *et al.*, 1996) expands a relational table by computing a set of aggregations over all the possible subspaces created from the combinations of the attributes of such a table (Vassiliadis and Sellis, 1999). The attributes defining the aggregation space are called *dimensions*, while the aggregates are called *measures*. The measures are functionally dependent on the dimension values. With N dimensions and M measures, we obtain an N-dimensional data cube. For example, the two arrays `A` and `B` from Example 2.1 can be seen as two data cubes `DC_A,DC_B<r:int,s:int>[i=1,3;j=1,4]`. While these data cubes seem identical to the corresponding N-dimensional arrays, there are both conceptual and physical representation differences between the two. They stem from the original application that led to their creation — on-line analytical processing (OLAP) for data cubes (Chaudhuri and Dayal, 1997), respectively science for arrays (Baumann, 1994).

Conceptually, the measures of the data cube are the result of aggregation queries that join a fact table with one or more dimension tables. Thus, the data cube is obtained by organizing and storing multiple query results. This can be done by following either a relational representation (ROLAP) or a multidimensional representation (MOLAP). As a result, both relational (Gray *et al.*, 1996; Gyssens and Lakshmanan,

1997) and multidimensional-oriented (Agrawal *et al.*, 1997; Cabibbo and Torlone, 1998; Vassiliadis, 1998) models are proposed to conceptualize data cubes (Vassiliadis and Sellis, 1999; Pedersen *et al.*, 2001; Torlone, 2003). Additionally, data cubes are also modeled as statistical data elements (Shoshani, 1997). Since the dimension values in these models are categorical — not ordinal — a *complete ordering relationship* cannot be defined over the domain of dimensions. This precludes direct mapping of data cube models to index-based — or positional — accessible multidimensional arrays. Nonetheless, measures can be directly accessed based on the dimension values that functionally determine them. For arrays, this requires an intermediate mapping to the dimension index. Dimension hierarchies group dimension values into classes based on inclusion relationships. While the grouping is often based on some form of partitioning, it does not necessarily require a complete order of the dimension values. Moreover, the order at the upper levels of the hierarchy can also be undefined since inclusion does not determine ordering. In conclusion, the difference between data cubes and arrays can be summarized as the difference between hash-based and sort-based data structures. While both of them provide point access, only sort-based data structures support efficient range-based access. In fact, range is not even defined for categorical dimensions.

2.7 Summary

- Arrays, tensors, and data cubes are defined by a set of dimensions and a set of attributes — known as values for tensors and measures for data cubes. The attributes are instantiated for every valid combination of the dimensions.
- In the case of a dense array, all the dimension combinations are valid, while in a sparse array, only a subset of combinations is valid.
- An array can be seen as a function from dimension values to attribute values since there is a functional dependence between the two.
- While arrays and tensors are conceptually identical, tensors can include constraints on values based on their dimensions.

- An array can be represented as a relation with a composite key defined over the dimensions. However, this still does not allow for positional access due to the unordered set property of relations.
- Positional access is permitted in a data cube for equality — or point — conditions on dimensions. In the case of an array, more general range conditions are supported.

3

Multidimensional Array Operations

Array processing is a common operation across multiple domains, including image processing, scientific computing, and machine learning. This results in a multitude of array operation types with different parameters and processing requirements. The common characteristic across all these operations is *indexing*, which provides direct access to an array cell based on its dimensions or position. Concretely, the expression $A[1, 2]$ identifies the cell on the first row and second column in Figure 2.1(a).

In this section, we identify and categorize the most important classes of operations on arrays, relations, tensors, and data cubes starting from scientific and linear algebra applications. These operations are subsequently formalized in array algebras, used as drivers for designing optimized array storage methods, and implemented as primitive operators in array processing systems.

3.1 Array Operations in Scientific Applications

The SS-DB benchmark (Cudre-Mauroux *et al.*, 2010) is modeled based upon a real workflow for processing astronomical images from the Sloan Digital Sky Survey (SDSS) (Szalay, 2008). Although application-specific, SS-DB includes a full spectrum of operations over arrays representa-

tive across various scientific domains. SS-DB contains queries 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). The SS-DB benchmark defines complex pipelines of composable array operators for observation extraction, grouping, and querying. These pipelines or query plans are equivalent to the relational algebra trees that encode advanced operations on tables. We present the benchmark pipelines and identify the relevant array operations.

Observation extraction. Observations are extracted, i.e., "cooked", from the cell values of dense 2-D grids/images that satisfy a given condition. This condition can take the form of a simple predicate or be expressed as a complex user-defined function (UDF) over the cell values. Adjacent cells satisfying the condition are clustered together into an observation. As an example observation, consider the image labeling operation that clusters adjacent pixels with the R component in RGB having values larger than 50. Adjacency is defined as a rectangle with configurable size centered on every grid cell. In addition to the data corresponding to every cell belonging to an observation, a set of aggregated attributes — such as the center, the bounding box, and the boundary polygon — are derived for the entire observation. Observations are represented as sparse 2-D arrays in the same dimension domain as the grids they are extracted from. Cell values include the unique identifier assigned to the observation and the aggregated attributes.

Multiple array operations are performed during observation extraction. Filtering the cells based on their value — or a UDF applied to the cell attributes — is equivalent to the relational selection operator. Aggregation across multiple cells is identical to relational tuple aggregation. However, the cells that are aggregated satisfy an adjacency relationship given as a shape parameter. This corresponds to the *stencil* operator from scientific computing (Datta *et al.*, 2008; Maruyama *et al.*, 2011) and *convolution* from image processing (O'Gorman *et al.*, 2008; Lippmeier and Keller, 2011). As depicted in Figure 3.1, the stencil/convolution is applied to every cell in the input array — shape centered on every cell, to be precise. The value of the output array cell is the aggregate of the cells covered by the shape argument. Finally, a

positional join between the input array and the extracted observations is required to derive the aggregated properties of observations.

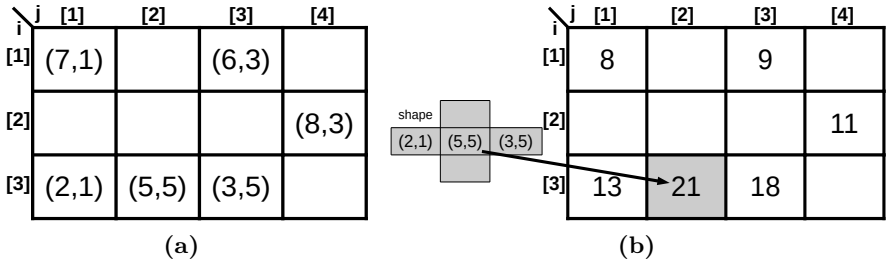

Figure 3.1: (a) `B<r,s>[i=1,3;j=1,4]`. (b) `STENCIL(B, SUM(r+s), shape)`.

We summarize the sequence of array operations that implement observation extraction — also depicted in Figure 3.2 — in the following:

1. `FILTER` the grid with the cell-based condition. This operation transforms the dense grid into a sparse array having as valid cells only those cells satisfying the predicate.
2. Assign a unique id to every valid cell in the sparse array. The `id` is an additional attribute in the sparse array and can be derived from the cell indices. While not necessarily required, the id provides an easier to understand identification of observations.
3. Perform a sequence of `STENCIL` operations with arguments the adjacency shape and the minimum aggregate over the `id` attribute of the sparse array. The result is the observation array with the same `id` in the cells belonging to an observation. `STENCIL` has to be invoked iteratively until the input and result arrays are identical.
4. Execute a structural/positional `JOIN` between the original grid and the observation array to get the attributes corresponding to the cells in the observation.
5. Derive the aggregate attributes of the observation by applying another `STENCIL` operation with the same shape and the corresponding aggregate functions.

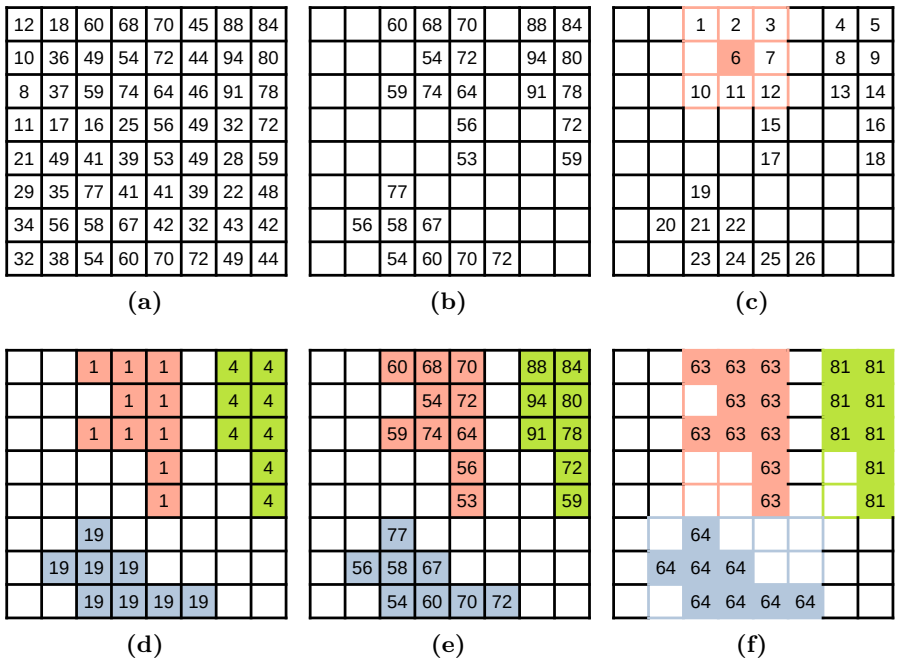

Figure 3.2: Observation extraction from the grid depicted in (a): `FILTER` out the cells with a value larger than 50, generating a sparse array (b); assign a unique id to every valid cell and perform a sequence of `STENCIL` operations with the shown shape and the minimum aggregate (c); the resulting three observations (d); structural `JOIN` between the (b) and (d) arrays to retrieve the data corresponding to every cell in the observation (e); the aggregate value — in this case, average — and the observation bounding box computed with another `STENCIL` operation (f).

Observation grouping. A group contains observations extracted from different grids that have their centers close to each other. Closeness is specified through a distance UDF rather than using a fixed shape. Nonetheless, adjacency can be represented as an irregular 3-D array derived based on a discretized version of the distance function. Thus, we can view grouping as a 3-D version of observation extraction with an irregular adjacency shape operating on observation centers. Consequently, the computation of grouping is similar to that of observation extraction and includes a sequence of stencil operations followed by a positional join and an aggregation.

Queries. SS-DB queries consist of sequences of array operations, including selection based on the cell attributes, stencil aggregations, and dimension translation. The stencil operations differ in terms of the cells where they are invoked, the shape of the neighborhood, and the aggregation function. A general characteristic across all the queries is that they operate on a portion of the space instead of the entire grid. This operation is known as *subsampling* or *range* filtering and provides access to the cells in a subspace of the domain identified by the range on every dimension. Subsampling is a direct application of indexing since it selects cells based on their position/dimensions. It is applied as an initial step before any of the other operations. The size and position of the filtering subspace control the difficulty of the query. The larger the subspace, the larger the number of cells considered by the other array operations.

3.2 Relational Operations

Since arrays are a particular type of relation in which the dimensions form a key, all the operations in relational algebra are directly applicable to arrays. However, due to the storage organization optimized for dimension — or index — access, operations on dimensions are more efficient than operations on attributes. Arguably, a similar effect can be obtained with a set of indexes layered on top of a relation. From this perspective, array operations can be seen as relational operations optimized for a particular storage organization.

The duality between arrays and relations is evident when considering the correspondence between relational algebra and the array operations in the SS-DB benchmark. While every operation from SS-DB can be expressed in terms of relational algebra operators, the operations that require the notion of adjacency or proximity — which is inexistent on relations — have a higher degree of complexity. Subsampling and stencil are two operations in this category. By default, the subsampling ranges are expressed as predicates on dimensions. When subsampling is applied around an anchor point, the predicates become more complex. The stencil operation takes this to another level by performing a subsampling-based aggregation at every cell. While relational window-

based aggregation may look as a good fit for this, it is intrinsically a 1-D linear operation that works optimally on streams. Its generalization to multiple dimensions is not straightforward — not to mention an optimal implementation. Distance-based operators such as similarity join and self-join are other relational extensions that have related semantics to the subsampling and stencil operations (Zhao *et al.*, 2016). The main difference is that they are defined over a continuous domain while subsampling and stencils are defined over discrete arrays. In addition to these conceptual issues, the physical organization of arrays is natively optimized for proximity operations. On the other hand, relational multidimensional indexes such as R-trees and quad trees are secondary data structures that target only a particular type of data access.

3.3 Tensor Operations

The NumPy API (NumPy Development Team, 2022) is one of the most extensive libraries for tensor operations. While the API includes multiple types of indexing and aggregations, value-based selection, reshaping, concatenating, padding, and sorting, the tensor-specific operations — which are not common among the scientific array operations and relational algebra — are the BLAS linear algebra operations. Basic Linear Algebra Subprograms (BLAS) (Wikipedia, 2020) are the de facto specification for tensor operations. BLAS classifies tensor operations at multiple levels based on the dimensionality of the tensor operands. BLAS level 1 includes operations between 1-D arrays, such as the dot product of two vectors. BLAS level 2 extends to matrix-vector multiplication — an operation between a 2-D matrix and a 1-D vector — while BLAS level 3 is centered around the generalized matrix-matrix multiplication operation. In the space of multidimensional tensors, matrix multiplication is a special form of *tensor contraction* (Shi *et al.*, 2016; Matthews, 2016; Springer and Bientinesi, 2018; Kim *et al.*, 2019) — the most general operation between two tensors proposed in BLAS level 4 (Springer and Yu, 2019). In tensor contraction, for every index in the two input and the single output matrix, one or more tensor dimensions are substituted. For example, the contraction of a 7-D tensor by a 5-D tensor into a 6-D tensor corresponds to the substitution of a 4-D by 3-D and a 3-D by

2-D tensor, respectively, with a 4-D by 2-D tensor — three common dimensions are eliminated. This operation is mathematically equivalent to *multidimensional matrix multiplication* when the metric tensor g defining the distance between two points is the identity matrix. Thus, we treat matrix multiplication as the most representative tensor operation.

Matrix multiplication. Consider the multiplication of two matrices A and B with dimensions $m \times k$ and $k \times n$, respectively. The result matrix $C = A \cdot B$ has dimensions $m \times n$. Its elements are computed as:

$$C_{ij} = \sum_{p=1}^{k} A_{ip} \cdot B_{pj}, \quad \text{where } 1 \leq i \leq m, 1 \leq j \leq n \tag{3.1}$$

In order to obtain the element C_{ij} of the result, the dot product between row i from A and column j from B has to be computed. This access pattern is applied to every row-column pair (i, j), which corresponds to a join on the common dimension followed by a summation. While this relational mapping is possible, it is likely not optimal. Instead, given the ubiquity across application domains, matrix multiplication is provided as a primitive operation in BLAS libraries.

3.4 Data Cube Operations

Given the exhaustive nature of the data cube (Gray *et al.*, 1996), in which the aggregates of all the possible dimension combinations are materialized, the most common data cube operation is to access a particular cell and extract the corresponding aggregate. This indexing operation requires the specification of the value for every dimension. If a dimension is omitted — represented as $*$ — the default semantics is to include all the values on the missing dimension in the result — which is not a problem because this aggregate is also materialized. Range predicates on dimensions are not permitted because there is no ordering among the categorical coordinates. Instead, a set of coordinates can be specified by enumerating their values. These indexing variations are restricted instantiations of subsampling.

3.5 Summary

- The most common array operations are *indexing* and *subsampling*. Indexing provides direct access to a cell specified by its dimensions/coordinates. Subsampling provides access to the cells in a subspace of the domain identified by the ranges on every dimension.
- The *stencil/convolution* is a ubiquitous array operation in scientific applications. The stencil computes the aggregate value of a group of adjacent/neighboring cells. Its access pattern is specified by a shape parameter, e.g., cross, hexagon, etc.
- *Matrix multiplication* and its extension to more than two dimensions — tensor contraction — are the most general linear algebra operations. Their access pattern pairs every row from one matrix with every column from the other.
- While most array operations can be written as SQL statements and sequences of relational algebra operators, this may lead to unacceptable complexity and performance.

4

Algebras and Query Languages for Multidimensional Arrays

When designing an array query language, the variety of array operation types across the many application domains has to be considered. While several attempts have been made over the years (Tomlin, 1990; Ritter *et al.*, 1990; Baumann, 1999), to date, there is no commonly accepted array query language similar to SQL. The common trend among the proposed languages is to first identify an array algebra — a set of primitive operators that can express as many array operations as possible — and then to design a query language that resembles SQL on top of the identified operators — typically array extensions to SQL. The challenge faced when identifying the primitive operators is the diversity of the array operations introduced in Section 3. The standard solution is to allow for second-order operators — operators that take user-defined functions as arguments — in the algebra. Writing composite expressions of array algebra operator invocations is the first step in designing a query language. Several attempts stop at this stage. Adding a more elevated syntax on top of the pure algebra operator invocation is the next stage. To encourage adoption, the proposed syntax is quite often a modification to SQL — if not simple extensions with new keywords corresponding to the array algebra operators. In this more advanced scenario, query

execution requires mapping the higher-level language constructs into array algebra operators, which represent the only implemented functions that can be executed. If multiple mappings are possible — the case when multiple implementations for the same operator are available or when the query expression permits it — the optimal mapping has to be determined. This process corresponds to query optimization.

In this section, we present the most important array algebras and query languages proposed in the database literature and discuss if and how the observation extraction operation introduced in Section 3.1 can be expressed in every case.

4.1 Array Query Language (AQL)

AQL (Libkin *et al.*, 1996) is a declarative query language for multidimensional 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 (NRCA), which plays the same role relational calculus and algebra play for the relational data model. Types and functions represent primitives in NRCA. The types include booleans, natural numbers, tuples, finite sets, and arrays defined over rectangular domains with indices ranging over initial segments of the natural numbers. Functions are defined from one type to another. The constructs supported in NRCA not involving arrays are standard in nested relational calculus and include functions, products, set constructs, ordering, nesting, and arithmetic operators for natural numbers.

There are four basic array operators in NRCA:

- Define or tabulate an array
- Extract an array element at a given index
- Extract the dimensions of an array
- Convert an indexed set into an array

These array operators together with the standard constructs in the nested relational calculus are sufficient to express any operation on multidimensional arrays, including mapping a function to every element of the array, zipping multiple arrays together, i.e., positional

natural join, extracting a subsequence — not necessarily contiguous — from an array, reversing, transposing, and projecting an array, and matrix multiplication. To simplify programming, a series of derived constructs such as comprehensions, patterns, and blocks are also added as operators in AQL — in a similar manner to the operators in extended relational algebra. This allows for expressing array operations in a higher-level language that hides the user from implementation details and is amenable to optimizations that would have to be implemented explicitly by the programmer otherwise. The negative effect of this is a reduction in the expressiveness of the operations that can be coded directly in the language. AQL addresses this drawback by providing extensible support for integrating user code dynamically in the language.

An implementation of AQL in the ML functional programming language (Milner *et al.*, 1997; Laboratory for Foundations of Computer Science at the University of Edinburgh, 2008) is introduced in Libkin *et al.* (1996). AQL constructs are supported as library functions written in ML and made available as language operators. Queries are written as ML programs invoking these operators on the input data. Thus, there is no higher-level query language beyond the NRCA constructs. AQL takes advantage of the advanced programming features, such as second-order functions, available in ML. This is the main feature used to provide support for user-defined functions (UDF), thus, extensible and customizable array processing. The operators are executed in full and intermediate results are materialized after every invocation. Part of query execution, the AQL constructs go through a series of transformations meant to generate an optimal execution plan that is eventually executed as calls to routines in the AQL library. Notice, though, that this type of optimization does not map a higher-level language to AQL — rather it rewrites a sequence of function invocations optimally.

Example 4.1 (AQL)**.** In order to implement the observation extraction operation in AQL, the three required array operations are `FILTER`, `STENCIL`, and index-based `JOIN`. While AQL supports `FILTER` and `JOIN` — called `zip` — it does not contain anything that closely resembles `STENCIL`. AQL does not have the concept of adjacency or shape pattern. However, since AQL has extensive support for UDFs, it is conceivable

that a `STENCIL` UDF combining multiple index expressions can be written.

4.2 Array Manipulation Language (AML)

The Array Manipulation Language (AML) (Marathe and Salem, 2002) is a functional query language defined over an algebra consisting of three operators that manipulate dense arrays. Every operator takes one or more arrays as arguments and produces an array as result. All of the AML operators take bit patterns as parameters. Patterns are not allowed to refer to array element values. This restriction implies that the shape of the result of an AML operation can always be statically determined — without actually evaluating the operator — if the shapes of the operator's array arguments are known — the same is true for the schema of the result relation in relational algebra. This property is useful when evaluating AML expressions since it implies that the space required to implement an AML operation can be determined in advance. AML expressions can be treated declaratively and can be subject to rewrite optimizations according to equivalence rules between operators.

The AML algebra operators are presented in the following:

- *SUB.* Subsample is a unary operator that extracts a sub-array along a given dimension. It takes an array, a dimension number, and a bit pattern as parameters, and produces an array, i.e., $B = SUB_i(P, A)$, where A is the input array, P is the bit pattern, and i is the dimension. SUB divides A into slabs along dimension i and then retains or discards the slabs based on pattern P. If P[k] = 1, then slab k is concatenated to the result array B. Two subsequent SUB applications to two different dimensions of the same array produce the same result independent of their order, i.e., SUB is commutative across dimensions. The resulting array can be inferred from the two bit patterns without the need to actually compute the result of each individual SUB operation — property applied in query optimization.
- *MERGE.* Merge is a binary operator that combines two arrays defined over the same domain. The merge operator takes as pa-

rameters two arrays A and B, a dimension number i, a bit pattern P, and a default value δ. It merges the two arrays to produce the result array C, i.e., $C = MERGE_i(P, A, B, \delta)$, by dividing A and B into slabs along dimension i and then merging the slabs according to pattern P. In this case, the pattern defines the order in which the slabs from A and B are added to the result C. Due to a shape mismatch between A and B, some values in C may be undefined. δ is assigned to all such undefined values. It is important to remark that MERGE can be used to increase the dimensionality of an array. MERGE is commutative and associative when applied to the same dimension with different patterns. The corresponding patterns can be easily determined. SUB and MERGE can be reordered both when applied to the same dimension as well as when applied to different dimensions. The corresponding patterns have to be determined from the patterns in the original expression. Choosing the optimal rule to apply is handled in query optimization.

- *APPLY.* Apply applies a user-defined function to an array to produce a new array. It is written as $B = APPLY(f, A, P_0, P_1, \ldots, P_{N-1})$, where f is the function to be applied, A is the array to apply it to, P_i's are patterns, and N is A's dimensionality. The structural relationship among the array cells f is applied to is made explicit by the patterns P. The stencil/convolution operations (Figure 3.1) can be expressed through pattern combinations. f is required to be defined such that it maps sub-arrays of A of some fixed shape D_f to sub-arrays of B of some fixed shape R_f. APPLY applies f to some or all of the sub-arrays of shape D_f of A. The pattern arguments specify to which of the possible sub-arrays of the input array A function f is applied. Pattern P_i selects the slabs in dimension i. f is applied to the sub-array with origin at x only if x belongs to selected slabs in all the N dimensions. Moreover, the sub-arrays to which f is applied to must be entirely contained within A. The results of these applications are concatenated to generate B. The arrangement of the resulting sub-arrays in B preserves the spatial arrangement of the selected sub-arrays in A. Applying a function to every cell and to the entire array are

special instances of APPLY. The structural locality captured by patterns can be used to reduce the number of applications of f and to prune portions of the input array.

The three logical AML operators are implemented with six physical operators that produce their result array based on the order of the input array(s). Marathe and Salem (2002) provide concrete examples with the mapping between the logical and physical operators. The underlying characteristic of the physical operators is generating the output array one-cell-at-a-time, which simplifies processing at the expense of extensive caching. In order to reduce the overall cache utilization, adjacent operators have to share the same processing order. This is achieved by optimally inserting array reordering operators between mappings.

AML is designed starting from an image algebra that defines the most common operations in image processing. AML defines only those operators that are amenable to declarative optimization, which includes a sufficiently large class of image processing algorithms. With singleton APPLYs, i.e., APPLY is defined for every array cell individually, AML encompasses almost all the image processing algorithms. AML is a functional programming language in which operators are nested as arguments to other upper-level operators to form queries. Processing functions are also passed as functor arguments — second-order functions — to operators, i.e., the function argument to APPLY. Query optimization involves simple rewriting rules that replace combinations of algebra operators with other such combinations deemed optimal. Thus, AML is more like an elevated execution plan description rather than a declarative array query language. Another AML limitation is that it contains only structural operators, i.e., operators that consider the indexes. While image processing represents a large class of array manipulations, it is interesting to investigate how AML can be extended to other array operations that are not originating from image processing.

Example 4.2 (AML). The implementation of observation extraction in AML uses only the `APPLY` and `MERGE` operators. `FILTER` is implemented as an instance of `APPLY` with the identity shape pattern and the predicate as a condition to filter the valid cells. `STENCIL` is the exact equivalent of `APPLY`. Positional `JOIN` requires a `MERGE` followed by `APPLY`. `MERGE` is

the only AML operator that creates an array from two input arrays. It is used to generate an array with twice as many rows as the array (e) in Figure 3.2 by interleaving rows from array (b) and (d). `APPLY` combines groups of two consecutive non-overlapping rows into a single row using a vertical two-cell shape and the identity value function. This results in array (e) from which `APPLY` can derive the observation properties.

4.3 Relational Array Mapping (RAM)

RAM (Ballegooij, 2004) is an array processing system built on top of the MonetDB (Idreos *et al.*, 2012) relational database. While RAM deals with dense arrays, SRAM (Cornacchia *et al.*, 2008) is targeted at sparse arrays commonly used in information retrieval applications. Nonetheless, both systems employ similar array formalizations based on the comprehension syntax, which represents arrays as functions defined over dimensions and taking primitive type values. Dimensions are defined over continuous integer intervals starting at 0 for a regular array shape — not necessarily symmetric, though. Array decomposition — an array with composite type values is represented as a set of aligned arrays with primitive type values — is default in RAM due to the columnar data representation in MonetDB. Since the execution happens inside a relational database engine, array queries follow a sequence of transformations that map 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.

The RAM query language consists of methods to extract values from arrays and methods to construct arrays. Value extraction is supported natively through array application since arrays are functions that can be applied to index values in order to yield results. Array construction is supported through a generative comprehension constructor and a concatenation operator. There is no query language syntax defined for these functions. They are exclusively theoretic notations expressed in comprehension syntax.

The RAM array algebra consists of six operators that implement the query language, create arrays and extract values based on indexes only. The *const* operator fills a new array with a constant value, whereas the *grid* operator creates an array with values taken from one of the indexes. *map*, *apply*, and *choice* are induced operators that operate on cell values. *map* creates a new array by applying a given function to one or multiple aligned arrays. *apply* replaces the function in *map* with an array interpreted as a function from indexes to values. *choice* is a combination of *map* and *apply* in which an array with boolean values selects the elements of a newly created array from the elements of two arrays passed as arguments. The *aggregate* operator applies an aggregate function to the array elements having the same value for the first k indexes, resulting in an array with smaller dimensionality.

In addition to the RAM operators, the SRAM array algebra (Cornacchia *et al.*, 2008) introduces a series of structural operators. *pivot* permutes the dimensions of an array according to an axis order permutation. *rangeSel* is the standard subsample operator, which extracts a sub-array with the same dimensionality from an array passed as argument. *replicate* generates an array with dimensionality $N + 1$ by replicating the original array a specified number of times. *topK* is a specialized operator that works only for vectors and creates an array with the indexes of the first K values in a specified order.

The mapping of the extended SRAM array algebra operators to relational operators is presented in Cornacchia *et al.* (2008). It is specific to the chosen relational representation of arrays in MonetDB. Sparse arrays are stored as relations clustered and indexed based on the array dimensions. The order is chosen arbitrarily as the lexicographical dimension order, i.e., the order in which dimensions are specified in the array definition. Only the cells with valid values are stored explicitly. The mapping of *apply* as a series of joins followed by a projection is presented as a canonical mapping for all the structural operators — *pivot*, *rangeSel*, and *replicate*. *map* between two dense arrays corresponds to relational join followed by function application. In the case of sparse arrays, the general form of outer join is used instead. *aggregate* can be mapped into a standard group-by aggregate relational operator on the dimensions, while *topK* does not have a relational equivalent. In

addition to the mapping rules from array algebra operators to relational operators, a series of simplification and rewriting rules are also proposed. They form the basis of the query optimization process.

Example 4.3 (RAM)**.** It is not clear how the three operators from observation extraction can be expressed in RAM. `FILTER` is the easy one because it is a *map* instance with the predicate as the function applied to every cell. *choice* is the only alternative for implementing structural `JOIN` since it is the only operator taking two input array arguments. Array (d) in Figure 3.2 is used as the boolean selector while array (b) provides the values. Although *apply* may seem sufficient to implement `STENCIL`, it lacks the aggregate function applied to the shape pattern. This is assuming that the shape can be represented as the array argument passed to *apply*. Moreover, the RAM *aggregate* operator implies dimensionality reduction on the input array — which is not the case for `STENCIL`. Therefore, it is fair to say that `STENCIL` has no immediate RAM representation.

4.4 RasDaMan Query Language (RasQL)

The RasDaMan array algebra (Baumann *et al.*, 1998; Baumann, 1999) conceptualizes arrays as functions from rectangular domains to cell values, similar to AQL (Libkin *et al.*, 1996). The algebra contains three core constructs that can express every array operation when composed together (Baumann, 1994). The execution of each of these constructs is iteration-based and safe — it does not require recursion. While user-defined functions can be integrated in the algebra, they are not fundamental. The authors advocate against their use due to the complications they introduce in query optimization.

The three core constructs in the RasDaMan array algebra are:

- *MARRAY.* The array constructor MARRAY creates new arrays by indicating a spatial domain and an expression that is evaluated for every cell position of the spatial domain. An iteration variable bound to a spatial domain is available in the cell expression so that a cell's value can depend on its position.

- *COND.* The condenser COND takes the values of an array's cells and combines them through the operation provided — which has to be commutative and associative — thereby obtaining a scalar value. An iterator variable is bound to the array spatial domain to address cell values in the condensing expression.
- *SORT.* The array sorter SORT proceeds along a selected dimension to reorder the corresponding hyper-slices. It rearranges a given array along a specified dimension based on an order-generating function that associates a sequential position to each (N-1)-dimensional hyper-slice, without changing its value set or the spatial domain.

While these three operators are minimal to make the array algebra complete, a series of derived operators are added to the algebra to enhance usability. They include trimming and slicing, operators induced by the underlying type of the array cells, and multiple aggregates that are particular condenser instances. The result is an extended array algebra identical in spirit to the extended relational algebra.

The RasDaMan array algebra is integrated in relational algebra and SQL following the array-as-attribute approach (Misev and Baumann, 2014). This requires the extension of relational operators with support to handle arrays. While extended projection is straightforward, selection and join require that array expressions return boolean values. The integration also requires the definition of operators that convert between arrays and relations. These operators are NEST and UNNEST, originally introduced in Jaeschke and Schek (1982) and refined in Ozsoyoglu *et al.* (1987) and Cao and Badia (2007).

Having the proposed array algebra as a theoretical foundation, RasQL is a declarative query language that extends SQL-92 with support for arrays. In RasQL, array expressions can appear in the SELECT and WHERE clauses of a SQL query. Special language constructs are introduced for the core array algebra operators — MARRAY, COND, and SORT — which can then be integrated with standard SQL. However, following the SQL standard, arrays are treated as a composite attribute type with a set of corresponding operators. Nonetheless, RasQL is the first complete array query language that integrates both an algebra and a higher-level declarative query language.

The generality of RasQL. Baumann and Holsten (2011) provide a comparison of AQL, AML, (S)RAM, and RasQL. In all these models, arrays are conceptualized as functions from hyper-rectangles to primitive or composite values. Array creation is specified using either tabulation (RasQL) or comprehension (AQL and RAM). Operations are defined as functionals, i.e., second-order functions taking other functions as parameters. While this generates a small set of operators, a large part of the complexity is hidden in the functional parameters. An important issue that has to be addressed is how many physical operators to implement and make available through the language syntax? The answer varies from all operators to only the operators in the algebra. Baumann and Holsten (2011) also show 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 — the comprehension syntax is the basis for AQL and RAM. The equivalence between AML and RasQL is proven directly. Since it is valid in both directions, AML and RasQL are equally expressive.

Example 4.4 (RasQL)**.** Since RasQL has the same expressive power as AML and observation extraction can be expressed in AML, it follows that the extraction can also be expressed in RasQL. `FILTER` and index-based `JOIN` are mapped as two instances of the MARRAY constructor operator. For `FILTER`, MARRAY keeps the value of the cells that satisfy the selection predicate and assigns a special value to the other cells. `JOIN` is an induced MARRAY operator that uses the observation array (d) in Figure 3.2 as a boolean selector for array (b). Since `STENCIL` requires aggregation, it is expressed with the COND operator. The shape pattern is provided as an explicit array to COND. The application of COND to every cell is controlled by an instance of MARRAY that iterates over the input array. When these two are grouped together, `STENCIL` consists of the sequence $MARRAY_{input\_array}(COND_{shape}(cell))$. This expression is written in the SQL-compatible RasQL language as follows:

```
SELECT MARRAY array_(d) IN domain(array_(a))
       VALUES COND MIN OVER x IN domain(shape)
FROM array_(c)
```

We do not include the statements for `FILTER` and `JOIN` because they have exactly the same syntax as SQL — the `WHERE` clause includes predicates while multiple arrays in the `FROM` clause imply a structural join.

4.5 Science Query Language (SciQL)

SciQL (Kersten *et al.*, 2011; Zhang *et al.*, 2011) 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. It is heavily targeted at experienced SQL programmers. While this is considered to be one of the most distinctive characteristics of SciQL from a database perspective, it may also be a drawback given the reduced familiarity the science community has with SQL.

SciQL provides all the benefits of a declarative query language that isolates an abstract data model from the physical data representation. Arrays are defined by specifying the dimensions, their corresponding ranges, and the array cell content. Named dimensions allow for direct indexing of the array elements. A default value is assigned to all the cells in the array at declaration. Arrays can appear wherever tables are allowed in SQL statements. The result of a query is an array only when the column list of a SELECT statement contains dimensional expressions. The SQL iterator semantics associated with tables extends to arrays, but iteration is confined to the cells whose values are not NULL. However, this may be quite inefficient for operations that require array traversal in a particular order.

Array creation and modification statements follow entirely the syntax corresponding to tables. The only difference is that dimensions have to be defined explicitly for arrays. Converting arrays to tables can be done by simply selecting all the array cells without specifying any dimensional expression. The reverse is not that straightforward since the designated dimensions have to form a primary key in both representations. If the result of a query is an array, it has to be specified explicitly in the SELECT clause. Cell selection and array slicing are performed using the bracketed index syntax from `C/C++`.

The most specific array operator introduced in SciQL is structural grouping — in fact, it is a syntactic representation for the *APPLY* operator in AML (Marathe and Salem, 2002). It consists in placing a template at every position in the array and computing an aggregate for all the cells in the neighborhood that are covered by the template. The result is an array with the same dimensions. Two versions are proposed — with and without overlap — corresponding to sliding and tumbling window aggregates from data streaming — which are both supported in SQL. Essentially, structural grouping is a multidimensional generalization of relational window aggregates. SciQL provides extensibility through user-defined operators. They can be implemented using primitive SciQL constructs — similar to stored procedures — or can be imported from an imperative programming language such as `C/C++` — similar to UDFs in relational databases. In addition to multidimensional array operations, SciQL supports a large range of time-series operators, which are most of the time instances of 1-D structural grouping.

Example 4.5 (SciQL). The `STENCIL` operation is written in SciQL as structural grouping:

```
SELECT [x], [y], MIN(id)
FROM array_(c)
GROUP BY array_(c)[x-1:x+1][y-1:y+1]
```

`x` and `y` correspond to the array dimensions. The shape pattern is included in the `GROUP BY` clause as ranges on dimensions. The other two operations in observation extraction — `FILTER` and `JOIN` — follow the standard SQL syntax.

4.6 SciDB Query Languages

SciDB (Stonebraker *et al.*, 2011) is a shared-nothing parallel database system designed specifically for array processing. SciDB queries can be written in two languages — Array Functional Language (AFL) and Array Query Language (AQL). AFL is a functional language in which the execution plan is expressed exactly in the same format as in

AML (Marathe and Salem, 2002). A slight difference is that the number of operators is larger than in AML. The reason is that instances of APPLY that execute a specific operation are promoted to stand-alone operators with their own name. We present AQL and its formal ArrayQL algebra (Maier, 2012) in the following.

ArrayQL algebra. In ArrayQL, arrays are defined 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. ArrayQL 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. We present the most important operators defined in the 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 — it transforms a sparse array into a dense one.
- `APPLY` applies a function to every valid cell of an array. Unlike the AML APPLY (Marathe and Salem, 2002), the value of the output cell is a function of only the input cell — not multiple adjacent cells — since no shape parameter is specified.
- `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.

- `REDUCE` generates a reduced version of an array by aggregating over one or more dimensions. Supported aggregate functions include the standard relational algebra and SQL aggregates.

AQL. AQL 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 constraints. 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. However, not all ArrayQL algebra operators are specified in the language. Moreover, not all the operations possible in the language by means of intricate index mappings are part of ArrayQL algebra.

Array joins. Consider two N-dimensional arrays α and β given in the functional representation (2.1):

$$\alpha : \{\mathcal{D}^\alpha = D_1^\alpha \times \cdots \times D_N^\alpha\} \longmapsto \{\mathcal{A}^\alpha = (A_1^\alpha, \ldots, A_M^\alpha)\}$$
$$\beta : \left\{\mathcal{D}^\beta = D_1^\beta \times \cdots \times D_N^\beta\right\} \longmapsto \left\{\mathcal{A}^\beta = (A_1^\beta, \ldots, A_M^\beta)\right\}$$

A join between arrays α and β, $\tau = \alpha \bowtie_P \beta$, is written in AQL as: `SELECT expression INTO` τ `FROM` α `JOIN` β `ON P`, where P is the join predicate, which consists of pairs of attributes and/or dimensions from the two source arrays. The output is a new array $\tau : \mathcal{D}^\tau \longmapsto \mathcal{A}^\tau$ having the schema:

$$\mathcal{D}^\tau = \mathcal{D}^\alpha \cup \mathcal{D}^\beta \qquad \mathcal{A}^\tau = \mathcal{A}^\alpha \cup \mathcal{A}^\beta$$

in which both the dimensions and the attributes from the input schemas are merged. Essentially, the result array has dimensionality equal to the sum of the dimensionality of the input arrays and every cell contains the union of the attributes. It is important to notice that the non-empty cells are given exclusively by the combination of non-empty cells in the input arrays. An array outer join generates a valid cell as long as one input cell is valid. As is the case with the relational cross product, the default array join is not of particular practical importance since it pairs every cell from α with every cell from β.

Duggan *et al.* (2015b) introduce a series of array equi-joins — *dimension:dimension*, *attribute:attribute*, and *attribute:dimension* — for the case in which predicate P contains only equality conditions. These joins have corresponding `INNERDJOIN` and `INNEREJOIN` operators in the ArrayQL algebra. Out of the three types of array equi-join, dimension:dimension join is specific to array databases while the others are instances of the relational join operator. ChronosDB (Zalipynis, 2018) extends dimension:dimension joins to more than two arrays based on a reducer function that derives the result cell from the input cells.

The *array similarity join* operator (Zhao *et al.*, 2016) is a generalization of the dimension:dimension join to predicates other than equality. The join predicate between arrays α and β is expressed by extending the AML APPLY operator (Marathe and Salem, 2002) with a mapping function that assigns a unique cell Ψ in β to every cell Υ in α and applying the shape/pattern to Ψ rather than Υ. The mapping function and similarity shape can encode a large range of relationships between cells — including all the L^p norms and Hamming distances. Moreover, the array similarity join supports uncommon discrete shapes such as arrays with empty cells and non-symmetric arrays, which cannot be expressed as implicit distance functions. Xing and Agrawal (2019) introduce the value similarity join operator as a combination of dimensional equi-join and attribute similarity join. This operator outputs cells that have identical indices and attribute values within the specified range ϵ.

Example 4.6 (SciDB)**.** The ArrayQL operators `FILTER` and `INNERDJOIN` can be immediately applied to observation extraction. `STENCIL`, however, does not have a corresponding operator. It has to be implemented as a UDF. Thus, observation extraction is not directly expressible in SciDB.

4.7 Algebras for Domain Specific Data

AQUERY (Lerner and Shasha, 2003) uses the concept of "arrables", i.e., ordered relational tables, and SQL queries extended with an ASSUMING ORDER clause to represent one-dimensional time series data. A blob-based approach where an algebra for the manipulation of irregular topological structures is applied to the natural science domain is proposed in Howe and Maier (2004).

The ChronosDB array data model (Zalipynis, 2018) is an abstract representation for geospatial data stored in raster file formats. It provides array mapping to a raster file that is independent of the storage format and supports arrays partitioned across multiple files distributed over the nodes of a cluster. This is achieved with a two-level structure consisting of a user-level array mapped over a set of system-level subarrays corresponding to the raster files. The array operations defined in ChronosDB include retiling, join, aggregation, resampling, hyperslabbing, reshaping, and chunking. They are defined abstractly over the user-level array and implemented concurrently over the distributed system-level subarrays.

SAVIME (Lustosa and Porto, 2019) defines the typed array data model (TAR), which consists of two mapping functions — position mapping and data mapping — that translate between data values and memory addresses. These mappings provide support for sparse arrays, non-integer dimensions, heterogeneous memory layouts, and functional partial dependencies with respect to dimensions — all of which are characteristics of simulation data.

The array-based genomic data model (Horlova *et al.*, 2020) defines a representation for genomic data consisting of three dimensions — coordinate, sample, and signal — that provide fast associative index access to the corresponding values. The operations defined in the genomic data model include standard relational operators such as selection, projection, grouping, join, union, and difference, and genomic-specific operators such as histogram, cover, and map. These operators are classified as region preserving — no new regions are created — and space-localized or space-rearranged. In a space-localized operation, every region can be processed independently from all the other regions, while space-rearranged operations require merging several input regions.

The statistical data transformation data model (SDTDM) (Song *et al.*, 2021) is a hierarchical model for the metadata associated with statistical operations. In SDTDM, metadata on statistical operations are organized into hierarchical meta tables consisting of meta rows and meta columns, which encode positional information. This allows for direct index access to data. SDTA defines an algebra over meta tables. It includes operations derived from relational algebra, such as selection, projection, aggregation, and join. Additionally, SDTA defines order-preserving operations over meta rows and meta columns. These are aimed at maintaining the ordering information through statistical transformations. The SDTL language provides syntactic sugar shortcuts for composite operations derived from nested SDTA operations.

4.8 Relational Algebra

The array representations as relations given in Section 2.4 can be divided into relational mapping — *array as table* — and object-oriented mapping — all the other representations. In the relational mapping, every array cell is represented explicitly as a tuple containing both the indices as well as the values. Array operations are directly mapped into expressions of relational algebra operators and SQL statements. However, SQL is not particularly well-suited for array operations due to the lack of positional indexing. In the object-oriented mapping, the original relational model is extended with a composite array data type and corresponding operators. The array becomes an attribute of a larger relation. Array operations are included in queries by making calls to the methods defined for the array data type. The set of supported methods is extensible with user-defined functions. Queries consist of a relational component and a non-relational component with expressions involving array methods. Since method invocations are treated as black boxes, the optimization of the non-relational part is mostly limited to the correct placement of the operators in the query plan.

4.9 Tensor Algebras

Operations over multidimensional tensors — including contractions and derivatives — can be written declaratively using index notations. In such a notation, an index variable is assigned to every dimension. Variables that appear in a single tensor are called free and their corresponding dimension is part of the result. Variables that are shared among tensors correspond to dimensions that are eliminated through aggregation. The two most common index notations are the Einstein notation and the Ricci notation. The matrix multiplication $C = A \cdot B$ is written in these notations as:

$$\text{Einstein: } A_{ip} \cdot B_{pj} \qquad \text{Ricci: } A^i_p \cdot B^p_j \tag{4.1}$$

Although the two notations are syntactically quite similar — the Einstein notation uses only lower indexes, while the Ricci notation uses both lower and upper indexes — they have semantic differences. The most important difference is that the Ricci notation differentiates between co- and contra-variant dimensions/indices and the Einstein notation does not. This allows for element-wise operations — among others — to be expressible only with the Ricci notation. Nonetheless, the Einstein notation is used more extensively — if not exclusively — in programming languages due to its uniform index handling.

4.9.1 Relational Algebra Extensions and Generalizations

Since tensors can be represented as relations, operations over tensors — including the Einstein notation — can be expressed in terms of relational algebra operators. This can be achieved either through `JOIN GROUP BY AGGREGATE` statements or through language extensions such as user-defined functions (UDF) and aggregates (UDA). The former requires a pure relational *array as table* representation for tensors, while the later works on any of the other array formats presented in Section 2.4. The initial solutions implemented in MADLib (Hellerstein *et al.*, 2012) and GLADE (Qin and Rusu, 2015) adopt the UDA approach because of its flexibility and better performance. AC/DC (Khamis *et al.*, 2018) on the other hand uses the *array as table* representation and enhances

its performance through factorization, functional dependencies, and worst-case optimal join algorithms. Given their reliance on SQL, these approaches are not aimed at defining a formal tensor algebra.

In the following, we present in detail several tensor algebras that extend or generalize the relational algebra. Each of them selects a relational array representation as a primitive data structure and defines a set of operations — relational and matrix — over this primitive. In many cases, the relational operators are second-order functions parametrized by matrix operations.

Tensor-Relational Model (TRM). TRM (Kim and Candan, 2011; Kim and Candan, 2014) introduce a representation of unordered relations as two types of tensors — occurrence tensor and value tensor. In the occurrence tensor, all the relation attributes become tensor dimensions and the cell value — only 0 or 1 — indicates whether a tuple with the corresponding attributes exists or not. The value tensor is the functional formalization of relations as multidimensional arrays from keys (dimensions) to functionally dependent attributes (Section 2.1). The standard relational operations — including selection, projection, Cartesian product, join, and set-based operations — can be performed on the tensor representation with minimal changes. Tensor decomposition is the only tensor-specific operation defined in TRM. Its interaction with the relational algebra operators is carefully analyzed and rule-based transformations to optimize complex expressions are designed. TRM is implemented with the ArrayQL algebra operators (Maier, 2012), which process tensors as tiled chunks.

LARA. LARA (Hutchison *et al.*, 2017) is a reduced algebra consisting only of three operators that generalizes the operations and rules of both linear algebra and relational algebra. LARA is a formalization of the concepts first introduced in Kunft *et al.* (2016). The only abstract data structure in LARA is the *associative table*, which corresponds to the functional formalization of relations as multidimensional arrays (Section 2.1). The associative table represents arrays by explicitly storing the mapping from dimensions to attributes using the *array as table*

representation (Section 2.4). The physical implementation is using partitioned sorted maps, which require massive space in the case of dense arrays since all the indices have to be materialized. This implementation is refined to a specialized key-value data structure for linear algebra operations in LevelHeaded (Aberger *et al.*, 2018). The LARA operators are second-order functions parameterized by UDFs defined over a restricted semiring structure — UDFs that are associative, commutative, and have an identity/zero element. The three operators are Union (vertical concatenation), Join (horizontal concatenation), and Ext (flatmap). They are applied to the keys/dimensions, while the UDF is applied to the corresponding values. Matrix multiplication consists of Join on the shared dimension with a multiply UDF followed by Union with a sum UDF. For physical optimizations, the Sort operator — which sorts an associative table based on its keys — is added to the algebra.

MATLANG. MATLANG (Brijder *et al.*, 2019) is a query language for matrices built up from basic linear algebra operations that are closed under composition. The only variables in MATLANG are matrices, i.e., 2-D tensors, specified by a type definition for their dimensions. The type associated with a matrix is either given at declaration or induced from the expression operands. The type distinguishes among matrices, row/column vectors, and scalars. The MATLANG operators are inherited from linear algebra. They include matrix multiplication and transposition, the constant $\mathbf{1}$ vector and column vector diagonalization, and the pointwise function application. These operators are composed into expressions — or queries — that are evaluated over the set of matrices in the input schema. The expressive power of MATLANG is similar to that of relational algebra with aggregates (Libkin *et al.*, 1996). This follows from the *array as table* representation given in Section 2.4. In order to express more complicated matrix operations such as inverse and eigenvector decomposition, MATLANG is extended with the `inv` and `eigen` operations.

Relational Matrix Algebra (RMA). RMA (Dolmatova *et al.*, 2020) extends the relational model with matrix operations performed exclusively on relations. RMA applies the *array as set of row vectors* representation

(Section 2.4) to a relation in order to construct a corresponding matrix. This requires the specification of the row dimension — which has to be a key attribute of the relation — and of the column attributes. Since the induced rows are indexed based on the dimension index, the columns have to be sorted accordingly. The order of the columns is given by the order of the attributes in the relation schema. Thus, there has to be an attribute for every column of the matrix — resulting in explicit storage of all the matrix indices. Essentially, matrices are constructed by sorting and generalized projection — both of which are relational algebra operators. Moreover, they have an exact relational representation. RMA supports all the matrix operations from the statistical programming language `R`, implemented as function calls to an external library from the MonetDB database. The result matrix is reversely mapped to a corresponding relation based on the dimension and column attributes specified in the matrix constructor.

Tensor Relational Algebra (TRA). TRA (Yuan *et al.*, 2021) is an extension of classical relational algebra (Codd, 1970) that models tensors concisely as binary relations from a vector encoding dimensions to the corresponding values represented as a tile. In other words, TRA takes as its primitive abstraction the *array as set of tiles* representation (Section 2.4) and defines closed operations over it. The TRA operations consist of second-order functions that have as arguments groups of tiles identified based on their indices. For example, matrix multiplication is expressed in TRA as a Join followed by an Aggregation. The Join operation pairs the tiles based on the common dimension and computes a matrix multiply kernel function. The resulting 3-D tiles are input to an Aggregation operator with a sum kernel function that sums up the values along the common dimension to generate the result matrix. The important aspect here is that join and aggregation are relational algebra operations extended with linear algebra kernels that work on tiles. This allows for a direct integration of optimized numerical kernels such as ScaLAPACK (Choi *et al.*, 1992) into relational databases. The other operations in TRA include ReKey, Filter, and Transform — which work only on dimensions — and Tile and Concat — which alter the structure of the tiles by regrouping the indices.

4.9.2 Tensor Algebras for Machine Learning

The Einstein notation is adopted to express the linear algebra operations standard in machine learning computations more abstractly. Optimized low-level kernels that implement these abstractions for specific target architectures — such as GPU — are subsequently automatically generated. Tensor Comprehensions (Vasilache *et al.*, 2018) are a direct instantiation of the Einstein notation as a domain-specific language in which index variables are defined implicitly by using them in an expression. Their range is inferred automatically from the tensors they are assigned to. The dimensions of the result tensor correspond to the index variables on the left side of an expression. All the index variables that appear only on the right side are assumed to be reduction dimensions. An extension of tensor comprehensions that specifies the indices of the result tensor explicitly is used to define a tensor calculus for automatic higher order differentiation in Laue *et al.* (2020). These declarative tensor algebras are integrated in ML libraries such as NumPy, TensorFlow, and PyTorch.

The applicability of LARA (Hutchison *et al.*, 2017) and MATLANG (Brijder *et al.*, 2019) to express common ML problems is studied by Barceló *et al.* (2019). They prove that the Einstein notation/summation is supported by both LARA and MATLANG, convolution is expressible only in LARA, and matrix inversion cannot be expressed in LARA when index comparisons are not allowed.

4.10 Data Cube Algebras

Following the concepts and operations introduced in the relational model (Codd, 1970), closed data cube models are proposed in Agrawal *et al.* (1997) and Vassiliadis (1998) among others. These models provide a formal representation for data cubes and define composable operations that generate a data cube as their result. The model introduced in Agrawal *et al.* (1997) treats dimensions and measures symmetrically, and supports multiple hierarchies along every dimension. The model in Vassiliadis (1998) is based on the notion of basic cubes as the storage element for the original data in a cube, and focuses on the support of sequences of navigational operations. Both models include standard

data cube operations such as slicing and dicing, roll-up and drill-down, and navigation along dimension hierarchies. Moreover, both models provide mappings to the relational model and to a restricted form of multidimensional arrays. This restriction is imposed by the lack of a complete order on dimensions, which precludes the application of range operations. Nonetheless, selection — or restriction — on dimensions is possible with set membership conditions. An alternative mapping from data cubes to multidimensional arrays — also lacking complete ordering — is through intermediate relations (Vassiliadis, 1998).

4.11 Summary

- Arrays are modeled as functions from dimensions to attribute values. This allows for an equivalent relational tuple representation where the dimensions form a key. An alternative representation — with wider support in SQL — is to declare arrays as attributes having a container data type. This way, the entire array is kept together in a single tuple instead of having every cell separately.
- Array operators are defined as second-order functions with dimensional operands that apply the functional argument to the attributes. In SQL, the array operators are implemented as user-defined functions (UDF). Without sufficient knowledge of the UDF processing, the integration of array operators in query optimization is difficult.
- The various algebras proposed in the literature share a set of similar single-array operators. Moreover, they are semantically equivalent, RasQL and AML being the most general.
- Array join formalizations distinguish between dimensions and attributes. While attribute joins are equivalent to relational joins, dimensional joins are defined based on discrete shape predicates that encode the neighborhood relationship among cells.
- Tensor algebras extend relational algebra by defining matrix operations over a primitive relational array representation. Matrix operations are embedded as functional parameters into second-order relational operators. This allows seamless integration of optimized linear algebra kernels into a relational processing engine.

5

Multidimensional Array Storage

In this section, we consider array storage in the following context. The size of the array — which can be bounded by $|D_1| * |D_2| * \cdots * |D_N| * |sizeof\ (A_1, A_2, \ldots, A_M)|$, where $|D_i|$ is the maximum range on dimension i — is too large to fit entirely in memory. However, it is possible to access the array elements based on their index. This is a major departure from table — or relation — storage where tuples cannot be identified based on their position. Nonetheless, the segment where a tuple is stored can be determined when data partitioning methods (DeWitt and Gray, 1991) are applied. Thus, there is a connection between data partitioning for parallel processing and array storage.

The array is organized on secondary storage into *chunks* — or partitions — that contain a group of array cells (in Figure 5.1). Whenever an element from a chunk has to be read into memory, the entire chunk is read — the I/O unit is the chunk, similar to the page for file systems and the block/partition for relational databases, respectively. Moreover, chunks can be allocated to different hosts or tasks for storage and concurrent processing. The optimal chunking is dependent on the operations performed on the array. While some operators, such as matrix multiplication, have static access patterns, the stencil operator applies a

shape-based aggregate to an array region, resulting in a highly-variable access to the array cells. Based on these considerations, array chunking has to address the following questions:

- What is the optimal size of a chunk?
- What is the shape of the chunk?
- What is the mapping function from an array index to the corresponding chunk? The mapping of a chunk to disk storage? To processes and processing nodes?
- How are the chunks allocated to processes and processing nodes?
- How are the cells organized inside the chunk?

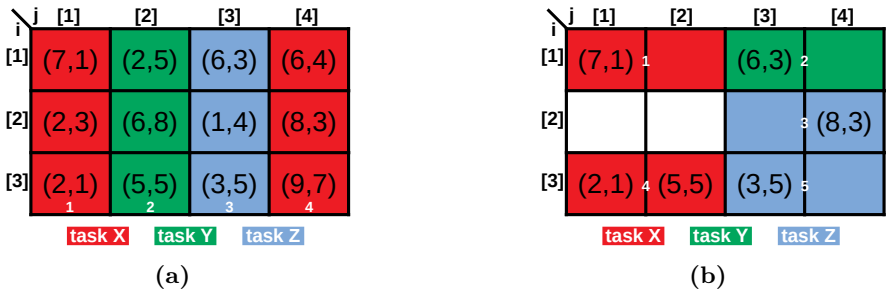

Figure 5.1: Chunking for dense (a) and sparse (b) arrays.

5.1 Optimal Chunk Size

Let us consider B to be the chunk size. This is a global system parameter similar to the block size in relational databases. Determining the optimal B value is the first question array chunking has to address. In early work (Sarawagi and Stonebraker, 1994), it was common to set B to the size of the file system page or the database block size, e.g., 4 to 64 KB. This strategy keeps the chunks tight without wasting space due to fragmentation. It is also optimal when small portions of the array are retrieved by the majority of the accesses, e.g., direct cell access based on the indexes or selective range queries. In more recent work (Soroush *et*

al., 2011; Cheng and Rusu, 2014; Papadopoulos *et al.*, 2016), B is set to much larger values, in the order of tens to hundreds of megabytes — for example, the chunk size in SciDB and EXTASCID is set to 64-256 MB. There are multiple reasons for this significant increase. First, scanning larger contiguous segments from disk does not take considerably longer due to the logic implemented in the disk controller. Second, memory capacity has increased considerably, thus, allowing for more data to be cached in memory. If a chunk is placed in contiguous memory blocks — which is the case for larger chunks — access is further improved. Finally, distributed processing across multiple servers has become the dominant array data management architecture. Network transfer throughput and latency are optimized when the data volume in a single transfer is higher and the number of transfers is reduced. In distributed array storage systems — as well as file systems — this is achieved through large chunk sizes and batching.

5.2 Chunking Strategies

A chunking strategy specifies the size and shape of a chunk, as well as what array cells are grouped together in the chunk. Different chunking strategies answer these questions in different ways. We consider the most important strategies in the following. We start with general strategies that can be applied equally to dense and sparse arrays. Then, we consider more specific solutions.

5.2.1 Arbitrary Chunking

Arbitrary chunking is the most straightforward chunking strategy. It does not require any mathematical formulation or any other kind of information. The main idea behind this strategy is to group together in the same chunk cells that are close to each other. Closeness is measured based on dimensions. A common simplification is to enforce that the shape of the resulting chunks is a multidimensional hypercube aligned with the dimension axes. Then, two questions have to be answered:

- Where to position the hyperplanes corresponding to every axis?

- Are the hyperplanes bounded or unbounded, i.e., do they cover the entire axis or only a segment?

Based on the answers to these questions, three types of arbitrary chunking — depicted in Figure 5.2 — are introduced.

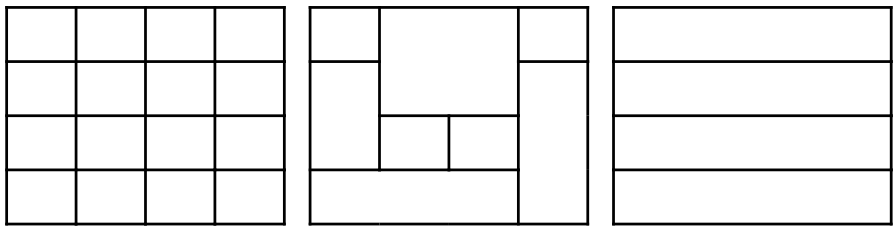

Figure 5.2: Arbitrary chunking: (a) regular, (b) directional, and (c) sliced.

Regular chunking. Regular chunking (Soroush *et al.*, 2011) or aligned tiling (Furtado and Baumann, 1999) divides every dimension into equal segments. The segments cover the entire axis. The result is a set of identical hypercubes aligned with the axes. A chunk corresponds to each such hypercube. In regular chunking, the number of hyperplanes on every axis is chosen arbitrarily. In aligned tiling, it is chosen such that the resulting chunks represent a uniform scaling down of the entire domain that fits in the allocated chunk size B, i.e., the ratio between the chunk size and the domain size is identical on all dimensions.

Example 5.1 (Regular chunking)**.** Consider a 3-D grid of integers. The domain sizes along the 3 dimensions are $(7,500, 7,500, 20)$. Aligned tiling this dense array requires chunk sizes that have the same ratio across all the dimensions. Thus, if we consider the constant ratio to be 10, then the chunk shape is $(750, 750, 2)$ and we get 1,000 chunks. Regular chunking does not require the same ratio. For example, chunks with the shape $(750, 375, 4)$ have different ratios on every dimension.

Directional tiling. In directional tiling (Furtado and Baumann, 1999), every dimension is treated independently. The position of the hyperplanes is given for every dimension. Chunks are obtained at the intersection of the hyperplanes. They do not necessarily have the same shape. While this results in chunks having different number of cells in the case of dense arrays, for sparse arrays the chunks can be determined such that they contain the same number of cells (Papadopoulos *et al.*, 2016). Nonetheless, chunks are aligned and *irregular*. Careful consideration is required for the cases when the volume of a chunk is smaller than the maximum allowed volume B — merging is possible — and when the volume is greater — further splitting is required. When any of these operations are applied, chunks become *nonaligned*, i.e., the hyperplane is only a segment that does not cover the entire axis domain. This case is depicted in Figure 5.2(b), which is obtained by merging several groups of adjacent chunks from the regular chunking in Figure 5.2(a).

Sliced chunking. A special case of arbitrary chunking corresponds to slicing a particular dimension with hyperplanes at every position in its domain. The resulting hypercubes have dimensionality N-1 and they can be chunked further, independently of each other. Any processing that can be confined to a slice becomes simpler due to the reduced dimensionality. Processing across multiple slices has to be decomposed into separate processing on every slice — a loop over the slices. The default representation of multidimensional arrays in general-purpose programming languages, e.g., `C/C++` or `Java`, is based on slicing. Starting with the most significant dimension, arrays of lower dimensionality are obtained by fixing the value of the outer indexes. Due to the linear representation in memory, these arrays are straightforward to generate. Problems appear when a lower-dimensional array has to be obtained by fixing the value of an index that does not match the linearization order. In this case, the lower-dimensional array has to be explicitly created by individually accessing every element. Consider, for example, a `C` language 2-D matrix `matrix[10][10]` linearized in standard row-major. Accessing the 3^{rd} row is straightforward, i.e., `matrix[2]`, but accessing the 3^{rd} column requires explicitly accessing every element.

Example 5.2 (Sliced chunking)**.** If we consider the same setting as in Example 5.1, we generate sliced chunks by treating each of the 20 points on the 3^{rd} dimension separately. We first generate 20 2-D arrays with size $(7,500, 7,500)$. Then, we chunk each of them individually.

5.2.2 Workload-based Chunking

The storage organization of an array is strongly dependent on the access patterns used to access its cells — which are application and workload specific. Thus, there is no organization that provides optimal performance for all possible queries. In the worst case, the entire array has to be read from secondary storage in order to compute the result. This is equivalent to a complete table scan in the relational model. In the best case, only the chunks containing data relevant to the query at hand are read from storage. Given these two extremes, the organization has to minimize the number of chunks read from storage for the majority of queries in the workload. A second parameter that requires careful consideration when deciding upon the chunking strategy is the size of the query result. It is likely that — in the case of queries returning a large number of cells — the difference between strategies is not that significant. However, when only a handful of cells are returned, the chunking strategy plays an important role. This problem is closely related to the effectiveness of indexes in relational databases.

Subsample access pattern. In Furtado and Baumann (1999), the authors identify a set of frequent access patterns:

- *Subsample multidimensional area with the same dimensionality.* The result of such a query is a hypercube having the same dimensionality as the original array. Splitting the array into chunks across all the dimensions is the optimal strategy in this situation. Notice that accessing the full array is a particular case of this access pattern.
- *Section of lower dimensionality across a subset of dimensions.* In this case, the query result is typically a hypercube with a lower dimensionality. The storage organization matching the section pattern provides optimal access in this case.

The authors propose a chunking strategy that takes the workload queries into account. Based on the query log, the access frequency is measured for every hypercube of the array. Hypercubes accessed frequently enough are designated *areas of interest*. Directional tiling is directly applied by taking the sides of the areas of interest as the splitting hyperplanes across every dimension. Further partitioning or merging may be required, as in directional tiling. Merging is different due to the requirement that only tiles from the same area(s) of interest can be put together. The objective is to put together as much data as possible from a single area of interest and to minimize the number of tiles for a given area.

Overlap access pattern. Soroush *et al.* (2011) provide more varied access patterns in addition to range selection patterns across all — or a subset — of dimensions. These diversified access patterns are:

- *Structural join between two arrays.* This operation requires combining data from the same index position in two arrays having the same dimensionality. The straightforward and optimal solution is to partition the arrays identically and — in the case of parallel processing — store corresponding partitions on the same processing node.
- *Overlap operations that access adjacent cells.* If the accessed adjacent cells are confined to a bounded region, the data can be replicated in every array chunk. This allows for independent parallel processing of every chunk without communication across nodes. If the number of adjacent cells is not bounded, merging and communication across chunks are required — this is the more general solution.

The chunking strategies proposed by Soroush *et al.* (2011) to handle these two access patterns are variations on regular and directional tiling. The main idea is to apply *two-level regular or directional tiling*. At the upper level, the chunk is determined as in any of these strategies. Inside a chunk — the lower level — another chunking is executed using again one of the regular or directional algorithms. The mini-chunks resulted

at the lower level can be accessed as a unit of processing — the I/O unit is still the upper level chunk. This strategy is beneficial exactly when data from adjacent chunks are needed. Instead of moving the entire chunk, only the overlapping mini-chunks have to be transferred.

The second chunking strategy proposed for overlapped processing requires replicating cells across multiple chunks. This *materialized view* can be stored either with the main chunk or separated. Additional space is used in both cases. The advantage of storing the materialized view separated is that it can be read on demand, only when needed. Otherwise, it has to be read whenever the chunk is read and this has the potential to incur significant overhead. As a variation on the same idea, multiple concentric materialized views with increasing radii can be created. The exterior materialized views include the interior ones. Dong *et al.* (2017) apply replication at chunk boundaries dynamically based on the specific processing task. The replicated regions — known as *ghost zones* — are only transient in this case. They are not materialized within or outside the chunk. Thus, they do not survive across tasks.

Query shape model. The approach taken in Sarawagi and Stonebraker (1994) is to model all the observed access patterns as a probability distribution function (pdf) over the shapes of the accesses — query shape model (Otoo *et al.*, 2007). Essentially, accesses are represented as N-dimensional hypercubes with a corresponding length on every dimension, i.e., $(s_1, s_2, \ldots, s_N)$. While this is the most general form of access, notice that it also encompasses degenerated patterns such as accessing a single cell or a hyperplane, i.e., $s_i = 1$. A probability is assigned to every access pattern independent of the actual occurring position in the array — the positions are assumed to be uniformly distributed across the entire domain. Then, access patterns can be grouped into classes of the form $\{[P_i, (s_{i_1}, s_{i_2}, \ldots, s_{i_N})] \,|1 \leq i \leq K\}$, where K is the number of different hypercube shapes and P_i is the probability corresponding to every class. In order to determine the optimal chunking — only regular chunking is considered — an optimization formulation that minimizes the number of blocks read from storage across all the classes is solved.

Example 5.3 (Query shape model)**.** In this formulation, the only constraint is given by the size of the storage block and the requirement that a chunk has to fit in a single block:

$$\begin{aligned} &\min_{(c_1,c_2,\ldots,c_N)} \sum_{i=1}^{K} \left(\prod_{j=1}^{N} \left\lceil \frac{s_{i_j}}{c_j} \right\rceil \right) P_i \\ &\text{such that } \prod_{i=1}^{N} c_i \leq \frac{B}{|sizeof\,(A_1, A_2, \ldots, A_M)|} \\ &\text{where } (c_1, c_2, \ldots, c_N) \text{ is the shape of the regular chunk} \end{aligned} \tag{5.1}$$

The authors make the assumption that — independent of the relative position of the chunk and the query hypercube — at most one additional chunk is read from storage for every dimension — that is where the ceiling comes from in the objective function. Clearly, this assumption is dependent on the actual position of the chunk, the shape of the chunk and what range query has to be answered — in some cases, two additional chunks have to be read. What amplifies the error effect is the multiplication of the factors across dimensions, while assuming dimensionality independence. Thus, the error becomes significantly higher if the same error is made for all the dimensions of a given class — the higher the dimensionality of the array, the higher the error. As a result, the solution of this formulation is only approximate and can incur significant errors.

Example 5.4 (Expected query shape model)**.** In this formulation, the authors modify the objective function by observing that the assumption made in Sarawagi and Stonebraker (1994) on the number of chunks to be read is problematic:

$$\min_{(c_1,c_2,\ldots,c_N)} \sum_{i=1}^{K} \left(\prod_{j=1}^{N} \left(\frac{s_{i_j} - 1}{c_j} + 1 \right) \right) \tag{5.2}$$

The factors in this formulation represent the expected value of the number of chunks to be read for every query class under the assumption that the position of the queries is uniformly distributed over the entire array domain. Notice that the objective function takes real values in

this case. These values do not represent the exact number of chunks to be read from storage for a given set of queries.

No matter which formulation we consider, it is not possible to compute a closed-form solution. Since it is not feasible to search the entire solution space, i.e., $|D_1| * |D_2| * \cdots * |D_N|$, methods to prune the space are required. The solution proposed in Sarawagi and Stonebraker (1994) reduces the search space to shapes of the form $(2^{y_1}, 2^{y_2}, \ldots, 2^{y_N})$ that are maximal, i.e., $\sum_{i=1}^{N} y_i = \left\lceil \log_2 \frac{B}{|sizeof(A_1, A_2, \ldots, A_M)|} \right\rceil$, where y_i, $1 \leq i \leq N$, are positive integers. Essentially, only hypercubes with side length of powers of 2 are considered that fill the maximum chunk size with minimal waste. The search over such shapes is exhaustive. Once the optimal shape with this restricted form is found, an additional search around the solution can be triggered to find an even better solution that allows more general shapes. In Otoo *et al.* (2007), the exhaustive search at the first level is replaced with a greedy algorithm that starts with 0 lengths for all dimensions and then chooses optimally which dimension to increase at every step. The computations required at each step are intricate and the authors do not show what benefit this brings when compared to exhaustive search over the parameter space. While the solution to the optimization problem is guaranteed to provide optimal chunking under the given assumptions, it would be interesting to see how far is this from the best solution. None of Sarawagi and Stonebraker (1994) and Otoo *et al.* (2007) provide such results or mention if other chunking strategies are better for practical query classes.

Independent attribute range model. The query workload can also be modeled through the size of the ranges it accesses on every dimension. Instead of considering a hypercube as the access unit, the query is decomposed into its corresponding segments on every dimension. Thus, two 2-D query shapes $\{< 4, 4 >, < 4, 6 >\}$ are part of two different pattern classes in the query shape model, but they are part of the same class based on the first dimension D_1. In this case, the probability distribution is defined separately for every dimension (Otoo *et al.*, 2007).

Example 5.5 (Independent attribute range model)**.** The optimization formulation for the independent attribute range model is:

$$\begin{aligned} &\min_{(c_1,c_2,\ldots,c_N)} \prod_{i=1}^{N} \left[\sum_{j=1}^{m_i} \left(\frac{s_{i_j} - 1}{c_i} + 1 \right) P_{i_j} \right] \\ &\text{such that } \prod_{i=1}^{N} c_i \leq \frac{B}{|sizeof\,(A_1, A_2, \ldots, A_M)|} \end{aligned} \tag{5.3}$$

where $(c_1, c_2, \ldots, c_N)$ is the shape of the regular chunk; $m_1, m_2, \ldots, m_N$ are the number of ranges on dimension i; s_{i_j} is the j^{th} range on dimension i; P_{i_j} is the probability of the j^{th} range on dimension i.

In this case, a closed-form formula can be computed if we give up the requirement that c_i's have to be positive integers and we impose maximality in the sense defined for the query shape model. The formula obtained using the Lagrange multiplier method gives us the c_i's:

$$c_i = \bar{A}_i \left(\frac{\frac{B}{|sizeof(A_1,A_2,\ldots,A_M)|}}{\prod_{i=1}^{N} \bar{A}_i} \right)^{\frac{1}{n}}, \text{where } \bar{A}_i = \sum_{j=1}^{m_i} s_{i_j} P_{i_j} - 1 \tag{5.4}$$

In order to obtain the integral solution, i.e., positive integer values for c_i's, the authors propose a method that rounds up some of the c_i's, while others are rounded down. By carefully choosing the two partitions, the optimal integral solution is obtained.

5.2.3 Recursive Chunking

In recursive — or hierarchical — chunking, the starting point is a single chunk corresponding to the entire array. The output is a set of chunks that form a directional tiling (Figure 5.2(b)) of the original array. At each step in the process, an existing chunk — or chunks — are chosen to be split. This can be done either by considering a query (Zhao *et al.*, 2018) or a cost function that ranks the chunks (Li *et al.*, 2020). The chosen chunk can be split into any number of additional chunks varying from 1 to 3^N, where N is the array dimensionality. In practice, a chunk is split into two chunks by first choosing a dimension and then a point along the dimension. The coordinates on all the other dimensions stay the same. The choice of the splitting dimension and the point differentiate between recursive chunking algorithms. The splitting process is applied recursively — or iteratively — until a stopping criterion is met. The most

common such criterion considers the number of non-empty cells in every resulting chunk — if the number of non-empty cells is below a threshold, the chunk is not considered for further splitting. Figure 5.3 depicts the first, fourth, and final split in the recursive chunking that produces the directional tiling in Figure 5.2(b). A careful reader immediately observes that this process is similar to how a *kd-tree* index is built.

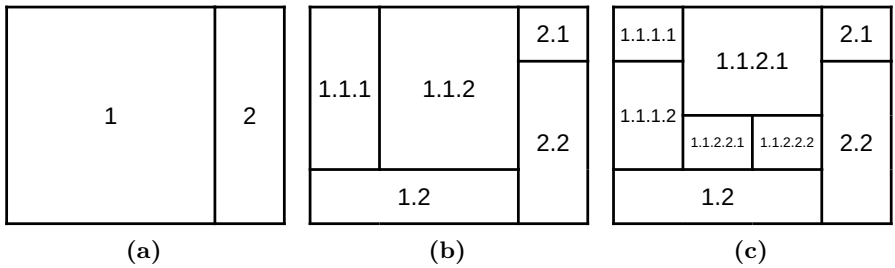

Figure 5.3: Recursive chunking: (a) first split, (b) fourth split, and (c) final tiling.

Raw array chunking. The goal of raw array chunking (Zhao *et al.*, 2018) is to infer the chunks dynamically at runtime from the query workload. Instead of creating arbitrary chunks during loading — which is time-consuming, delays the time-to-query, and may not be optimal for the actual workload — chunks are built incrementally one query-at-a-time. Given a subsample query, the relevant cells have to be identified, while minimizing the total number of inspected cells. Raw array chunking is an incremental algorithm that builds an evolving *R-tree* based on the queries executed by the system. The invariant of the algorithm is that the set of chunks cover all the valid cells of the array at any time instant. Moreover, the chunks are non-overlapping. However, the resulting chunks do not necessarily cover the complete array — they cover only the valid cells, which are non-empty in a sparse array. The central point of this algorithm is splitting a chunk that overlaps with the query. A chunk is split in two cases. First, if there is a sufficiently large number of cells in the chunk. In the second case, even when the number of cells is small, if the query does not contain any cell, the

chunk is split further. Raw array chunking always splits a chunk into two chunks. This is done by selecting a single splitting dimension. The algorithm enumerates over the query boundaries that intersect with the chunk bounding box and chooses to split into those two chunks that have the minimum combined hyper-volume. Rather than computing the hyper-volume from the query-generated chunks, the bounding box of a chunk is derived only from the cells assigned to it. Chunks that cover a smaller hyper-volume are more compact, thus the probability to contain relevant cells is higher. As a result, it is likely that the resulting chunks do not completely cover the range of the array dimensions. This is not a problem because the uncovered ranges do not contain valid cells. Therefore, they are not part of any computation. Nonetheless, the management of the chunk bounding boxes — the metadata — becomes more intricate since this is where the operation scheduling is performed. For optimal performance, a balance has to be achieved between the number of chunks and the array domain coverage.

RecPart. Given a distributed band-join query between two arrays, the RecPart algorithm (Li *et al.*, 2020) computes a recursive chunking that optimizes a composite objective function consisting of two factors. The first factor is the number of cells that have to be replicated across chunks, while the second is the overall size of chunks processed at a computing node — load balancing. Similar to *decision trees* training in machine learning, RecPart's chunking is performed from the root, each time recursively splitting some chunk. As RecPart splits chunks, cell replication is monotonically increasing, while large chunks are broken up into smaller ones. Given the targeted objective, the rank function — used to choose the best chunk to split and the splitting point — is the ratio between load balance improvement and additional cell replication. This gives priority to chunks that do not introduce replication through splitting. When a chunk becomes small enough that all cells join with each other, then the chunk is not split any further. However, if the load induced by that chunk is high, then a grid-style partitioning is applied internally for scheduling. The rank function is evaluated over random samples, which can introduce serious errors for sparse and skewed arrays. Moreover, sampling has to be also performed on the output of the band-join.

5.2.4 Adaptive Rechunking

Chunking imposes a strict processing strategy for certain array operations that access neighboring cells, such as subsample and similarity join. In such cases — depending on the existing chunking — adjacent chunks have to be accessed to execute the operation. However, with a different chunking, it may be the case that the operation can be performed for every chunk independently, or by accessing a smaller number of adjacent chunks. Thus, a choice is required — perform the operation based on the existing chunking or rechunk the arrays first — the `reshape` and `repart` operations in SciDB (Cudre-Mauroux *et al.*, 2009; Stonebraker *et al.*, 2011; Paradigm4, 2022a) — and then perform the operation? ChronosDB (Zalipynis, 2018) applies an heuristic to determine when to rechunk an array. Whenever the size of a chunk becomes two times larger or smaller than a user-defined threshold, the entire array is rechunked. A third alternative — possible for operations such as array similarity join (Zhao *et al.*, 2016) — is to overlap or fuse rechunking and the operation at the expense of considerable increase in complexity. While the optimal choice depends on the relative cost of the available alternatives, in practice the decision is largely binary. With the exception of similarity joins — band-join (Li *et al.*, 2020) included — which always perform rechunking, the other operations preserve the existing chunking.

5.2.5 Update-optimized Chunking

The *array fragment* concept as a timestamped snapshot of a batch of updates is introduced in Papadopoulos *et al.* (2016). A fragment is a collection of array cells modified via write operations. Rather than performing the updates in-place — as in the case of HDF5 (The HDF5 Group, 2020) — the modified cells are grouped into fragments of fixed size. The order of cells inside a fragment follows the order of dimensions in the chunk representation. The shape of a fragment is given by the *minimum bounding rectangle (MBR)* that encompasses it. Since fragments are built based on the order of the modification operations, overlapped fragments can result. However, a cell always belongs to a single fragment. Update operations are optimized because

the modifications from a fragment are applied all at once. Moreover, the *consolidation* between chunks and fragments is performed only when the array read access becomes suboptimal due to a large number of fragments — array reads have to consider both chunks and fragments.

5.3 Mapping Cells to Chunks

Independent of the actual chunking strategy, neighboring cells in the case of dense grids or proximal points in sparse arrays — measured based on dimensions — are grouped together in chunks. Moreover, the typical shape of a chunk is a hypercube aligned with the dimensions. Whenever a cell or a range of cells have to be retrieved based on their index, the chunk(s) containing the cell(s) have to be found first. This requires a mapping function from the cell index to the corresponding chunk:

$$\begin{aligned} &Map : D_1 \times D_2 \times \cdots \times D_N \longmapsto [1 \ldots Z_{chunks}], \\ &\text{where } Z_{chunks} \text{ is the total number of chunks} \end{aligned} \tag{5.5}$$

A chunk is identified by its position across all the chunks — chunks are linearized similarly to how they are materialized on storage. The position is given as an index on a discrete axis. The mapping function can be implicit — given by a closed-form formula — or explicit — stating the corresponding position for every chunk. In the case of distributed storage, a second function maps chunks from the linear index to the assigned server. The chunks allocated to the same server can either follow the global order or go through another local mapping.

5.3.1 Implicit Mapping

An implicit mapping requires a pre-determined order in which chunks are considered in the dimension space, e.g., row-major or column-major for 2-D arrays. The order impacts how chunks are accessed from storage — precisely, the order determines how long are the sequential scans. An implicit mapping function requires regular chunks having the same size and can be applied only to dense grids. When applied to sparse arrays,

all the empty cells have to be represented explicitly, e.g., NULLs, thus requiring additional storage space.

Example 5.6 (Implicit mapping)**.** Consider the 3-D grid from Example 5.1, i.e., $(7,500, 7,500, 20)$, chunked into 1,000 regular tiles with shape $(750, 750, 2)$. The mapping function used to linearize the chunks considers the dimensions from the first to the third, with the index on the third increasing the fastest, i.e., row-major extended to three dimensions. Thus, the order in which chunks are stored follows the indexes as: $[(1,1,1),(1,1,2),\ldots,(1,1,10),(1,2,1),\ldots,(1,10,10),(2,1,1),\ldots]$. In this case, the formula for the implicit mapping function is:

$$Map_{implicit}(x,y,z) = \left\lfloor \frac{x}{750} \right\rfloor \cdot \frac{7,500}{750} \cdot \frac{20}{2} + \left\lfloor \frac{y}{750} \right\rfloor \cdot \frac{20}{2} + \left\lceil \frac{z}{2} \right\rceil \qquad (5.6)$$

Given an array index, it is straightforward to find the chunk that contains the corresponding cell. For example, the cell corresponding to index $(1111, 308, 7)$ is in chunk $\left\lfloor \frac{1111}{750} \right\rfloor \cdot 100 + \left\lfloor \frac{308}{750} \right\rfloor \cdot 10 + \left\lceil \frac{7}{2} \right\rceil = 100+0+4 = 104$. Identifying the chunks corresponding to a range/subsample query is more intricate. Consider the range $([3,000, 4,000], [1,000, 7,000], [5, 11])$. We have to treat every dimension separately in order to identify the range of chunks covered by the query interval. For dimension x, we have $[4, 5]$; for y, $[1, 9]$; and $[3, 6]$ on z, respectively. To find the chunks that overlap this range, we have to compute all the possible combinations of indexes, i.e., $2 \cdot 9 \cdot 4 = 72$. Thus, there are 72 chunks that have to be accessed in order to answer this range query. Some of them are: $[413, 414, 415, 416, \ldots, 445, 446, \ldots, 593, 594, 595, 596]$. Since $1,001 \cdot 6,001 \cdot 7 = 42,049,007$ cells are covered by the query — which fit into 38 compact chunks — the effectiveness of the chunking scheme is poor for this range query — almost a double number of chunks is accessed. While the effectiveness can be improved by reducing the chunk size, this may impact the access speed negatively.

Intermediate mapping with regular chunking. An alternative approach is to first partition the array into regular chunks using any of the methods presented previously. The domain on every dimension is

reduced from the original size to the number of chunks along the dimension. Based on the original dimension ordering, a chunk corresponds to every point in the new domain. A chunk is identified by its position along every dimension. Given the new coordinate system, the mapping function is defined as a dimensionality reduction transformation from the multidimensional coordinates of the chunk to an integral position along a linear axis. This method is introducing an intermediate mapping from the original domain to the chunk domain, in the same multidimensional space. Only then we are mapping the chunks to the linear axis. Formally, this corresponds to two functions:

$$\begin{aligned} &Map_{chunk} : D_1 \times D_2 \times \cdots \times D_N \longmapsto C_1 \times C_2 \times \cdots \times C_N \\ &Map_{linear} : C_1 \times C_2 \times \cdots \times C_N \longmapsto [1 \ldots Z_{chunks}] , \\ &\text{where } C_i, \text{ is the number of chunks along a dimension,} \\ &\qquad Z_{chunks} = |C_1| * |C_2| * \cdots * |C_N| \text{ is the number of chunks} \end{aligned} \tag{5.7}$$

While Map_{chunk} is straightforward to define, there are a variety of choices for Map_{linear}. The most common choices are row-major (column-major), snake row-major (snake column-major), and their extensions to multidimensional spaces.

Mapping functions based on space-filling curves are presented in Jagadish (1990). They are defined recursively for a given domain size and do not have a simple closed-form formula. Out of the three methods presented — Z-curve, Gray code, and Hilbert curve — it is shown that Hilbert curve mapping provides the best performance for partial exact match selection — slicing along one dimension — and range selections in 2-D space. The performance metric used for the theoretical analysis is the number of runs of consecutive grid points, which is equivalent to the number of non-consecutive disk blocks fetched. Lower values correspond to a reduced number of disk seek jumps. This translates indirectly to continuous scans, thus better disk I/O throughput. In addition to this metric, the total number of disk blocks fetched and the size of the linear span for a given selection — the difference between the maximum and minimum linear coordinate — are also used in the experimental evaluation.

Dimension ordering. An important question that requires attention in the case of implicit mapping is the order in which to consider the dimensions when linearizing the chunks on storage. Notice that the same number of chunks has to be accessed for a query no matter what the order is. What is highly sensitive to the order, though, are the length of the sequential scans and that of the seeks between chunks that are within query range. Longer sequential scans and shorter seek jumps are better. The arbitrary — and most common — solution is to use the order in which dimensions are specified in the array definition. In Sarawagi and Stonebraker (1994), the authors provide a heuristic which orders the dimensions based on the ratio of the number of chunks accessed on a dimension — across the queries in the workload — and the number of chunks on that dimension. The dimension with the largest ratio is the inner-most one. Intuitively, this corresponds to having the dimensions with the largest accessed number of chunks in the inner loops of the traversing order. Or, equivalently, execute the longer sequential scans more often — and the longer seek jumps less frequent.

5.3.2 Explicit Mapping

An explicit mapping function bypasses the conversion to a chunk position and maps N-dimensional hypercubes specified by their left-bottom and right-upper corners to the starting position of the chunk on storage:

$$Map_{explicit} : [C_{l_1}, C_{u_1}] \times \cdots \times [C_{l_N}, C_{u_N}] \longmapsto [1 \ldots Z_{disk}], \quad \text{where } Z_{disk} \text{ is the maximum storage index} \tag{5.8}$$

In this case, finding the array cell corresponding to a given index requires identifying the chunk which contains the index. Since the mapping function is represented explicitly, this reduces to inspecting every entry in the function domain and checking inclusion. Building a multidimensional index — such as an R-tree — over the hypercubes is an alternative to reduce the number of entries inspected — at the expense of maintaining the index.

Example 5.7 (Explicit mapping)**.** Let us consider a modification of the 3-D array used in the previous examples. Instead of having a dense

grid, consider the $(7,500, 7,500)$ squares positioned in a plane of size $(10^6, 10^6)$. The resulting 20 2-D arrays — when considered together over the $(10^6, 10^6, 20)$ space — form a sparse array. One strategy to chunk the sparse array is to slice each square out and chunk it using a dense strategy. Thus, if we use $(750, 750)$ rectangles as before, we obtain 2,000 chunks with 6 coordinates corresponding to every chunk. They have the form $[(x, y, z_1), (x + 750, y + 750, z_1)]$ or, equivalently, $[x, x + 750] \times [y, y + 750] \times [z_1, z_1]$. An explicit mapping function stores the corresponding position on storage for every such hypercube — without first mapping to the chunk position. While this example is for a sparse array, notice that irregular chunking always requires the mapping function to be represented explicitly.

5.4 Chunk Organization

Once array cell membership to chunk is determined, the next step is to organize the cells inside the chunk. It is important to remember that the I/O unit is the chunk. Even if only one cell is needed for a given task, the entire chunk has to be read from disk into memory. While I/O is supposed to be the most time-consuming operation, memory access and CPU processing are also relevant. Thus, it is important to also consider optimization strategies for these operations.

Array cell organization inside a chunk can be viewed as another chunking problem — at lower scale. Thus, recursive chunking can be applied. Everything discussed earlier applies directly to the more confined space. The depth of the recursion can be decided during chunking. In Soroush *et al.* (2011), the authors set for a two-level recursion. The benefit of such a strategy is again the reduction on the number of cells that are inspected in range queries. Notice, though, that the I/O unit at the upper-most level remains the chunk.

5.4.1 Dense Chunks

The standard order in which cells are stored for dense grids is identical to the order in which chunks are linearized on storage — same order for dimensions. While other dimension orderings are possible, it is not

clear what effect they have on query response time. However, the most important criterion for dense grids is the storage reduction that can be obtained by discarding the indexes corresponding to array cells inside a chunk. This technique is known as *dimension suppression* (Stonebraker *et al.*, 2011; Cheng and Rusu, 2014). It reduces significantly the size of the chunk — for highly-dimensional grids — thus, the amount of data that has to be read from storage. The only requirement for dimension suppression to be applicable is the existence of an implicit mapping function from an index to the corresponding cell inside the chunk — exactly the same idea as for chunk linearization.

5.4.2 Sparse Chunks

In the case of sparse arrays, it is not that clear how to store cells inside a chunk. The simplest idea is to completely ignore the ordering and to process any query by scanning all the cells. This is perfectly reasonable since checking if a cell has to be included in the processing of a given query takes only a conditional `if` instruction. Given the purely relational format of sparse array data, any indexing technique based on dimensions or attributes — including bitmap indexing — is equally applicable. In particular, bitmap indexing along dimensions (Wu *et al.*, 2006) represents a secondary method to discard overlapping chunks for range queries. The only effect of any indexing technique is reducing the number of cells that have to be inspected — at the cost of building the index. As mentioned before, the benefits are unclear.

In Goil and Choudhary (1997), the authors provide a complete overview on how to organize cells inside a chunk for sparse arrays. They analyze the storage requirement of every technique as a function of multiple parameters, such as dimensionality, density, and size of the array cell. They also provide detailed analytical cost formulas for the time it takes to answer point and range queries for each of the analyzed schemes. The storage schemes presented in Goil and Choudhary (1997) are:

- *Index-value pairs.* This is the straightforward relational representation of sparse data. Index-value pairs are known as the coordinate (COO) representation for sparse matrices in BLAS libraries for

linear algebra. The order of the pairs inside the chunk can be arbitrary or it can follow the dimensions.

- *Offset-value pairs.* The same principle behind linearizing chunks on storage is applied to linearizing array cells inside the chunk. While absolute coordinates have to be stored for chunks, only the relative position in the chosen order is stored for array cells.
- *Compressed sparse dimensions.* In this representation, one of the dimensions is chosen as the principal dimension. In the case of sparse 2-D tensors, we have compressed sparse row (CSR) and compressed sparse column (CSC) format. The position of every non-empty array cell is stored on this principal dimension in a 1-D vector. The cells are also stored in a corresponding 1-D vector. For the remaining dimensions, the transition from one index value to the next is recorded as positions in the 1-D vectors with indexes and array cells, respectively. While the size of the first two vectors depends only on the number of non-empty cells, the size of the last vector is equal to $\sum_{j=1}^{N-1} \left(\prod_{i=j}^{N-1} |D_i| \right)$, where D_N is the principal dimension. Determining the ordering of the dimensions in order to minimize the storage is quite straightforward. That is not the case for determining the principal dimension.
- *Sparse-dense split storage.* Dimensions are split into dense and sparse. When the dimensionality of the original array is reduced to the number of dense dimensions, the resulting arrays — one for every combination of the sparse dimensions — are either dense or empty. Empty arrays do not need to be stored at all. However, what have to be stored are the indexes of the sparse dimensions for which there exist dense arrays.
- *Bit-encoded sparse storage.* Rather than storing the index of each dimension as a basic numeric type, e.g., `int` or `long`, the minimum number of bits sufficient to represent the cardinality of every dimension is used. This has the potential to result in storage reduction — especially when chunking is used. Point queries benefit from this representation since index matching becomes a bit manipulation operation. However, answering range queries becomes more intricate.

5.5 Mapping Chunks to Storage

Declustering (Moon and Saltz, 1998) studies how to distribute chunks across disks in a multi-disk environment — shared-disk or shared-nothing architecture. The formal declustering representation requires a function to be defined from the chunk linearization to the disk domain:

$$Map_{disk} : [1 \dots Z_{chunks}] \longmapsto [1 \dots HD]\,, \quad \text{where } Z_{chunks} \text{ is number of chunks, } HD \text{ is number of disks} \tag{5.9}$$

Map_{disk} partitions the chunks over the available disks. The objective is to find mappings that evenly distribute the chunks across all the available disks (Moon and Saltz, 1998). This results in spreading the disk I/O evenly across disks, thus maximizing the overall throughput. While this can be achieved on average, there will always be queries for which more chunks — if not all — have to be read from the same disk, resulting in degraded I/O performance.

In order to get access to a given array cell, two mappings have to be applied in sequence, i.e., mapping composition. First, the chunk that contains the array cell has to be identified using either $Map_{implicit}$ or $Map_{explicit}$. Then, Map_{disk} is applied on the result. The same procedure is followed for range queries, with individual calls to Map_{disk} for every chunk in the overlapped region.

5.5.1 Data Partitioning Declustering

We introduce possible forms for Map_{disk} that are immediate extensions from data partitioning schemes in parallel databases (DeWitt and Gray, 1991). The main difference is that there is no mapping from an array cell to a given chunk in data partitioning. Rather, the mapping is from a tuple attribute to a chunk — if present at all.

Round-robin. The declustering mapping function for a chunk with index x is defined as:

$$Map_{round\text{-}robin}(x) = (x + c) \mod HD + 1 \tag{5.10}$$

where $\mod$ is the modulo operation and c is a constant that determines the index of the disk to which the first chunk is assigned. Incrementing

the result is necessary only because numbering the disks starts from 1 instead of 0. The idea is to assign chunks sequentially to disks based on their position in the linear order. The distance between two chunks assigned to the same disk is HD. Each disk receives at least $\left\lfloor \frac{Z_{chunks}}{HD} \right\rfloor$ chunks. To make things concrete, we consider an example with 9 chunks indexed 1 to 9 and 3 disks, i.e., $HD = 3$, indexed 1 to 3. The value of c is set to 1. $Map_{round\text{-}robin}$ assigns chunks $\{2, 5, 8\}$ to disk 1, chunks $\{3, 6, 9\}$ to disk 2, and chunks $\{1, 4, 7\}$ to disk 3, respectively.

Range. In range-based mapping, the chunks are split into HD groups, every group containing $\frac{Z_{chunks}}{HD}$ chunks — we assume that Z_{chunks} is a multiple of HD. The difference from round-robin is that the groups contain consecutive chunks, assigned by the mapping:

$$Map_{range}(x) = \left\lceil \frac{x \cdot HD}{Z_{chunks}} \right\rceil \tag{5.11}$$

Following the example given in round-robin mapping, the chunks assigned to disk 1 are $\{1, 2, 3\}$. Disk 2 is assigned chunks $\{4, 5, 6\}$, while disk 3, chunks $\{7, 8, 9\}$.

Hash or pseudo-random. The standard mapping function used in hash-based partitioning is defined as:

$$Map_{hash}(x) = [(a \cdot x + b) \mod P] \mod HD + 1 \tag{5.12}$$

where a and b are random numbers, while P is a large prime number. On average, the same number of chunks are assigned to every disk. However, the chunks assigned to a disk depend strictly on the parameters of the hash mapping. In the example with 9 chunks and 3 disks, we consider $a = 3$, $b = 2$, and $P = 13$. With these parameters, chunks $\{8, 9\}$ are assigned to disk 1, chunks $\{4, 5, 6, 7\}$ to disk 2, and chunks $\{1, 2, 3\}$ to disk 3, respectively. In order to enforce that chunks are uniformly distributed across disks — not in groups, as in this example — a combination between round-robin and hash can be devised such that in every run of HD chunks, every disk gets a chunk. Inside a run, chunks are randomly assigned to disks, rather than following a fixed pattern.

5.5.2 Intermediate Mapping Declustering

Instead of assigning chunks to disks based on the linearization produced by the implicit or the explicit mapping, the assignment can be done starting from the intermediate mapping Map_{chunk}. In this case, the input to Map_{disk} is the multidimensional coordinate in the chunk space:

$$Map_{disk\text{-}chunk} : C_1 \times C_2 \times \cdots \times C_N \longmapsto [1 \ldots HD] \tag{5.13}$$

where the symbols have the same meaning as defined previously. There are various forms $Map_{disk\text{-}chunk}$ can take. In the following, we present the most common declustering methods that use the intermediate mapping as introduced in Moon and Saltz (1998).

Disk modulo (DM). In the DM scheme, chunk $[i_1, i_2, \ldots, i_N]$ is assigned to disk:

$$Map_{DM}([i_1, i_2, \ldots, i_N]) = (i_1 + i_2 + \cdots + i_N) \mod HD + 1 \tag{5.14}$$

Even though the assignment may seem simple, the DM mapping is known to be strictly optimal — exactly the minimum number of chunks is read from every disk — for many cases of partial match queries, including all partial match queries with only one unspecified attribute (Moon and Saltz, 1998). However, DM does not scale for range queries as the number of disks is increased. This limits drastically its applicability.

Fieldwise XOR (FX). The FX scheme replaces the summation operation in DM with a bitwise XOR operation on the binary representation of the chunk coordinates. Chunk $[i_1, i_2, \ldots, i_N]$ is assigned to disk:

$$Map_{FX}([i_1, i_2, \ldots, i_N]) = (i_1 \oplus i_2 \oplus \cdots \oplus i_N) \mod HD + 1 \tag{5.15}$$

where i_j is the binary representations of index j in the chunk space. FX has similar characteristics to DM — when the number of disks and the size of each dimension are a power of two, FX is optimal for partial match queries. The scalability for range queries remains problematic.

Cyclic. The cyclic allocation scheme introduced in Prabhakar *et al.* (1998) is a general declustering method for 2-D dense grids. Chunk $[i_1, i_2]$ is assigned to disk:

$$Map_{cyclic}([i_1, i_2]) = (H * i_1 + i_2) \mod HD + 1 \tag{5.16}$$

where H is chosen to be relatively prime with HD. This results in separating proximal chunks in both dimensions on different disks — neighboring chunks on the same row are assigned to consecutive disks, while neighboring chunks on the same column are assigned to disks having distance H apart. The condition that H and HD are relatively prime guarantees that chunks are assigned to all the available disks before considering the same disk again. It is straightforward to remark that DM is an instantiation of the cyclic allocation scheme when $H = 1$. Given a value for HD, it is possible to create an entire class of cyclic allocations that choose all the relatively prime values between 1 and HD for H — if HD is prime, the number of classes is the largest. However, not all of the classes provide the same performance. Identifying the best value for H requires a time-consuming exhaustive search. Even if the search space is drastically reduced, a close to optimal value for H can be found with high probability.

The scheme with the best performance that avoids the exhaustive search is based on Fibonacci numbers (Prabhakar *et al.*, 1998). Given a value for HD, H is chosen such that $H = F\left(F^{-1}\left(HD\right) - 1\right)$, where $F(x)$ is the closed-form equation for the x^{th} Fibonacci number obtained after solving the recursion — if HD is a Fibonacci number, H is the previous Fibonacci number based on this equation. If HD is not a Fibonacci number and the resulting H is not even relatively prime with HD, H is forced to be a relatively prime number with HD by finding the closest such number to the result obtained from the equation.

There are two problems with this approach. First, it is limited to 2-D arrays. It is not clear how to generalize it to higher multidimensional spaces and if the analysis holds in higher dimensions. The second problem is the performance measure used in the paper. For a given size, all the queries across the entire space are considered and their error is averaged. Then, the errors are averaged again over all possible sizes. The problem is that the number of queries is highly different across

sizes and the maximum error is also highly dependent on the query size. As a result, the impact an individual query error has on the overall error is not uniform across all the query sizes.

5.5.3 Space-filling Curves

Another alternative to assign chunks to disks is based on the linearization provided by space-filling curves, rather than by multidimensional chunk coordinates. A space filling curve visits all the points in a multidimensional space exactly once and never crosses itself. In this solution, chunks are first linearized using a space-filling curve that maps a multidimensional space into a linear sequence, while preserving spatial proximity. Then, they are assigned to disks in round-robin fashion. Unlike cyclic declustering — which enforces that neighboring chunks on both dimensions are spread apart as far as possible — space-filling curves guarantee this property only for a subset of the dimensions. Formally, chunk $[i_1, i_2, \ldots, i_N]$ is assigned to disk:

$$Map_{space}\left([i_1, i_2, \ldots, i_N]\right) = space\left(i_1, i_2, \ldots, i_N\right) \mod HD + 1 \quad (5.17)$$

where *space* is a space-filling curve — a complicated function at the border between implicit and explicit mappings. Out of the many space-filling curves proposed in the literature, the linearization based on Hilbert curves (Faloutsos and Bhagwat, 1993) is shown to provide the best performance both for partial match, as well as range queries across multidimensional spaces, when the number of disks is large.

5.5.4 Similarity-based Graph-theoretic Declustering

The main idea behind the previously presented declustering methods is to make sure that neighboring chunks get assigned to different disks. This results in spreading the I/O throughput across many disks in the case of queries that select spatially close regions, thus improved execution time. The degree to which this goal is achieved is a property of every method. The approach taken in the similarity-based methods presented in Moon *et al.* (1996) and Liu and Shekhar (1995) is to formulate declustering as a graph partitioning problem. The graph is generated by creating a vertex for every chunk and creating an edge

for every pair of chunks — complete graph. The edges are weighted by the probability that their adjacent vertices are accessed together by a query. Declustering corresponds to a multi-way partitioning of the graph. Since the goal is to minimize response time by maximizing parallelism in disk access, chunks — vertices in the graph — that are likely to be accessed together should be assigned to different disks — separate convex components in the graph. This problem is a variant of the Max-Cut problem, which is known to be NP-complete. As a result, the similarity-based graph-theoretic methods for declustering are heuristic algorithms for Max-Cut and its converse — the Min-Cut problem.

5.5.5 Block-cyclic Declustering

Instead of applying declustering to a full array, a different alternative is to partition the array into multiple sub-arrays and then apply declustering for each sub-array separately. The same or different declustering strategies can be applied for every sub-array. This approach is known as block-cyclic declustering (Soroush *et al.*, 2011). It splits an array into regular blocks of chunks and declusters every block individually, e.g., with round-robin partitioning. In certain cases, block-cyclic declustering spreads dense array regions more evenly than one-level methods. Moreover, due to the compatibility with BLAS level 2 and level 3 operations, block-cyclic declustering is the preferred representation for dense matrix operations in ScaLAPACK (Choi *et al.*, 1992).

5.6 Relational Chunking

Data partitioning (DeWitt and Gray, 1991) is the concept corresponding to array chunking on relational data. While data partitioning represents the main strategy for parallel data processing, it is also a simplified form of indexing. In a relational setting, the tuples of a relation are split into multiple segments and assigned to different execution nodes for processing. Since every process works on a smaller dataset, a speedup proportional to the number of processing nodes can be obtained in optimal conditions. Moreover, some segments can be ignored when exe-

cuting certain types of queries. For example, in the case of a range query, a segment that does not overlap with the range can be safely discarded. The minimum and maximum value in the segment are necessary for this decision. They represent the primitive index data.

There are three general data partitioning schemes. In *round-robin partitioning*, tuples are assigned to segments sequentially based on their position in the database file. This guarantees that all the segments have the same number of tuples — plus/minus one. In *range partitioning*, the tuples having values in the same range are grouped together in a segment. This can be done either by dividing the domain into equal size segments — equi-width — or by having the same number of tuples in every segment — equi-depth. The last scheme — *hash partitioning* — assigns tuples to segments based on the value of a random hash function applied to the tuple. This guarantees that tuples with the same value are located in the same segment. While the segment to which a tuple is assigned can be determined for every scheme, only range partitioning groups tuples with consecutive values together. This is exactly the goal of array chunking — assign neighboring cells to the same chunk. As such, array chunking is equivalent to multidimensional range-based data partitioning across the array dimensions (Cheng and Rusu, 2014).

5.7 Tensor Chunking

In the context of tensors, the most common chunking strategies are targeted at the matrix multiplication primitive. Given the relatively simple — but complete — access pattern, in which every row from one matrix is paired with every column from the other matrix, the space of chunking alternatives is rather limited. This space is further constrained by the dependency between the chunking of the two input matrices — which determines the chunking of the result matrix. Specifically, in order to reduce the number of accessed cells, the left matrix has to be stored in row-major format, while the right matrix in column-major format. If that is not the case, rechunking is necessary — which requires expensive network traffic in a distributed setting.

The chunking strategies for a matrix follow directly from the representation of an array as a relation (Section 2.4). The most common

strategies are row slice or row strip (Figure 5.2(c)), column slice or column strip, and regular or tile (Figure 5.2(a)). These strategies are all arbitrary chunking methods introduced in Section 5.2.1. Storing the entire matrix as a single chunk — or single tuple — is also relevant in the context of matrix multiplication because it is amenable to full replication — or broadcasting — which eliminates the requirement for rechunking and, thus, can reduce network traffic.

5.7.1 Optimal Chunking for Linear Algebra Programs

The optimal chunking for a linear algebra program consisting of matrix operations over a set of input matrices — which are pre-chunked at creation — depends upon the types of operators and the available implementations. It requires finding the chunking type and its corresponding parameters, e.g., the tile sizes, for every operator, such that the fastest execution of the overall program is achieved.

Cost-based adaptive rechunking for linear algebra matrices is implemented in the BUDS system (Gao *et al.*, 2017). In this case, only four chunking strategies are allowed — exhaustive sparse, row-sliced, column-sliced, and compressed dense. A storage format corresponds to every form of chunking. Moreover, conversion procedures are defined between any two forms of chunking. When a matrix is part of a linear algebra operation, the optimal chunking for that operation is determined. If this is not the same as the storage chunking, a conversion is executed as long as its cost is below the cost of performing the operation on the suboptimal chunking. It is important to notice that this strategy does not consider the chunk shape as a parameter since linear algebra operations are always performed on the complete matrix — there is no subsampling for a matrix.

Optimal chunking is modeled as a graph annotation optimization problem in Luo *et al.* (2021). Operators are represented as vertices, while edges correspond to the matrix operands. Vertices are annotated with the implementation of the operator and the format — or chunking — of the result matrix. Edges are annotated with the rechunking transformation — if any — applied to the associated matrix operand. An execution time cost function is assigned to every annotation in the graph. The

parameters of the cost function — including the number of floating point operations, the network traffic in bytes, and the size of the intermediate matrix operands — are estimated with a pre-trained regression model. The goal is to find those annotations that minimize the overall cost of the graph. This can be done efficiently for a tree-shaped graph — in time linear in the number of vertices — with a dynamic programming algorithm — which can be extended to a general directed acyclic graph (DAG) at the expense of an exponential increase in time complexity.

Other solutions that optimize the execution of linear algebra programs do not consider the combination of chunk layout, operator implementation, and rechunking transformation over the entire program. Cumulon (Huang *et al.*, 2013) uses a fixed size tiled matrix layout and automatically optimizes linear algebra programs in terms of operator implementations. DMac (Yu *et al.*, 2015) considers only the row/column strip and matrix as single tuple chunking configurations. Based on the chunking of the input matrices, the transformations required by every operator are determined based on the order and dependency among operators. This is done independently for every operator. The implementation of an operator is chosen based on the computed chunking using heuristics that define a holistic communication cost model. In SystemML (Boehm *et al.*, 2016), the chunk layout is fixed for the base matrices and is passed unchanged through the other linear algebra operators. MatFast (Yu *et al.*, 2017) introduces a cost model for matrix rechunking based on the required data transfer. This cost model is used to determine the optimal chunking of a particular operator. A linear algebra expression consisting of several operators is greedily optimized by tuning the layout of every operator independently. The only supported layouts are row/column strip and matrix as single tuple. DistME (Han *et al.*, 2019) considers tiled chunking as a generalization of all the other strategies for stand-alone matrix multiplication. The optimal tile size is determined through exhaustive search such that the required communication is minimized. The optimization space is limited to tiles having size a multiple of the basic tile size.

5.7.2 Loop Tiling

While array chunking is static — the range on every dimension is fixed during processing — tensors are often dynamically (re-)chunked based on the characteristics of the hardware platform. This dynamic rechunking has been studied extensively in the compiler and high performance computing (HPC) communities under the name *loop tiling* or *blocking* (Irigoin and Triolet, 1988; Wolf, 1989; Renganarayana and Rajopadhye, 2008; Luo, 2020). The goal of loop tiling is to generate optimal chunk sizes for the given memory capacity and configuration in order to increase data locality and parallelism. The focus on memory access and parallel processing are the main differences from array chunking.

Loop tiling is formulated as an optimization problem with the objective to determine the optimal chunk size that results in the fastest execution under the constraints of cache locality and access reuse (Wolf and Lam, 1991). Since the optimization problem cannot be solved efficiently, heuristics are often applied (Dongarra and Schreiber, 1990). Lowenthal (2000) assigns different tile sizes to the CPUs in a multiprocessor environment automatically. The running time of the first few block operations is collected and used to decide the tile size for the subsequent blocks. Kisuki *et al.* (2000) extend this approach into an iterative algorithm to choose the tile sizes, while (Nikolopoulos, 2004) uses different tile sizes for single-thread and multi-thread execution to avoid cache conflicts between threads. Jordan *et al.* (2012) present a multi-objective optimization formulation designed to find the optimal tile sizes and number of threads for a parallel linear algebra program. At compile time, a set of candidate solutions is generated. The best solution is selected at runtime by considering the execution context and the weight of the different terms in the objective. Leung *et al.* (2010) present an optimized tiling strategy that exploits the parallelism and data locality on GPUs. Li *et al.* (2019b) extend this approach with a batching engine, while Kernert *et al.* (2016) and Hong *et al.* (2019) design adaptive tiling strategies for sparse matrices.

5.8 Data Cube Chunking

As discussed in Section 2.6, data cubes can be stored either in a multidimensional (MOLAP) format or a relational (ROLAP) format. Consequently, the chunking strategies for arrays apply to MOLAP, while relational data partitioning applies to ROLAP. In the case of MOLAP, several aspects specific to data cubes have to be considered in chunking. The first aspect is the categorical property of the dimensions. Since this precludes ordering, the set of values assigned to a chunk has to be associated with every chunk — unlike the range boundaries for numerical dimensions. Second, due to the heterogeneous structure of some data cubes — consisting both of dense and sparse regions — a hybrid chunking strategy is more appropriate. For example, dense regions of the cube are stored in dense chunks, whereas the sparse regions are grouped together in sparse chunks. This strategy can be extended further, such that the sparse regions are stored in their original relational format and the measures are computed on-demand — as in the case of ROLAP. This hybrid storage strategy is known as HOLAP (Chaudhuri *et al.*, 2011).

5.9 Summary

- From the multitude of chunking strategies proposed in the literature — including arbitrary, workload-driven, recursive, update-optimized, and adaptive — the most popular are regular and sliced chunking, which are also among the simplest.
- Regular chunks are linearized on storage in row/column major order, snake row/column major order, or with space-filling curves. The chunks can be subsequently partitioned/declustered across multiple devices — disks or nodes in a cluster — based on round-robin, range, hash, cyclic, or graph partitioning functions.
- Dimension suppression is the generally optimized layout for dense chunks, while for sparse chunks, there are multiple compressed layouts — including COO, CSR, and CSC.
- Tensors/matrices are chunked using the regular or sliced strategy, resulting in three common layouts — row strip, column strip, and tile.

6

Multidimensional Array Processing

In this section, we study strategies and algorithms that implement the array algebra primitives introduced in Section 4 over the chunked array storage presented in Section 5. We focus on parallel processing techniques in a distributed array database having a *shared-nothing architecture* over a cluster of workers — or nodes — each hosting an instance of the query processing engine and having its local attached storage. This processing architecture is depicted in Figure 6.1, which illustrates how the chunks of the sparse array in Figure 5.1(b) are distributed across the three nodes X, Y, and Z.

The coordinator is the single query input point into the system. The coordinator stores the system catalog and manages the nodes and their access to the catalog. The chunks of an array are distributed — and possibly replicated — across all the workers, which share access to a centralized system catalog that maintains information about active nodes, array schemas, and chunk distribution. In order to determine the chunks required by a query and their location, a multidimensional index — such as an R-tree (Guttman, 1984) — is built over the chunk boundaries. The index provides the first level of pruning for the efficient evaluation of dimensional indexing and subsampling queries. Access to

the index and all the other catalog data structures has to incur minimal overhead. Thus, the catalog is stored in memory. The query optimizer — also resident on the coordinator — is responsible for computing the optimal execution plan that minimizes the overall query processing time. The parameters that are considered include the chunks transferred among nodes, the transfer schedule, the chunk access plan, and the overlap between processing and communication. Chunk metadata is the main source of data used to compute the execution plan.

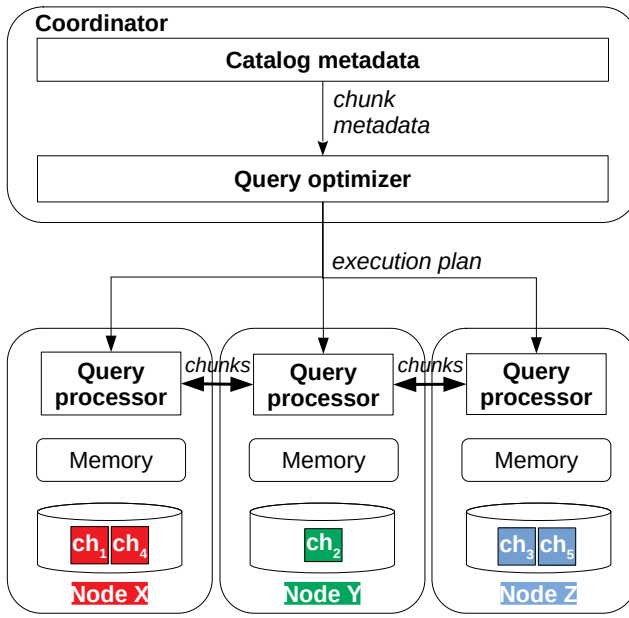

Figure 6.1: Multidimensional array processing system architecture.

The query processor resident on every node implements algorithms for the array algebra primitives. The defining characteristic of the primitives is that they operate on chunks of variable dimensionality and size (Widmann and Baumann, 1998). The algorithms include accessing the chunks from local storage and performing the computation corresponding to the algebra operators. Additionally, the query processor contains a data transfer module that moves chunks between nodes. Data transfer is handled by the underlying distributed computing infrastructure — such as Hadoop (Hadoop Development Team, 2020) and

Spark (Zaharia *et al.*, 2010) — or is implemented on top of optimized communication libraries such as MPI (MPI Forum, 2022). The execution plan provided by the query optimizer specifies the processing details and the communication strategy, which is overseen by the coordinator.

Duggan and Stonebraker (2014) consider an elastic environment — specific to cloud computing — in which the number of worker nodes can increase at runtime proportionally with the size of the arrays. As more chunks are appended to an array — for example, based on a time dimension — more working nodes are added to the cluster. Duggan and Stonebraker (2014) introduce the cyclic workload model consisting of three phases — data ingestion, rechunking, and query processing — that are performed repeatedly. Rechunking is the main operation in this model. It is performed when a sufficiently large number of chunks are ingested to require the addition of nodes. In order to execute rechunking efficiently, incremental algorithms that allocate the new chunks to the added nodes are devised. These algorithms are derived from the declustering techniques presented in Section 5.5. In addition to transferring the chunks to the assigned nodes, only the catalog at the coordinator has to be updated. Queries are executed separately, as in a static environment. This is the setting we follow in the presentation.

6.1 Array Processing Paradigms

In this section, we introduce several paradigms to implement the array algebra primitives. These paradigms define a common chunk-based interface for all the primitives, which allows for their composition into queries that implement complex array processing tasks. Since some of the algebra primitives are second-order functions having a functional parameter, the processing paradigms can also include an interface for the definition of the functional parameter. We classify the paradigms based on this functional interface. At one end of the spectrum, we have the UDF/UDO paradigm, which supports general functions without any particular interface. Map-Reduce imposes a strict interface of three functions, while the GLA paradigm defines a more extensive interface consisting of both required and optional functions.

6.1.1 User-defined Functions (UDF) and Operators (UDO)

ArrayDB. ArrayDB (Marathe and Salem, 2002) implements the AML algebra (Section 4.2), in which arbitrary UDFs are applied to arrays following a structured strategy. AML execution plans pipeline chunks from the input array through operators and generate results one chunk at a time by choosing the optimal result generation order. The execution plan is an operator tree that contains an internal node for every SUB, MERGE, and APPLY primitive, and a leaf node for the input array. The leaf node is a special instance of APPLY that serves input chunks. Based on the catalog metadata, the nodes in the tree are annotated with dimensionality and schema information. During query optimization, the original query tree is transformed into an equivalent one that is more efficient to evaluate using a multi-step top-down tree traversal heuristic. The cost measure is the number of UDF invocations in the APPLY operators in the tree. The tree with the smallest number of UDF invocations — which are treated as black boxes — is considered optimal. Since successive operators must have compatible chunk shapes and generation orders, several rechunking operators are added to the plan. This introduces a second round of query optimization that minimizes the memory used by the operators. The position of the rechunking operators is determined with a bottom-up dynamic programming algorithm. The optimal query plan is executed bottom-up in a pipelined pull-based strategy anchored by calls to the `GetNext(chunk)` iterator.

ArrayUDF. ArrayUDF (Dong *et al.*, 2017) provides a generic implementation for the second-order AML APPLY primitive that is parametrized with different functions for the neighborhood cells, depending on their position relative to the center. In addition to the generalization to multiple UDFs, these functions can be defined only on a subset of the neighborhood and they are not restricted to simple aggregations. ArrayUDF uses the *stencil* abstraction to define the UDFs corresponding to a pattern/shape. The execution of a stencil is split across the array chunks, which are determined dynamically based on the neighborhood shape. This is possible because ArrayUDF is targeted at a shared-disk architecture in which the storage is separated from compu-

tation. In this architecture, the nodes incur similar overhead to access all the chunks of an array since there is no distinction between local and remote chunks. In order to allow for fully parallel execution and eliminate communication altogether, cells at chunk boundaries are replicated into ghost zones. Complex operations are decomposed into a sequence of stencils that are treated independently. There are no inter-stencil optimizations since all the intermediate arrays are materialized.

SciDB. SciDB (Cudre-Mauroux *et al.*, 2009; Stonebraker *et al.*, 2011) implements the ArrayQL algebra primitives (Section 4.6) as UDFs and UDOs. UDFs are scalar functions with cell arguments that return a single value as result. UDOs take one or more arrays as arguments and produce a new array. In addition to the primitives provided by SciDB, the user is given the possibility to implement other types of operators — structural or value-based. A SciDB query execution plan consists of a series of successive UDFs and UDOs, which are instances of the APPLY primitive with different functions as argument. APPLY provides a standard pull-based interface with a `GetNext(chunk)` function that operates over array chunks. The main idea behind the SciDB query execution is to identify segments of successive operators that can be executed in pipelined fashion on a single node. These operators are scheduled and executed on the nodes where the input chunks are stored. Unless successive UDFs/UDOs are commutative, the structure of the query execution plan cannot be altered. Thus, query optimization focuses on parallelizing individual UDF/UDO operators and pipelining array chunks between operators. Whenever chunks have to be transferred across nodes, this is done in a carefully coordinated process. After every processing step, intermediate chunk statistics are gathered and used for the optimization and scheduling of the next segment of operators. The optimization and scheduling are executed dynamically at runtime.

6.1.2 Map-Reduce

The multidimensional data processing system Titan (Chang *et al.*, 1997) and its successor T2 (Chang *et al.*, 1998) introduce the Map-Reduce paradigm as a generic parallel architecture to implement the array

algebra primitives using a unified interface. Later, Map-Reduce has been popularized by Google for highly parallel/distributed computing (Dean and Ghemawat, 2008). Map-Reduce applies the following second-order functions to input chunks in order to produce the result chunks:

- Transform: transform input chunk cells into items. This function is performed concurrently by the nodes.
- Map: map the transformed items to output chunk cells. The coordinator assigns cells to output chunks and determines the node storing the chunk. The nodes transfer data among themselves directly.
- Reduce: aggregate all the items mapped to the same cell to compute the output value. This function is performed concurrently.

The defining characteristic of Map-Reduce is customization — Transform, Map, and Reduce can be parametrized with any processing function. If the required functions are not already available, the user has the ability to implement them by following a well-defined interface. When the functional parameters passed to the Map-Reduce interface are associative decomposable, i.e., commutative and associative, both distributive and algebraic, the input and output chunks can be efficiently processed in parallel and in any order.

Albeit originally introduced as public implementations of the Map-Reduce paradigm for customizable tasks over arbitrary formatted datasets, Hadoop and Spark have evolved into much more comprehensive ecosystems that also encompass array processing. The chunking provided by these systems does not follow the semantics of array chunking, in which adjacent cells are grouped together in order to achieve data locality. As a result, array chunking has to be implemented as the initial step of any computation — which is expensive and unnecessary. Moreover, the processing strategy in these systems is batch-oriented and does not readily support range predicates on dimensions. These shortcomings have led to the development of array-optimized extensions in Hadoop and Spark. SciHadoop (Buck *et al.*, 2011) introduces primitive array chunking and the dimension subsampling primitive in Hadoop. SIDR (Buck *et al.*, 2012) decouples the sequential Hadoop

execution of Map and Reduce into separate asynchronous tasks for every result chunk. Spangle (Kim *et al.*, 2021) extends Spark with array chunking, and bitmasks and bitwise operations that encode valid cells in sparse arrays. Spangle supports both array algebra primitives as well as linear algebra operations on matrices, implemented using the Spark functionality. Overall, these extensions to Hadoop and Spark require a significant engineering effort to embed standard array techniques into Map-Reduce implementations targeted at unordered key-value data collections. However, the contributions to multidimensional array data management are rather limited.

6.1.3 Generalized Linear Aggregates (GLA)

The GLA interface for massively parallel data aggregation is introduced in the GLADE system (Cheng *et al.*, 2012) and extended to multidimensional array processing in EXTASCID (Cheng and Rusu, 2014). A GLA is an associative-decomposable aggregate interface consisting of four user-defined functions — `Init`, `Accumulate`, `Merge`, and `Finalize` — that process array cells independently before combining their partial states into the final result. The semantic of these functions is similar to the Map-Reduce paradigm. `Init` corresponds to Transform, `Accumulate` to Map, while the compound `Merge` and `Finalize` to Reduce.

Additionally, GLAs are enhanced with functions specific to array operations. `BeginChunk` is invoked before the cells inside a chunk are processed. `EndChunk` is similar to `BeginChunk`, invoked after processing the chunk cells instead. These two functions operate at chunk granularity. They are the places where side-effect operations are executed. For example, array cells can be sorted according to a dimension that makes the processing more efficient in `BeginChunk`. In `EndChunk`, data that 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. `LocalMerge` puts together local GLAs created on the same

processing node, while `RemoteMerge` is invoked 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 locally after the GLAs at a processing node are merged.

Cheng and Rusu (2014) show how all the primitives in the ArrayQL algebra can be expressed in terms of functions from the GLA interface. They even provide a mapping for the more complex AML APPLY operator — which is not optimally supported in Map-Reduce. It is important to notice that not all the interface methods have to be implemented for every array algebra primitive. Overall, while GLAs are similar in spirit to Map-Reduce since they provide a unified interface to express array operations, they are more generalizable. Moreover, they also integrate organically with relational data.

6.2 Array Operators

In this section, we present algorithms for the array algebra primitives introduced in Section 4. We focus on the primitives that require data processing beyond the catalog metadata stored at the coordinator. These include SUBSAMPLE, FILTER, APPLY, REDUCE, and JOIN. Their corresponding relational algebra operators are selection, projection, group by aggregation, and join, respectively.

6.2.1 Indexing on Dimensions (SUBSAMPLE)

Subsampling — or positional indexing on dimensions — can take multiple forms depending on the type of indexing — point or range — and the number of dimensions it is applied to — all or a subset. In the most selective case, there is a point predicate on every dimension, which requires access to a single chunk. Dimensions without conditions — which include their entire range — lead to the inclusion of all the chunks in the corresponding hyperplane to the result. Subsampling is implemented as an index scan operator based on chunking (Section 5.2). The goal

is to access from storage only the chunks that overlap with the range predicate. In a subsequent refinement step — applied to the accessed chunks — only the cells within the range predicate are extracted. Since the execution time is dominated by the number of accessed chunks, the optimization strategies are targeted at designing effective chunking methods — which are discussed extensively in Section 5.

6.2.2 Filter on Attributes (FILTER)

Since the array is chunked and organized based on dimensions, filters on attribute values require inspecting all the chunks. This operations corresponds to a full sequential scan. In the worst case, all the array data have to be accessed. However, if columnar storage is used for the attributes (Papadopoulos *et al.*, 2016), the amount of data read from storage can be reduced — only the required attributes are read. The number of accessed chunks can be further reduced by storing the minimum and maximum value — the range — of each attribute across chunks (Cheng and Rusu, 2014). While the range can be large since the attribute values are not clustered, reductions are possible for certain range queries. Another alternative is to build an unclustered index that stores the chunks where every distinct value of the attribute appears (Blanas *et al.*, 2014). However, this solution requires additional space for storing the index and additional time to access the index prior to the data — as with any index structure.

COMPASS (Xing and Agrawal, 2018) partitions the cells of a chunk based on the value of an attribute. This results in buckets that group cells with close values together — instead of adjacent cells. Every bucket stores the dimensional indices of the assigned cells together with the residual attribute values normalized to the lower bucket boundary. Given the unknown cell assignment to buckets, the indices have to be stored explicitly for every cell. Dimension suppression is not possible anymore. The COMPASS storage layout is essentially combining dimension chunking with attribute range-based partitioning in order to optimize queries with selections on both dimensions and attributes. However, improvements are limited only to the case when the range on dimensions matches exactly chunk boundaries and an attribute selection is present.

6.2.3 Aggregation (APPLY + REDUCE)

Unlike relational aggregation — which groups tuples based on their value — array aggregations are structural — they group array cells based on their positional relationship. The structural relationship is encoded as a shape/pattern argument to the APPLY primitive that is evaluated for every cell in the array and results in an array with the same size. This is different from relational aggregation, which generally results in a smaller table. The values in the new array are obtained by performing the REDUCE primitive over the cells covered by the shape. Wang *et al.* (2014) classify structural aggregations based on the shape into grid, sliding, hierarchical, and circular, while Choi *et al.* (2019) define top-k aggregates over overlapped and disjoint subarrays.

There are two approaches to perform structural aggregations over chunked arrays. The first approach uses overlapped chunking (Section 5.2.2) in order to confine the computation to a single chunk and avoid data transfer. The main benefit is that all the chunks can be processed concurrently without any synchronization — aggregation becomes trivially parallel. The requirement is that the shape parameters for all the queries are known beforehand since the shapes are used to perform chunking. Additionally, overlapped chunking increases storage because of cell replication. ArrayStore (Soroush *et al.*, 2011) and ArrayUDF (Dong *et al.*, 2017) — among others — implement this strategy.

The second strategy for structural aggregation is characteristic to the GLA processing paradigm presented in Section 6.1.3. It consists of two stages. During local aggregation, all the cells that are internal to a chunk are processed in parallel. The cells at chunk boundaries require access to cells from other chunk(s). These are handled in the merging stage by transferring them to the same node. Merging can also be performed concurrently across chunks. Further optimizations are possible for reduce functions that are associative-decomposable since they are insensitive to the execution order. Compared to overlapped chunking, the merged strategy is independent of chunking and the shape parameter, thus, more general. Moreover, the overhead incurred by data transfer can be overlapped with the aggregate computation.

6.2.4 Joins (JOIN)

Structural — or dimension:dimension — joins (Section 4.6) are specific only to array databases because the chunking of the arrays is used to optimize data transfer among nodes and local processing at a node. The other types of array joins require value-based repartitioning, which is standard for relational joins. As such, we focus on algorithms for structural array equi-joins and their generalization to shape-based similarity join.

Structural join. The standard algorithm to implement array equi-join is a special form of nested-loop join operating at chunk level (Algorithm 3 in Soroush *et al.* (2011)). The join iterates over chunks of the outer array α. For every chunk, it looks up the corresponding chunks in the inner array β, retrieves them all, and joins the outer chunk with every of the inner chunks in turn. The join between two chunks is itself implemented as nested loops iterating over chunk cells. If the cells in the chunks are sorted according to dimensions, the optimal merge join algorithm can be executed instead.

The structural join algorithm can be readily implemented in a distributed array database. Once the joining cells are determined for an outer chunk — or all the outer chunks — the node identifies their location by querying the catalog on the coordinator. A message is sent to the corresponding node for every chunk and, when the chunk is received, the output cells in the output array τ are computed. While the degree of parallelism across nodes is maximized, there are several problems with such an asynchronous decentralized approach. Although every node aims to minimize the amount of transferred data — it behaves locally optimal — there is no guarantee that the overall data are minimized. In fact, this is very unlikely since nodes do not coordinate at all. As a consequence, the actual data transfer can be severely imbalanced due to the contention for network bandwidth. In the extreme case, all the nodes in the cluster send/receive data to/from the same node. Load-balancing beyond what is achievable with a uniform chunk distribution to nodes is not considered at all in structural join.

Baumann and Merticariu (2015) propose an alternative in which the joining chunks across the two arrays are grouped into components that are processed as a single unit. The components are computed as an Euler circuit in a bipartite graph, in which chunks are the vertices and edges correspond to joins. Every component is assigned to a node for processing. The node requests the chunks — which can belong to any of the two arrays — following a sequential order that guarantees that result cells are incrementally produced and chunks are minimally accessed. The benefit of this solution is that it reduces the amount of transferred data by moving a chunk only once — in some cases, multiple transfers are still necessary. The drawback is the decrease in the number of tasks executed concurrently, from one for every chunk in the outer array to one for every component. The overall impact of these changes to structural join depends on the initial chunking of the two input arrays.

Map-Reduce join. In the structural join algorithm, the computation is executed exclusively at nodes storing chunks from the outer array α. Unless these chunks are distributed across the entire cluster, there are nodes that do not participate in join processing. Moreover, if chunk distribution is not even, there is load imbalance. Map-Reduce join (Blanas *et al.*, 2010) — as a direct extension of distributed Grace hash join (Dewitt *et al.*, 1990) — guarantees that all the nodes in the cluster participate in join processing. Load-balancing is enforced at runtime through dynamic assignment of chunks to nodes. The tradeoff to achieve these two goals is network traffic. Map-Reduce join is far from network-optimal because it transfers the complete arrays over the network.

Map-Reduce join works as follows. The result array is divided into logical non-overlapping chunks, i.e., join units. These are computed from the schema of the result array — specified by the user, or inferred by the system in some restricted situations. Join units are computed at the coordinator and sent to all the nodes storing data from arrays α and β, or they are encoded directly into the Map hash function. Every node partitions the cells it stores over the join units. This is done concurrently across all the nodes. Cells in array α are assigned to a single join unit.

Cells in array β are assigned either to a single join unit — for equi-join — or they can be replicated in several units — for similarity join. The data alignment phase, i.e., shuffling, transmits all the partitions belonging to the same join unit to a single node for the computation of the result τ. In the case of distributed hash join, the assignment of join units to nodes is static and uniform. In Map-Reduce join, tasks are assigned dynamically at runtime to better adapt to the processing capacity of the nodes, resulting in more adaptive load-balancing.

Shuffle join. In structural join, every node minimizes its local data receiving, while Map-Reduce join ignores communication completely. Shuffle join (Duggan *et al.*, 2015b) aims to minimize the overall data transfer imposed by the execution of an array equi-join, while guaranteeing some form of load-balancing. It extends upon the track join minimal network traffic distributed hash algorithms introduced in Polychroniou *et al.* (2014). The main idea is to consider the assignment of join units to nodes as a global optimization problem and solve it after all the nodes finish their local partitioning. The amount of data each node has in a join unit is the principal decision variable. Several algorithms are considered, including a simple minimum bandwidth greedy heuristic that assigns a join unit to the node storing the largest portion of cells in the unit; a tabu search algorithm that incorporates load-balancing into the minimum bandwidth heuristic; and an integer programming formulation that optimizes the end-to-end execution time. The proposed analytical cost model has the inherent limitation that communication and computation cannot be overlapped across the join units assigned to the same node. Moreover, the order in which a node has to send its partitions is not computed — it is arbitrary. A global synchronization mechanism that enforces a single node to transmit data to a destination at any time instant is deployed in order to prevent network congestion. However, this can have the negative effect of stalling nodes.

Array similarity join. Zhao *et al.* (2016) design a parallel algorithm for the shape-based array similarity join operator introduced in Section 4.6. This algorithm is an optimized structural join that minimizes the overall data transfer and network congestion while providing load-balancing

across the nodes that store array chunks. The algorithm has two stages — optimization and execution. In the optimization phase, the algorithm computes an optimal execution plan for every worker node. The plan consists of three components — transfer graph, transfer schedule, and data access plan. Finding the optimal plan is challenging because it involves solving a complex non-linear optimization problem. The proposed solution decomposes the original optimization problem into three separate sub-problems — one for every plan component — and solves them independently using graph-based heuristic algorithms. At query execution, the algorithm overlaps I/O — disk and network — with join computation at chunk granularity. Network transfer and local disk I/O are each handled by a separate thread. The join between two chunks is executed in a separate worker thread. The algorithm is configured with a pool of worker threads, allocated based on the number of CPU cores available in the system. This allows for several pairs of chunks to be joined concurrently. All the threads — I/O and workers — execute asynchronously and coordinate through message passing.

Li *et al.* (2020) introduce a shuffle-based algorithm for band-joins, which are a restricted form of array similarity joins with symmetric shapes. Since the input arrays are not grid-partitioned — or chunked — the join requires a complete data shuffle, similar to Map-Reduce join. The algorithm applies a recursive partitioning scheme based on the band parameter and using samples from the input arrays. This partitioning allows for a finer-grained replication of the array cells across nodes, which results in less data transfer compared to standard grid chunking.

Join algorithm comparison. Table 6.1 summarizes the properties of the array join algorithms. Structural and similarity join are the most general of these algorithms. Map-Reduce and shuffle join are in the same family of equi-join algorithms. Extensions to array similarity join and other types of joins are possible, however, they incur costly modifications. Shuffle join is the only algorithm that aims to minimize the overall data transfer. Map-Reduce join incurs heavy all-to-all communication, while structural join targets only local optimizations at every node. Network congestion is addressed only by shuffle join through a runtime global synchronization mechanism that gives writing access

on a link to a single sender. Load-balancing is supported as a runtime reactive process in Map-Reduce join. The approach in shuffle join is to embed load-balancing into the data transfer scheduling. Thus, only Map-Reduce and shuffle join include all the nodes — not only the ones storing the join arguments — in the processing. Since they do not consider the initial chunking, Map-Reduce and shuffle join require complete data repartitioning and mapping to the output array space. As a result, repartitioning incurs unnecessary data replication. The array similarity join operator extends the benefits of the other algorithms by minimizing the overall transfer and network congestion while providing load-balancing across the nodes that store data, but without completely repartitioning and replicating the arrays.

Table 6.1: Comparison of array join algorithms.

	Structural	Map-Reduce	Shuffle	Array similarity
Targeted join	general	equi-join	equi-join	general
Data transfer	locally optimal	suboptimal	globally optimal	globally optimal
Network congestion	ignored	ignored	runtime	optimal scheduling
Load-balancing	ignored	runtime reactive	static optimized	static optimized
Processing nodes	store one of α or β	all	all	store α or β
Repartitioning	not required	complete	complete	not required
Replication	minimally required	suboptimal	suboptimal	minimally required

6.3 Advanced Array Processing Techniques

In this section, we present advanced processing techniques for array data. These techniques build upon the general processing paradigms and the algorithms implementing the array primitives. We discuss materialized views, versioning, provenance, and uncertainty — all of which are current research topics in data management and databases.

6.3.1 Views

Zhao *et al.* (2017) introduce the concept of *materialized array views* defined over complex shape-based similarity join aggregate queries. Since shape-based array similarity join is a generalization of array equi-join and distance-based similarity join, materialized array views cover an extensive class of array algebra operations. With regard to SQL, array

views include the class of join views with standard aggregates. The incremental array view maintenance is considered under batch updates to the base arrays. Batch updates are essential for amortizing the cost of network communication and synchronization in a distributed environment. There are two primary challenges posed by incremental array view maintenance under batch updates. The first challenge is identifying the cells in the base arrays that are involved in the maintenance computation and the cells that require update in the array view. The second challenge is due to the distributed nature of array databases. Given the current distribution of the arrays and the view, the challenge is to find the optimal strategy — data transfer and computation balancing — to integrate the updates into the view.

Zhao *et al.* (2017) model distributed array view maintenance as an optimization formulation that computes the optimal plan to update the view. The optimization continuously repartitions the array and the view based on a window of past batch updates. In the long run, repartitioning improves view maintenance time by grouping relevant portions of the array and the view and by distributing join computation across the cluster. Meanwhile, repartitioning does not incur additional time because it takes advantage of the communication required in view maintenance. Since the optimization cannot be solved efficiently, the formulation is decomposed into three separate stages — differential view computation, view chunk reassignment, and array chunk reassignment — that are solved independently by effective cost-based heuristics. The materialized views are integrated in optimizing similarity join queries using an analytical cost model that chooses the best alternative between a complete similarity join and a differential query on the view.

6.3.2 Versioning

The idea behind versioning is the requirement to never modify the array in place. Every modification has to generate another version of the original array. Some of the versions are given names, while the majority are identified based on a sequential identifier assigned automatically by the system. Maintaining versions allows for novel *time travel queries*. These queries retrieve a particular version of the array at a given instance

in time or return the transformations data go through across a subset of the versions. Abstractly, array versioning corresponds to adding a new time dimension to the original array, while time travel queries are either slice or range queries along the time dimension.

An array versioning system has to address several problems. The first — and most important — is how to minimize the storage space occupied by the versions? The naive solution to materialize every version independently results in storage proportional with the number of versions. The observation allowing for improved solutions is that new versions modify only a small portion of the array. Thus, materializing only the modifications has the potential to save significant storage. Alternatively, a delta array that contains only the difference between the base array and a version can be generated. The second question is which version to materialize? Under the assumption that the newest version is queried more frequently, the newest version should be materialized. To reduce the number of previous versions that have to be re-encoded based on every new version, older versions are maintained as the difference from the immediately successive version. This way, a chain of delta versions in which version one is materialized as the difference from version two, version two as the difference from version three, and so on, results. When versions are maintained as deltas, answering time travel queries is more complicated since heavier computation — combining deltas with a materialized version — is required. Thus, efficiently answering time travel queries with deltas is the third versioning problem to consider.

Three solutions for maintaining and querying versioned arrays are built on top of SciDB. All of them consider the simplified problem of versioning single-valued arrays with regular chunking. The main idea is to store a single materialized version of a chunk and all its deltas inside the same chunk.

Seering *et al.* (2012) study how to optimally encode a series of consecutive versions given at once under the assumption that queries across all the versions are equally probable — materializing the last version is not optimal in this case. The goal is to determine which versions to materialize and based on which materialized version to create deltas for the non-materialized versions. The problem is formulated as a graph having versions as vertices. Edges represent deltas from one

version to another and are annotated with the storage required by the delta. The optimal version to materialize and the sequence of deltas are computed as the minimum spanning tree of the resulting graph — where the root corresponds to the materialized version.

Soroush and Balazinska (2013) and Xing *et al.* (2018) consider a scenario where versions of the same array are created incrementally, one after another. The assumption is that the most recent version is queried more frequently than the previous ones and the probability of querying a version decreases with its age. Thus, the most recent version is materialized and every other version is stored as a delta from the immediately successive version. When a new version is created, only the second newest version has to be delta encoded from the newly created version. The storage space required by the versions is reduced by delta and run-length compression applied at chunk level. The execution time increases with the age of the queried version since a larger number of deltas have to be merged. Soroush and Balazinska (2013) introduce skip links — delta encodings between non-consecutive similar versions — that are built lazily while evaluating queries. Only skip links to the most recent version are considered and only when querying an old version. Skip links are also used to provide faster approximate answers to other non-similar time travel queries.

6.3.3 Provenance

Scientific processing consists of workflows of operators that take as input arrays, apply multiple transformations, and generate an output array. Given an output cell, it is common to ask what are the cells in the input arrays on which the output cell depends? Or the inverse equivalent query, what are the output cells that depend on a given input cell? To complicate the problem further, these types of queries can be asked for any pair of operators in the workflow, not necessarily the source and the result arrays. To answer these queries after the workflow is processed — without entirely re-executing it — lineage data have to be stored for every operator in the workflow — in both directions. If such data are generated at cell level for all the arrays in the workflow, the amount of additional space may be larger than the original data.

Also, it is not guaranteed that answering the provenance queries is going to be faster than re-executing the workflow. Determining which data to materialize and which to recompute is the fundamental question in array provenance.

SubZero (Wu *et al.*, 2013) is a prototype system for managing array provenance data. It is based on the idea of region lineage as an intermediate level to generate and store lineage data based on locality. SubZero introduces a lineage API that allows developers to expose lineage data from UDFs through mapping functions. Given a workflow consisting of a series of operators, SubZero uses an optimization framework to select the optimal strategy to generate lineage data for a given workload. Multiple strategies to generate lineage data are considered for every operator and their corresponding cost. SubZero can record and store the lineage data at workflow runtime or it can decide that it is more efficient to re-execute the workflow, case in which the lineage data are generated only during the execution of the provenance query — after answering the query, the provenance data are discarded.

Every operator in the workflow can support multiple types of lineage data. Black-box lineage data record only the input and output arrays of every operator together with the execution parameters. Cell-level lineage records pairs of (input, output) array cells, where the output cell is dependent on the input cell. An input/output cell can be part of many pairs. Region lineage is similar to cell-level at a coarser granularity — all-to-all cell-level lineage applies between every cell in the input region and every cell in the output region. Multiple strategies to generate and store the region lineage data are presented. Operators can implement one or more strategies. In full lineage, all the region pairs are stored explicitly. In mapping lineage, only two mapping functions — forward and backward — have to be specified for an operator. Every function specifies the output coordinates as a function of the input cell coordinates. The functions do not depend on cell content. They are structural array primitives. No lineage data are stored in this case. At query time, the lineage can be computed for every cell based on the coordinates.

6.3.4 Uncertainty

Uncertainty can manifest in array databases in two different ways — value uncertainty and position uncertainty (Peng and Diao, 2015). Value uncertainty corresponds to the situation when the value of an attribute is modeled by a probability density function, and is an immediate extension of probabilistic databases. Position uncertainty applies to dimensions. In this case, the indices of a cell are uncertain. They are modeled by a multidimensional probability density function. Essentially, the values at a particular index combination can belong to multiple cells — identified by different indices. From a chunking perspective, position uncertainty implies cell replication across chunks. Moreover, value uncertainty is a direct consequence of position uncertainty. The inclusion of uncertainty in the structure and content of an array requires defining the semantics of the array primitives over probability density functions and designing efficient algorithms to evaluate the uncertainty-enhanced primitives.

Ge and Zdonik (2010) assume that the values of an array attribute are drawn from an unknown probability density function. They also assume that the values exhibit positional correlation, which means that neighboring cells are more likely to have similar values. The proposed A^*-*tree* is designed to capture these positional correlations in a hierarchical tree structure similar to a quad-tree index. The leaves of the tree contain the cell values in the array, while the intermediate nodes encode the correlation among the corresponding children nodes. The nodes in the first intermediate level encode correlations among adjacent cells. The nodes at higher levels capture the correlation among cells that are farther apart. A^*-*tree* supports the efficient execution of reduce and structural join array algebra primitives through Monte Carlo sampling (Ge *et al.*, 2011), which can be performed top-down from the root of the tree by applying aggressive pruning.

Peng and Diao (2015) define the possible range of an array cell as the subarray within the bounds of which the cumulative probability to find the cell is approximately one. The range on a dimension is determined as a constant number of standard deviations from the position of the cell. Based on the possible range associated with every cell, probabilistic

subsampling and structural join operators are defined. They both take a threshold parameter as input argument and include a cell in the result if its probability at a location that overlaps with the range predicate is larger than the threshold. This change requires the transformation of set membership operations from the original primitives into range overlap checks in probabilistic subsampling and structural join. The standard solution to handle range overlapping is boundary cell replication across chunks. Peng and Diao (2015) introduce a variant that replicates only the cells having large positional variance according to their probability density function. In order to support efficient structural joins, an index is built over the cell range overlap of the inner array. This allows fast identification of the relevant cell pairs in the inner loop of the nested loop join algorithm — which becomes indexed nested loop join.

6.4 In-situ Array Processing

Dense arrays — or grids — generated by scientific applications are stored in raster files with a self-describing format, which makes them queryable, portable, and sharable. A raster file can contain one or more arrays, each with its own chunking. There are multiple popular raster file formats, including FITS (The FITS Support Office, 2022), GeoTIFF (Open Geospatial Consortium, 2022), HDF5 (The HDF5 Group, 2020), and netCDF (UniData, 2022). Every format is accompanied by a corresponding I/O library that provides access to the arrays at chunk granularity. The access is done programmatically through function calls that identify the chunks based on their dimensions. More advanced libraries that implement a large range of array algebra primitives are either integrated with the access libraries or built on top of them. Two such advanced libraries are netCDF Operators (NCO) (The netCDF Development Team, 2022) and Geospatial Data Abstraction Library (GDAL) (The GDAL Development Team, 2022) — which supports more than 100 different raster formats and implements primitives on a general intermediate array representation.

The goal of in-situ array processing is to access data directly in the raster file format — without loading in the array database. This can be achieved by integrating the raster I/O libraries within the array scan

operators. The new bridge operators produce chunks by retrieving the corresponding data from the file and mapping to the in-memory format of the array database — known to all the other operators. This is the approach taken by Wang *et al.* (2013a) who implement a subsampling scan operator that maps complex dimensional range predicates into I/O library calls. In subsequent work, Wang *et al.* (2014) push the execution of structural — or chunk-level — aggregates to the scan operator. SDS/Q (Blanas *et al.*, 2014) introduces an index scan operator for cell value filters based on an external bitmap index. ArrayBridge (Xing *et al.*, 2018) implements both a scan and a save operator that converts SciDB chunks to HDF5 files. The save operator allows for the creation of multiple versions of an array under modification operations and the execution of time travel queries — which are pushed to the scan operator. Data Vaults (Ivanova *et al.*, 2012) implement just-in-time loading and caching to preserve the converted arrays inside the MonetDB database. Zhao *et al.* (2018) design a distributed caching layer on top of raster files to speed up access to frequently queried cells. The chunking in the cache is based on the workload and can be different from the one in the raw files. In summary, the purpose of all these solutions is to reduce the number of raster file accesses and minimize the number of chunks passed to the upstream array primitives.

A more advanced form of in-situ processing performs array operations by invoking the already existing functions in some of the raster file libraries. This approach is taken by ChronosDB (Zalipynis, 2018), which delegates the complete execution of array operations to the NCO and GDAL libraries. The main limitation of these libraries is the single-node single-file architecture without support for array chunking and distributed processing. ChronosDB addresses these shortcomings by implementing a coordinator that manages the invocation of local library instances running on the cluster nodes. Upon receiving a command — that has the same syntax as the corresponding library function — the coordinator forwards it to the nodes, which perform it on their local files. However, since the files are chunks of a distributed logical array, the local outputs of the command have to be composed to generate the complete result. This composition introduces two problems. First, the operations required may be different from the original library function.

Thus, in order to implement the distributed version, the composition required by every function has to be determined. The composition can be quite different from the original command and may not even be available as a library function — case in which additional logic has to be implemented. The second problem is that the composition requires data transfer among nodes, which has to be newly added on top of the raster library. In addition to these issues, ChronosDB has to create a unified catalog with chunk metadata over the raster files. This requires the identification of the files that store subarrays — or chunks — from the same logical array and, subsequently, dimension alignment and rechunking. Since these operations are performed in-situ on the raster file, the user has to manually copy the files to the nodes. Consequently, converting a single-node raster library into a distributed version requires significant effort that goes well beyond invoking separate instances of the same function at every node.

6.5 Relational Array Processing

The standard approach to support array processing inside a database is to integrate arrays in the relational data model. Given the multiple representations of arrays as relations — introduced in Section 2.4 — several processing strategies are possible. In the case of *array as table* representation, the array primitives can be mapped directly to relational algebra operators and SQL. However, the mapping can result in complex sequences of operators that are both unintuitive and inefficient. The reason for this is the complete reliance on relational operators, which leads to the inability of applying array-specific optimizations. The processing strategies for the other representations depend on the availability of a composite array data type and corresponding operators. When these are supported, array operations can be included in queries as direct function calls. This leads to mixed queries that consist of relational operators and array primitives. Since the optimizations across the two types of operators are mostly limited to the placement of the array primitives within the relational execution plan, achieving the best performance is challenging. When the composite data type is unavailable, arrays are encoded as binary large objects (BLOB), which

are processed exclusively through external user-defined functions (UDF). These can reside inside the database or at the application level. In the first case, the optimizations are minimal since UDFs are treated as black boxes by the query optimizer. For the later, the database provides only storage while the application has to implement all the necessary array processing — including optimizations across primitives.

(S)RAM. (S)RAM (Ballegooij, 2004; Cornacchia *et al.*, 2008) implement the (S)RAM array algebra (Section 4.3) on top of the MonetDB/X100 relational database system (Idreos *et al.*, 2012; Boncz *et al.*, 2005) using the *array as table* representation. The array algebra primitives are mapped into relational algebra expressions containing selection, join, group by, and other relational algebra operators. The execution is purely relational. The benefit of such an approach is that an existing system — MonetDB in this case — is used for processing. No system has to be rebuilt from scratch. The disadvantage is that the mapping is not always optimal due to the impedance mismatch between relations and arrays. RAM is further extended to a parallel setting in Ballegooij *et al.* (2005). Two rules — partitioning and aggregation — that allow array-specific query decomposition as an extension of the relational set semantics are introduced. Partitioning allows for sub-arrays of the same array to be evaluated in parallel whenever there is no dependency between cells. Aggregation allows for commutative and associative functions over arrays to be evaluated concurrently.

6.6 Tensor Processing

Optimized matrix multiplication implementations follow the approach pioneered by Goto and van de Geijn (Goto and Geijn, 2008; Goto and Geijn, 2009). The logical operations involved in this approach are matrix partitioning, sub-matrix packing, and the invocation of highly-optimized multiplication kernels. The input matrices A and B, as well as the result matrix C, are successively partitioned into sub-matrices labeled A_i, B_j, and C_p that fit into the various levels of the cache hierarchy. The sub-matrices A_i and B_j are packed — or copied — into temporary buffers in a special storage format that facilitates vectorization and

memory locality. The size of these sub-matrices is determined by cache blocking parameters, such that A_i is resident in the L2 cache while B_j is retained in the L3 cache. These sub-matrices are then fed into an optimized inner kernel that performs the actual matrix multiplication operation. In the BLIS approach (Zee and Geijn, 2015), the sub-matrices are further partitioned according to register block sizes, such that a pair of input blocks fit into L1 cache while the result C_p is stored in CPU registers.

6.6.1 Distributed Matrix Multiplication

The three-stage approach can be extended to large-scale matrix multiplication in a distributed environment (Choi *et al.*, 1992; Thomas and Kumar, 2018). In this case, the three stages of the process are repartitioning the input matrices among tasks (matrix repartition), performing local matrix multiplication within each task (local multiplication), and aggregating the intermediate results of local matrix multiplication (matrix aggregation). Since the number of operations in local multiplication is the same, the challenge is reducing the communication overhead that occurs in matrix repartition and aggregation. There are methods that perform communication only in the repartition stage (Boehm *et al.*, 2016), methods that incur most of their communication in aggregation (Gu *et al.*, 2017), methods that have high overall communication while minimizing memory usage (Seo *et al.*, 2010; Yu *et al.*, 2015; Yu *et al.*, 2017), and methods that optimize across both communication and memory usage (Han *et al.*, 2019). Among these methods, three algorithms stand out.

In broadcast matrix multiplication, matrix A is chunked into row strips while matrix B is fully replicated across all the cluster nodes. This allows for the computation of the result matrix C in a single step, without the need for aggregation. Matrix C has the same chunking as A. Broadcast matrix multiplication requires that at least a row from the larger matrix and the entire smaller matrix fit in memory.

In cross product matrix multiplication, matrix A is chunked into column strips while matrix B is chunked into row strips. The multiplication between a column strip and its corresponding row strip generates

only a partial aggregate of the result cell. In order to compute the complete result, a subset of the partial aggregates equal in size to the result matrix has to be replicated across all the cluster nodes. However, this does not result in having result matrix C fully replicated across all the cluster nodes — C is generated chunked into column strips, just as A.

Replicated matrix multiplication — known as 3D matrix multiplication (Agarwal *et al.*, 1995) — reduces the amount of data transferred during matrix aggregation by replicating the input matrices A and B — which are chunked as tiles — multiple times in the matrix repartitioning stage. A is replicated J times, where J is the number of tiles along dimension j in B, while B is replicated I times, where I is the number of tiles along dimension i in A. The result matrix C is replicated P times, where P is the number of tiles along dimension p in A and B, respectively. The allocation of the replicated tiles to nodes is controlled by their contribution to the result matrix tiles. The optimal allocation has to consider the overall communication across the repartitioning and aggregation stages (Jankov *et al.*, 2021). These three methods for distributed matrix multiplication are implemented in some shape by all the tensor processing systems.

Cumulon. Cumulon (Huang *et al.*, 2013) considers the multiplication of tiled matrices stored in a distributed file system. The multiplication consists only of local multiplication and matrix aggregation. It is assumed that the matrices are properly partitioned, thus, there is no need for matrix repartition. In local multiplication, all the intersecting pairs of tiles from the two matrices are multiplied following the 3D matrix multiplication algorithm. This is done by reading a pair of tiles from the distributed file system, performing the multiplication using the JBLAS library, and writing the intermediate result tile back to the distributed file system. In matrix aggregation, all the intermediate tiles that contribute to a result cell are read and aggregated to produce the final result. Reading and writing from/to the distributed file system may require network data transfer. Cumulon optimizes for the minimum number of processing nodes that can perform a matrix multiplication within a given time budget. For this, it considers both the execution

time of an in-memory matrix multiplication task — constrained by the memory capacity — as well as the network transfer time required by a node. Estimates for these quantities are derived from simulation with different matrix sizes and tiling factors. The optimal number of nodes is found using a search algorithm that considers values between easy-to-determine lower and upper bounds. Overall, the main characteristic of Cumulon is that it considers a dynamic cloud environment in which the number of processing nodes can be assigned at runtime.

DMac. DMac (Yu *et al.*, 2015) optimizes the execution of a chain of matrix multiplication and transpose operations. Matrix repartition is necessary to convert between row and column strip chunking, and for broadcasting a matrix to all the processing nodes. Repartitioning can be performed at every operation in the chain and incurs an afferent communication cost. DMac aims to minimize the overall communication cost across the entire chain of operations. This corresponds to finding the optimal chunking across the entire chain. DMac introduces two heuristics that traverse the chain of operations in order and make local partitioning decisions based on the previous operations in the chain. The cost of local matrix multiplication is not included in the optimization. Moreover, since full rows and columns belong to the same chunk, the result matrix is directly obtained without further aggregation. The main limitations of DMac are the reduced chunking strategies with fixed sizes, the simple communication cost model that depends only on the matrix size, and the suboptimal local matrix multiplication kernels. Nonetheless, DMac is the first attempt that includes repartitioning in the optimization of a chain of matrix operations.

SystemML. SystemML (Boehm *et al.*, 2016) optimizes the implementation of every matrix operator independently — or a small group of operators fused into a compound operator (Boehm *et al.*, 2018). The core of SystemML is a hybrid sparsity-aware fixed block size matrix library — with square blocks of 1000×1000 cells — which operates on the entire matrix on a single node, or blocks of a matrix in a distributed setting. The matrix library provides seven matrix multiplication physical operators requiring different degrees of data repartitioning and network

communication. The best operator for a given task is selected based on memory estimates, data, and cluster characteristics. While matrix repartitioning is avoided as much as possible, this is not done with a principled cost-based solution since data transfer is not considered in the selection. The reason for this approach is the reliance on a multi-level distributed cache buffer pool that stores matrices in-memory, evicts them when necessary, and handles data exchange among processing nodes transparently. For a chain of matrix multiplications, SystemML exploits the associativity property and orders the multiplications in a way that avoids large intermediate results. Moreover, if a sparse matrix is involved in the multiplication, the result cells are computed only for the non-zero entries.

MatFast. MatFast (Yu *et al.*, 2017) estimates the sparsity of intermediate matrices in order to minimize the memory usage and communication cost of a chain of sparse matrix multiplications. For this, the optimizer uses data dependency among matrices, dynamic cost-based analysis, and rule-based heuristics to determine how to partition the input and intermediate matrices. The sparsity of a matrix multiplication is estimated by sampling rows and columns from the two matrices and computing the number of non-zero entries in their outer product. In the case of a chain, the multiplication that results in the sparsest matrix is performed first. Then, the process is repeated until a single matrix is obtained. The intuition behind this approach is that sparser matrices are smaller and result in less data transfer. The assignment of the partitioning scheme to every matrix in the chain is done using a cost model that considers the network traffic required to repartition a matrix. The cost model is based on the size of the matrix, which corresponds to sparsity for sparse matrices. The cost model is integrated in a greedy plan generator that assigns the partitioning to a matrix in the chain starting from the result matrix, following with the intermediate multiplications, and ending with the input matrices. This planning applies repartitioning closer to the input matrices since their cost can be estimated more accurately. Although MatFast supports tile-based partitioning, the cost model is defined only for row/column strip and matrix as single tuple chunking. Thus, tile-based matrix multiplication is not considered in processing.

DistME. DistME (Han *et al.*, 2019) performs distributed multiplication of tile-based partitioned matrices in a cloud setting where the number of processing nodes is variable. Given a number of nodes, DistME determines the maximum size of the tile-based chunks that can be multiplied in the available memory. Then, the two matrices are repartitioned accordingly in a preprocessing step executed before the multiplication is performed. Multiplication starts with the repartitioned matrices and proceeds by pairing the corresponding tiles, which are locally multiplied with an optimized BLAS GPU kernel. Since the GPU memory is limited, additional repartitioning is required. Moreover, the execution of the GPU kernels is streamed in order to reduce data transfer. This results in optimal performance for matrix multiplication.

TRA. TRA (Yuan *et al.*, 2021) defines a set of equivalence rules for kernel function composition and repartitioning applied to the physical operators in the tensor algebra. There are four physical operators in TRA — two for repartitioning, broadcast and shuffle, and two for local tensor operations, multiplication and aggregation. The equivalence rules provide mappings among combinations of these operators. However, TRA does not provide an enumeration algorithm for the generation of the equivalent mappings. The execution cost of an expression of operators is assessed with a cost model that measures only the data transferred over the network, which corresponds to the cardinality of the tensor. This is appropriate only for dense tensors. The computation of an operation is not included in the cost model because operations are executed through functions from optimized kernel libraries. These functions have the same execution cost when performed on tensors with the same dimensionality. Thus, the only relevant decision for performance is whether to shuffle or broadcast a tensor. The TRA cost model is enhanced with terms corresponding to execution in subsequent work by the same authors (Jankov *et al.*, 2021). In this case, tiled matrix multiplication is also considered in addition to broadcast and shuffle. The enhanced cost model takes as input the distribution of the tiles across the cluster nodes and performs a pilot run over the tile dimensions in order to compute the cost of each matrix multiplication algorithm. The execution plan corresponding to the minimum cost

solution is determined with a heuristic algorithm that assigns pairs of relevant tiles greedily to the computing nodes.

6.6.2 Compressed Sparse Linear Algebra

Matrix compression is implemented in SystemML (Elgohary *et al.*, 2016) in order to reduce the size of the tiled matrices that have to be transferred among the processing nodes. Compression is applied to every partition of the matrix independently only if all the operations the matrix is part of can be performed on the compressed matrix. The set of operations is limited to element-wise operations and matrix-vector multiplication, and does not include the general matrix multiplication. Moreover, only the input matrices are compressed — not the intermediates. Matrix compression is a combination between lossless compression techniques such as dictionary coding, run-length encoding, and offset-list encoding, and sparse matrix formats such as COO, CSR, and CSC. The compression method is applied independently to a single column or a group of correlated columns. The optimal method is selected based on the format and sparsity of the data (Sommer *et al.*, 2019). An alternative row oriented compression scheme derived from the Lempel-Ziv-Welch (LZW) algorithm is introduced in Li *et al.* (2019a), which has a higher compression ratio for wide short matrices.

6.7 Data Cube Processing

The data cube processing algorithms proposed in the literature can be classified into four main categories based on the format they use to compute and store the data cube. Relational-OLAP (ROLAP) algorithms use relational operators and standard SQL to compute the data cube, and materialized views for storage. Multidimensional-OLAP (MOLAP) methods use multidimensional arrays for storage and access to the aggregates. Graph-based algorithms employ specialized tree-like graph data structures. Finally, integrated algorithms compute the data cube while compressing the aggregates at the same time. Out of these four classes, ROLAP algorithms are the most common because of their direct application of relational processing — which allows for their immediate

integration with a standard database. Morfonios *et al.* (2007) provide a comprehensive survey of the ROLAP algorithms. Several of these algorithms (Wang *et al.*, 2013b; Nandi *et al.*, 2012; Milo and Altshuler, 2016) are extended to the parallel/distributed Map-Reduce framework by maximizing the degree of parallelism used for the computation of overlapping cuboids while minimizing the amount of transferred data.

Given our focus on multidimensional arrays, we present here only the concepts behind the MOLAP algorithms. MOLAP stores the base data from which the data cube is computed — as well as the data cube — as multidimensional arrays, where the measure values are determined by the position of the cell in the dimension space. The cube construction algorithms generate smaller dimensionality arrays from the base array by slicing along all the possible combinations of dimensions and aggregating the measure values along the eliminated dimension(s). A naive implementation of this approach considers each of the exponential number of dimension combinations — or cuboids — separately and always starts from the base data. Optimized algorithms overlap the computation of multiple cuboids in a single pass over the base data and reuse the higher dimensionality cuboids in the computation of the lower dimensionality ones. The Multi-Way Array algorithm (Zhao *et al.*, 1997) works on a base data array regularly chunked and generates all the cuboids that make the data cube in a single scan over the chunks of the base array. This requires simultaneously loading a significant number of the base array chunks in memory. The Multi-Way Array algorithm introduces several optimizations to reduce the amount of required memory. These optimizations target the order in which to scan the base array chunks and the identification of the most suitable cuboid from which to derive a lower dimensional cuboid — multiple choices are possible since the cuboids form a lattice. The MM-Cubing algorithm (Shao *et al.*, 2004) reduces the memory usage of Multi-Way Array by identifying and computing only the high density cuboid cells — the iceberg cells. Thus, MM-Cubing is optimized for skewed high-dimensional data.

6.8 Summary

- The traditional architecture for multidimensional array processing is a shared-nothing cluster consisting of a coordinator and multiple worker nodes. The chunks of the array are distributed across the nodes and processed concurrently. Optimizations are targeted at minimizing the network traffic among the worker nodes.
- The array processing paradigms have a two-level functional programming structure. The top level is an execution framework that implements a determined workflow consisting of higher level functions — or functionals — that take actual processing operations — or functions — as arguments. The bottom level consists of an application programming interface (API) for implementing concrete array operations as UDFs. Functionals execute operations by invoking the UDF arguments using the fixed API. Map-Reduce and GLA are the most common functional array processing paradigms.
- The implementation of array algebra operators is characterized by handling the neighboring chunks resulted from the range-based partitioning along the array dimensions. The array similarity join operator, which generalizes shape-based neighborhood relationships, is the most illustrative example of this type of processing.
- In-situ processing is an important requirement for scientific array data because of the extensive use of specialized data formats and the breadth of the corresponding processing libraries. As illustrated by ChronosDB, the challenge of in-situ processing consists in efficiently composing primitive functions into complex workflows that implement general array operations.
- It is recommended to implement tensor operations as wrappers over linear algebra functions from highly optimized libraries. In the case of matrix multiplication, this is best achieved with tiled or block chunking. The efficient execution of linear algebra expressions containing chains of matrix multiplications requires the rechunking of intermediate results — which becomes the main objective to optimize. The TRA system achieves this goal with a rechunking optimizer and 3D replication matrix multiplication.

7

Multidimensional Array Systems

A recent survey by Baumann *et al.* (2021) aggregates the most important array technologies and categorizes their implementation in real systems. However, its reach is limited since it does not include tensor and data cube systems. In this section, we start from the categorization introduced in Baumann *et al.* (2021) and extend it with the missing parts. We present full-stack array databases, raster extensions to relational databases, tensor processing systems, and OLAP data cubes. We focus our attention on functional systems that are available for use, are under consistent development, and provide reasonable documentation to understand how they implement the defining array concepts. This excludes from discussion unmaintained prototypes and undocumented systems without open-source code — which is the case for most of the Map-Reduce prototypes and the distributed tensor processing systems.

7.1 Array Databases

Array database systems are characterized by an array algebra and query language in which the array operations are declaratively specified, an execution engine that implements the algebra operators and an eventual query optimizer that selects among multiple operator implementations,

and a chunk-based storage manager. Additionally, concurrent multi-user operation with transactional support and an access control mechanism provide the complete functionality of a relational database. In this section, we present four systems that implement these requirements at different levels of completeness starting from scratch, while in the subsequent section we introduce alternatives that start from a full-fledge relational database and enhance it with multidimensional array support.

RasDaMan. RasDaMan ("Raster Data Manager") (RasDaMan Development Team, 2022) is the pioneer array database, which is at version 10.0 as of this writing. RasDaMan is available both as open-source in a community edition as well as a more extensive commercial enterprise version with dedicated support. According to the developers, the source code in the two versions is identical. Queries can be submitted to RasDaMan both from a command line interface as well as through connection APIs from multiple programming languages, including `C++`, `Java`, `Python`, and `R`. Moreover, RasDaMan is integrated in a RESTful web server for geographical (geo) services.

RasDaMan supports dense multidimensional arrays of arbitrary size, dimension, and structure through the declarative query language RasQL (Section 4.4) paired with internal execution, storage, and query optimization. RasQL supports a wide range of array operations, including dimensional transformations, geometric mappings, clipping, scaling, concatenation, cross product, and grouped aggregations. Additionally, RasQL supports INSERT/DELETE/UPDATE modification operations on arrays and CREATE/DROP/ALTER operations on collections. Conceptually, arrays are defined as types consisting of a spatial domain for dimensions and a subtype for cells. An array type can be encapsulated into a set supertype that includes a specification for empty cells. This is how sparse arrays are declared in RasQL. Concrete instances of an array type are created as tuples of a collection having a set supertype. The collection is the equivalent of the table in relational databases. Physically, arrays are chunked using the arbitrary strategies presented in Section 5.2.1 and stored either as files in the operating system or BLOBS in a PostgreSQL database. In the former case, chunk metadata — which takes the form of a multidimensional index — is stored in an

embedded SQLite database, while in the latter, it is stored together with the data in PostgreSQL. RasDaMan also provides direct external array access to a variety of file formats through the GDAL library (The GDAL Development Team, 2022). Moreover, the arrays resulted from RasQL queries can be exported to different file formats for display in visualization and web applications.

The RasDaMan engine (Baumann *et al.*, 1998) includes all the components of a standard relational DBMS. The query parser transforms a RasQL query into an execution tree of array algebra operators. The query optimizer transforms the query tree into a more efficient execution plan based on algebraic query rewriting rules and chunk layout information. The goal is to access only the necessary chunks and find the optimal order in which to process the chunks. The execution engine has a materialized architecture in which the array operators are invoked through function calls that take as input chunks and produce output chunks. Since chunks are processed in a streaming fashion, this strategy minimizes memory usage. Moreover, chunks can be processed concurrently by separate threads — as long as there is no dependency among them. Extensive implementation details for chunk processing in RasDaMan are presented in Widmann and Baumann (1998). RasDaMan does not provide intra-query distributed processing of arrays chunked across multiple nodes — a RasQL query is executed entirely at a single node. Nonetheless, multiple RasQL queries can be processed concurrently across a federation of peer RasDaMan servers that fully replicate their array storage. However, this configuration has to be set up manually by an administrator.

Overall, RasDaMan is a complete database system for dense arrays — or rasters. It provides an extensive set of functional and user features, including a large variety of operations over raster data in different formats, programming interfaces, management utilities, and web access. The lack of optimal support for sparse arrays and distributed processing are some of the most important limitations in RasDaMan. Moreover, while linear algebra operations can be expressed as RasQL grouping aggregations, their implementation is not fully optimized.

SciDB. SciDB (Paradigm4, 2022a) is a parallel database with a shared-nothing architecture initiated as the technological solution to process the high-resolution sky images acquired by the Large Synoptic Survey Telescope (LSST). This direction has been completely abandoned by now as the focus of SciDB has become drug discovery and precision medicine. Currently, SciDB is the computing platform of the REVEAL suite of medical applications commercialized by the Paradigm4 company. Since workflow management and reproducibility are paramount in medicine, SciDB includes a complex multi-versioning control system with no in-place data updates. As of this writing, SciDB is available both as open-source in a community edition as well as a more extensive commercial enterprise version. Unlike RasDaMan, the enterprise edition includes considerably more features (Paradigm4, 2022b) and is at least two releases ahead — version 21.8 for enterprise compared to version 19.11 for community. Queries can be submitted to SciDB using the command line interface `iquery` client. The commercial REVEAL platform also provides `Python` and `R` connection APIs. Queries have to be expressed in the Array Functional Language (AFL), which consists of function compositions that allow for coding complex features. The declarative Array Query Language (AQL) (Section 4.6) is only under experimental development since its functionality is completely subsumed by AFL.

Sparse arrays are implemented as a special encoding of dense arrays in SciDB. While this allows for a single common processing interface, it also precludes certain optimizations specific to sparse arrays. The storage layer includes array decomposition, chunk overlapping, and uses chunk compression. Array decomposition consists in splitting an array with multiple values in a cell into separate arrays with a single value in every cell. This is a generalization of the column-stores ideas to arrays. Chunk overlapping consists in storing the same array cells in multiple chunks to increase the level of parallelism in execution. Immediate drawbacks of this include increased storage and more complex management. Several compression techniques are extended from column-stores, including null suppression, run-length encoding (RLE), subtraction from an average value, and delta encoding. They are applied on a chunk-by-chunk basis, with different chunks possibly compressed differently.

All the SciDB operators are implemented as UDFs. While some operators are built-in and provided with the system, the user is given the freedom to implement any other operators using the extension features provided by UDFs. Extensibility in SciDB follows the pattern introduced in PostgreSQL (The PostgreSQL Development Team, 2020). A user can add to the system: user-defined data types (UDT); scalar user-defined functions (UDF) taking arguments user-defined types and returning a single value; user-defined aggregates (UDA), which allow special aggregate computation for the newly defined types, expressed as a group of functions invoked according to a well-established pattern; and user-defined operators (UDO) taking arrays as arguments and producing an array as the result. The SciDB operators can be divided into several classes: structural, e.g., slice, subsample, reshape, concatenate, cross-product, join, etc.; value-based, e.g., filter, aggregate, apply, project, etc.; statistical, e.g., bernoulli, kendall, pearson, quantile, etc.; and linear algebra, e.g., gemm, gesvd, spgemm, etc. The linear algebra operators are wrappers over the optimized ScaLAPACK implementations (ScaLAPACK Development Team, 2022). Additionally, a large variety of statistical functions applied in biology and medicine are also available. However, many of these are included only in the commercial enterprise edition.

While both RasDaMan and SciDB implement the chunked array model, only SciDB supports parallel chunk processing across computing nodes. RasDaMan executes a query entirely at a single node. Thus, it has limited scalability. SciDB includes a larger variety of statistical and optimized linear algebra operators. However, RasDaMan supports a considerably larger number of array/raster formats. Data loading in SciDB is a serious bottleneck as it requires two steps. First, an array is ingested as a 1-D vector. Second, the vector is repartitioned into the corresponding multidimensional array. The RasDaMan community edition is feature complete compared to its enterprise version. This is not the case for SciDB, whose community edition is rather unmaintained.

Ophidia. Ophidia (CMCC Foundation, 2022b) is an open-source parallel framework for processing multidimensional arrays. It is targeted at scientific applications that process heterogeneous data and require inten-

sive analysis. Ophidia provides a native API written in `C`, exposed also as a `Python` interface (CMCC Foundation, 2022a). The API consists of standard array operators, including indexing, subsampling, rechunking, and aggregation, statistical primitives imported from various libraries such as the GNU Scientific Library (GSL), and import/export functions from/to various data formats, including FITS, NetCDF, JSON, and HTML. New operators can be added to the API as long as they follow a template implementation. The array operators can be invoked independently or combined in composite workflows defined as JSON objects. Similar to a query execution tree made of relational operators, a workflow specifies the dependencies and the arrays passed among operators. Additionally, workflows can include control flow primitives such as branches and loops. This brings workflows closer to an imperative interface instead of a declarative query language.

Ophidia splits the dimensions of an array into explicit and implicit. Only the explicit dimensions are used for chunking. The implicit dimensions impact the rendering of the cells inside a chunk. A separate chunk is created for every combination of the explicit dimensions' values, which are used as the key for node assignment and parallel processing. Since Ophidia does not support stencil operations, this chunking does not impact processing negatively. However, in order to reduce the number of chunks, the number of non-empty cells on the explicit dimensions has to be small. This implies that explicit dimensions are sparse, while implicit dimensions are dense. This insight is specific to data cubes, where is applied to minimize the number of materialized cuboids. As such, Ophidia can be viewed as a data cube system for scientific data.

TileDB. TileDB (TileDB, Inc., 2022a) is an open-source embedded array storage library with support for both dense and sparse arrays. TileDB's goal is to be the equivalent of `sqlite` for array databases. It has a native `C++` API exposed through a variety of other programming languages, including `Python`, `R`, `Java`, and `Go`. TileDB provides efficient array storage with zero-copy access in multiple formats — in file systems and cloud object stores. At the application level, TileDB is integrated with geospatial libraries such as GDAL, distributed computing frameworks such as Spark, and relational databases such as MariaDB.

A user can interact with TileDB only through its functional API (TileDB, Inc., 2022b). There is no support for function composition in declarative queries. Moreover, the only operation provided by the API is range selection — or subsampling — on dimensions. Although this functionality is quite limited, the update-optimized chunking (Section 5.2.5) implemented by TileDB is its defining feature. It allows for the efficient execution of time travel queries over a sequence of array versions as well as highly concurrent read/write array access. This type of processing is not transparently supported by any other system or library.

7.2 Relational Array Systems

In this section, we present two relational databases that provide specialized support for arrays and raster data through object-relational extensions such as user-defined data types (UDT) and user-defined functions (UDF). Both of these systems implement storage optimizations for dense arrays in the form of index suppression and chunking. They also have a deep integration with the GDAL library (The GDAL Development Team, 2022), which implements a large variety of functions on many raster file formats.

PostgreSQL arrays & PostGIS rasters. PostgreSQL (The PostgreSQL Development Team, 2020) supports variable-length dense multidimensional arrays as table attributes. The array type is a collection of elements having the same base type. Array attributes can be referred in SQL queries exactly as any other attributes. The main operations on arrays are indexing and a series of structural functions such as containment, append, and concatenation. None of the array algebra operators introduced in Section 4.6 are built-in. Function `unnest` is of particular relevance because it expands an array attribute into a relation with a tuple for every element in the array. The layout of the array on storage is row-major — without chunking. Consequently, the array functions take the complete array as an argument. Since functions are treated as black boxes by the optimizer, no optimizations are applied during query execution.

PostGIS (The PostGIS Development Team, 2022), the spatial PostgreSQL package, provides extensive support for rasters — or dense arrays — by integrating the GDAL library. The `raster2pgsql` data loader is at the core of PostGIS. It converts rasters from any of the formats supported by GDAL to the internal PostgreSQL representation and loads them as tuples into a table. Chunking can be applied during loading, case in which every chunk becomes a separate tuple. The type of the raster attribute is a special user-defined data type specified using the object-relational PostgreSQL extension mechanism. In order to take advantage of chunking in subsampling queries, a spatial index has to be built on the raster column after loading. The only solution to include raster attributes in SQL queries is through functions. PostGIS includes an extensive set of raster functions, which are wrappers over the corresponding GDAL functions. The arguments to the GDAL functions are passed as string expressions in the wrappers. Overall, the PostGIS approach is a SQL frontend for GDAL functions, where cross-function optimizations are not possible due to the PostgreSQL execution mechanism.

Oracle Spatial GeoRaster. Multidimensional arrays are supported in Oracle through the GeoRaster feature (Oracle, 2022b) available in the Oracle Spatial package. GeoRaster manages the storage and processing of both the array metadata — as an XML document — and the array cells — as a GeoRaster table consisting of a spatial extent attribute for the dimensions and a BLOB attribute for the values. A raster is chunked into regular chunks — or blocks — that are stored as the tuples of a GeoRaster table. Incomplete chunks are padded. The maximum size of a chunk is 4 GB, which can be compressed using the JPEG algorithm. Sparse arrays are supported through bitmap masks indicating the non-empty cells, which are associated with every chunk. In the case of a completely empty chunk, no BLOB is associated with the spatial extent. Rasters can be grouped into pyramids of different sizes and degrees of resolution through resampling and interpolation operations. A pyramid is treated as a composite object with the individual rasters identified by their resolution level. GeoRaster provides an extended set of array algebra operators, including polygon subsampling, stencil-based

interpolation, and pyramid construction. These operators are formalized in a raster algebra language that is an extension to Oracle PL/SQL. This combination allows the specification of raster analyses as closed algebraic expressions of raster operators. The operators are exposed both as functions in PL/SQL queries and directly through their native `Java` API. Some of the operators have parallel chunk- and cell-based implementations. GeoRaster is fully integrated with the GDAL library and provides concurrent batch loading and exporting to the supported raster file formats. Overall, GeoRaster is a complete solution for raster processing inside a relational database using object-relational extensions — with their corresponding advantages and limitations.

7.3 Tensor Systems

The BLAS operations (Wikipedia, 2020) introduced in Section 3.3 are widely implemented in linear algebra libraries optimized for various computing architectures. Examples of such libraries include the Intel Math Kernel Library (MKL) (Wikipedia, 2022b) optimized for Intel CPUs, the cuBLAS library (NVIDIA, 2022) optimized for NVIDIA GPUs, and the ScaLAPACK library (ScaLAPACK Development Team, 2022) for parallel distributed memory architectures. These libraries contain a large variety of operations, making their complete reimplementation virtually impossible. They also provide bindings from many programming languages, including `C/C++`, `Python`, `R`, and `Julia`. Finally, the linear algebra kernels provided by these libraries are highly optimized through an extensive development cycle. Given the significant amount of effort necessary to replicate the functionality and performance of these libraries, the reasonable approach is to integrate the existing kernels in higher-level systems. This is exactly the approach taken by data analytics systems such as MADlib and SystemML, and deep learning frameworks such as PyTorch, TensorFlow, and Apache MXNet. While all these systems include tensors in their API, the underlying implementation is inherited from a tensor library. In the following, we discuss the integration in more detail and present the NumPy array library, which has become the standard API for tensor operations.

BLAS library wrappers. The common approach to integrate BLAS library functions into a tensor processing system is to encapsulate them into a wrapper. The wrapper performs two tasks — translation between the data representations corresponding to tensor processing and the BLAS library, and function invocation. First, the tensor operands are mapped into the BLAS library data structures. Then, the BLAS function is executed. Finally, the result is mapped back to the data structures in the tensor system. For this process to be efficient, the overhead of data translation has to be minimized — which can be quite challenging. This is the reason why dual solutions consisting of a wrapper and a limited reimplementation are proposed. The wrapper provides generality by supporting the invocation of any function from the BLAS library while the reimplementation avoids data translation, which can result in better performance. This approach is taken by the MADlib (MADlib Development Team, 2022) and RMA (Dolmatova *et al.*, 2020) libraries for in-database analytics. A subset of the linear algebra operations are implemented both as standalone UDFs — in MADlib — or a sequence of MonetDB low-level operators — in RMA — as well as wrappers to functions from the Eigen library (Eigen Development Team, 2010) — in MADlib — or Intel MKL — in RMA. AIDA (D'silva *et al.*, 2018), which integrates MonetDB and NumPy (NumPy Development Team, 2022), exploits the use of `C arrays` as internal data structures in both systems. This allows for sharing the same memory space and passing pointers to arrays as function arguments — which eliminates data translation.

NumPy, `xarray`, and Dask. NumPy (Harris *et al.*, 2020) is the primary array programming library in the `Python` programming language. What makes NumPy so extensively used are the breadth and depth of its API. The API includes a tremendous variety of array processing functions, ranging from multiple types of indexing, value-based selection, vectorization, broadcasting, and reductions to reshaping, concatenating, padding, searching, sorting, and counting on arrays. Additionally, NumPy implements an extensive set of linear algebra and statistical operations — CPU-accelerated by the OpenBLAS and Intel MKL libraries. Moreover, NumPy can read and write arrays from/to different file formats. All this functionality is integrated into an intuitive API that

closely mimics a mathematical formalism. The NumPy API is adopted by other specialized array libraries, including PyData/Sparse for sparse arrays, CuPy for GPU-optimized arrays, Dask for distributed arrays, and xarray for labeled arrays. Moreover, the API is becoming a standard for adding hardware acceleration to other libraries — including TensorFlow (TensorFlow Development Team, 2022) and PyTorch (PyTorch Development Team, 2022). To facilitate interoperability among libraries, NumPy provides protocols that allow these specialized arrays to be passed as arguments to NumPy functions. In turn, NumPy dispatches the operations to the corresponding library based on the type of the arguments. This allows programmers to port their code across platforms with minimal modifications.

NumPy arrays are the data structure at the core of the library. A NumPy array consists of a pointer to a contiguous memory region and associated metadata to interpret the data stored there. The metadata include the shape, the cell data type, and the strides. They correspond to the main components of a multidimensional array. The shape defines the dimensions of the array while the cell data type defines the array attributes. The strides specify the physical layout of the array in memory as the number of bytes at which the next cell on every dimension is located. The strides depend on the cell data type size. Since the stride on a dimension is constant, only row- and column-major chunking — and their multidimensional extensions — are supported. Overall, NumPy arrays are a standard in-memory multidimensional array implementation.

The `xarray` library (Hoyer and Hamman, 2017) decorates NumPy arrays with labels in the form of dimensions, coordinates, and attributes. These labels are explicitly assigned to the corresponding elements in the array definition. While dimensions and attributes have a direct correspondent, coordinates are aliases for values on the range of a dimension. The labels can be integrated in NumPy expressions by replacing the positional indexing with a more intuitive named notation. This notation is a direct extension of the `pandas` library relational API to multidimensional arrays. While a `pandas.DataFrame` is a collection of vectors — or series — aligned based on their position, an `xarray.Dataset` is a collection of NumPy arrays aligned along their shared dimensions.

The `xarray.Dataset` allows for the creation of multidimensional arrays with cells having a composite data type by grouping NumPy arrays that share all their dimensions. This corresponds to slicing the composite array into a group of identical arrays, with one basic array for every attribute. At the implementation level, `xarray` builds a map from labels to NumPy arrays and transparently converts the named notation to NumPy API function calls (NumPy Development Team, 2022). Thus, it extensively reuses the NumPy functionality. At the I/O level, the similarity between the `xarray.Dataset` and the NetCDF file format allows for the direct memory mapping of a NetCDF file to an in-memory `xarray.Dataset` object. Overall, the `xarray.Dataset` follows closely our definition of multidimensional arrays from Section 2.1. The labels allow the creation of named array algebra queries that go beyond the indexed expressions from NumPy.

A Dask Array (The Dask Development Team, 2022) consists of multiple NumPy arrays arranged into a grid. The individual NumPy arrays represent the physical chunks of the complete Dask array. The Dask library manages the storage, location, and processing of the chunks. It supports streamed execution on CPU and GPU — in which chunks are processed one at a time in order to reduce memory usage — and distributed processing across multiple computing nodes. The Dask library API inherits the NumPy API while adapting the implementation to chunk-based processing, which can be performed in parallel. While much of the NumPy API is implemented, the linear algebra functions are the most notable omission to date. Additionally, the API includes functions to rechunk and reshape an array. The chunks are identified by their index combination and are organized into a Dask graph for processing. Moreover, chunks can have overlapped boundaries. Overall, Dask arrays are a scalable extension of NumPy arrays with an almost identical interface. They come the closest to a Python array database.

7.4 Data Cube Systems

A comprehensive list of commercial and open source OLAP databases — as well as their detailed comparison — is available online (Wikipedia, 2022a). Three of these systems — Oracle Essbase, IBM Cognos, and

Apache Kylin — provide comprehensive data cube support. Although these three systems are classified as MOLAP, the input data are stored either in flat files or relational tables. Since only the data cube aggregates are materialized as multidimensional chunked arrays, the more appropriate category for these systems is Hybrid OLAP — or HOLAP. As a comparison, Pentaho Mondrian (Pentaho, 2022) stores both the input data and the data cube in relational tables. Thus, it is purely ROLAP. While SQL is heavily used to build the data cube over relational data, access to the cube cells and cuboids is expressed in the MultiDimensional eXpressions (MDX) query language (Whitehorn *et al.*, 2005) developed at Microsoft and adopted extensively in OLAP servers. With the exception of Apache Kylin and a few other open source systems, all the other OLAP servers support MDX as their user API. For execution, MDX is mapped either to SQL or positional access to the data cube cells. In the following, we present more details on the three data cube systems introduced above.

Oracle Essbase. The Oracle Essbase multidimensional database (Oracle, 2022a) allows the user to specify dense and sparse dimensions when defining a data cube. The sparse dimensions are used to chunk the data cube. A dense chunk — or block — is created for every combination of the sparse dimensions for which there exists at least one non-empty cell. A good partitioning of the dimensions into dense and sparse groups results in a small number of chunks with as few empty cells as possible. The cells of a chunk are linearized in the order in which the dimensions are defined. A chunk is fully expanded in memory when processed while being compressed when materialized on secondary storage. Accessing a data cube cell is a two-stage process. First, the chunk containing the cell is identified based on the values of the sparse dimensions. This is efficiently achieved with a multidimensional index over the sparse dimensions. Second, the cell is directly accessed in the chunk — which is completely loaded in memory — by indexing along the dense dimensions. Only point access to the data cube cells is possible — no range access is supported. The computation of the data cube is specified in a scripting language — or rules file — that defines the measures corresponding to a data cell and what source data they are derived from. Supported data

sources include flat files, relational tables, and spreadsheets. By default, data cube computation is serial, thus, inefficient.

IBM Cognos. The IBM Cognos Dynamic Cubes (Beryoza *et al.*, 2015) are an in-memory middleware that builds and provides efficient access to a data cube using the MDX language. The data cube is defined over a star schema relational database following the referential integrity constraints embedded in such a schema. This requires only the identification of the fact table — the dimensions are derived from the referential integrity constraints. Cognos Data Cubes maximize the portion of the data cube that is cached in memory based on the cube definition and the user access patterns — which are closely logged and monitored. Cells that are not cached are automatically computed on-demand by running queries against the underlying database. Upon startup, the data cube is built bottom-up until the memory capacity is exhausted. Given that the data cube is memory resident and is not materialized to secondary storage, the access is based on the dimension values used as the key in a hash table. This allows for the pruning of empty cells, which impacts positively the memory utilization. Moreover, dimension based access simplifies cache management. Since data cube building is mapped as SQL queries, parallel execution is delegated to the multi-query processing capabilities of the underlying database.

Apache Kylin. Kylin (Kylin Development Team, 2022) is an open-source project that supports the distributed building and querying of data cubes. As of version 4.0, a cube is built over relational data extracted from a database — or flat files — using the Spark framework. The cuboids making the data cube are stored as separate columnar Parquet files, which can be queried independently using Spark SQL. The first step in the workflow requires the definition of the data cube model as a star schema consisting of a fact table and several lookup tables connected by key/foreign-key relationships. The dimensions and measures of the data cube are selected from the attributes in these tables. In order to reduce the number of computed — and stored — cuboids, dimensions can be split into aggregation groups and classified as mandatory or joint. This is equivalent to partitioning the overall

dimension space into smaller sub-spaces and selecting only a subset for evaluation. Kylin also provides a cube planner advisor that recommends the most relevant cuboids to build based on their estimated size and query frequency. Once the data cube model is complete, the cuboids are built concurrently in an optimized Spark application that shares computation among connected cuboids and reuses the already computed cuboids whenever possible. By storing every cuboid as a separate Parquet file, Kylin implements a form of chunking that allows direct access to the relevant dimensions. This is done in a relational SQL syntax with SparkSQL.

7.5 Summary

- RasDaMan and SciDB are the only two full-stack array databases that have all the components of a database — from an array query language to chunk-based storage. While RasDaMan is optimized for single machine, SciDB has a scalable distributed architecture.
- TileDB aims to become an embedded array database similar to `sqlite`. The update-optimized storage layer and support for time-travel queries based on multi-versioned updates are its distinguishing features. However, TileDB uses the NumPy API to specify operations on arrays instead of a declarative query language.
- Rasters are integrated in relational databases such as PostgreSQL and Oracle through storage optimizations and external function invocation to generic libraries such as GDAL. The query expressiveness of this approach is limited by the SQL support for function composition. Since sparse arrays map to relations naturally, multidimensional indexes are sufficient to achieve reasonable efficiency.
- The NumPy API is the accepted notation to abstractly express linear algebra operations over tensors. It provides the desired separation between the mathematical specification and the target architecture — be it (multi-) CPU, GPU, or a distributed cluster.
- Most of the OLAP data cube systems fall under the category of HOLAP because they use the multidimensional representation only for the cube — which is built over data extracted from

relational tables. In order to minimize the cube building time and storage space, the dimensions are split based on density, resulting in many low-dimensional cuboids instead of a single high-dimensional cube.

8

Future Directions

In this work, we survey the research on multidimensional array data management from a database perspective. Unlike previous surveys that are limited to raster processing in the context of scientific data (Rusu and Cheng, 2013; Baumann *et al.*, 2021; Zalipynis, 2021), our perspective on multidimensional array data management considers all types of arrays — rasters, data cubes, and tensors. We identify and analyze the most important research ideas on arrays proposed over time. We cover all data management aspects, from array algebras and query languages to storage strategies, execution techniques, and operator implementations. Moreover, we discuss which research ideas are adopted in real systems and how are they integrated in complete data processing pipelines. We also compare the differences between arrays and the unordered set-based relational data model at every step in the presentation. Up to this point, the survey summarizes concisely the most relevant work on multidimensional array data management and organizes the material to provide an accurate perspective on the state-of-the-art in array processing. In this section, we provide several suggestions for future work in the field following the organization of this survey.

Array algebras and query languages. Although no array algebra and query language have gained general acceptance to date, there are proposals that have become the de-facto standard for every type of array. RasQL (Misev and Baumann, 2014) lies at the foundation of the SQL/MDA standard for querying raster data through SQL. While SQL/MDA follows the RasDaMan approach of integrating rasters as table attributes, it does not include an exhaustive set of raster operations. The GDAL library (The GDAL Development Team, 2022) represents a good starting point in this direction. However, the effective integration of GDAL into SQL/MDA requires further research. The MDX query language (Whitehorn *et al.*, 2005) — which provides direct access to data cube cells and cuboids — is used extensively in OLAP servers. Its formalization as a standard for data cube navigation is the natural next step. NumPy (Harris *et al.*, 2020) defines a complete API for tensor operations, including linear algebra and many other operators. Nonetheless, its array data structures are primitive. They do not have support for chunking and partitioning. In order to scale the NumPy API to large distributed infrastructures, these issues have to be carefully addressed. The Tensor Relational Algebra (Yuan *et al.*, 2021), which models tensors concisely as binary relations from dimensions to valued tiles, is a first step in that direction. However, its applicability is limited due to the reduced set of supported operations. Moreover, the optimization space is constrained by two communication patterns.

While the existing solutions are targeted at a specific array type — raster, data cube, or tensor — and address the integration with the relational data model, a fundamental question that remains unanswered is how to design a generic data model and query language that encompass these specializations? Furthermore, how to include other data structures — such as polygons, hierarchies, and graphs — that can be represented as multidimensional arrays? One could argue that the relational data model already satisfies these requirements. The major problem is that the corresponding relational expressions are not practical and do not have efficient implementations. Hybrid data models and polystores (Duggan *et al.*, 2015a; Alotaibi *et al.*, 2019; Koutsoukos *et al.*, 2021) are recent alternatives that address these shortcomings with a layered abstraction that maps to any of the underlying data structures. Further work is needed in order to assess their generality and evaluate their performance.

Array storage. Chunking is the defining characteristic of array data management. As such, it has received extensive attention in the database literature, as illustrated in Section 5. However, the vast majority of the work focuses on optimally chunking persistent arrays for a single class of operations — most commonly, dimension subsampling. Sequences of linear algebra operations, which are specific to machine learning workloads, make chunking considerably more complicated because they generate intermediate tensors. In this case, chunking becomes a dynamic problem that has to be solved independently for every expression at runtime. The optimal chunking depends both on the sequence of operations as well as the input tensors — including their dimensions and cell density. Since this problem is related to database query optimization — albeit more complicated because it requires the simultaneous identification of the operators and the chunking — the initial solutions proposed in the literature follow a relational database approach. They are based on similar data synopses and cost models. However, these do not capture well the intrinsic dimensionality of tensors and the characteristics of tensor operations. Consequently, specialized solutions tailored at multidimensional tensors have to be devised. These include novel dimension-aware data statistics and chunk-based cost models.

The use of heterogeneous architectures consisting of a diverse set of computing devices — such as CPU, GPU, TPU, and FPGA — has become more common — especially in raster processing and machine learning applications. Since memory capacity and hierarchy vary significantly from device to device, the optimal chunking has to be adapted to every configuration. This includes the chunking of the base arrays as well as that of intermediate arrays. Similar to relational databases — where the format of a table is abstracted out — the chunking of an array has to be separated from its definition. This separation allows the user to focus on the logical array operations while the system can automate and optimize the physical implementation details. For something like this to be feasible, the user has to at least specify the target architecture. The system has to provide optimized implementations as well as rechunking routines for every architecture. The rechunking includes both packing and unpacking functions for converting between different array formats. While this entire process can be automated, we also anticipate the need

for “chunking advisors” that enhance the existing tuning advisors from relational databases in order to keep the user in the loop.

Array processing. The development of novel array processing techniques is largely driven by domain-specific applications. This has started with raster images in different science domains and has continued with materialized data cubes in business analytics and tensor processing in machine learning and AI. Given the extensive use of multidimensional arrays as a representation formalism, we see this trend to continue. However, going forward, the tools developed by domain experts have to be better integrated with data management capabilities in order to reduce the amount of work replication. The common approach of adding functionality to a data management system is not sustainable, as illustrated by the development of in-situ processing techniques. To address this issue, foundational work on designing generic interfaces that allow the conceptual composition of various libraries is further needed. The NumPy API (NumPy Development Team, 2022) is a good initial step in this direction.

Array systems. None of the existing array data management systems fully supports all three types of arrays — rasters, data cubes, and tensors. Systems are optimized for a specific array type and may provide some functionality for the others. While this approach is perfectly motivated by manageable complexity and better performance, the design of a generic multidimensional array data management system remains an intriguing topic to explore.

Acknowledgments

We want to thank Professor Joe Hellerstein, Editor-in-Chief of *Foundations and Trends in Databases*, and Mike Casey and Alet Heezemans from *Now Publishers* for their unlimited patience and continuous encouragement to finish this work.

We also want to thank the anonymous reviewers for their insightful comments that improved the quality of this work significantly.

This work is supported by NSF award number 2008815 and by a U.S. Department of Energy Early Career Award (DOE Career).

References

Aberger, C., A. Lamb, K. Olukotun, and C. Re. (2018). "LevelHeaded: A Unified Engine for Business Intelligence and Linear Algebra Querying". In: *Proceedings of 2018 IEEE ICDE International Conference on Data Engineering*. 449–460.

Agarwal, R. C., S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. (1995). "A Three-dimensional Approach to Parallel Matrix Multiplication". *IBM Journal of Research and Development*. 39(5): 575–582.

Agrawal, R., A. Gupta, and S. Sarawagi. (1997). "Modeling Multidimensional Databases". In: *Proceedings of 1997 IEEE ICDE International Conference on Data Engineering*. 232–243.

Alotaibi, R., D. Bursztyn, A. Deutsch, I. Manolescu, and S. Zampetakis. (2019). "Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue". In: *Proceedings of 2019 ACM SIGMOD International Conference on Management of Data*. 1660–1677.

Ballegooij, A. van, R. Cornacchia, A. P. de Vries, and M. Kersten. (2005). "Distribution Rules for Array Database Queries". In: *Proceedings of 2005 DEXA International Conference on Database and Expert Systems Applications*. 55–64.

Ballegooij, A. R. van. (2004). "RAM: A Multidimensional Array DBMS". In: *Proceedings of 2004 EDBT Extended Database Technology Workshops*. 154–165.

Barceló, P., N. Higuera, J. Pérez, and B. Subercaseaux. (2019). "Expressiveness of Matrix and Tensor Query Languages in Terms of ML Operators". In: *Proceedings of 2019 DEEM International Workshop on Data Management for End-to-End Machine Learning*.

Baumann, P., A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. (1998). "The Multidimensional Database System RasDaMan". In: *Proceedings of 1998 ACM SIGMOD International Conference on Management of Data*. 575–577.

Baumann, P. (1994). "On the Management of Multi-Dimensional Discrete Data". *VLDB Journal (VLDBJ)*. 4(3): 401–444.

Baumann, P. (1999). "A Database Array Algebra for Spatio-Temporal Data and Beyond". In: *Proceedings of 1999 NGITS International Workshop on Next Generation Information Technologies and Systems*. 76–93.

Baumann, P. and S. Holsten. (2011). "A Comparative Analysis of Array Models for Databases". In: *Proceedings of 2011 FGIT-DTA/BSBT*. 80–89.

Baumann, P. and V. Merticariu. (2015). "On the Efficient Evaluation of Array Joins". In: *Proceedings of 2015 IEEE BIG DATA International Conference on Big Data*. 2046–2055.

Baumann, P., D. Misev, V. Merticariu, and B. P. Huu. (2021). "Array Databases: Concepts, Standards, Implementations". *Journal of Big Data*. 8(1).

Beryoza, D., M. Campbell, C. Cardorelle, T. Creasey, D. Cushing, V. D. Silva, S. David, A. Hagleitner, I. Henderson, D. Howell, I. Kozine, P. Prieto, P. Thompson, J. Vazquez, and Y. Zhang. (2015). *IBM Cognos Dynamic Cubes*. IBM Redbooks.

Blanas, S., J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. (2010). "A Comparison of Join Algorithms for Log Processing in MapReduce". In: *Proceedings of 2010 ACM SIGMOD International Conference on Management of Data*. 975–986.

Blanas, S., K. Wu, S. Byna, B. Dong, and A. Shoshani. (2014). "Parallel Data Analysis Directly on Scientific File Formats". In: *Proceedings of 2014 ACM SIGMOD International Conference on Management of Data*. 385–396.

Boehm, M., M. W. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. R. Reiss, P. Sen, A. C. Surve, and S. Tatikonda. (2016). "SystemML: Declarative Machine Learning on Spark". *PVLDB*. 9(13): 1425–1436.

Boehm, M., B. Reinwald, D. Hutchison, P. Sen, A. V. Evfimievski, and N. Pansare. (2018). "On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML". *PVLDB*. 11(12): 1755–1768.

Boncz, P., M. Zukowski, and N. Nes. (2005). "MonetDB/X100: Hyper-Pipelining Query Execution". In: *Proceedings of 2005 CIDR Conference on Innovative Database Research*. 225–237.

Brijder, R., F. Geerts, J. V. D. Bussche, and T. Weerwag. (2019). "On the Expressive Power of Query Languages for Matrices". *ACM Transactions on Database Systems (TODS)*. 44(4).

Brown, T. B., B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. (2020). "Language Models are Few-Shot Learners". *CoRR*. arXiv:2005.14165.

Buck, J., N. Watkins, G. Levin, A. Crume, K. Ioannidou, S. Brandt, C. Maltzahn, and N. Polyzotis. (2012). "SIDR: Efficient Structure-Aware Intelligent Data Routing in SciHadoop". *Tech. rep.* No. UCSC-TR-SOE-12-08. UC Santa Cruz.

Buck, J. B., N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt. (2011). "SciHadoop: Array-based Query Processing in Hadoop". In: *Proceedings of 2011 SC International Conference for High Performance Computing, Networking, Storage and Analysis*. 66:1–66:11.

Cabibbo, L. and R. Torlone. (1998). "A Logical Approach to Multi-dimensional Databases". In: *Proceedings of 1998 EDBT Extended Database Technology Workshops*. 183–197.

Cao, B. and A. Badia. (2007). "SQL Query Optimization Through Nested Relational Algebra". *ACM Transactions on Database Systems (TODS)*. 32(3).

Chang, C., A. Acharya, A. Sussman, and J. H. Saltz. (1998). "T2: A Customizable Parallel Database for Multi-Dimensional Data". *SIGMOD Rec.* 27(1): 58–66.

Chang, C., B. Moon, A. Acharya, C. Shock, A. Sussman, and J. H. Saltz. (1997). "Titan: A High-Performance Remote Sensing Database". In: *Proceedings of 1997 IEEE ICDE International Conference on Data Engineering*. 375–384.

Chaudhuri, S. and U. Dayal. (1997). "An Overview of Data Warehousing and OLAP Technology". *ACM SIGMOD Record*. 26(1): 65–74.

Chaudhuri, S., U. Dayal, and V. Narasayya. (2011). "An Overview of Business Intelligence Technology". *Commun. ACM*. 54(8): 88–98.

Cheng, Y., C. Qin, and F. Rusu. (2012). "GLADE: Big Data Analytics Made Easy". In: *Proceedings of 2012 ACM SIGMOD International Conference on Management of Data*. 697–700.

Cheng, Y. and F. Rusu. (2014). "Formal Representation of the SS-DB Benchmark and Experimental Evaluation in EXTASCID". *Distributed and Parallel Databases*.

Choi, D., C.-S. Park, and Y. D. Chung. (2019). "Progressive Top-k Subarray Query Processing in Array Databases". *PVLDB*. 12(9): 989–1001.

Choi, J., J. J. Dongarra, R. Pozo, and D. W. Walker. (1992). "ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers". In: *Proceedings of 1992 Symposium on the Frontiers of Massively Parallel Computation*. 120–127.

CMCC Foundation. (2022a). "Ophidia Big Data Code Repository". URL: https://github.com/OphidiaBigData.

CMCC Foundation. (2022b). "Ophidia Project". URL: https://ophidia.cmcc.it/.

Codd, E. (1970). "A Relational Model for Large Shared Data Banks". *Comm. ACM*. 13(6): 377–387.

Cornacchia, R., S. Héman, M. Zukowski, A. P. de Vries, and P. Boncz. (2008). "Flexible and Efficient IR using Array Databases". *VLDB Journal (VLDBJ)*. 17: 151–168.

Cudre-Mauroux, P., H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, and S. Zdonik. (2009). "A Demonstration of SciDB: A Science-Oriented DBMS". *PVLDB*. 2(2): 1534–1537.

Cudre-Mauroux, P., H. Kimura, K.-T. Lim, J. Rogers, S. Madden, M. Stonebraker, S. B. Zdonik, and P. G. Brown. (2010). "SS-DB: A Standard Science DBMS Benchmark". URL: http://www.xldb.org/science-benchmark/.

D'silva, J. V., F. De Moor, and B. Kemme. (2018). "AIDA: Abstraction for Advanced in-Database Analytics". *PVLDB*. 11(11): 1400–1413.

Datta, K., M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. (2008). "Stencil Computation Optimization and Auto-tuning on State-of-the-art Multicore Architectures". In: *Proceedings of 2008 SC International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

Dean, J. and S. Ghemawat. (2008). "MapReduce: Simplified Data Processing on Large Clusters". *Commun. ACM*. 51(1): 107–113.

Dewitt, D. J., S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. (1990). "The Gamma Database Machine Project". *IEEE Transactions on Knowledge and Data Engineering (TKDE)*. 2(1): 44–62.

DeWitt, D. J. and J. Gray. (1991). "Parallel Database Systems: The Future of Database Processing or a Passing Fad?" *SIGMOD Rec.* 19.

Dolmatova, O., N. Augsten, and M. H. Böhlen. (2020). "A Relational Matrix Algebra and Its Implementation in a Column Store". In: *Proceedings of 2020 ACM SIGMOD International Conference on Management of Data*. 2573–2587.

Dong, B., K. Wu, S. Byna, J. Liu, W. Zhao, and F. Rusu. (2017). "ArrayUDF: User-Defined Scientific Data Analysis on Arrays". In: *Proceedings of 2017 ACM HPDC International Symposium on High-Performance Parallel and Distributed Computing*.

Dongarra, J. and R. Schreiber. (1990). "Automatic Blocking of Nested Loops". *Tech. rep.* University of Tennessee.

Duggan, J., A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik. (2015a). "The BigDAWG Polystore System". *ACM SIGMOD Record*. 44(2): 11–16.

Duggan, J., O. Papaemmanouil, L. Battle, and M. Stonebraker. (2015b). "Skew-Aware Join Optimization for Array Databases". In: *Proceedings of 2015 ACM SIGMOD International Conference on Management of Data*. 123–135.

Duggan, J. and M. Stonebraker. (2014). "Incremental Elasticity for Array Databases". In: *Proceedings of 2014 ACM SIGMOD International Conference on Management of Data*. 409–420.

Eigen Development Team. (2010). "Eigen". URL: http://eigen.tuxfamily.org.

Elgohary, A., M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald. (2016). "Compressed Linear Algebra for Large-Scale Machine Learning". *PVLDB*. 9(12): 960–971.

Faloutsos, C. and P. Bhagwat. (1993). "Declustering Using Fractals". In: *Proceedings of 1993 International Conference on Parallel and Distributed Information Systems*. 18–25.

Furtado, P. and P. Baumann. (1999). "Storage of Multidimensional Arrays Based on Arbitrary Tiling". In: *Proceedings of 1999 IEEE ICDE International Conference on Data Engineering*. 480–489.

Gao, Z. J., S. Luo, L. L. Perez, and C. Jermaine. (2017). "The BUDS Language for Distributed Bayesian Machine Learning". In: *Proceedings of 2017 ACM SIGMOD International Conference on Management of Data*. 961–976.

Ge, T., D. Grabiner, and S. Zdonik. (2011). "Monte Carlo Query Processing of Uncertain Multidimensional Array Data". In: *Proceedings of 2011 IEEE ICDE International Conference on Data Engineering*. 936–947.

Ge, T. and S. Zdonik. (2010). "A*-Tree: A Structure for Storage and Modeling of Uncertain Multidimensional Arrays". *PVLDB*. 3(1): 964–974.

Goil, S. and A. N. Choudhary. (1997). "Sparse Data Storage Schemes for Multidimensional Data for OLAP and Data Mining". *Tech. rep.* No. CPDC-TR-9801-005. Northwestern University.

Goto, K. and R. van de Geijn. (2008). "Anatomy of High-performance Matrix Multiplication". *ACM Transactions on Mathematical Software (TOMS)*. 34(3).

Goto, K. and R. van de Geijn. (2009). "High-performance Implementation of the Level-3 BLAS". *ACM Transactions on Mathematical Software (TOMS)*. 35(1).

Gray, J., A. Bosworth, A. Layman, and H. Pirahesh. (1996). "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total". In: *Proceedings of 1996 IEEE ICDE International Conference on Data Engineering*. 152–159.

Gu, R., Y. Tang, C. Tian, H. Zhou, G. Li, X. Zheng, and Y. Huang. (2017). "Improving Execution Concurrency of Large-Scale Matrix Multiplication on Distributed Data-Parallel Platforms". *IEEE Transactions on Parallel and Distributed Systems (TPDS)*. 28(9): 2539–2552.

Guo, H. (2021). *What Are Tensors Exactly?* World Scientific.

Guttman, A. (1984). "R-trees: A Dynamic Index Structure for Spatial Searching". In: *Proceedings of 1984 ACM SIGMOD International Conference on Management of Data*. 47–57.

Gyssens, M. and L. V. Lakshmanan. (1997). "A Foundation for Multi-Dimensional Databases". In: *Proceedings of 1997 VLDB International Conference on Very Large Data Bases*. 106–115.

Hadoop Development Team. (2020). "Hadoop". URL: http://hadoop.apache.org/.

Han, D., Y.-M. Nam, J. Lee, K. Park, H. Kim, and M.-S. Kim. (2019). "DistME: A Fast and Elastic Distributed Matrix Computation Engine using GPUs". In: *Proceedings of 2019 ACM SIGMOD International Conference on Management of Data*. 759–774.

Harinarayan, V., A. Rajaraman, and J. D. Ullman. (1996). "Implementing Data Cubes Efficiently". In: *Proceedings of 1996 ACM SIGMOD International Conference on Management of Data*. 205–216.

Harris, C. R., K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gerard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. (2020). "Array Programming with NumPy". *Nature*. 585(7825): 357–362.

Hellerstein, J. M., C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. (2012). "The MADlib Analytics Library: Or MAD Skills, the SQL". *PVLDB*. 5(12): 1700–1711.

Hong, C., A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan. (2019). "Adaptive Sparse Tiling for Sparse Matrix Multiplication". In: *Proceedings of 2019 PPoPP Symposium on Principles and Practice of Parallel Programming*. 300–314.

Horlova, O., A. Kaitoua, and S. Ceri. (2020). "Array-based Data Management for Genomics". In: *Proceedings of 2020 IEEE ICDE International Conference on Data Engineering*. 109–120.

Howe, B. and D. Maier. (2004). "Algebraic Manipulation of Scientific Datasets". In: *Proceedings of 2004 VLDB International Conference on Very Large Data Bases*. 924–935.

Hoyer, S. and J. Hamman. (2017). "xarray: N-D labeled Arrays and Datasets in Python". *Journal of Open Research Software*. 5(1).

Huang, B., S. Babu, and J. Yang. (2013). "Cumulon: Optimizing Statistical Data Analysis in the Cloud". In: *Proceedings of 2013 ACM SIGMOD International Conference on Management of Data*. 1–12.

Hutchison, D., B. Howe, and D. Suciu. (2017). "LaraDB: A Minimalist Kernel for Linear and Relational Algebra Computation". In: *Proceedings of 2017 ACM SIGMOD BeyondMR Workshop on Algorithms and Systems for MapReduce and Beyond*.

Idreos, S., F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. (2012). "MonetDB: Two Decades of Research in Column-Oriented Database Architectures". *IEEE Data Eng. Bull.* 35(1): 40–45.

Irigoin, F. and R. Triolet. (1988). "Supernode Partitioning". In: *Proceedings of 1988 ACM POPL Symposium on Principles of Programming Languages*. 319–329.

Ivanova, M., M. L. Kersten, and S. Manegold. (2012). "Data Vaults: A Symbiosis between Database Technology and Scientific File Repositories". In: *Proceedings of 2012 SSDBM International Conference on Scientific and Statistical Database Management*. 485–494.

Jaeschke, G. and H. J. Schek. (1982). "Remarks on the Algebra of Non First Normal Form Relations". In: *Proceedings of 1982 PODS Symposium on Principles of Database Systems*. 124–138.

Jagadish, H. (1990). "Linear Clustering of Objects with Multiple Attributes". In: *Proceedings of 1990 ACM SIGMOD International Conference on Management of Data*. 332–342.

Jankov, D., B. Yuan, S. Luo, and C. Jermaine. (2021). "Distributed Numerical and Machine Learning Computations via Two-Phase Execution of Aggregated Join Trees". *PVLDB*. 14(7): 1228–1240.

Jordan, H., P. Thoman, J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch. (2012). "A Multi-objective Auto-tuning Framework for Parallel Codes". In: *Proceedings of 2012 SC International Conference for High Performance Computing, Networking, Storage and Analysis*. 10:1–10:12.

Kernert, D., W. Lehner, and F. Kohler. (2016). "Topology-aware Optimization of Big Sparse Matrices and Matrix Multiplications on Main-memory Systems". In: *Proceedings of 2016 IEEE ICDE International Conference on Data Engineering*. 823–834.

Kersten, M. L., Y. Zhang, M. Ivanova, and N. Nes. (2011). "SciQL, A Query Language for Science Applications". In: *Proceedings of 2011 AD EDBT/ICDT Array Databases Workshop*. 1–12.

Khamis, M. A., H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. (2018). "AC/DC: In-Database Learning Thunderstruck". In: *Proceedings of 2018 DEEM International Workshop on Data Management for End-to-End Machine Learning*.

Kim, J., A. Sukumaran-Rajam, V. Thumma, S. Krishnamoorthy, A. Panyala, L. Pouchet, A. Rountev, and P. Sadayappan. (2019). "A Code Generator for High-Performance Tensor Contractions on GPUs". In: *Proceedings of 2019 IEEE/ACM CGO International Symposium on Code Generation and Optimization*. 85–95.

Kim, M. (2014). *TensorDB and Tensor-Relational Model (TRM) for Efficient Tensor-Relational Operations, Ph.D. Thesis*. Arizona State University.

Kim, M. and K. S. Candan. (2011). "Approximate Tensor Decomposition within a Tensor-Relational Algebraic Framework". In: *Proceedings of 2011 ACM CIKM International Conference on Information and Knowledge Management*. 1737–1742.

Kim, M. and K. S. Candan. (2014). "Efficient Static and Dynamic In-Database Tensor Decompositions on Chunk-Based Array Stores". In: *Proceedings of 2014 ACM CIKM International Conference on Information and Knowledge Management*. 969–978.

Kim, S., B. Kim, and B. Moon. (2021). "Spangle: A Distributed In-Memory Processing System for Large-Scale Arrays". In: *Proceedings of 2021 IEEE ICDE International Conference on Data Engineering*. 1799–1810.

Kisuki, T., P. M. W. Knijnenburg, and M. F. P. O'Boyle. (2000). "Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation". In: *Proceedings of 2000 PACT International Conference on Parallel Architectures and Compilation Techniques*. 237–248.

Kolda, T. G. and B. W. Bader. (2009). "Tensor Decompositions and Applications". *SIAM Rev.* 51(3): 455–500.

Koutsoukos, D., S. Nakandala, K. Karanasos, K. Saur, G. Alonso, and M. Interlandi. (2021). "Tensors: An Abstraction for General Data Processing". *PVLDB*. 14(10): 1797–1804.

Kunft, A., A. Alexandrov, A. Katsifodimos, and V. Markl. (2016). "Bridging the Gap: Towards Optimization across Linear and Relational Algebra". In: *Proceedings of 2016 ACM SIGMOD BeyondMR Workshop on Algorithms and Systems for MapReduce and Beyond*.

Kylin Development Team. (2022). "Kylin". URL: https://kylin.apache.org/.

Laboratory for Foundations of Computer Science at the University of Edinburgh. (2008). "The Standard ML Language". URL: http://www.lfcs.inf.ed.ac.uk/software/ML/.

Laue, S., M. Mitterreiter, and J. Giesen. (2020). "A Simple and Efficient Tensor Calculus". In: *Proceedings of 2020 AAAI Conference on Artificial Intelligence*. 4527–4534.

Lerner, A. and D. Shasha. (2003). "AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments". In: *Proceedings of 2003 VLDB International Conference on Very Large Data Bases*. 345–356.

Leung, A., N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin. (2010). "A Mapping Path for Multi-GPGPU Accelerated Computers from a Portable High Level Programming Abstraction". In: *Proceedings of 2010 Workshop on General-Purpose Computation on Graphics Processing Units*. 51–61.

Li, F., L. Chen, Y. Zeng, A. Kumar, X. Wu, J. F. Naughton, and J. M. Patel. (2019a). "Tuple-Oriented Compression for Large-Scale Mini-Batch Stochastic Gradient Descent". In: *Proceedings of 2019 ACM SIGMOD International Conference on Management of Data*. 1517–1534.

Li, R., W. Gatterbauer, and M. Riedewald. (2020). "Near-Optimal Distributed Band-Joins through Recursive Partitioning". In: *Proceedings of 2020 ACM SIGMOD International Conference on Management of Data*. 2375–2390.

Li, X., Y. Liang, S. Yan, L. Jia, and Y. Li. (2019b). "A Coordinated Tiling and Batching Framework for Efficient GEMM on GPUs". In: *Proceedings of 2019 PPoPP Symposium on Principles and Practice of Parallel Programming*. 229–241.

Libkin, L., R. Machlin, and L. Wong. (1996). "A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques". In: *Proceedings of 1996 ACM SIGMOD International Conference on Management of Data*. 228–239.

Lim, K.-T., D. Maier, J. Becla, M. Kersten, Y. Zhang, and M. Stonebraker. (2012). "Array QL Syntax". URL: http://www.xldb.org/wp-content/uploads/2012/09/ArrayQL-Draft-4.pdf.

Lippmeier, B. and G. Keller. (2011). "Efficient Parallel Stencil Convolution in Haskell". In: *Proceedings of 2011 Haskell ACM Symposium on Haskell*. 59–70.

Liu, D.-R. and S. Shekhar. (1995). "A Similarity Graph-based Approach to Declustering Problems and Its Application towards Parallelizing Grid Files". In: *Proceedings of 1995 IEEE ICDE International Conference on Data Engineering*. 373–381.

Lowenthal, D. K. (2000). "Accurately Selecting Block Size at Runtime in Pipelined Parallel Programs". *Int. J. Parallel Program.* 28(3): 245–274.

Luo, S. (2020). *Automatic Matrix Format Exploration for Large Scale Linear Algebra, Ph.D. Thesis*. Rice University.

Luo, S., Z. J. Gao, M. N. Gubanov, L. L. Perez, and C. M. Jermaine. (2017). "Scalable Linear Algebra on a Relational Database System". In: *Proceedings of 2017 IEEE ICDE International Conference on Data Engineering*. 523–534.

Luo, S., D. Jankov, B. Yuan, and C. Jermaine. (2021). "Automatic Optimization of Matrix Implementations for Distributed Machine Learning and Linear Algebra". In: *Proceedings of 2021 ACM SIGMOD International Conference on Management of Data*. 1222–1234.

Lustosa, H. and F. Porto. (2019). "SAVIME: A Multidimensional System for the Analysis and Visualization of Simulation Data". *CoRR*. arXiv:1903.02949v2.

MADlib Development Team. (2022). "MADlib". URL: https://madlib.apache.org/.

Maier, D. (2012). "ArrayQL Algebra: version 3". URL: http://www.xldb.org/wp-content/uploads/2012/09/ArrayQL_Algebra_v3+.pdf.

Maier, D. and B. Vance. (1993). "A Call to Order". In: *Proceedings of 1993 PODS Symposium on Principles of Database Systems*. 1–16.

Marathe, A. P. and K. Salem. (2002). "Query Processing Techniques for Arrays". *VLDB Journal (VLDBJ)*. 11(1): 68–91.

Maruyama, N., T. Nomura, K. Sato, and S. Matsuoka. (2011). "Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers". In: *Proceedings of 2011 SC International Conference for High Performance Computing, Networking, Storage and Analysis*. 11:1–11:12.

Matthews, D. A. (2016). "High-Performance Tensor Contraction without Transposition". *CoRR*. arXiv:1607.00291v4.

Milner, R., M. Tofte, R. Harper, and D. MacQueen. (1997). *The Definition of Standard ML (revised)*. MIT Press.

Milo, T. and E. Altshuler. (2016). "An Efficient MapReduce Cube Algorithm for Varied Data Distributions". In: *Proceedings of 2016 ACM SIGMOD International Conference on Management of Data*. 1151–1165.

Misev, D. and P. Baumann. (2014). "Extending the SQL Array Concept to Support Scientific Analytics". In: *Proceedings of 2014 SSDBM International Conference on Scientific and Statistical Database Management*.

Moon, B., A. Acharya, and J. Saltz. (1996). "Study of Scalable Declustering Algorithms for Parallel Grid Files". In: *Proceedings of 1996 Parallel Processing Symposium*. 434–440.

Moon, B. and J. H. Saltz. (1998). "Scalability Analysis of Declustering Methods for Multidimensional Range Queries". *IEEE Transactions on Knowledge and Data Engineering (TKDE)*. 10(2): 310–327.

Morfonios, K., S. Konakas, Y. Ioannidis, and N. Kotsis. (2007). "ROLAP Implementations of the Data Cube". *ACM Comput. Surv.* 39(4).

MPI Forum. (2022). "Message Passing Interface (MPI)". URL: https://www.mpi-forum.org/.

Nandi, A., C. Yu, P. Bohannon, and R. Ramakrishnan. (2012). "Data Cube Materialization and Mining over MapReduce". *IEEE Transactions on Knowledge and Data Engineering (TKDE)*. 24(10): 1747–1759.

Nikolopoulos, D. (2004). "Dynamic Tiling for Effective Use of Shared Caches on Multithreaded Processors". *International Journal of High Performance Computing and Networking*. 2(1): 22–35.

NumPy Development Team. (2022). "NumPy". URL: https://numpy.org/.

NVIDIA. (2022). "cuBLAS". URL: https://docs.nvidia.com/cuda/cublas/.

O'Gorman, L., M. J. Sammon, and M. Seul. (2008). *Practical Algorithms for Image Analysis, 2nd edition*. Cambridge University Press.

Open Geospatial Consortium. (2022). “GeoTIFF Standard”. URL: https://www.ogc.org/standards/geotiff.

Oracle. (2022a). “Essbase”. URL: https://docs.oracle.com/en/database/other-databases/essbase/index.html.

Oracle. (2022b). “Spatial GeoRaster”. URL: https://docs.oracle.com/en/database/oracle/oracle-database/21/geors/index.html.

Otoo, E. J., D. Rotem, and S. Seshadri. (2007). “Optimal Chunking of Large Multidimensional Arrays for Data Warehousing”. In: *Proceedings of 2007 ACM DOLAP International Workshop on Data Warehousing and OLAP*. 25–32.

Ozsoyoglu, G., Z. M. Ozsoyoglu, and V. Matos. (1987). “Extending Relational Algebra and Relational Calculus with Set-valued Attributes and Aggregate Functions”. *ACM Transactions on Database Systems (TODS)*. 12(4): 566–592.

Papadopoulos, S., K. Datta, S. Madden, and T. G. Mattson. (2016). “The TileDB Array Data Storage Manager”. *PVLDB*. 10(4): 349–360.

Paradigm4. (2022a). “SciDB”. URL: https://github.com/Paradigm4/SciDB.

Paradigm4. (2022b). “SciDB Documentation”. URL: https://paradigm4.atlassian.net/wiki/spaces/scidb/overview?homepageId=2694289094.

Pedersen, T. B., C. S. Jensen, and C. E. Dyreson. (2001). “A Foundation for Capturing and Querying Complex Multidimensional Data”. *Information Systems*. 26(5): 383–423.

Peng, L. and Y. Diao. (2015). “Supporting Data Uncertainty in Array Databases”. In: *Proceedings of 2015 ACM SIGMOD International Conference on Management of Data*. 545–560.

Pentaho. (2022). “Mondrian OLAP Server”. URL: https://mondrian.pentaho.com/documentation/index.php.

Polychroniou, O., R. Sen, and K. A. Ross. (2014). “Track Join: Distributed Joins with Minimal Network Traffic”. In: *Proceedings of 2014 ACM SIGMOD International Conference on Management of Data*. 1483–1494.

Prabhakar, S., K. Abdel-Ghaffar, D. Agrawal, and A. E. Abbadi. (1998). "Cyclic Allocation of Two-Dimensional Data". In: *Proceedings of 1998 IEEE ICDE International Conference on Data Engineering*. 94–101.

PyTorch Development Team. (2022). "PyTorch". URL: https://pytorch.org/.

Qin, C. and F. Rusu. (2015). "Speculative Approximations for Terascale Distributed Gradient Descent Optimization". In: *Proceedings of 2015 ACM SIGMOD DanaC Workshop on Data Analytics in the Cloud*.

RasDaMan Development Team. (2022). "RasDaMan". URL: http://rasdaman.org/.

Renganarayana, L. and S. Rajopadhye. (2008). "Positivity, Posynomials and Tile Size Selection". In: *Proceedings of 2008 ACM/IEEE SC Conference on Supercomputing*.

Ritter, G., J. Wilson, and J. Davidson. (1990). "Image Algebra: An Overview". *Computer Vision, Graphics, and Image Processing*. 49(1): 297–331.

Rusu, F. and Y. Cheng. (2013). "A Survey on Array Storage, Query Languages, and Systems". *CoRR*. arXiv:1302.0103.

Salton, G., A. Wong, and C. S. Yang. (1975). "A Vector Space Model for Automatic Indexing". *Commun. ACM*. 18(11): 613–620.

Sarawagi, S. and M. Stonebraker. (1994). "Efficient Organization of Large Multidimensional Arrays". In: *Proceedings of 1994 IEEE ICDE International Conference on Data Engineering*. 328–336.

ScaLAPACK Development Team. (2022). "Scalable Linear Algebra PACKage (ScaLAPACK)". URL: http://www.netlib.org/scalapack/.

Seering, A., P. Cudre-Mauroux, S. Madden, and M. Stonebraker. (2012). "Efficient Versioning for Scientific Array Databases". In: *Proceedings of 2012 IEEE ICDE International Conference on Data Engineering*. 1013–1024.

Seo, S., E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng. (2010). "HAMA: An Efficient Matrix Computation with the MapReduce Framework". In: *Proceedings of 2010 IEEE CLOUDCOM International Conference on Cloud Computing Technology and Science*. 721–726.

Shao, Z., J. Han, and D. Xin. (2004). "MM-Cubing: Computing Iceberg Cubes by Factorizing the Lattice Space". In: *Proceedings of 2004 SSDBM International Conference on Scientific and Statistical Database Management*.

Shi, Y., U. N. Niranjan, A. Anandkumar, and C. Cecka. (2016). "Tensor Contractions with Extended BLAS Kernels on CPU and GPU". *CoRR*. arXiv:1606.05696.

Shoshani, A. (1997). "OLAP and Statistical Databases: Similarities and Differences". In: *Proceedings of 1997 PODS Symposium on Principles of Database Systems*. 185–196.

Sommer, J., M. Boehm, A. V. Evfimievski, B. Reinwald, and P. J. Haas. (2019). "MNC: Structure-Exploiting Sparsity Estimation for Matrix Expressions". In: *Proceedings of 2019 ACM SIGMOD International Conference on Management of Data*. 1607–1623.

Song, J., H. V. Jagadish, and G. Alter. (2021). "SDTA: An Algebra for Statistical Data Transformation". In: *Proceedings of 2021 SSDBM International Conference on Scientific and Statistical Database Management*. 109–120.

Soroush, E. and M. Balazinska. (2013). "Time Travel in a Scientific Array Database". In: *Proceedings of 2013 IEEE ICDE International Conference on Data Engineering*.

Soroush, E., M. Balazinska, and D. L. Wang. (2011). "ArrayStore: A Storage Manager for Complex Parallel Array Processing". In: *Proceedings of 2011 ACM SIGMOD International Conference on Management of Data*. 253–264.

Springer, P. and P. Bientinesi. (2018). "Design of a High-Performance GEMM-like Tensor–Tensor Multiplication". *ACM Transactions on Mathematical Software (TOMS)*. 44(3).

Springer, P. and C.-H. Yu. (2019). "cuTENSOR: High-Performance CUDA Tensor Primitives". URL: https://developer.nvidia.com/cutensor.

Stonebraker, M., P. Brown, A. Poliakov, and S. Raman. (2011). "The Architecture of SciDB". In: *Proceedings of 2011 SSDBM International Conference on Scientific and Statistical Database Management*. 1–16.

Szalay, A. S. (2008). "The Sloan Digital Sky Survey and Beyond". *SIGMOD Rec.* 37(2): 61–66.

TensorFlow Development Team. (2022). "TensorFlow". URL: https://www.tensorflow.org/.

The Dask Development Team. (2022). "Dask Arrays". URL: https://docs.dask.org/en/latest/array.html.

The FITS Support Office. (2022). "Flexible Image Transport System (FITS)". URL: https://fits.gsfc.nasa.gov/fits_home.html.

The GDAL Development Team. (2022). "Geospatial Data Abstraction Library (GDAL)". URL: https://gdal.org/.

The HDF5 Group. (2020). "HDF5". URL: http://www.hdfgroup.org/HDF5/.

The netCDF Development Team. (2022). "netCDF Operators (NCO)". URL: http://nco.sourceforge.net/.

The PostGIS Development Team. (2022). "PostGIS Raster Data Management and Applications". URL: http://postgis.net/docs/manual-dev/using_raster_dataman.html.

The PostgreSQL Development Team. (2020). "PostgreSQL". URL: http://www.postgresql.org/.

Thomas, A. and A. Kumar. (2018). "A Comparative Evaluation of Systems for Scalable Linear Algebra-based Analytics". *PVLDB.* 11(13): 2168–2182.

TileDB, Inc. (2022a). "TileDB". URL: https://github.com/TileDB-Inc.

TileDB, Inc. (2022b). "TileDB Documentation". URL: https://docs.tiledb.com/.

Tomlin, C. D. (1990). *Geographic Information Systems and Cartographic Modelling.* Prentice Hall.

Torlone, R. (2003). "Multidimensional Databases". In: IGI Global. Chap. Conceptual Multidimensional Models.

UniData. (2022). "Network Common Data Form (NetCDF)". URL: https://www.unidata.ucar.edu/software/netcdf/.

Vasilache, N., O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. (2018). "Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions". *CoRR.* arXiv:1802.04730.

Vassiliadis, P. (1998). "Modeling Multidimensional Databases, Cubes and Cube Operations". In: *Proceedings of 1998 SSDBM International Conference on Scientific and Statistical Database Management*. 53–62.

Vassiliadis, P. and T. Sellis. (1999). "A Survey of Logical Models for OLAP Databases". *ACM SIGMOD Record*. 28(4): 64–69.

Wang, Y., A. Nandi, and G. Agrawal. (2014). "SAGA: Array Storage as a DB with Support for Structural Aggregations". In: *Proceedings of 2014 SSDBM International Conference on Scientific and Statistical Database Management*.

Wang, Y., Y. Su, and G. Agrawal. (2013a). "Supporting a Light-Weight Data Management Layer over HDF5". In: *Proceedings of 2013 IEEE/ACM CCGRID International Symposium on Cluster, Cloud and Grid Computing*. 335–342.

Wang, Z., Y. Chu, K. Tan, D. Agrawal, A. E. Abbadi, and X. Xu. (2013b). "Scalable Data Cube Analysis over Big Data". *CoRR*. arXiv:1311.5663.

Whitehorn, M., R. Zare, and M. Pasumansky. (2005). *Fast Track to MDX*. Springer-Verlag.

Widmann, N. and P. Baumann. (1998). "Efficient Execution of Operations in a DBMS for Multidimensional Arrays". In: *Proceedings of 1998 SSDBM International Conference on Scientific and Statistical Database Management*. 155–165.

Wikipedia. (2020). "Basic Linear Algebra Subprograms". URL: https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms.

Wikipedia. (2022a). "Comparison of OLAP Servers". URL: https://en.wikipedia.org/wiki/Comparison_of_OLAP_servers.

Wikipedia. (2022b). "Math Kernel Library". URL: https://en.wikipedia.org/wiki/Math_Kernel_Library.

Wolf, M. (1989). "More Iteration Space Tiling". In: *Proceedings of 1989 ACM/IEEE SC Conference on Supercomputing*. 655–664.

Wolf, M. E. and M. S. Lam. (1991). "A Data Locality Optimizing Algorithm". In: *Proceedings of 1991 ACM SIGPLAN PLDI Conference on Programming Language Design and Implementation*. 30–44.

Wu, E., S. Madden, and M. Stonebraker. (2013). "SubZero: A Fine-Grained Lineage System for Scientific Databases". In: *Proceedings of 2013 IEEE ICDE International Conference on Data Engineering*.

Wu, K., E. J. Otoo, and A. Shoshani. (2006). "Optimizing Bitmap Indices with Efficient Compression". *ACM Transactions on Database Systems (TODS)*. 31(1): 1–38.

Xing, H. and G. Agrawal. (2018). "COMPASS: Compact Array Storage with Value Index". In: *Proceedings of 2018 SSDBM International Conference on Scientific and Statistical Database Management*.

Xing, H. and G. Agrawal. (2019). "Accelerating Array Joining with Integrated Value-Index". In: *Proceedings of 2019 SSDBM International Conference on Scientific and Statistical Database Management*. 145–156.

Xing, H., S. Floratos, S. Blanas, S. Byna, Prabhat, K. Wu, and P. Brown. (2018). "ArrayBridge: Interweaving Declarative Array Processing in SciDB with Imperative HDF5-Based Programs". In: *Proceedings of 2018 IEEE ICDE International Conference on Data Engineering*. 977–988.

Yu, L., Y. Shao, and B. Cui. (2015). "Exploiting Matrix Dependency for Efficient Distributed Matrix Computation". In: *Proceedings of 2015 ACM SIGMOD International Conference on Management of Data*. 93–105.

Yu, Y., M. Tang, W. G. Aref, Q. M. Malluhi, M. M. Abbas, and M. Ouzzani. (2017). "In-Memory Distributed Matrix Computation Processing and Optimization". In: *Proceedings of 2017 IEEE ICDE International Conference on Data Engineering*. 1047–1058.

Yuan, B., D. Jankov, J. Zou, Y. Tang, D. Bourgeois, and C. Jermaine. (2021). "Tensor Relational Algebra for Distributed Machine Learning System Design". *PVLDB*. 14(8): 1338–1350.

Zaharia, M., M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. (2010). "Spark: Cluster Computing with Working Sets". In: *Proceedings of 2010 USENIX HotCloud Conference on Hot Topics in Cloud Computing*.

Zalipynis, R. A. R. (2018). "ChronosDB: Distributed, File Based, Geospatial Array DBMS". *PVLDB*. 11(10): 1247–1261.

Zalipynis, R. A. R. (2021). "Array DBMS: Past, Present, and (near) Future". *PVLDB*. 14(12): 3186–3189.

Zee, F. van and R. van de Geijn. (2015). "BLIS: A Framework for Rapidly Instantiating BLAS Functionality". *ACM Transactions on Mathematical Software (TOMS)*. 41(3).

Zhang, Y., M. Kersten, M. Ivanova, and N. Nes. (2011). "SciQL: Bridging the Gap between Science and Relational DBMS". In: *Proceedings of 2011 IDEAS Symposium on International Database Engineering and Applications*. 124–133.

Zhao, W., F. Rusu, B. Dong, and K. Wu. (2016). "Similarity Join over Array Data". In: *Proceedings of 2016 ACM SIGMOD International Conference on Management of Data*. 2007–2022.

Zhao, W., F. Rusu, B. Dong, K. Wu, A. Y. Q. Ho, and P. Nugent. (2018). "Distributed Caching for Processing Raw Arrays". In: *Proceedings of 2018 SSDBM International Conference on Scientific and Statistical Database Management*.

Zhao, W., F. Rusu, B. Dong, K. Wu, and P. Nugent. (2017). "Incremental View Maintenance over Array Data". In: *Proceedings of 2017 ACM SIGMOD International Conference on Management of Data*. 139–154.

Zhao, Y., P. M. Deshpande, and J. F. Naughton. (1997). "An Array-Based Algorithm for Simultaneous Multidimensional Aggregates". In: *Proceedings of 1997 ACM SIGMOD International Conference on Management of Data*. 159–170.